Semestrální projekt NI-PDP 2021/2022:

Paralelní algoritmus pro nalezení bipartitního podgrafu s maximální váhou

Tomáš Petříček

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

19. dubna 2022

1 Zadání

Naším úkolem bylo implementovat zadaný algoritmus čtyřmi různými způsoby: sekvenčně, pomocí taskového paralelismu, s využitím datového paralelismu a distribuovaným způsobem. V letošním roce byly vytvořeny dvě úlohy, jedna pro sudá a druhá lichá cvičení. Jelikož jsem se účastnil sudých cvičení, tak mi byl přidělen problém nalezení **bipartitního podgrafu s maximální váhou**. Vstupem algoritmu byl graf s následujícími vlastnostmi:

- n - počet uzlů grafu G, $50 > n >= 10$
- k - průměrný stupeň uzlu grafu G, $n/2 >= k >= 3$
- $G(V, E)$ - jednoduchý neorientovaný hranově ohodnocený souvislý graf o n uzlech a průměrném stupni k, váhy hran jsou z intervalu $< 80, 120 >$

Úkolem bylo nalézt podmnožinu hran F takovou, že podgraf $G(V, F)$ je souvislý a bipartitní a váha F je maximální v rámci všech možných bipartitních souvislých podgrafů G nad V. Graf $G(V, F)$ je bipartitní, pokud lze množinu uzlů V rozdělit na disjunktní podmnožiny U a W tak, že každá hrana v F spojuje uzel z U s uzlem z W. Bipartitní graf lze uzlově obarvit 2 barvami 0 a 1.

Dále jsme obdrželi několik testovacích vstupních grafů, které byly uložených jako **matice sousednosti** v textových souborech. Pro každý graf jsme také dostali referenční dobu běhu sekvenčního algoritmu, celkový počet volání rekurzivní prohledávací funkce, maximální váhu bipartitního podgrafu a počet optimálních řešení. Byl nám poskytnut také generátor grafů s instrukcemi, jak jej použít v případě, že bychom chtěli vygenerovat více instancí vstupního grafu v daném formátu. Nakonec jsme dostali také doporučený způsob implementace sekvenčního algoritmu.

2 Popis implementace sekvenčního algoritmu

Nejprve bylo potřeba načíst graf pomocí funkce `read_graph`. Pro uložení grafu byla zvolena datová struktura **seznam hran** reprezentovaná třídou `graph::edge_list`. Vzhledem k tomu, že počet hran není v době kompilace znám, byl použit `std::vector` pro ukládání hran. Hrany byly reprezentovány pomocí struktury `graph::edge`, která si ukládá váhu a indexy vrcholů, které spojuje.

Poté byl graf předán do konstruktoru třídy `finder::sequential`, která implementuje sekvenční algoritmus. Účelem třídy je najít nejlepší stav. Každý podgraf je reprezentován stavem ve vyhledávacím prostoru. Jedinečně se vyznačuje seznamem vybraných hran. Každý stav je reprezentován třídou `finder::state`, která si kromě seznamu vybraných hran uchovává i obarvení jednotlivých vrcholů, podgraf, potenciální váhu, index hrany, se kterou aktuálně pracuje a další informace. Podgraf je reprezentován třídou `graph::adjacency_list` a používá se pouze ke kontrole, zda jsou všechny hrany v grafu spojené. Pro kontrolu se používá algoritmus **prohledávání do hloubky** (DFS), a proto je graf uložen pomocí datové struktury seznamu sousedů. Má časovou složitost $\mathcal{O}(V + E)$, kde V je počet vrcholů a E je počet hran.

Pro samotné hledání nejlepšího stavu, byl použit algoritmus **prohledávání do hloubky s metodou větví a hranic** (BB-DFS). Vyhledávání pouze pomocí DFS má časovou složitost $\mathcal{O}(3^n)$, což řádí problém do kategorie *NP-těžkých* problémů. Účelem metody BB je snížit složitost tím, že se nebudou prohledávat stavy, které nemohou být lepší než již nalezený nejlepší stav (ořezávání shora). Metoda sama o sobě řádově zkracuje čas na vyřešení problému.

Sekvenční algoritmus začíná řazením hran grafu podle jejich váhy v sestupném pořadí. To také zrychluje vyhledávání, protože metoda BB může lépe (dříve) prořezávat stavový prostor. Jeden vrchol je pak obarven, což snižuje počet vyhledávacích stavů, jelikož tím zabraňuje vybarvování v obráceném pořadí.

