Semestrální projekt NI-PDP 2022/2023:

Paralelní algoritmus pro řešení problému

Ondřej Wrzecionko

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

25. dubna 2023

1 Definice problému a popis sekvenčního algoritmu

Problémem, který budu ve svém semestrálním projektu řešit, je problém **BPO**: nalezení bipartitního souvislého podgrafu hranově ohodnoceného grafu s maximální vahou.

Vstupem programu bude **graf** $G(V, E)$, což je jednoduchý neorientovaný hranově ohodnocený souvislý graf o n uzlech a průměrném stupni k, kde váhy hran jsou z intervalu 80 a 120.

Naším cílem je najít **podmnožinu hran** F takovou, aby byl podgraf $G(V, F)$ bipartitní, souvislý a zároveň součet ohodnocení jeho hran byl **maximální** mezi všemi takovými bipartitními grafy.

Sekvenční algoritmus prohledává stavový prostor všech stavů. Zde je třeba si uvědomit, že stav je určen "obarvením" (přiřazením do jedné nebo druhé partity) jednotlivých vrcholů. Máme-li dáno rozdělení vrcholů do partit, je pak už jednoznačně definováno také, které hrany budou součástí grafu – mezi dvěma vrcholy stejné partity hrana **nepovede**, mezi dvěma vrcholy různé partity hrana **povede** právě tehdy, když vedla v původním grafu. Tímto postupem máme zaručenou maximalitu, protože váhy hran jsou kladné *(takže hranu mezi různě obarvenými vrcholy přidat chceme)*.

Konstrukci budeme tedy provádět pomocí **BB-DFS**, kdy si v každém stavu držíme pozici v poli hran *(procházíme hrany)* a v jednom kroku projdeme jednu hranu *(hloubka rekurze bude tedy maximálně počet hran)*. U hrany rozhodujeme o tom, zda ji **přidáme**, nebo **nepřidáme**. Podíváme se na to, jaké barvy mají **vrcholy**, mezi kterými hrana vede. Pokud se jedná o vrcholy **stejné** barvy, hranu nepřidáváme, pokud **různé** barvy, hranu přidáme. Pokud je obarven pouze **jeden** vrchol, druhý obarvíme stejnou barvou a hranu nepřidáme, nebo jinou barvou a hranu přidáme. Pokud není obarven zatím **ani jeden** vrchol, vyzkoušíme **všechny 4** možnosti obarvení (přidat hranu, bílá - černá, černá - bílá a nepřidat hranu, bílá - bílá, černá - černá).

Už tato složitější logika nám v některých případech **snižuje** počet rekurzivních větvení (ze 4 na 2 nebo 1). Dalším vylepšením je **prořezávání**. V každém kroku si držíme váhu nejlepšího řešení a součet vah zbývajících hran. Pokud je už jasné, že ani kdybychom přidali všechny hrany, tak **nepřekonáme** nejlepší řešení, tak nemá smysl pokračovat a výpočet **ukončíme**.

Pro zjednodušení výpočtu funkce `weight_inner` vrací pouze číslo (`size_t`) symbolizující váhu nejlepšího řešení. Kdybychom chtěli zobrazit či vypsat všechny hrany použité v řešení, stačilo by upravit typ proměnné `currentBest` na `State` a do `currentBest` pak ukládat tento stav. Po skončení programu bychom pak na základě obarvení a logiky v odstavci výše vypsali použité hrany.

Program pro sekvenční řešení úloh jsem **spustil** na všech instancích ze sady na Courses. Ukázalo se ale, že je můj program tak **rychlý**, že i nejtěžší instance, pro kterou referenční řešení běželo 5700 sekund moje řešení běželo pouhých 0.017s *(a provedlo 2 509 822 rekurzivních volání oproti 1.3 T referenčního)*.

