Semestrální projekt NI-PDP 2022/2023

Paralelní algoritmus pro hledání minimálního hranového řezu hranově ohodnoceného grafu

Bc. Luboš Zápotočný

FIT ČVUT
Thákurova 9, 160 00 Praha 6

16. května 2023

1 Definice pojmů a popis sekvenčního algoritmu

V této kapitole se budeme věnovat definici problému a popisu sekvenčního algoritmu pro řešení minimálního hranového řezu v hranově ohodnoceném grafu. Nejprve připomeneme základní pojmy týkající se grafů a hranového řezu a poté představíme konkrétní problém, který se v této práci řešil.

Problém minimálního hranového řezu v hranově ohodnoceném grafu spočívá v nalezení dvou disjunktních podmnožin uzlů, takzvaných X a Y, takových, že součet ohodnocení všech hran spojujících uzly z obou množin je minimální. Tento problém má mnoho praktických aplikací, například v oblasti sítí, kde se hledají nejlevnější cesty mezi uzly sítě.

Vstupní data obsahují číslo reprezentující počet vrcholů v grafu a následně reprezentaci ohodnocení hran pomocí matice sousednosti. Ukázka vstupních dat je vyobrazena v ukázce 1.

```
10
     0   112     0     0    98    80     0     0    91   102
   112     0    90     0     0     0     0   119    96     0
     0    90     0     0   104   111    82     0     0   107
     0     0     0     0     0   114    96     0     0     0
    98     0   104     0     0   118    80    88     0     0
    80     0   111   114   118     0   105   106     0   105
     0     0    82    96    80   105     0   109    93    99
     0   119     0     0    88   106   109     0     0    83
    91    96     0     0     0     0    93     0     0    95
   102     0   107     0     0   105    99    83    95     0
```

Kód 1: Vstupní data

Sekvenční algoritmus pro řešení tohoto problému je založen na postupném procházení některých (nikoli všech) kombinací podmnožin uzlů X a Y a hledání takové kombinace, která minimalizuje součet ohodnocení hran mezi nimi.

Některé kombinace algoritmus vynechává, jelikož nemůžou vést k lepšímu řešení, než je aktuálně nalezené minimum. Při průchodu stromem možných kombinací některé podstromy můžeme vynechat. Této metodě se říká Branch and Bound.

```
if (currentWeight >= bestWeight) {
    upperBoundCounter++;
    return;
}
```

Kód 2: Horní řez

```
int lowerBound = graph.cutLowerBound(cut, index);

if (currentWeight + lowerBound >= bestWeight) {
    lowerBoundCounter++;
    return;
}
```

Kód 3: Dolní řez

Princip prořezávání pomocí Branch and Bound spočívá v systematickém prohledávání stromu kombinací s cílem najít optimální řešení s co nejmenším počtem vyhodnocených kombinací. Algoritmus začíná v kořeni stromu a postupně prochází všechny jeho větve, přičemž se snaží minimalizovat horní hranici ceny řešení a zároveň maximalizovat dolní hranici ceny řešení. Toho se dosahuje pomocí prořezávání podstromů, které nemohou obsahovat optimální řešení.

Prořezávání probíhá pomocí dvou technik: dolních a horních odhadů. Dolní odhady se používají k identifikaci podstromů, které neobsahují optimální řešení, a mohou být tedy bezpečně prořezány. Horní odhady se používají k minimalizaci počtu vyhodnocených kombinací tím, že prořezávají podstromy, jejichž řešení by bylo horší než nejlepší nalezené řešení. Ukázka kódu 2 zobrazuje implementaci horního řezu. Ukázka kódu 3 znázorněna podmínka pro řez pomocí dolního odhadu.

Dolní odhad je spočítán tak, že pro každý zatím nepřiřazený uzel grafu v daném mezistavu s částečným řezem vypočteme, o kolik by se váha řezu zvýšila, pokud by tento uzel patřil do X, o kolik by se zvýšila, pokud by tento uzel patřil do Y a vezmeme menší z těchto dvou hodnot a tato minima posčítáme pro všechny nepřiřazené uzly.

Cílem prořezávání stromu kombinací pomocí Branch and Bound je minimalizovat počet vyhodnocených kombinací a najít optimální řešení. Tento algoritmus je velmi účinný pro řešení problémů, kde je prostor kombinací velký a výpočetní čas je omezený. [1]

Rekurzivní části algoritmu jsou vyobrazeny na ukázkách 4 a 5.