Dále začne prohledávání. Metoda `bb_dfs` (viz 1) pro vyhledávání se volá rekurzivně. Při každém volání se snaží aktualizovat nejlepší stav a následně vstoupí do smyčky, která se pokouší přidat hranu do podgrafu. Existují dva způsoby, jak přidat hranu do podgrafu, protože existují dva způsoby obarvení. Pokud lze hranu přidat, je zahájeno další volání rekurzivní metody. Jestli byly vyzkoušeny oba způsoby barvení, vyzkouší se poslední stav a to znamená, že se hrana vůbec nepřidá a je započata další iterace. Smyčka končí, jakmile index hrany dosáhne počtu hran grafu. Důležitým detailem je, že aktuální stav je zkopírován pokaždé, když je předán rekurzivní metodě, což usnadňuje paralelizaci v dalších implementacích.

```
    void bb_dfs(state curr)
    {
        try_update_best(curr);

        while (curr.edge_idx_<graph_.n_edges()) {
            // check upper bound
            if (!can_be_better(best_, curr))
                return;

            // update potential weight
            curr.potential_weight_ += graph_.edge(curr.edge_idx()).weight;

            select_edge(green, red, curr);

            select_edge(red, green, curr);

            // update index
            curr.edge_idx_++;
        }
    }
```

Listing 1: Metoda pro prohledávání stavového prostoru

Zda má současný stav potenciál být lepší než nejlepší stav, se vypočítá na základě jeho aktuální váhy, potenciální váhy a celkové váhy grafu. Potenciální váha podgrafu je váha, kterou by graf měl, kdyby do něj byly přidány všechny hrany, které mohly být v daném stavu přidány. Vzorec použitý pro porovnání stavů je vidět na implementaci metody `can_be_better` (viz 2).

```
    bool can_be_better(const state& best, const state& other)
    {
        int max_weight = other.total_weight()+
                (graph_.total_weight()-other.potential_weight());
        return max_weight>best.total_weight();
    }
```

Listing 2: Metoda pro ořezávání shora

Jakmile je nalezen nejlepší stav, výsledky programu jsou vypsány do konzole. Příklad výpisu (viz 3) ukazuje, že jsou zobrazeny informace, jako je název souboru, rozměry grafu, výpočetní čas a atributy nejlepšího stavu. Celková váha podgrafu poskytnutá programem byla porovnána s referenční celkovou váhou ze zadání. Srovnání bylo provedeno pro každý testovací graf, aby byla ověřena správnost programu.

```
filename: "graf_10_3.txt"
n vertices: 10
n edges: 15
res:
duration: 0.00066782
best:
vertex_colors: 0,1,0,0,1,1,0,0,1,1
selected_edges: 1,1,1,1,0,1,1,1,1,0,1,1,1,1,1
total_weight: 1300
edge_idx: 15
--------------------------------------------------
```

Listing 3: Příklad výpisu výsledků

3 Popis implementace algoritmu pomocí taskového paralelismu

Dalším způsobem, jak efektivněji implementovat stejný algoritmus, bylo použití taskového paralelismu, který pro urychlení výpočtu používá více vláken. Základní princip taskového paralelismu spočívá v tom, že nejprve vlákno vytvoří úlohu, kterou umístí do task poolu, a poté ji další vlákno z task poolu vezme a provede. Úlohy jsou vkládány do task poolu více vlákny a jsou také z něho odebírány více vlákny.

Sekvenční algoritmus lze paralelizovat, jelikož jsou všechny stavy ve vyhledávacím prostoru na sobě nezávislé. K implementaci taskového paralelismu byla použita knihovna **OpenMP**, která k paralelizaci algoritmu používá *pragma* direktivy. Třída `finder::task_parallel` implementuje paralelní algoritmus. Má podobné rozhraní jako třída `finder::sequential`.

Kromě grafu se konstruktoru předává i poměr sekvenčně prohledávaných stavů. Z poměru se vypočítá hloubka, od které jsou stavy prohledávány sekvenčně. Pokud je hodnota poměru jedna, pak se prohledávají sekvenčně všechny stavy, na druhou stranu, pokud je hodnota nula, pak žádný z nich.