Pomocí generátoru z Courses jsem tedy vygeneroval **obtížnější** instance s 20 až 30 vrcholy a odpovídajícími počty průměrných stupňů uzlu. Na těch už program běžel v desítkách sekund až desítkách minut. Jednotlivé doby běhů pro sekvenční řešení uvádím v tabulkách Tabulka časů běhů pro jednotlivé instance z Courses a Tabulka časů běhů pro jednotlivé vygenerované instance

Poznámka: Časy běhu v tabulkách na následující stránce byly měřeny na školních počítačích, tedy na **Intel Xeon CPU E3-1245** v6 @ 3.70GHz. Paralelní konfigurace byly spuštěny na **8 vláknech**.

Graf	Váha	Rekurzivních volání	Čas (sekvenční)	Čas (parallel task)	Čas (parallel data)
10_3	1300	259	0.007s	0.025s	0.025s
10_5	1885	4 950	0.010s	0.012s	0.011s
10_6	2000	7 104	0.010s	0.011s	0.011s
10_7	2348	15 725	0.011s	0.012s	0.012s
12_3	1422	3 518	0.010s	0.030s	0.007s
12_5	2219	22 389	0.007s	0.016s	0.006s
12_6	2533	40 433	0.006s	0.007s	0.013s
12_9	3437	133 K	0.007s	0.007s	0.006s
13_12	4182	498 K	0.009s	0.008s	0.008s
13_9	3700	228 K	0.007s	0.009s	0.006s
15_12	5380	1.6 M	0.018s	0.014s	0.007s
15_14	5578	2.5 M	0.017s	0.012s	0.008s
15_4	2547	21 232	0.006s	0.005s	0.005s
15_5	2892	148 K	0.007s	0.008s	0.006s
15_6	3353	220 K	0.006s	0.008s	0.006s
15_8	3984	753 K	0.009s	0.011s	0.007s
17_10	5415	5.3 M	0.031s	0.027s	0.011s

Table 1: Tabulka časů běhů pro jednotlivé instance z Courses

Graf	Váha	Rekurzivních volání	Čas (sekvenční)	Čas (parallel task)	Čas (parallel data)
20_16	9353	111 M	0.517s	0.235s	0.229s
20_17	9768	132 M	0.628s	0.231s	0.127s
20_19	10288	154 M	0.727s	0.253s	0.140s
21_15	9570	197 M	0.968s	0.444s	0.393s
22_17	11015	517 M	2.266s	0.952s	0.461s
23_20	12902	1.4 G	6.326s	2.638s	1.429s
24_23	14844	3.5 G	16.149s	5.019s	3.152s
25_16	12105	3.9 G	16.677s	7.635s	3.409s
25_22	15594	6.4 G	28.416s	9.895s	5.755s
26_25	17477	17.3 G	76.146s	23.859s	15.349s
27_19	15470	21.3 G	87.856s	35.597s	19.900s
28_19	15758	38.7 G	170.7s	72.1s	63.2s
28_24	18729	65.5 G	234.4s	90.6s	54.3s
29_26	20810	161.2 G	586.9s	204.5s	125.8s

Table 2: Tabulka časů běhů pro jednotlivé vygenerované instance

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

Prvním krokem pro paralelizaci bylo použít knihovnu OpenMP, a to konkrétně **taskový paralelismus**. Ten je v OpenMP velmi jednoduchý na implementaci, jelikož stačilo v podstatě pouze přidat direktivu preprocesoru `#pragma omp task if(state.matrixPos < state.matrixLen / TASK_LEN)` ke každému rekurzivními volání funkce `weight_inner`, přidat aktualizaci globální proměnné nejlepšího stavu do `#pragma omp critical` a zaobalit první volání funkce do paralelního regionu s `#pragma omp single`, aby se DFS nepustilo na každém vlákně.

