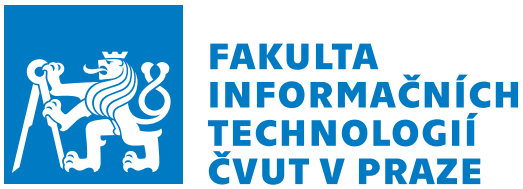
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



NI-PDP – Paralelní a distribuované programování

Minimální hranový řez hranově ohodnoceného grafu

Autor: Jan Babák

PRAHA, DUBEN 2023

Obsah

1 Definice problému

1.1 Vstupní data

1. n = přirozené číslo představující počet uzlů grafu G, $150 > n \geq 10$
2. k = přirozené číslo představující průměrný stupeň uzlu grafu G, $3n/4 > k \geq 5$
3. $G(V, E)$ = jednoduchý souvislý neorientovaný hranově ohodnocený graf o n uzlech a průměrném stupni k, váhy hran jsou z intervalu $< 80, 120 >$
4. a = přirozené číslo, $5 \leq a \leq n/2$
5. Mnou vytvořený program načítá tato data ze vstupních souborů poskytnutých na Courses.

1.2 Úkol

Nalezněte rozdělení množiny uzlů V do dvou disjunktních podmnožin X, Y tak, že množina X obsahuje a uzlů, množina Y obsahuje $n - a$ uzlů a součet ohodnocení všech hran $\{u, v\}$ takových, že u je z X a v je z Y (čili velikost hranového řezu mezi X a Y), je minimální.

1.3 Výstup algoritmu

V ukázkovém výstupu je na první řádce jméno vstupního souboru, na druhé řádce je minimální váha řezu. Následuje pole nul a jedniček, které značí rozdělení vrcholů na 2 disjunktní podmnožiny X a Y. Index v poli značí id vrcholu.

```
graf_mro/graf_20_12.txt
Minimal weight: 5060
[0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0]
```

2 Sekvenční algoritmus

2.1 Popis algoritmu

Minimální hranový řez hranově ohodnoceného grafu je NP-úplný problém, tedy zatím neexistuje žádný polynomiální algoritmus, který by ho efektivně řešil a je nutné použít hrubou sílu. Přestože používám různé přístupy, pomocí kterých hledám řešení rychleji, v nejhorším případě musím vyzkoušet všechny možnosti rozdělení vrcholů do množin X a Y.

Jedná se o rekurzivní algoritmus, který začíná s polem délky počtu vrcholů. Index v poli značí id vrcholu a hodnota na indexu značí ve které množině se vrchol nachází (0 pro X, 1 pro Y a -1 pro dosud nezařazené vrcholy. Na začátku jsou všechny vrcholy nezařazené. Rekurzivní funkce vytvoří kopii tohoto pole (máme tedy 2 pole), první nezařazený vrchol v prvním poli zařadí do X a první nezařazený vrchol ve druhém poli zařadí do Y a rekurzivně zavolá sama sebe – jednou s prvním polem a podruhé s druhým. Používám přístup DFS – tedy zanořování se do hloubky a tzv. backtracking. Pokud jsou všechny vrcholy v poli zařazené, tak v rekurzi nepokračuji, spočítám minimální váhu řezu a pokud je lepší než nejlepší zatím nalezené řešení, uložím tuto váhu a konfiguraci vrcholů.

2.2 Optimalizace

K základnímu algoritmu jsem přidal několik optimalizací, které ukončují rekurzi dříve – tzv. prořezávání (branch and bound) – pokud vím, že v této větvi stromu rekurzivních volání řešení není, neprohledávám ji. Přidal jsem tyto optimalizace:

- Pokud počet vrcholů v množině X je větší než a, není splněno zadání a není potřeba zkoušet rozřazovat další vrcholy a je možné větev zahodit.
- Stejně tak pokud počet vrcholů v množině Y přesáhne $n - a$.
- Pokud váha řezu doposud rozřazených vrcholů je větší než minimální nalezená váha, nemá smysl pokračovat.
- Dále počítám lowerbound (dolní odhad) váhy. Pokud je lowerbound vyšší než minimální nalezená váha, nemá smysl pokračovat. Lowerbound počítám tak, že všechny doposud nezařazené vrcholy zkusím vložit do X a potom do Y a spočítám váhu řezu, pro každý vrchol beru minimum z obou vah.

2.3 Měření

Měření bylo provedeno (stejně jako všechna následující měření) na školním klastru Star na výpočetním uzlu s dvěma CPU na základní desce, kde každé má k dispozici 10 vláken a 8GB paměti.

Ve všech měřeních budu značit vstupní úlohy písmeny a, b, c... Významy těchto písmen jsou v tabulce 1 (seřazeny od nejsnazších instancí po ty nejnáročnější). Vybral jsem pouze ty instance, jejich sekvenční řešení nepřesáhlo limit počítače (10 minut).

úloha	vstupní graf	velikost množiny X
a	graf_10_5.txt	5
b	graf_10_6b.txt	5
c	graf_20_7.txt	7
d	graf_20_7.txt	10
e	graf_20_12.txt	10
f	graf_30_10.txt	10
g	graf_30_10.txt	15
h	graf_30_20.txt	15
i	graf_40_8.txt	15

Tabulka 1: Legenda vstupních dat

úloha/čas	a	b	c	d	e	f	g	h	i
Sekvenční alg.	0.010s	0.011s	0.080s	0.128s	0.491s	7.988s	22.656s	295.930s	330.765s

Tabulka 2: Časy sekvenčního algoritmu pro různé vstupy

3 Task paralelismus v OpenMP

Po tvorbě nejjednoduššího algoritmu (sekvenčního) jsem pokračoval paralelním řešením využívající task paralelismus.