První velký rozdíl v implementaci byl v metodě `find` (viz 4). Počáteční volání metody rekurzivního vyhledávání bylo obaleno omp direktivami. Direktiva **omp parallel** označuje oblast, která je vykonávána více vlákny. Direktiva **omp master** určuje oblast, kterou provádí pouze hlavní vlákno, což je velmi důležité, jinak se celé vyhledávání provede tolikrát, kolik je vláken.

```
state find(state init)
{
    ...

    // find best state
    #pragma omp parallel
    #pragma omp master
    bb_dfs(init);

    return best_;
}
```

Listing 4: Metoda pro nalezení nejlepšího stavu

V samotné metodě `bb_dfs` se pár věcí změnilo. Nejprve volá metodu `try_update_best` (viz 5), která se snaží aktualizovat nejlepší stav. Implementace této metody byla obalena direktivou **omp critical**, která poskytuje výhradní přístup do dané oblasti právě jednomu vláknu.

```
void try_update_best(const state& candidate)
{
    #pragma omp critical
    {
        if (candidate.n_colored()==graph_.n_vertices() && candidate.subgraph_connected()
                && best_.total_weight()<candidate.total_weight()) {
            best_ = candidate;
        }
    }
}
```

Listing 5: Metoda pro pokus o aktualizaci nejlepšího stavu

Uvnitř hlavní smyčky je volána metoda `select_edge` (viz 6), která musela být také změněna. Pokud se rozhodne, že hranu lze vybrat a hloubka nedosáhla sekvenční hloubky, tak se úloha přidá do task poolu pomocí direktivy **omp task**.

```
void select_edge(color from, color to, state curr)
{
    ...

    // can be added
    if ((curr_from==from || curr_from==colorless)
        && (curr_to==to || curr_to==colorless)) {
        ...
```

```
            if (graph_.n_edges()-curr.edge_idx()-1>n_sequential_) {
                #pragma omp task
                bb_dfs(curr);
            }
            // search sequentially
            else {
                bb_dfs(curr);
            }
        }
    }
```

Listing 6: Metoda pro označení hrany

V samotné metodě (viz 7) byl přidán pouze jeden řádek s direktivou **omp taskwait**, který čeká na dokončení vytvořených úloh, aby mohla pokračovat v prohledávání.

```
    void bb_dfs(state curr)
    {
        try_update_best(curr);

        while (curr.edge_idx_<graph_.n_edges()) {
            ...

            select_edge(green, red, curr);

            select_edge(red, green, curr);

            #pragma omp taskwait

            ...
        }
    }
```

Listing 7: Metoda pro prohledávání stavového prostoru

K výsledkům zapsaným do konzole (viz 8) byl přidán počet vláken a poměr sekvenčně prohledávaných stavů. Počet vláken byl nastaven voláním funkce `omp_set_num_threads` před voláním metody `find`.

```
filename: "graf_10_3.txt"
n vertices: 10
n edges: 15
sequential ratio: 0.5
n threads: 2
res:
duration: 0.000385726
best:
vertex_colors: 0,1,0,0,1,1,0,0,1,1
selected_edges: 1,1,1,1,0,1,1,1,1,0,1,1,1,1,1
total_weight: 1300
edge_idx: 15
--------------------------------------------------
```

Listing 8: Příklad výpisu výsledků

4 Popis implementace algoritmu pomocí datového paralelismu

Dalším způsobem paralelizace programu bylo využití datového paralelismu, který používá také více vláken, ale jiným způsobem. Pro funkci, jejíž volání jsou na sobě nezávislá, je připraveno několik instancí vstupních dat. Volání funkce jsou rozdělena mezi vlákna, která jsou umístěna v *thread poolu*. Pro implementaci datového

paralelismu byla opět použita knihovna **OpenMP**. Třída `finder::data_parallel` implementuje paralelní algoritmus. Kromě grafu bere konstruktor jako parametr instanci třídy `finder::explorer`, která se používá pro sběr počátečních stavů.

Pro vytvoření instance vyžaduje třída `finder::explorer` právě jeden parametr, který určuje hloubku, ve které se nacházejí počáteční stavy. Hloubka se může pohybovat v rozsahu od nuly, což znamená, že by byl shromážděn pouze kořenový stav, až po počet hran grafu, což by znamenalo, že by byly shromážděny všechny stavy.