OpenMP pak na prvním vlákně spustí první `weight_inner`, a při každém dalším volání vytvoří nový **task**, který přidá do task poolu a distribuuje jednotlivým vláknům. Aby **nedocházelo** k vytváření velkého množství tasků, které by pak kvůli režii synchronizaci mezi vlákny **zpomalilo** program, používám podmínku, že task se spustí pouze tehdy, pokud je pozice v matici v první `1 / TASK_LEN`-tině. Jako nejvhodnější konstanta `TASK_LEN` se mi po spuštění několika běhů na několika instancích osvědčila 25.

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

Implementace v OpenMP pomocí **datového paralelismu** již byla o něco složitější. Nejprve jsem změnil **wrapper** funkci `weight`, která v taskovém paralelismu pouze zavolala `weight_inner` a to tak, aby s pomocí BFS vygenerovala 2 · počet vláken stavů. Pro to jsem vytvořil lehce upravenou funkci `weight_inner_bfs`, která má úplně stejnou implementaci, jako `weight_inner` s tím rozdílem, že místo rekurzivního volání sebe sama přidává nově vzniklý stav do **fronty**. Tato funkce se volá tak dlouho, dokud není dosažen potřebný počet stavů.

Jakmile je vygenerován dostatečný počet stavů, s pomocí direktivy `#pragma omp parallel for` jednotlivé stavy distribuuji mezi jednotlivá vlákna, na které zavolám původní rekurzivní funkci `weight_inner` (přejmenovanou na `weight_seq_rec`), která oproti sekvenčnímu řešení akorát používá ono `#pragma omp critical`.

Stejně jako sekvenční program, tak i oba dva paralelní programy (taskový i datový paralelismus) jsem spustil na školním počítači. Na ukázkových datech z Courses byly programy ještě **pomalejší**, jelikož byla tato data tak malá a jednoduchá, že režie **převážila** ono paralelní zrychlení. Na složitějších instancích již bylo zrychlení znát a jako **efektivnější** se ukázal **datový** paralelismus – především díky tomu, že pomocí BFS rozdělí stavový prostor rovnoměrněji mezi jednotlivá vlákna než taskový pomocí DFS.

Pokud se podíváme na **graf zrychlení**, je zajímavé, že zatímco zrychlení u taskového paralelismu je (na 8 vláknech) **stabilně** mezi 2-3x, zrychlení u datového paralelismu je většinou okolo 5x, ale na některých instancích **spadne** na 2,5 – na některých instancích bude tedy graf stavového prostoru značně nevyvážený *(jedno vlákno pak počítá v situaci, kdy už ostatní dopracovala).*

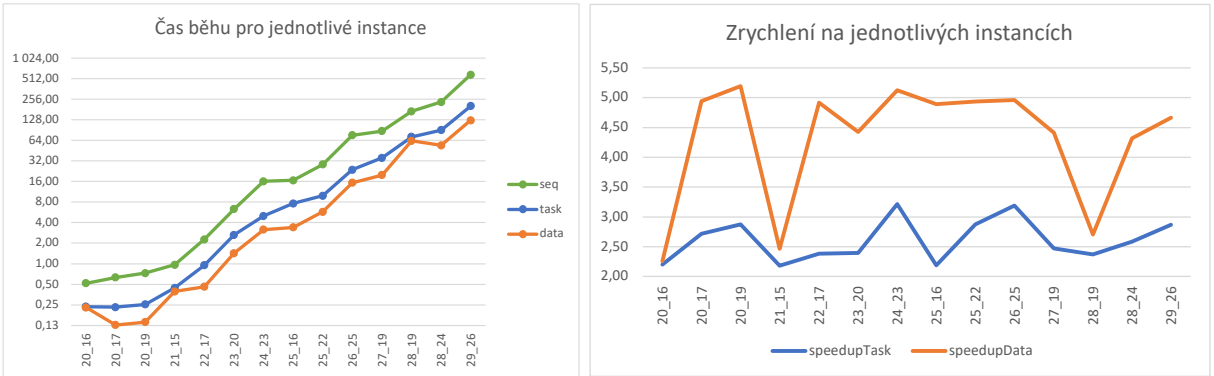

Figure 1: Časy běhu a zrychlení na jednotlivých instancích

4 Popis paralelního algoritmu a jeho implementace v MPI

Při tvorbě paralelního algoritmu pro **MPI** jsem vycházel z verze pro OpenMP. Na základě provedených měření se ukázal **datový paralelismus** jako efektivnější způsob, rozhodl jsem se proto pro model, kdy na jednotlivých **Slave** uzlech poběží paralelní program s datovým paralelismem a **Master** bude tento výpočet pouze řídit.