Za předpokladu, že je nalezeno lepší řešení, je tato hodnota uložena a rekurzivní prohledávání této cesty ve stromu kombinací je u konce. Kód 6 zobrazuje část rekurzivní funkce, která kontroluje

```
void DFS_BB(const Graph &graph, int maxPartitionSize,
            bool *cut, int count, int index,
            int currentWeight, int &bestWeight);
```

Kód 4: Prototyp rekurzivní funkce

```
// try with this vertex
cut[index] = true;

// try with this vertex (need to extend current cut)
DFS_BB(graph, maxPartitionSize,
       cut, count + 1, index + 1,
       currentWeight + graph.vertexWeight(cut, index, index), bestWeight);

// restore the status of the cut
cut[index] = false;

// try without this vertex (need to extend current cut)
DFS_BB(graph, maxPartitionSize,
       cut, count, index + 1,
       currentWeight + graph.vertexWeight(cut, index, index), bestWeight);
```

Kód 5: Rekurzivní volání

```
if (index == graph.size) {
    if (count != maxPartitionSize) {
        return;
    }

    if (currentWeight < bestWeight) {
        bestWeight = currentWeight;
    }

    return;
}
```

Kód 6: Prototyp rekurzivní funkce

aktuální váhu řezu a aktualizuje globální proměnnou v případě nalezení lepšího řešení.

2 Popis paralelního algoritmu a implementace v OpenMP - taskový paralelismus

Taskový paralelismus se používá v OpenMP pro paralelizaci běhu algoritmu. Jednotlivé úlohy (nebo podúkoly) jsou přiděleny vláknům k vykonání. V OpenMP lze k vyznačení úloh použít direktivu „#pragma omp task“, která říká, že se má daná část kódu vykonat jako samostatná úloha. Ukázka této modifikace je zobrazena v kódu 7. Přidáním těchto direktiv povolujeme spuštění podúkolů paralelně. Je však důležité, aby tuto rekurzivní funkci započalo pouze jedno vlákno. Proto byla vytvořena separátní funkce, jejíž obsah je zobrazen v ukázce kódu 8.

Více podúkolů může být spuštěno současně a vlákna, která jsou k dispozici, mohou vykonávat

```
// try with this vertex (need to extend current cut)
#pragma omp task default(none) firstprivate(graph, cut, count, index,
    currentWeight)
{
    // try with this vertex
    cut[index] = true;

    // compute weight with this vertex
    int nextWeight = currentWeight+graph->vertexWeight(cut, index, index);

    DFS_BB(cut, count + 1, index + 1, nextWeight);
}

// try without this vertex (need to extend current cut)
#pragma omp task default(none) firstprivate(graph, cut, count, index,
    currentWeight)
{
    // restore the status of the cut
    cut[index] = false;

    // compute weight without this vertex
    int nextWeight = currentWeight+graph->vertexWeight(cut, index, index);

    DFS_BB(cut, count, index + 1, nextWeight);
}
```

Kód 7: Modifikace na taskový paralelismus v rekurzivní funkci

```
bestWeight = std::numeric_limits<int>::max();

#pragma omp parallel default(none) firstprivate(graph)
{
    #pragma omp single
    DFS_BB(Cut(graph->size), 0, 0, 0);
}

return bestWeight;
```

Kód 8: Začátek taskového paralelismu

úlohy paralelně. Každé dvě podúlohy nezávislé a mohou běžet současně bez čekání na dokončení jiných úloh.

Taskový paralelismus je vhodný pro aplikace s dynamickým vytvářením úloh, které mohou být vykonány asynchronně. To může zvýšit využití výpočetních zdrojů a zlepšit výkon aplikace.

3 Popis paralelního algoritmu a implementace v OpenMP - datový paralelismus

Datový paralelismus je druh paralelizace, který spočívá v rozdělení dat na části a zpracování těchto částí v paralelních výpočetních jednotkách.

V programu jednotlivá vlákna zpracovávají různé části dat a zpracování probíhá paralelně. Vhodnou implementací datového paralelismu lze výrazně zlepšit výkon programu, zvláště u algoritmů, které pracují s velkými objemy dat.

Datový paralelismu začíná vygenerováním několik počátečních stavů, které se následně v paralelní smyčce nechávají sekvenčně dopočítat.