Maximální počet počátečních stavů lze určit umocněním čísla 3 na hodnotu hloubky, což je dáno způsobem prohledávání stavového prostoru. Počet počátečních stavů je ve skutečnosti mnohem nižší, protože při sběru byla také použita metoda větví a hranic. Každý počáteční stav vytváří podprostor, jehož tvar tvoří ternární strom a maximální počet stavů, které lze v každém z nich prozkoumat, je stejný. Důležitou vlastností podprostorů je, že jsou navzájem *disjunktní*, což znamená, že každý stav může být prohledán právě jednou.

Interně si ukládá počáteční stavy do `std::vector`, ke kterému lze přistupovat pomocí getteru `states`. Hlavní metoda volaná při shromažďování počátečních stavů je `keep_exploring` (viz 9), která přidá stav k počátečním stavům, pokud bylo dosaženo dané hloubky, a vrátí false pro zastavení hledání v dané větvi, nebo vrací true pro pokračování hledání.

```
bool keep_exploring(const finder::state& state)
{
    if (state.edge_idx()==max_depth_) {
        states_.emplace_back(state);
        return false;
    }
    else {
        return true;
    }
}
```

Listing 9: Metoda pro řízení prohledávání

Třída `finder::data_parallel` používá metodu `prepare_states` (viz 10) k přípravě počátečních stavů pro paralelní oblast. Zde se používá instance třídy `finder::explorer`. Nejprve je převedena na ukazatel, aby mohla simulovat volitelný parametr metody `bb_dfs`.

```
std::vector<state> prepare_states(state init)
{
    ...

    auto expl = std::make_unique<explorer>(explorer_);
    bb_dfs(init, expl.get());
    return expl->states();
}
```

Listing 10: Metoda pro přípravu počátečních stavů

Jediné, co se v metodě `bb_dfs` (viz 11) změnilo, bylo to, že před prozkoumáním nového stavu zkontroluje, zda má pokračovat v hledání. Metoda `try_update_best` je umístěna do paralelní oblasti stejným způsobem jako při implementaci taskového paralelismu.

```
void bb_dfs(state curr, explorer* explorer = nullptr)
{
    try_update_best(curr);

    while (curr.edge_idx_<graph_.n_edges()) {
        if (explorer)
            if (!explorer->keep_exploring(curr))
                return;
        ...
    }
}
```

Listing 11: Metoda pro prohledávání stavového prostoru

Metoda `find` (viz 12) nejprve shromáždí počáteční stavy pomocí metody `prepare_states` a poté spustí datový paralelismus. Používá direktivu `omp parallel for` ke spuštění paralelního cyklu for. Jedna věc, kterou je třeba poznamenat, je, že metoda `bb_dfs` se volá bez parametru explorer.

```
    state find(const state& init)
    {
        std::vector<state> states = prepare_states(init);

        // find best state
        #pragma omp parallel for
        for (int i = 0; i<states.size(); i++) {
            bb_dfs(states[i]);
        }

        return best_;
    }
```

Listing 12: Metoda pro nalezení nejlepšího stavu

Maximální hloubka, ve které se nacházejí počáteční stavy, byla přidána do výstupu konzole (viz 13).

```
filename: "graf_10_3.txt"
n vertices: 10
n edges: 15
max depth: 5
n threads: 5
res:
duration: 0.000411941
best:
vertex_colors: 0,1,0,0,1,1,0,0,1,1
selected_edges: 1,1,1,1,0,1,1,1,1,0,1,1,1,1,1
total_weight: 1300
edge_idx: 15
-------------------------------------------------
```

Listing 13: Příklad výpisu výsledků

5 Popis implementace distribuovaného algoritmu

Poslední způsob, jak rozdělit práci, byla distribuce mezi více procesorů. Doposud bylo využito pouze výhody více jádrových procesorů. Nicméně jsme měli na clusteru přístup k více uzlům, na kterých bylo možné současně spustit hledání nejlepšího stavu. Vzor **master-slave** byl použit k implementaci distribuovaného algoritmu. Koncept spočívá v tom, že je jeden proces zvaný *master*, jehož úkolem je připravit práci, a zbylé procesy se nazývají *slave*, jejichž úkolem je provést práci. Z principu vzoru vyplývá, že pro implementaci musí být spuštěny alespoň 2 procesy. Procesy spolu musí komunikovat a k tomu byl použit **MPI** framework poskytující sadu funkcí, které se používají k odesílání a přijímání zpráv mezi procesy.