3.1 Popis algoritmu

Při tvorbě paralelního řešení pomocí task paralelismu jsem vycházel ze sekvenčního řešení. Tedy opět používám DFS k prohledávání do hloubky. Funkce na hledání nejlepšího řešení je ta samá rekurzivní funkce, v každém volání se větví na maximálně dvě rekurzivní volání. Rekurzivní funkci volám uvnitř paralelního regionu označeného direktivou `#pragma omp parallel`. První volání funkce je voláno s direktivou `#pragma omp single`, aby byla zavolána pouze jednou. Další rekurzivní volání jsou volána s `#pragma omp task` – tedy každé volání je zpracováno jiným vláknem. Počet vláken nastavuji jako parametr programu (provedl jsem měření pro různé počty vláken). Dále používám direktivu `#pragma omp taskwait`, která vytvoří bariéru – čekám, abych mohl po dokončení úlohy dealokovat použitou paměť. Poslední použitou direktivou je `#pragma omp critical`, kterou používám pro zajištění výlučného přístupu pro uložení nejlepšího řešení.

3.2 Optimalizace

Algoritmus používá všechny branch and bound optimalizace, které používá sekvenční algoritmus k prořezávání a zmenšení prohledávání stavového prostoru.

Dále jsem přidal optimalizaci paralelizace. Pokud algoritmus dorazí do fáze, kdy chybí prohledat tři a méně hladin, nezavolá paralelní rekurzivní fukci, ale sekvenční rekurzivní funkce, které prohledá zbylé hladiny. Výrazně se díky tomu sníží počet vytvořených vláken a tím i režie paralelizace.

Další optimalizace režie paralelizace je v kritické sekci, kde přepisuji nejlepší nalezené řešení. Nejdříve zkontroluji, jestli se opravu jedná o nejlepší řešení a až poté vstoupím do kritické sekce. Díky čemuž se nezamyká kritická sekce ve většině případů, kdy se nejedná o nejlepší řešení. Podmínka nejlepšího řešení zde musí být dvakrát, protože se řešení před vstupem do kritické sekce mohlo změnit.

```
        if (weight < minimalSplitWeight) {
#pragma omp critical
            {
                // if best, save it
                if (weight < minimalSplitWeight) {
                    minimalSplitWeight = weight;
                    for (int i = 0; i < graph.vertexesCount; i++) {
                        minimalSplitConfig[i] = config[i];
                    }
                }
            }
        }
```

3.3 Měření

Měření bylo provedeno na stejném hardwaru jako měření sekvenčního algoritmu. Byly použity ty samé vstupy. Na rozdíl od sekvenčního řešení jsem zde mohl testovat algoritmus pro různé počty vláken. Dle zadání jsem testoval pro 1, 2, 4, 6, 8, 12, 16 a 20 vláken (v názvu algoritmu značí 1t – jedno vlákno, 2t – dvě vlákna atd.

úloha/čas	a	b	c	d	e	f	g	h	i
omp task 1t	0.003s	0.003s	0.079s	0.146s	0.56s	9.632s	25.31s	348.184s	314.791s
opm task 2t	0.003s	0.003s	0.052s	0.103s	0.338s	5.342s	15.415s	192.093s	215.740s
opm task 4t	0.003s	0.003s	0.041s	0.088s	0.268s	3.801s	11.686s	151.349s	146.007s
omp task 6t	0.002s	0.002s	0.033s	0.066s	0.193s	2.912s	8.429s	97.451s	107.310s
omp task 8t	0.003s	0.003s	0.032s	0.070s	0.170s	2.578s	6.796s	94.021s	90.542s
omp task 12t	0.003s	0.003s	0.038s	0.079s	0.142s	1.919s	5.412s	52.767s	55.683s
omp task 16t	0.003s	0.004s	0.043s	0.096s	0.159s	2.080s	5.224	49.457s	53.268s
omp task 20t	0.006s	0.004s	0.050s	0.101s	0.175s	2.607s	5.635s	44.550s	63.473s

Tabulka 3: Časy task paralismu pro různé vstupy a počty vláken

4 Datový paralelismus v OpenMP

Dalším krokem byla implementace algoritmu opět pomocí knihovny OpenMP, tentokrát však s využitím datového paralelismu.

4.1 Popis algoritmu

Algoritmus využívající datový paralelismus je opět postaven na původním sekvenčním řešení, ale využívá jiný přístup k paralelizaci než task paralelismus. Task paralelismus v nových vláknech spouští paralelně funkci, kdežto datový paralelismus spouští paralelně iterace cyklu.

Můj algoritmus začne s generováním task poolu (zásobárny úloh). Tyto úlohy generuji pomocí DFS (stejně tak by bylo možné použít i BFS). Rozhodl jsem se použít DFS, protože jsem ho předtím používal v sekvenčním algoritmu. Tvorba taskpoolu je prakticky skoro stejná jako volání sekvenčního algoritmu, pouze úlohy nepočítám hned a jen je vytvářím. Takto vygeneruji předem stanovený počet konfigurací, které mají předvyplněný předem stanovený počet hladin. Předvyplněný počet hladin jsem experimentálně stanovil na 7.