Sekvenční algoritmus a taskový paralelismus využívání DFS průchod stromem kombinací. Datový paralelismus nejprve provede BFS expanzi z počátečního vrcholu do dané hloubky a tyto stavy poté nechává více vláknově dopočítat. Ukázka této expanze je v kódu 9. Tento kód projde stromem kombinací až do hloubky $(2 * maxPartitionSize)/3$, což je experimentálně zvolená hodnota, která vygeneruje dostatečný počet počátečních stavů. Na konci je výsledné pole počáteční stavů setříděné podle částečné aktuální váhy, což zrychluje nalezení menších hodnot, což poté využívá Branch and Bound pro horní řez některých podstromů výpočtu.

4 Popis paralelniho algoritmu a jeho implementace v MPI

MPI (Message Passing Interface) je knihovna umožňující programátorům psát paralelní aplikace pro distribuované systémy, které mohou běžet na mnoha procesorech. MPI aplikace jsou psány jako soubor nezávislých procesů, které se navzájem komunikují pomocí posílání zpráv.

MPI umožňuje programátorům psát různé typy paralelních aplikací, včetně datově paralelních a taskově paralelních aplikací. Datově paralelní aplikace pracují s velkými datovými sadami, které jsou rozděleny mezi různé procesory. Každý procesor pracuje s určitou částí dat a výsledky jsou posléze kombinovány. Úlohově paralelní aplikace jsou aplikace, které se skládají z mnoha úloh, které mohou být spuštěny na různých procesorech a mezi sebou komunikují pomocí posílání zpráv.

MPI umožňuje programátorům psát paralelní aplikace, které mohou běžet na různých typech hardware, včetně lokálních počítačů, clusterů a superpočítačů. MPI poskytuje také mnoho funkcí pro synchronizaci a komunikaci mezi procesy, což umožňuje programátorům psát robustní paralelní aplikace.

MPI aplikace se obvykle skládají z řídícího procesu, který koordinuje běh aplikace a koordinuje komunikaci mezi procesy, a mnoha pracovních procesů, které provádějí výpočetní práci. MPI aplikace mohou být napsány v mnoha jazycích, včetně C, C++ a Fortranu. [2]

```
auto initialStates = std::vector<State>();
    auto initialStatesQ = std::queue<State>();

    initialStatesQ.push(State(0, 0, 0, Cut(graph->size)));

    while (!initialStatesQ.empty() && initialStatesQ.front().index < (2 *
        maxPartitionSize) / 3) {
        auto state = initialStatesQ.front();
        initialStatesQ.pop();

        auto cut = Cut(state.cut);

        int nextWeight = state.currentWeight + graph->vertexWeight(cut,
            state.index, state.index);
        initialStatesQ.push(State(state.count, state.index + 1, nextWeight,
             cut));

        if (state.count + 1 > maxPartitionSize) continue;

        cut = Cut(state.cut);
        cut[state.index] = true;

        nextWeight = state.currentWeight + graph->vertexWeight(cut, state.
            index, state.index);
        initialStatesQ.push(State(state.count + 1, state.index + 1,
            nextWeight, cut));
    }

    while (!initialStatesQ.empty()) {
        initialStates.push_back(initialStatesQ.front());
        initialStatesQ.pop();
    }

    std::sort(initialStates.begin(), initialStates.end());
```

Kód 9: BFS expanze před zahájením datového paralelismu

```
enum MPITag {
    SLAVE_INITIALIZED,
    SLAVE_INITIALIZE_GRAPH_SIZE,
    SLAVE_INITIALIZE_GRAPH,
    SLAVE_TASK,
    SLAVE_JOB_STATE,
    SLAVE_JOB_CUT,
    SLAVE_AVAILABLE,
    SLAVE_RESULT
};

enum MPITask {
    JOB,
    SYNC,
    TERMINATE
};
```

Kód 10: MPI tagy a specifické tasky

Úprava z datového paralelismu byla komplikovanější než z taskového na datový paralelismus. Program je rozdělen na dva módy. Proces s $ID = 0$ se stává hlavním procesem a rozesílá mezistavy k dořešení na ostatní procesy, které mohou být i na úplně jiném serveru. MPI varaibta algoritmu stále používá datový paralelismus na úrovni vláken.