Chcete-li spustit program na více procesech, musíte použít příkaz jako **mpirun** ke spuštění a propojení procesů. Vlajka *np* se používá k určení počtu procesů, které se mají spustit. Na konci příkazu je uveden spustitelný soubor se svými vlajkami. Příklad příkazu je uveden ve výpisu 14.

```
mpirun -np 4 /MBPCombined -f graf_10_3.txt -o distrib.csv -m "DISTRIB" -dm 4 -ds 4 -t 3
```

Listing 14: Příklad použití příkazu mpirun

K funkcím MPI lze přistupovat zahrnutím hlavičkového souboru. Framework je však napsán v jazyku C na velmi nízké úrovni, a proto bylo výhodnější použít knihovnu **Boost.MPI**, která je objektově orientovaná. Používá také knihovnu **Boost.Serialization**, která se používá k serializaci a deserializaci objektu, aby jej bylo možné převést na pole bajtů, odeslat přes síť a znovu sestavit. Pro serializaci objektu bylo nutné implementovat metodu `serialize`, což bylo velmi jednoduché a intuitivní (viz 15).

```
    friend class boost::serialization::access;
    template<class Archive>
    void serialize(Archive& archive, const unsigned int version)
    {
        archive & BOOST_SERIALIZATION_NVP(init);
        archive & BOOST_SERIALIZATION_NVP(best);
    }
```

Listing 15: Příklad implementace metody serialize

Pro inicializaci prostředí pro zasílání zpráv je vytvořena instance třídy `boost::mpi::environment`. Funkce `MPI_Init` je volána během konstrukce objektu a `MPI_Finalize` je volána při jeho destrukci. Poté je vytvořena instance `boost::mpi::communicator`, která odpovídá funkci `MPI_COMM_WORLD`. Slouží ke komunikaci mezi procesy. Může být použita k identifikaci ranku procesu voláním metody `rank`. Proces s rankem 0 (`process::rank::master`) je master proces, všechny ostatní ranky identifikují slave procesy. Pro zapouzdření práce procesů byly vytvořeny třídy `process::master` a `process::slave`. Jelikož v každém procesu běží stejný kus kódu, bylo nutné implementovat jednoduchou logiku pro odlišení procesů (viz 16).

```
void run_distributed(...)
{
    boost::mpi::environment env;
    boost::mpi::communicator world;

    ...

    if (world.rank()==process::rank::master) {
        ...

        auto proc = process::master(world, graph, master_explorer, slave_explorer);
        auto res = measure_duration([&] { return proc.start(); });

        ...
    }
    else {
        auto proc = process::slave(world);
        proc.start();
    }
}
```

Listing 16: Rozdělení procesů

Třída `process::master` bere jako parametr konstruktoru komunikátor, graf a explorer pro master a slave proces. Jakmile je proces spuštěn, jsou připraveny počáteční stavy pomocí metody `prepare_states` (viz 17), která začíná hledat stavy od kořenového stavu a k přípravě používá instanci třídy `finder::data_parallel`. Po shromáždění stavů je třeba aktualizovat nejlepší stav třídy.

```
    std::vector<finder::state> prepare_states()
    {
        auto root = finder::state(graph_.n_vertices(), graph_.n_edges());
        std::vector<finder::state> states = finder_.prepare_states(root);
        best_ = finder_.best();
        return states;
    }
```

Listing 17: Metoda pro přípravu počátečních stavů

Poté je zavolána metoda `manage_slaves` (viz 18), která nejprve spustí slave procesy. Pošle jim instanci třídy `setting` složenou z grafu a slave exploreru a následně instanci třídy `config` složenou z nejlepšího stavu známého masterem a počátečního stavu. K sdělení obsahu zprávy používá štítky `tag::setting` a `tag::config`. Poté čeká na obdržení výsledků vyhledávání od jakéhokoli slave procesu, a proto používá `boost::mpi::any_source` jako parametr metody `recv`. Když je přijat nejlepší lokální stav, pokusí se o aktualizaci nejlepšího stavu a poté odešle