Následuje paralelní zpracování task poolu. Cyklem iteruji přes taskpool a paralelně provádím jednotlivé iterace. To je možné díky tomu, že jednotlivé tasky jsou datově nezávislé. Pro paralelní cyklus používám direktivu `#pragma omp parallel for schedule(dynamic)`. Zkoušel jsem různá nastavení parametru schedule a skončil jsem s možností dynamic, protože byla nejrychlejší. Schedule dynamic rozloží zátěž mezi jednotlivá vlákna vcelku rovnoměrně, což je zde vhodné, protože ne všechny úlohy jsou díky prořezávání stejně náročné.

4.2 Optimalizace

Použil jsem výše zmiňované optimalizace ze sekvenčního řešení. Dále jsem použil optimalizace režie kritické sekce stejně jako v task paralelismu.

4.3 Měření

Měření probíhalo na stejném hardwaru jako všechna měření za použití stejných vstupních dat. Stejně jako v task paralelismu pomocí OpenMP jsem úlohy spouštěl s různými počty vláken (1, 2, 4, 6, 8, 12, 16 a 20). Pro moje testovací vstupy mi vyšel datový paralelismus rychlejší než task paralelismus.

5 MPI paralelismus

Posledním úkolem bylo paralelizovat algoritmus nejen pomocí více vláken, ale i pomocí více procesů, což mi usnadnila knihovna MPI.

úloha/čas	a	b	c	d	e	f	g	h	i
omp data 1t	0.002s	0.003s	0.078s	0.131s	0.496s	8.016s	22.832s	299.725s	332.198s
opm data 2t	0.002s	0.002s	0.063s	0.105s	0.333s	4.789s	13.651s	161.419s	185.874s
opm data 4t	0.002s	0.002s	0.041s	0.075s	0.216s	3.158s	8.022s	91.469s	109.318s
omp data 6t	0.002s	0.002s	0.033s	0.058s	0.165s	2.564s	5.830s	64.402s	85.347s
omp data 8t	0.002s	0.003s	0.028s	0.048s	0.141s	2.161s	4.802s	49.353s	75.824s
omp data 12t	0.002s	0.003s	0.024s	0.034s	0.114s	1.762s	3.721s	36.631s	61.129s
omp data 16t	0.002s	0.003s	0.020s	0.045s	0.211s	1.634s	3.083s	29.930s	58.792s
omp data 20t	0.004s	0.008s	0.023s	0.050s	0.204s	1.630s	2.725s	25.450s	52.624s

Tabulka 4: Časy datového paralismu pro různé vstupy a počty vláken

5.1 Popis algoritmu

Tento algoritmus byl složitější než všechny předešlé, protože musel navíc zajistit i komunikace mezi jednotlivými procesy. Tentokrát jsem již nevycházel ze sekvenčního řešení, ale z datového paralelismu pomocí OpenMP. Nevycházel jsem z task paralelismu, protože ten měl horší výsledky než datový.

Algoritmus funguje na principu master-slave. To znamená, že jeden proces (master) řídí všechny ostatní procesy (slaves), které pracují (počítají úlohy, které jim master přidělí). Na začátku master proces vygeneruje task pool (stejně jako v datovém paralelismu hlavní vlákno). Následně master rozdistribuuje tento task pool jednotlivým slavům. Nejdříve pošle každému slavovi jeden task. Slavové počítají a jakmile skončí výpočet, pošlou výsledek master procesu a ten jim pošle další task k výpočtu nebo zprávu o ukončení, pokud v master task poolu nezbývají žádné další tasky.

Výpočet úlohy slavem je skoro totožný jako výpočet úlohy pomocí datového paralelismu v předchozím příkladě. Pouze navíc slave vygeneruje vlastní taskpool z tasku, který dostane od mastera. To udělá následujícím způsobem – master při generování tasků vyplní v každém tasku 6 hladin. Slave vygeneruje vlastní taskpool obsahující tasky, které mají prvních 6 hladin stejných jako úloha od master procesu, ale navíc dogeneruje daší 3 hladiny na výsledných 9. Tato číslo (6 a 9) byla opět určena experimentálně a je možné je změnit.

5.2 Implementace

Následující ukázka kódu ukazuje hlavní cyklus master procesu. Dokud existují pracující slave procesy, tak cyklus distribuuje tasky slavům a čeká na výsledky. Jakmile jsou všechny slave procesy ukončené, master sesbírá výsledky.

```
1  // collect results from slaves
2  void collectResults() {
3      int receivedResults = 0;
4      ConfigWeight resultMessage = ConfigWeight(configLength);
5      while (receivedResults < numberOfProcesses - 1) {
6          resultMessage.receive();
7          saveConfigIfBest(resultMessage);
8          receivedResults++;
9      }
10 }
11
12 // distribute tasks and collect results
13 void masterMainLoop() {
14     int workingSlaves = numberOfProcesses - 1;  // minus 1, because of master process
15     MPI_Status status;
```

```
16      ConfigWeight message = ConfigWeight(configLength);
17
18      while (workingSlaves > 0) {
19          status = message.receive(MPI_ANY_SOURCE, TAG_DONE);
20          saveConfigIfBest(message);
21
22          // if there left some task, assign it to finished process
23          if (taskPool.size() > 0) {
24              sendTaskToSlave(status.MPI_SOURCE);
25          } else {
26              // no task left -> terminate slave
27              MPI_Send(nullptr, 0, MPI_SHORT, status.MPI_SOURCE, TAG_TERMINATE, MPI_COMM_WORLD);
28              workingSlaves--;
29          }
30      }
31
32      collectResults();
33  }
```