Komunikace probíhá pomocí zpráv, které znázorňuje kód 10. Ukázka kódu 11 demonstruje čekání hlavního procesu na volný pracovní stroj. Pokud je některý stroj volný, odešle zprávu a hlavní proces mu následně pošle mezistav na dopočítání. Odesílání probíhá ve dvou zprávách. První zpráva obsahuje metadata o aktuálním stavu společně s poslední aktualizovanou hodnotu nejlepší nalezené váhy z ostatních procesů. Následující druhá zpráva obsahuje booleanské pole reprezentující aktuální rozdělení vrcholů do množin X a Y.

V kódu 12 je zobrazena část kódu pro výpočetní uzel, který přijme mezistav a dopočítá tento stav pomocí datového paralelismu.

Každých sto počátečních mezistavů se všechny pracovní stroje sesynchornizují, aby měli nejaktuálnější nejlepší váhu a jejich interní Branch and Bound řezy byly efektivní.

5 Naměřené výsledky a vyhodnocení

Na obrázcích 1, 2 a 3 jsou vyobrazeny časy běhů jednotlivých variant algoritmu. MPI variantz jsou pojmenované podle toho, kolik procesů měli k dispozici = na kolika oddělených strojích mohli pracovat. Například mpi_4 znamená, že program byl spuštěn na jednom stroji, který pracoval jako hlavní proces a na třech strojích, pracujících pouze na výpočtu přidělených podúkolů.

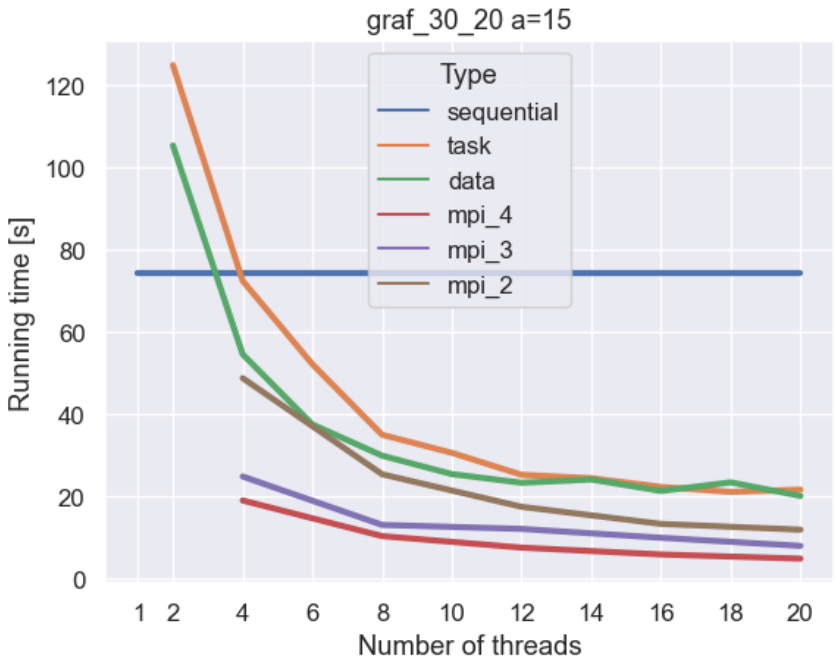
Obrázky 7, 8 a 9 zobrazují efektivitu paralelního zrychlení, které znázorňuje efektivitu zrychlení podle maximálního počtu vláken k dispozici. U MPI úloh je počet vláken násoben počtem výpočetních uzlů.

```
int slave;
MPI_Recv(&slave, 1, MPI_INT, MPI_ANY_SOURCE, MPITag::SLAVE_AVAILABLE,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);

int task = MPITask::JOB;
MPI_Send(&task, 1, MPI_INT, slave, MPITag::SLAVE_TASK, MPI_COMM_WORLD);

int stateBuffer[4] = {bestWeight, state.count, state.index, state.
    currentWeight};
MPI_Send(stateBuffer, 4, MPI_INT, slave, MPITag::SLAVE_JOB_STATE,
    MPI_COMM_WORLD);

MPI_Send(state.cut.data, graph->size, MPI_C_BOOL, slave, MPITag::
    SLAVE_JOB_CUT, MPI_COMM_WORLD);
```