další konfiguraci. Pokračuje v odesílání konfigurací, dokud nevyčerpá počáteční stavy. Nakonec odešle štítek `tag::stop`, který říká slave procesu, aby se ukončil. Komunikace mezi procesy probíhá blokovacím způsobem, tj. při přijímaní proces čeká na přijetí zprávy a v případě odesílání čeká na obdržení zprávy příjemcem. Metoda `send` třídy `boost::mpi::communicator` je mapována na funkci `MPI_Send` a metoda `recv` na `MPI_Recv`.

```
void manage_slaves(const std::vector<finder::state>& init_states)
{
    ...

    for (int i{0}; i<init_states.size()+world_.size(); i++) {
        // start working
        if (i<world_.size()-1) {
            world_.send(i+1, tag::setting, setting);
            world_.send(i+1, tag::config, config(init_states[i], best_));
        }
        // keep working
        else if (i<init_states.size()) {
            status = world_.recv(boost::mpi::any_source, tag::done, local_best);
            try_update_best(local_best);
            world_.send(status.source(), tag::config, config(init_states[i], best_));
        }
        // stop working
        else if (i>init_states.size()) {
            status = world_.recv(boost::mpi::any_source, tag::done, local_best);
            try_update_best(local_best);
            world_.send(status.source(), tag::stop, config());
        }
    }
}
```

Listing 18: Metoda pro obstarání slave procesů

Slave proces reprezentovaný třídou `process::slave` je mnohem jednodušší. Skládá se pouze z jedné veřejné metody `start` (viz 19), která nejprve příjme nastavení, a poté buď najde nejlepší stav pro danou konfiguraci nebo se ukončí. Jelikož neví, co obdrží, používá štítek `boost::mpi::any_tag` jako parametr metody `recv`. K nalezení nejlepšího stavu používá instanci `finder::data_parallel`.

```
void start()
{
    ...

    // receive setting
    pdp::setting setting;
    world_.recv(rank::master, tag::setting, setting);;
    finder::data_parallel finder{setting.graph, setting.explorer};

    // keep working
    while (true) {
        status = world_.recv(rank::master, boost::mpi::any_tag, config);

        // stop working
        if (status.tag()==tag::stop) return;

        // find best
        finder.best(config.best);
        best = finder.find(config.init);
        world_.send(rank::master, tag::done, best);
    }
}
```

Listing 19: Hlavní metoda třídy slave procesu

Do výpisu (viz 20) byl přidán počet běžících procesů. Lze jej získat voláním metody `size` třídy `boost::mpi::communicator`, která odpovídá funkci `MPI_Comm_size`. Dále byla do výpisu přidána maximální hloubka, kde se nacházejí počáteční stavy pro master a slave procesy.

```
filename: "graf_10_3.txt"
n vertices: 10
n edges: 15
n processes: 4
max depth master: 4
max depth slave: 4
n threads: 5
res:
duration: 0.00460724
best:
vertex_colors: 0,1,0,0,1,1,0,0,1,1
selected_edges: 1,1,1,1,0,1,1,1,1,0,1,1,1,1,1
total_weight: 1300
edge_idx: 15
-------------------------------------------------
```

Listing 20: Příklad výpisu výsledků

6 Naměřené výsledky a vyhodnoceni

Pro porovnání implementací byla provedena měření výpočetního času. Všechna měření byla provedena na clusteru **STAR**. Byl nám poskytnuty skripty pro spouštění našich aplikací. Jeden uzel byl přiřazen sekvenčním a paralelním implementacím a distribuovaná implementace mohla používat až 4 uzly. Každý uzel se skládá z 20 jader.

Doba výpočtu musela být měřena aplikací a k tomu byla použita funkce `measure_duration` (viz 21. Přebírá funkci `find`, která jako výsledek vrací nejlepší stav. Vrací strukturu `result`, který se skládá z nejlepšího stavu a doby výpočtu v sekundách. Čas je měřen pomocí instancí `std::chrono::high_resolution_clock` ze standardní knihovny.

```
result measure_duration(const std::function<finder::state()>& find)
{
    auto begin = std::chrono::high_resolution_clock::now();

    auto best = find();

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end-begin);

    return result(best, duration.count()*1e-9);
}
```

Listing 21: Funkce pro měření doby výpočtu

Výsledky každého spuštění aplikace byly uloženy jako řádek instance třídy `table`, která byla následovně uložena pomocí funkce třídy `csv` do souboru. Formát CSV byl použit z důvodu snadného použití ve vizualizační části vyhodnocení výsledků. Funkce `run_sequantial` (viz 22) ukazuje, jak se výsledky ukládají.

```
void run_sequential(...)
{
    ....

    // prepare table
    pdp::table<std::string, int, int, double> table{
            {"filename", "n vertices", "n edges", "time[s]"}
    };
```