Na klastru Star, který máme k dispozici, toto **není** optimální řešení, jelikož se zde nachází 4 stroje s 20 jádry, a při rozdělení na 1 Mastera a 3 Slavy tak pro výpočet budou použita pouze jádra na Slave procesech, tento model by byl tedy vhodnější pro klastr, kde by master uzel mohl být méně výkonný.

Princip **distribuovaného** algoritmu je následující: Master ve funkci `weight_master` načte graf a pomocí **BFS** vygeneruje $(slave\_count + 1) \cdot 2$ stavů, které rozešle jednotlivým pracovním uzlům s tagem `TAG_WORK`. Ty ve funkci `slaveWork` v cyklu čekají na práci, a jakmile ji dostanou, tak pomocí OpenMP datového paralelismu provedou paralelní výpočet a výsledek předají zpět pomocí tagu `TAG_DONE`. Master tak průběžně rozděluje práci, dokud nějaká zbývá. Když je fronta úloh prázdná, počká na dokončení práce na všech pracovních uzlech *(ty, kde práce doběhla, ukončí zprávou `TAG_TERMINATE`)*.

Aby byl program **efektivnější**, pokaždé, když Master od pracovního uzlu obdrží výsledek, tak si aktualizuje aktuálně nejlepší řešení a předá ho pracovnímu uzlu – probíhá tak částečná synchronizace nejlepších řešení napříč pracovními uzly, a může tedy docházet k efektivnějšímu **prořezávání**.

Distribuovaný paralelní program pomocí MPI jsem již na **všech** mnou vygenerovaných instancích nespouštěl, jelikož nemám k dispozici klastr školních počítačů. Spustil jsem jej pouze na třech instancích, které jsem zároveň pustil i sekvenčně, a to vše na klastru Star – více v následující kapitole Popis paralelního algoritmu a jeho implementace v MPI.

5 Naměřené výsledky a vyhodnocení

Pro **měření** na klastru Star jsem vybral tři instance, které mají **sekvenční** dobu běhu na jednom uzlu Star mezi 2 a 10 minutami – jedná se o mnou vygenerované instance 27_19, 28_19 a 28_24. Pro měření jsem upravil zdrojové kódy sekvenčního, OpenMP (datový paralelismus) a MPI programů tak, aby měřily pouze čas výpočtu *(jak dlouho trvá volání příslušné funkce `weight`)* a zároveň jsem přidal do OpenMP a MPI programu parametr nastavující **počet** vláken, na kterých se program spustí.

Následně jsem použil předvytvořené skripty z `/home/mpi`, které jsem vhodně **upravil** a umístil do složky `scripts`. Nachází se zde tedy **skripty** pro spuštění sekvenčního programu, OpenMP a MPI programu s 2, 4, 8, 10, 16 a 20 jádry. Dále jsem si vytvořil pomocné skripty pro **kompilaci** a **spuštění**:

```
# Build Sequential solution
g++ -Wall -pedantic main_seq.cpp -O3 -o main_seq

# Build OpenMP solution
g++ -Wall -pedantic -fopenmp main_par_data.cpp -O3 -o main_par

# Build MPI solution
mpic++ main.cpp -O3 -fopenmp -o main_pdp
```