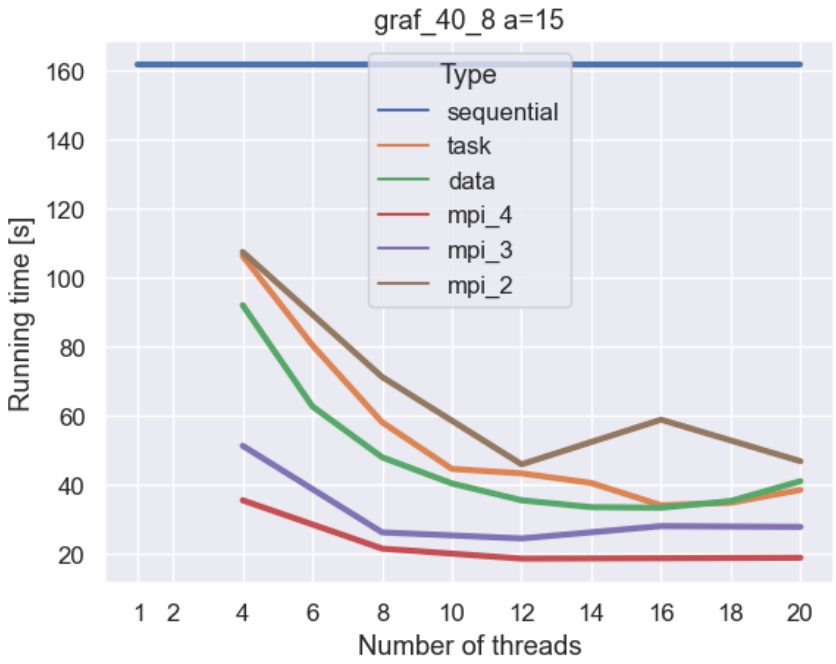
Kód 11: MPI tagy a specifické tasky

```
if (task == MPITask::JOB) {
    MPI_Recv(stateBuffer, 4, MPI_INT, 0, MPITag::SLAVE_JOB_STATE,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    MPI_Recv(cutBuffer, graph->size, MPI_C_BOOL, 0, MPITag::SLAVE_JOB_CUT,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    if (stateBuffer[0] < bestWeight) {
        std::cout << "[" << std::setw(3) << rank << "] "
                  << "Updating best weight" << stateBuffer[0] << std::endl;
        bestWeight = stateBuffer[0];
    }

    auto state = State(stateBuffer[1], stateBuffer[2], stateBuffer[3], Cut(
        graph->size, cutBuffer));
    auto initialStates = BFS_Expansion(state, state.index + 7);

    #pragma omp parallel for schedule(dynamic) default(none) shared(
        initialStates)
    for (const auto &state: initialStates) {
        DFS_BB(state);
    }
}
```

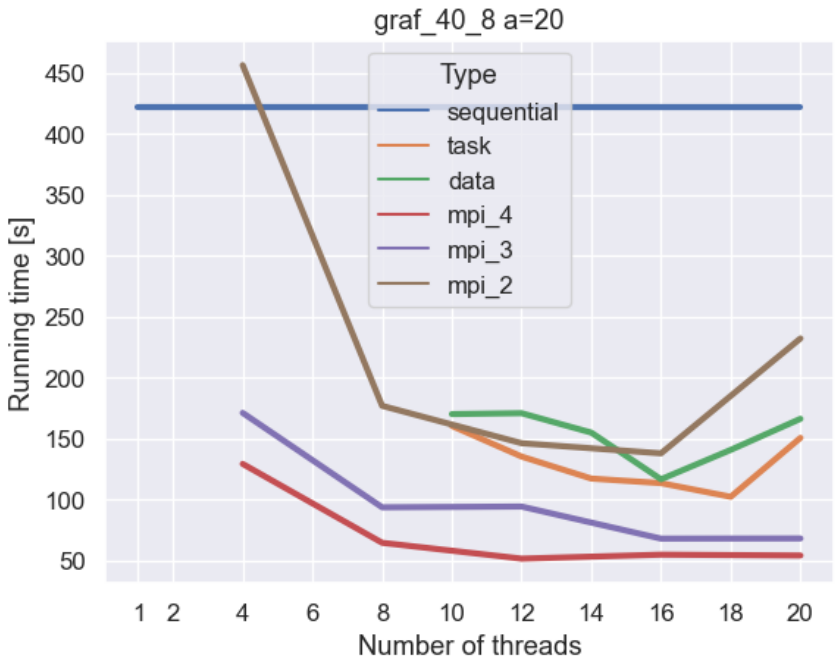
Kód 12: MPI přijmutí mezistavu a zahájení výpočtu