Dále zde mám ukázku kódu slave procesu. Slave proces čeká na zprávu. Po obdržení zprávy se dle jejího obsahu buďto ukončí nebo vypočítá obdrženou úlohu.

```
1   // slave process function
2   void slave() {
3       MPI_Status status;
4       ConfigWeightTask taskMessage = ConfigWeightTask(configLength);
5       ConfigWeight resultMessage = ConfigWeight(configLength);
6
7       while (true) {
8           status = taskMessage.receive(MASTER, MPI_ANY_TAG);
9
10          // send result to master and terminate
11          if (status.MPI_TAG == TAG_TERMINATE) {
12              resultMessage.setWeightAndConfig(minimalSplitWeight, minimalSplitConfig);
13              resultMessage.send(MASTER);
14              return;
15          }
16          // work - compute
17          else if (status.MPI_TAG == TAG_WORK) {
18              saveConfigIfBest(taskMessage);
19              produceSlaveTaskPool(taskMessage.getTask());
20              consumeTaskPool();
21              resultMessage.setWeightAndConfig(minimalSplitWeight, minimalSplitConfig);
22              resultMessage.send(MASTER, TAG_DONE);
23          } else {
24              printf("ERROR, BAD MESSAGE");
25          }
26      }
27  }
```

Poslední část kódu, kterou zde uvedu je posílání zpráv, se kterými jsem měl ze začátku potíže. Vzhledem k tomu, že jsem potřeboval posílat složitější datové struktury než skaláry nebo pole, hledal jsem elegantní řešení, jak zprávy poslat. Vytváření vlastního MPI datového typu bylo problematické, jelikož jsem potřeboval pracovat s dynamickým polem. Jako nejlepší řešení jsem vyhodnotil datový typ `MPI_PACKED`, který umožňuje zabalit do zprávy různé data a podobným způsobem je poté rozbalit.

```
// send self to destination process id
void ConfigWeight::send(int destination, int tag) {
    int position = 0;
    int bufferSize = size() / sizeof(char);
    char* buffer = new char[bufferSize];
    MPI_Pack(&weight, 1, MPI_LONG, buffer, bufferSize, &position, MPI_COMM_WORLD);
    MPI_Pack(config, configLength, MPI_SHORT, buffer, bufferSize, &position, MPI_COMM_WORLD);
    MPI_Send(buffer, position, MPI_PACKED, destination, tag, MPI_COMM_WORLD);
    delete[] buffer;
}

// receive self from destination process id
MPI_Status ConfigWeight::receive(int destination, int tag) {
    MPI_Status status;
    int position = 0;
    int bufferSize = size() / sizeof(char);
    char* buffer = new char[bufferSize];
    MPI_Recv(buffer, bufferSize, MPI_PACKED, destination, tag, MPI_COMM_WORLD, &status);
    if (status.MPI_TAG == TAG_TERMINATE) {
        delete[] buffer;
        return status;
    }
    MPI_Unpack(buffer, bufferSize, &position, &weight, 1, MPI_LONG, MPI_COMM_WORLD);
    MPI_Unpack(buffer, bufferSize, &position, config, configLength, MPI_SHORT,MPI_COMM_WORLD);
    delete[] buffer;
    return status;
}
```

5.3 Optimalizace

Krom optimalizací z datového paralelismu jsem přidal sdílení nejlepšího řešení uprostřed výpočtu. Algoritmus fungoval původně tak, že dílčí výsledky jednotlivých slavů sesbíral až na konci výpočtu. Poté jsem ho upravil tak, že slave při každé žádosti o nový task zároveň pošle doposud nejlepší nalezené řešení master procesu. Master proces, v případě, že je to globální nejlepší řešení, toto řešení uloží. Každý slav při žádosti o nový task dostane zároveň i nejlepší globální řešení, které si uloží.

5.4 Měření

Stejně jako v předešlých úlohách jsem prováděl měření stejných vstupních dat na stejném hardwaru. Měření se liší pouze v tom, že jsem volil nejen různé počty vláken, ale i různé počty slavů. Dle zadání jsem program spouštěl na 3 a 4 procesech (tedy se 2 a 3 slavy). V tabulce značí "mpi 3s 6t"3 slavy a 6 thredů (vláken).