Listing 1: Skript pro kompilaci na Staru

```
# Start sequential job
qrun 20c 1 pdp_serial scripts/serial_job.sh

for i in {2,4,8,10,16,20}
do
  # Start OpenMP jobs
  qrun 20c 1 pdp_serial "scripts/openmp_job$i.sh"

  # Start MPI jobs (master + 2 slaves)
  qrun 20c 3 pdp_long "scripts/parallel_job$i.sh"

  # Start MPI jobs (master + 3 slaves)
  qrun 20c 4 pdp_fast "scripts/parallel_job$i.sh"
done
```

Listing 2: Skript pro spuštění na Staru. Všimněte si, že k spuštění na masteru a 2 slavech používám frontu `pdp_long`, protože zvláště na nižším počtu jader zde hrozí, že by program nestihl doběhnout pod 3 minuty.

Skript jsem **spustil** na každé z těchto instancí a výsledky jsem zapsal do následující **tabulky** a vytvořil z nich následující grafy (upozornění: osa y používá logaritmickou škálu), které budu analyzovat v kapitole Závěr:

Graf	27_19	28_19	28_24
Váha	15 470	15 758	18 729
Rekurzivních volání	21.3 G	38.7 G	65.5 G
Čas (sekvenční)	166.583 s	321.817 s	530.419 s
Čas (OMP, 2 vlákna)	82.231 s	158.141 s	260.696 s
Čas (OMP, 4 vlákna)	96.659 s	80.066 s	291.926 s
Čas (OMP, 8 vláken)	20.707 s	122.108 s	65.995 s
Čas (OMP, 10 vláken)	20.801 s	79.593 s	98.695 s
Čas (OMP, 16 vláken)	10.583 s	20.267 s	98.644 s
Čas (OMP, 20 vláken)	14.709 s	20.239 s	65.750 s
Čas (MPI/3, 2 vlákna)	47.160 s	97.083 s	128.017 s
Čas (MPI/3, 4 vlákna)	20.385 s	39.226 s	142.958 s
Čas (MPI/3, 8 vláken)	29.564 s	43.320 s	32.337 s
Čas (MPI/3, 10 vláken)	20.321 s	33.892 s	48.161 s
Čas (MPI/3, 16 vláken)	5.255 s	19.590 s	48.055 s
Čas (MPI/3, 20 vláken)	7.479 s	17.185 s	32.178 s
Čas (MPI/4, 2 vlákna)	35.114 s	58.276 s	95.822 s
Čas (MPI/4, 4 vlákna)	15.251 s	29.345 s	106.708 s
Čas (MPI/4, 8 vláken)	22.264 s	38.630 s	24.295 s
Čas (MPI/4, 10 vláken)	14.984 s	29.035 s	36.120 s
Čas (MPI/4, 16 vláken)	3.923 s	7.572 s	35.722 s
Čas (MPI/4, 20 vláken)	5.602 s	10.785 s	24.098 s