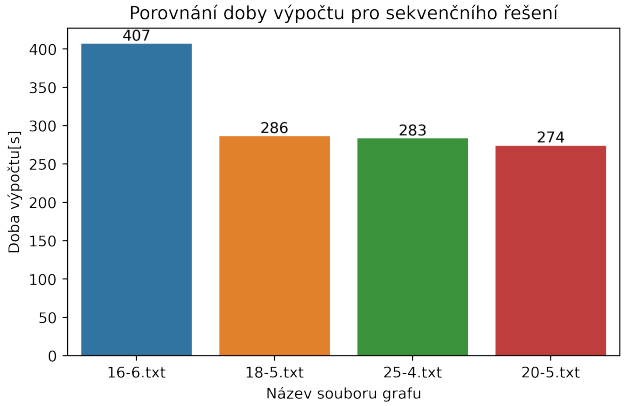
```
    ....

    // save results to csv
    table.add_row({graph_path.filename(), graph.n_vertices(), graph.n_edges(), res.
duration});
    pdp::csv csv{csv_path};
    csv.write(table);
}
```

Listing 22: Funkce pro měření doby výpočtu

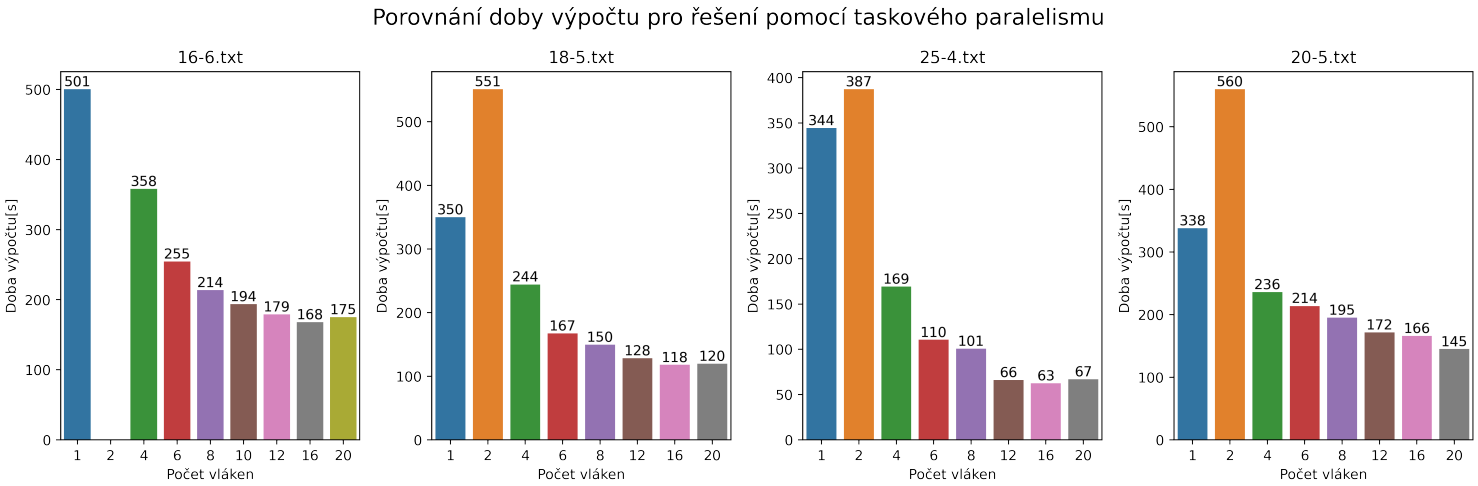
Pro porovnání implementací bylo nutné vybrat alespoň 3 grafy, které běžely 1 až 10 minut na clusteru pomocí sekvenční implementace. Jelikož všechny testovací grafy, které nám byly poskytnuty, běžely rychleji nebo pomaleji, bylo nutné vygenerovat grafy nové. Bylo vygenerováno přes sto grafů a 4 z nich, které běžely nejpomaleji, byly vybrány pro porovnání implementací. Paralelní implementace byly spuštěny pro 1, 2, 4, 6, 8, 12, 16, 20 vláken. Distribuovaná implementace byla spuštěna na 4 procesech a slave procesy měly přístup k 6, 8, 12, 16, 20 vláknům. Pro všechna nastavení byl spuštěn každý graf. Výsledky uložené v souborech csv byly poté vizualizovány pomocí jazyka `Python` a knihoven, jako jsou `Pandas`, `Matplotlib` a `Seaborn`.