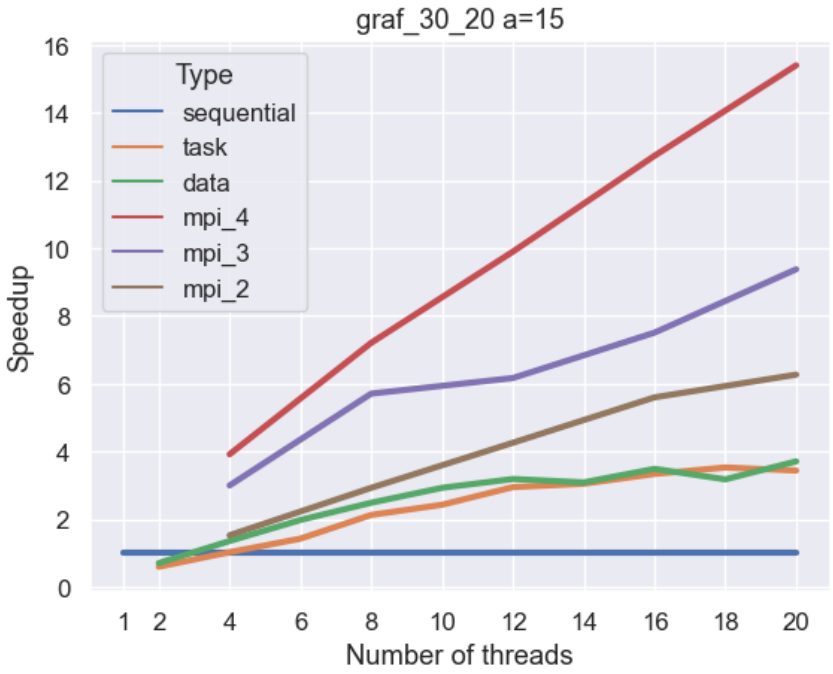
Obrázek 1: Čas běhu s parametrem 15 na grafu 30_20



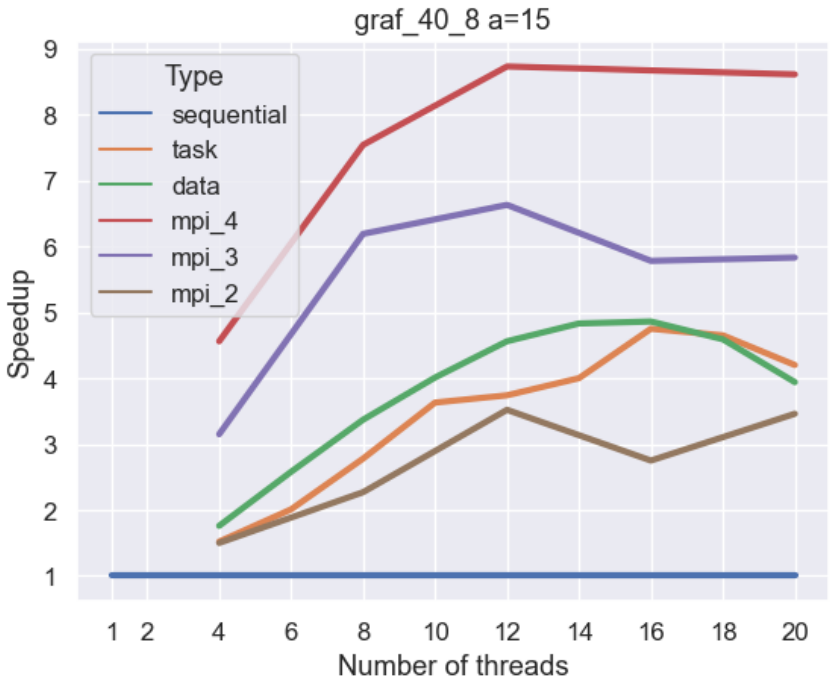
Obrázek 2: Čas běhu s parametrem 15 na grafu 40_8