Table 3: Tabulka naměřených časů běhů

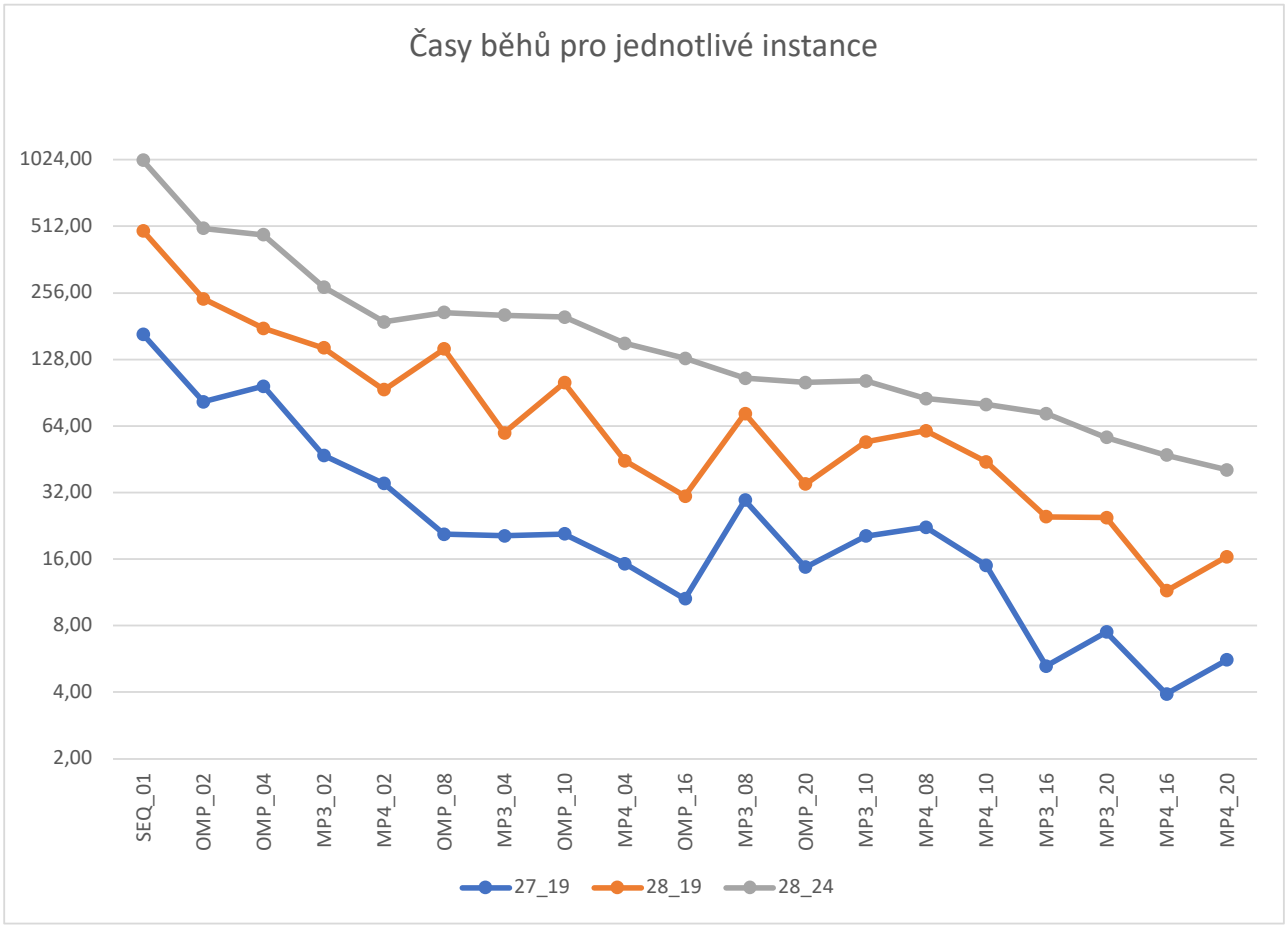

Figure 2: Časy běhu na jednotlivých instancích

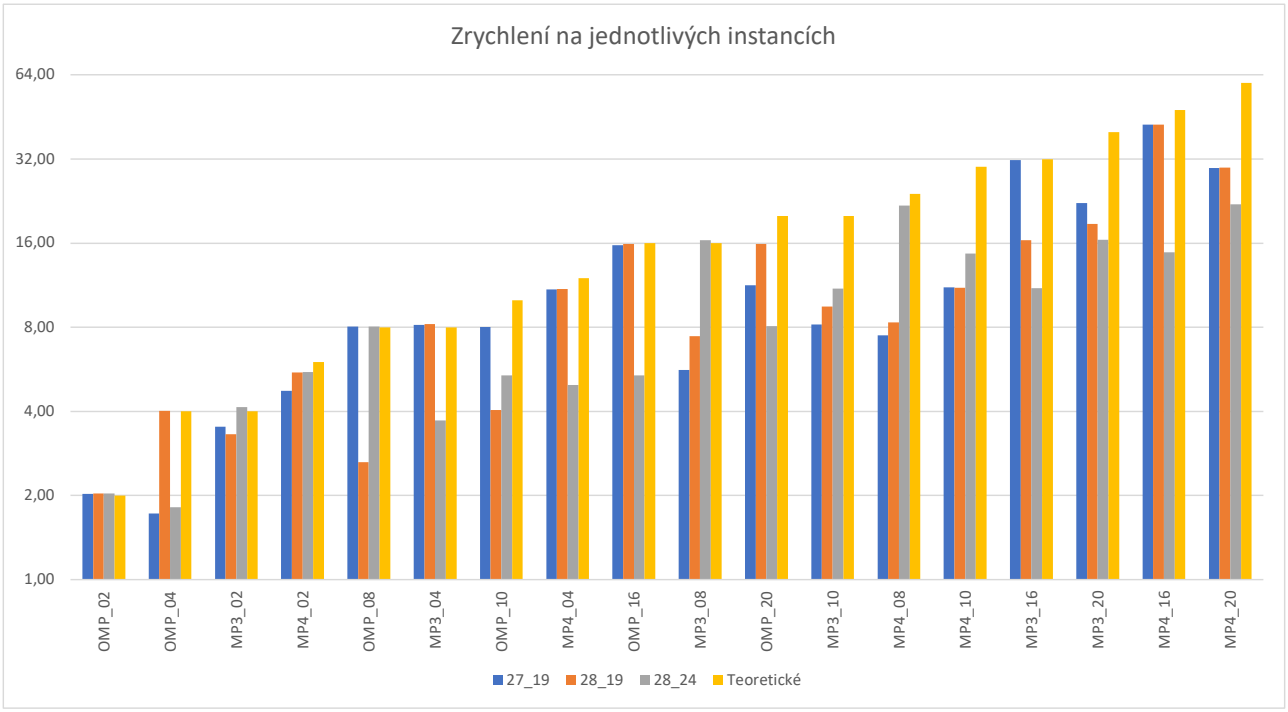

Figure 3: Zrychlení na jednotlivých instancích

6 Závěr

V semestrální práci jsem si vyzkoušel nejprve **návrh** sekvenčního řešení algoritmu a následně jeho **paralelizaci** – nejprve pomocí **OpenMP** na jednom zařízení, následně pomocí **MPI** na více zařízeních. Na závěr jsem **změřil** efektivitu mého řešení na třech vybraných instancích se sekvenční složitostí mezi 2 a 10 minutami.

Na základě Tabulka naměřených časů běhů a grafů Časy běhu na jednotlivých instancích Zrychlení na jednotlivých instancích můžu provést analýzu mého řešení. První věc, které si lze všimnout, je **superlineární** zrychlení (lehce nad 2) na dvou vláknech u OpenMP u všech tří instancí. Další zajímavý fakt je ten, že s **rostoucím** počtem vláken vždy **neklesá** doba běhu – například OpenMP program na 4 vláknech běží déle než na 2 vláknech na instancích 27_19 a 28_24 – nejspíše se práce dokáže mezi vlákna rozdělit **rovnoměrněji** právě v tomto případě. Z grafu lze vypozorovat také to, že program má pro 20 vláken mnohem **menší** zrychlení, než pro 16 *(nejspíše je už vláken tolik, že se nedaří tak dobře rozdělit práci)*. V neposlední řadě lze také zjistit, že zrychlení je mnohem **větší** u grafů s **menším** průměrným stupněm (19 vs 24).

Semestrální práce mi přišla jakožto velmi **přínosná**, bylo skvělé vidět, jak jde algoritmus, který **původně** v sekvenčním programu bez prořezávání **nedokáže** zpracovat ani grafy o 10 vrcholech s průměrným stupněm 3 pomocí prořezávání, **paralelizace** a použití distribuovaného programování vylepšit tak, že **dokáže** za 5 sekund zpracovat graf o 27 vrcholech s průměrným stupněm 19.