úloha/čas	a	b	c	d	e	f	g	h	i
mpi 3s 6t	0.092s	0.095s	0.117s	0.137s	0.247s	2.853s	5.187s	49.304s	63.227s
mpi 3s 8t	0.095s	0.093s	0.115s	0.136s	0.224s	2.674s	4.416s	40.635s	60.463s
mpi 3s 12t	0.092s	0.093s	0.111s	0.121s	0.190s	2.463s	3.668s	35.152s	54.044s
mpi 3s 16t	0.094s	0.093s	0.109s	0.119s	0.176s	2.314s	3.407s	28.141s	52.237s
mpi 3s 20t	0.100s	0.098s	0.120s	0.126s	0.176s	2.380s	3.493s	28.853s	51.376s
mpi 2s 6t	0.089s	0.088s	0.124s	0.150s	0.292s	4.217s	7.283s	72.293s	92.255s
mpi 2s 8t	0.089s	0.088s	0.121s	0.145s	0.272s	3.963s	6.410s	58.563s	87.367s
mpi 2s 12t	0.087s	0.085s	0.115s	0.134s	0.221s	3.617s	5.496s	51.543s	75.094s
mpi 2s 16t	0.088s	0.090s	0.113s	0.132s	0.210s	3.334s	5.030s	41.887s	75.245s
mpi 2s 20t	0.092s	0.090s	0.113s	0.146s	0.208s	3.542s	5.208s	41.059s	72.492s

Tabulka 5: Časy procesového paralismu pro různé vstupy a počty vláken a procesů

6 Naměřené výsledky a vyhodnocení

V tabulce 6 zobrazuji všechna předchozí měření. Poslední tři sloupce zobrazují úlohy, které jsou sekvenčním algoritmem zpracovány za 1 až 10 minut.

Dále graficky zobrazuji rychlosti všech řešení v obrázku 1. Čím menší čas, tím lepší. Můžeme si všimnout, že sekvenční algoritmus (jak bylo očekáváno) je zdaleka nejpomalejší. Paralelní algoritmy jsou na začátku velice vyrovnané. V nejnáročnější úloze se výsledky začínají trochu lišit – nejrychlejší je datový paralelismus pomocí OpenMP a MPI paralelismus s 3 slavy, nejpomalejší paralelní řešení je MPI s 2 slavy. Čekal jsem, že MPI bude zdaleka nejrychlejší, ale to se nejspíše ukáže až u velkých instancí. Zde na takto malé úlohy se nejspíše nevyplatí komunikační režie mezi procesy MPI. Když jsem však zkoušel lokální spuštění MPI na svém počítači (zde jsem nebyl limitován na maximálně 10 minut). Tak MPI řešení zrychlilo sekvenční čas u nejtěžší instance z 12 hodin na asi 100 minut. Další vliv na rychlost má počet generovaných hladin – některé hladiny mohou být standardním algoritmem prořezány rychleji, než když je generuji do taskpoolu, kde se začnou prořezávat až od první nevygenerované hladiny.

Na obrázcích 2, 3 a 4 zobrazuji časovou závislost na počtu vláken task, datového a MPI paralelismu. Lze si všimnout, že rychlost task a datového paralelismu klesá s narůstajícím počtem vláken exponenciálně, kdežto u MPI paralelismu pouze lineárně (avšak nutno podotknout, že zde jsem neměřil dle zadání běhy s 1, 2, a 4 vlákny – může se tedy jednat o zploštělou exponenciálu).

Obrázky 5 a 6 zobrazují grafy paralelního zrychlení které jsem počítal jako:

$$S(n,p) = \frac{sekvencniCas}{paralelniCas}$$

Z grafů paralelního zrychlení je vidět, že lineárnímu zrychlení jsou blíže OpemMP řešení než MPI, kde je nejspíše větší část procesového výkonu využita na komunikaci. Všechna paralelní řešení jsou slušně škálovatelná a fungovala mi dobře i na větších instancích.

7 Závěr

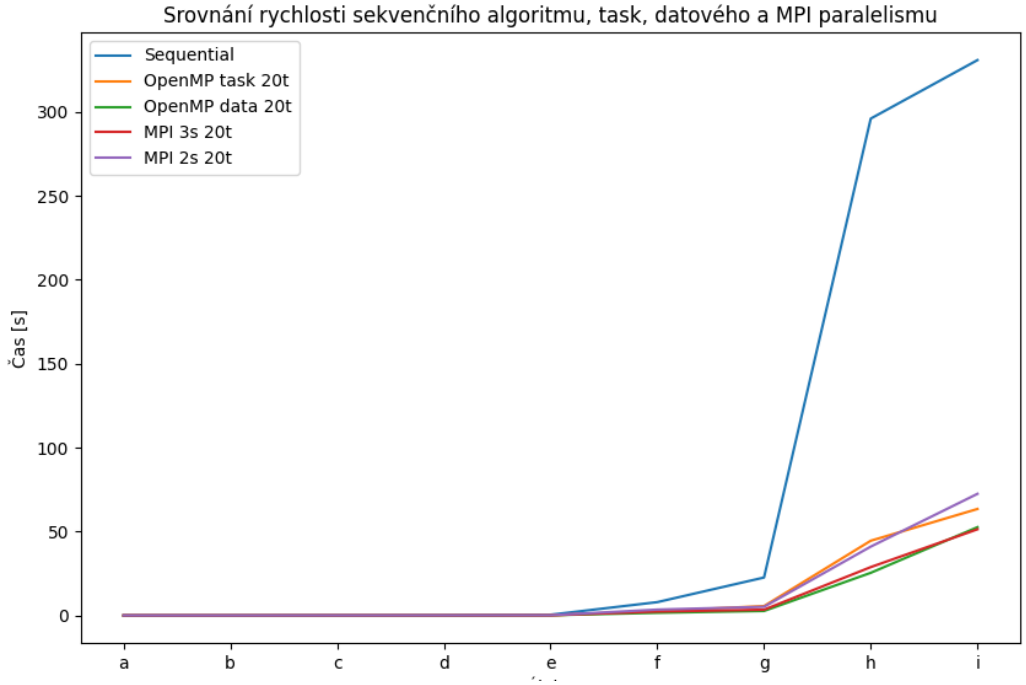
Na závěr bych rád zhodnotil svou práci a získané zkušenosti během semestru. S výsledky mých programů jsem vcelku spokojen. Programy jsou rychlé, ale pořád je zde prostor pro zlepšení. Například by bylo možné implementovat efektivní lowerbounding – posílat mu předpočítanou hodnotu z minulé iterace nebo něco podobného, aby se nepočítal pokaždé od nuly.