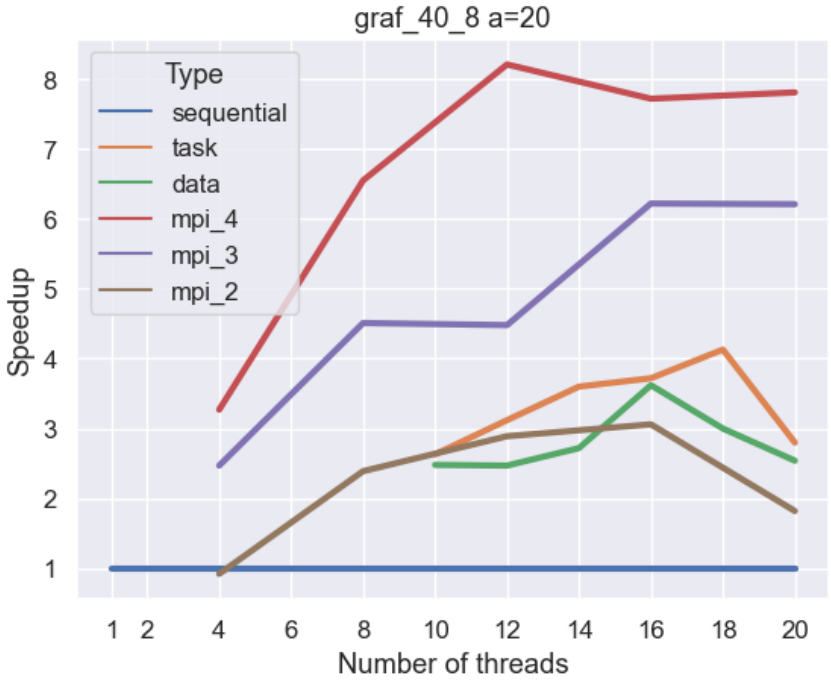
Obrázek 3: Čas běhu s parametrem 20 na grafu 40_8



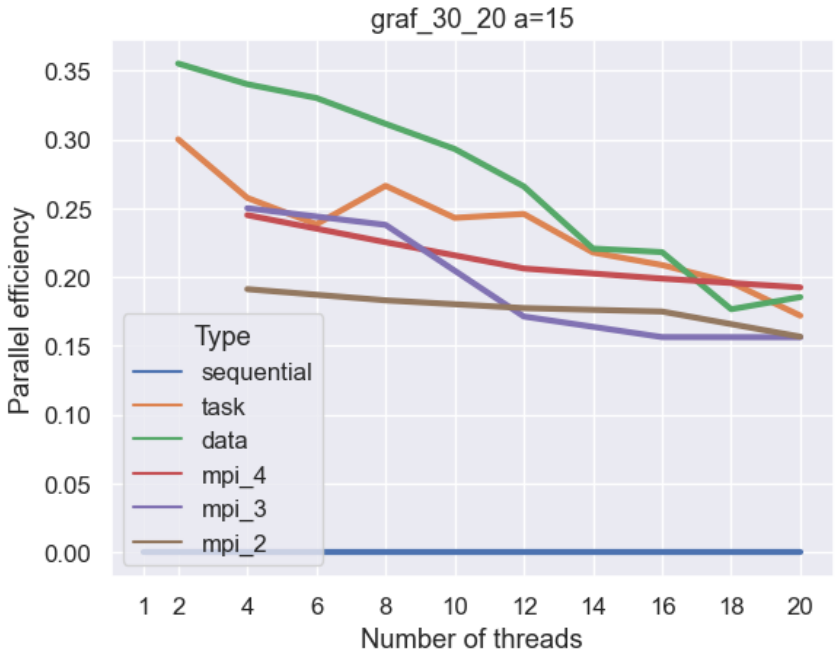
Obrázek 4: Zrychlení oproti sekvenčnímu řešení s parametrem 15 na grafu 30_20



Obrázek 5: Zrychlení oproti sekvenčnímu řešení s parametrem 15 na grafu 40_8



Obrázek 6: Zrychlení oproti sekvenčnímu řešení s parametrem 20 na grafu 40_8



Obrázek 7: Paralelní efektivita pro vstup s parametrem 15 na grafu 30_20

Na těchto obrázcích je zajímavé například to, že taskový paralelismus má o dost lepší efektivitu než mpi_2 (která má jeden výpočetní uzel s paralelním výpočtem). Tento jev dobře demonstruje, že komunikace mezi výpočetními uzly není nezanedbatelná.