Díky tomuto předmětu jsem získal nové znalosti z paralelního a distribuovaného programování. Mé předchozí

znalosti byly pouze z předmětu BI-OSY (bakalářský předmět Operační systémy), tedy ruční tvorba vláken pomocí p-thread funkcí a ruční synchronizace pomocí mutexů atd. Přístup využívající knihovny OpenmMP a MPI je mi daleko příjemnější, jelikož velice usnadňují práci a paralelizace původně sekvenčního programu nevyžaduje tak velké úsilí jako s použitím p-thread. Dále jsem rád, že jsem si mohl vyzkoušet tvorbu distribuovaného programu schopného běžet na více procesorech, což jsem nikdy před tím nedělal. Na rozdíl od předmětu BI-OSY jsem se s domácími úkoly netrápil a upřímně mě i bavili.

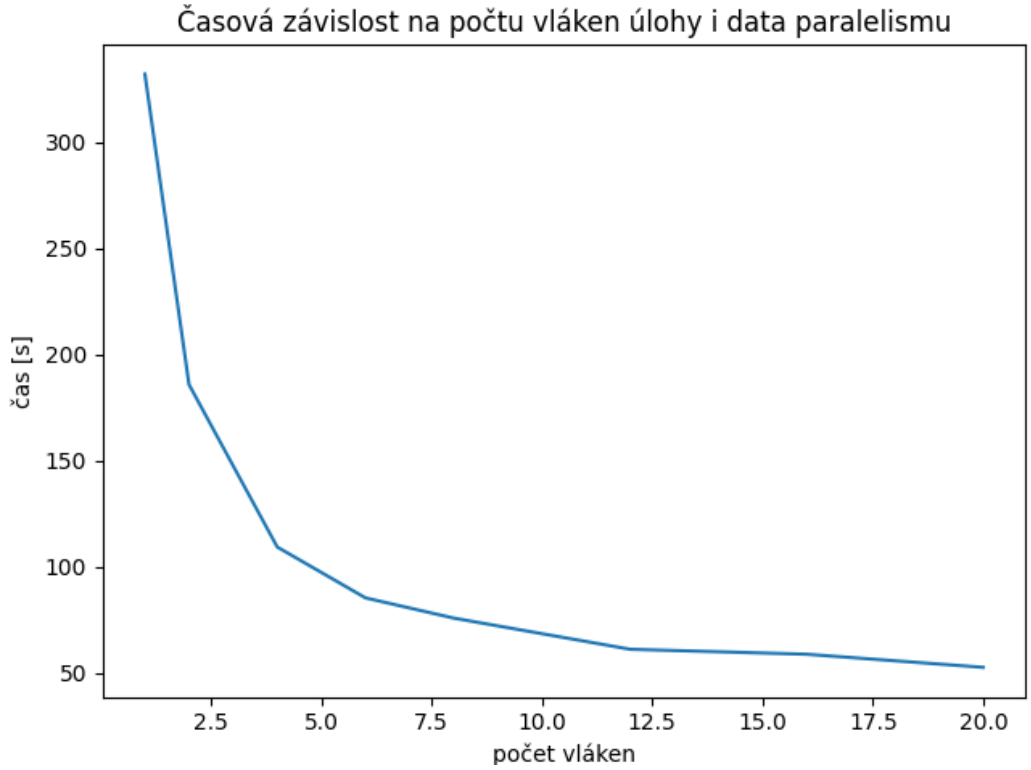
úloha/čas	a	b	c	d	e	f	g	h	i
Sekvenční alg.	0.010s	0.011s	0.080s	0.128s	0.491s	7.988s	22.656s	295.930s	330.765s
opm task 2t	0.003s	0.003s	0.052s	0.103s	0.338s	5.342s	15.415s	192.093s	215.74s
opm task 4t	0.003s	0.003s	0.041s	0.088s	0.268s	3.801s	11.686s	151.349s	146.007s
omp task 6t	0.002s	0.002s	0.033s	0.066s	0.193s	2.912s	8.429s	97.451s	107.31s
omp task 8t	0.003s	0.003s	0.032s	0.07s	0.17s	2.578s	6.796s	94.021s	90.542s
omp task 12t	0.003s	0.003s	0.038s	0.079s	0.142s	1.919s	5.412s	52.767s	55.683s
omp task 16t	0.003s	0.004s	0.043s	0.096s	0.159s	2.08s	5.224	49.457s	53.268s
omp task 20t	0.006s	0.004s	0.05s	0.101s	0.175s	2.607s	5.635s	44.55s	63.473s
omp data 1t	0.002s	0.003s	0.078s	0.131s	0.496s	8.016s	22.832s	299.725s	332.198s
opm data 2t	0.002s	0.002s	0.063s	0.105s	0.333s	4.789s	13.651s	161.419s	185.874s
opm data 4t	0.002s	0.002s	0.041s	0.075s	0.216s	3.158s	8.022s	91.469s	109.318s
omp data 6t	0.002s	0.002s	0.033s	0.058s	0.165s	2.564s	5.830s	64.402s	85.347s
omp data 8t	0.002s	0.003s	0.028s	0.048s	0.141s	2.161s	4.802s	49.353s	75.824s
omp data 12t	0.002s	0.003s	0.024s	0.034s	0.114s	1.762s	3.721s	36.631s	61.129s
omp data 16t	0.002s	0.003s	0.020s	0.045s	0.211s	1.634s	3.083s	29.930s	58.792s
omp data 20t	0.004s	0.008s	0.023s	0.050s	0.204s	1.630s	2.725s	25.450s	52.624s
mpi 3s 6t	0.092s	0.095s	0.117s	0.137s	0.247s	2.853s	5.187s	49.304s	63.227s
mpi 3s 8t	0.095s	0.093s	0.115s	0.136s	0.224s	2.674s	4.416s	40.635s	60.463s
mpi 3s 12t	0.092s	0.093s	0.111s	0.121s	0.190s	2.463s	3.668s	35.152s	54.044s
mpi 3s 16t	0.094s	0.093s	0.109s	0.119s	0.176s	2.314s	3.407s	28.141s	52.237s
mpi 3s 20t	0.100s	0.098s	0.12s	0.126s	0.176s	2.380s	3.493s	28.853s	51.376s
mpi 2s 6t	0.089s	0.088s	0.124s	0.150s	0.292s	4.217s	7.283s	72.293s	92.255s
mpi 2s 8t	0.089s	0.088s	0.121s	0.145s	0.272s	3.963s	6.410s	58.563s	87.367s
mpi 2s 12t	0.087s	0.085s	0.115s	0.134s	0.221s	3.617s	5.496s	51.543s	75.094s
mpi 2s 16t	0.088s	0.090s	0.113s	0.132s	0.210s	3.334s	5.030s	41.887s	75.245s
mpi 2s 20t	0.092s	0.090s	0.113s	0.146s	0.208s	3.542s	5.208s	41.059s	72.492s

Tabulka 6: Časy všech algoritmů

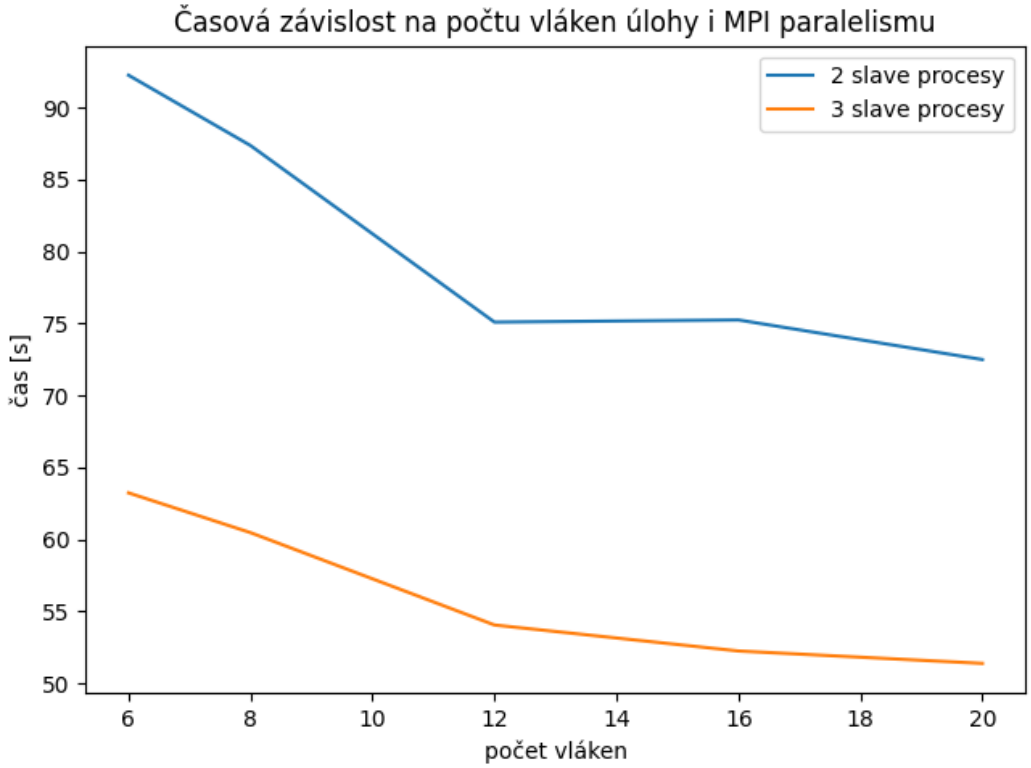


Obrázek 1: Graf rychlosti běhů všech algoritmů

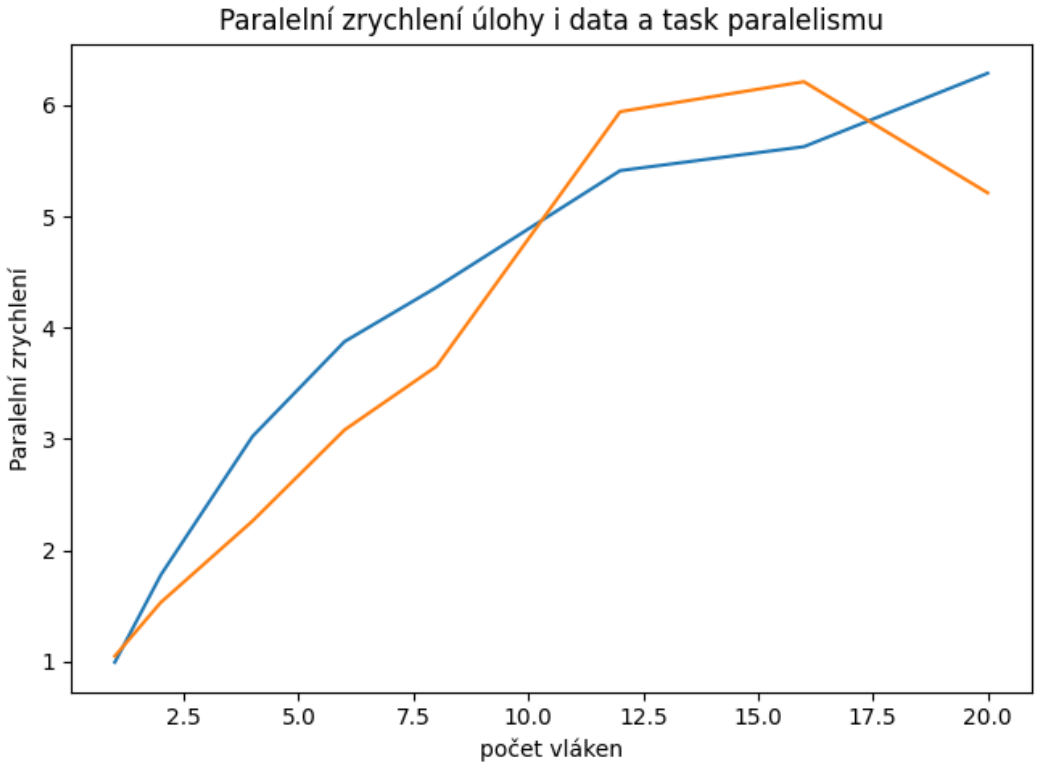
Obrázek 2: Časová závislost na počtu vláken úlohy i task paralelismu



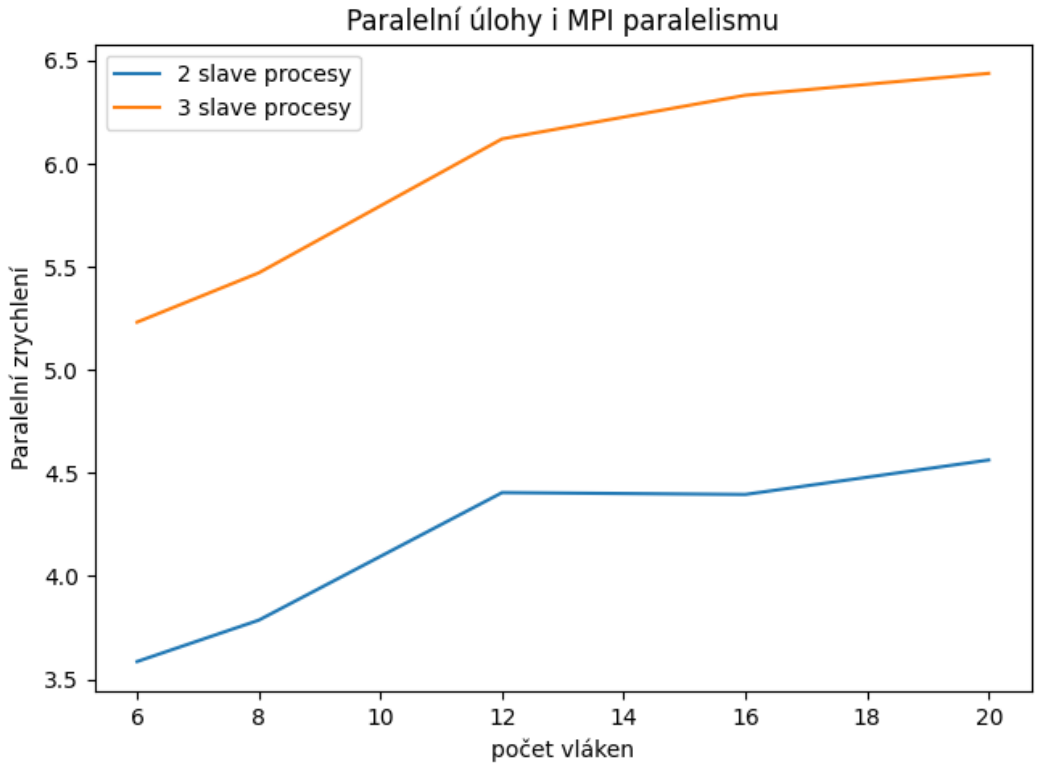
Obrázek 3: Časová závislost na počtu vláken úlohy i datového paralelismu



Obrázek 4: Časová závislost na počtu vláken úlohy i MPI paralelismu



Obrázek 5: Paralelní zrychlení úlohy i data a taks paralelismu



Obrázek 6: Paralelní zrychlení úlohy i MPI paralelismu