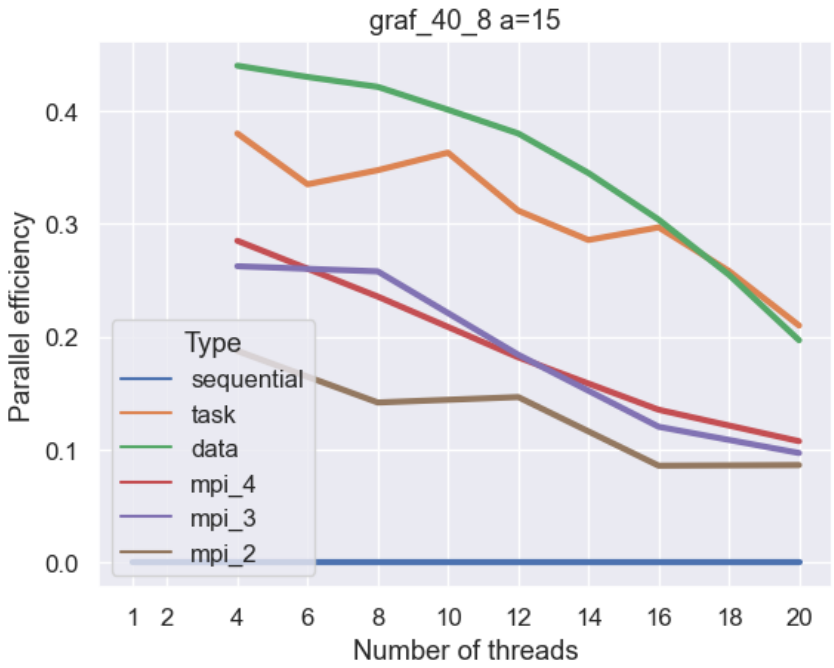
6 Závěr

Iterativní práce a postupné modifikace sekvenčního řešení mi dávalo po dobu semestr dobrý smysl a je to takto koncepčně dobře připravené. Doplňkové materiály ke cvičením jsou výborný zdroj pro inspiraci, jakým postupem danou problematiku řešit.

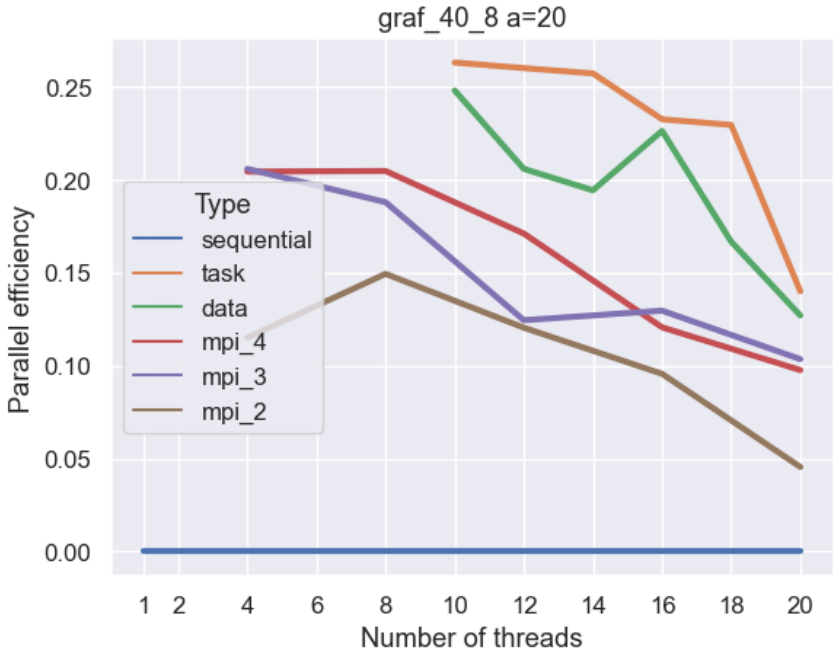
Při použití C++ se mi v některých částech zjednodušila práce s programem (například použitím copy konstruktorů).

Zároveň byla přínosná zkušenost s porovnáním běhu algoritmů na lokálním stroji oproti serverovému clusteru star.

Zde bych chtěl ještě moji práci detailněji prozkoumat, jelikož výsledky prezentované v této zprávě pocházejí ze staru. Na lokálním počítači programy běžely vždy téměř lineárně, což se na startu ani v případě taskového či datového paralelismu nepotvrdilo.



Obrázek 8: Paralelní efektivita pro vstup s parametrem 15 na grafu 40_8



Obrázek 9: Paralelní efektivita pro vstup s parametrem 20 na grafu 40_8

7 Literatura

Reference

[1] Šoch Michal. Ni-pdp: Prohledávání do hloubky, 4. 2. 2022. https://courses.fit.cvut.cz/NI-PDP/labs/prohledavani-do-hloubky.html.

[2] Mpi, 2023. https://en.wikipedia.org/wiki/Message_Passing_Interface.