Pro srovnání doby výpočtu byl zvolen sloupcový graf. Porovnání doby výpočtu pro sekvenční řešení je vidět na grafu 1. Pořadí grafů zůstává pro snazší čitelnost stejné pro grafy, které následují. Lze pozorovat, že nalezení nejlepšího stavu u prvního grafu trvá o poznání déle než u ostatních grafů, jejichž doba výpočtu je přibližně stejná.



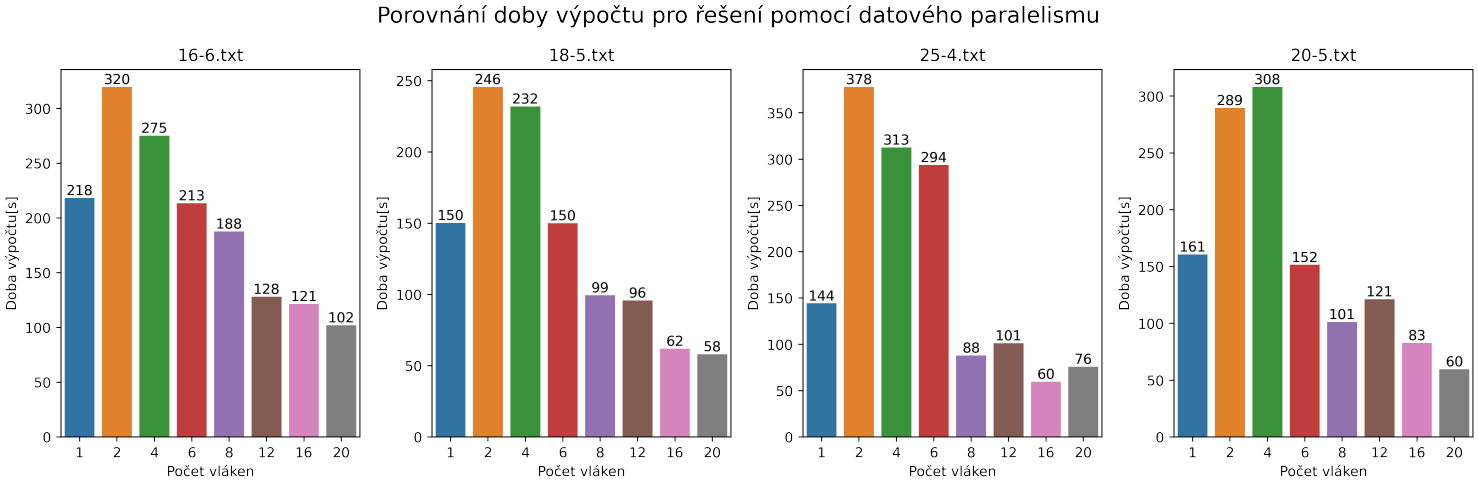
Obrázek 1: Porovnání doby výpočtu pro sekvenční řešení

Následovalo srovnání implementace taskového paralelismu. Poměr sekvenčně hledaných stavů byl nastaven na **0,5** na základě experimentů provedených v průběhu semestru. Z grafu (viz 2) je na první pohled vidět, že se zvyšujícím se počtem vláken se zkracovala doba výpočtu. V prvním grafu lze pozorovat, že chybí sloupec pro dvě vlákna, což je způsobeno překročením časového limitu. Z nějakého důvodu byl běh se dvěma vlákny mnohem pomalejší než s jedním vláknem, a to platilo pro všechny grafy. Po použití alespoň 4 vláken byla paralelní implementace rychlejší než sekvenční. Při použití několika málo vláken je tedy režie tak vysoká, že se nevyplatí spouštět algoritmus paralelně. V průměru pro všechny grafy byl nejrychlejší běh zhruba **dvakrát** rychlejší než sekvenční, což lze považovat za dobrý výsledek vzhledem k tomu, že v kódu bylo provedeno pouze několik málo změn.



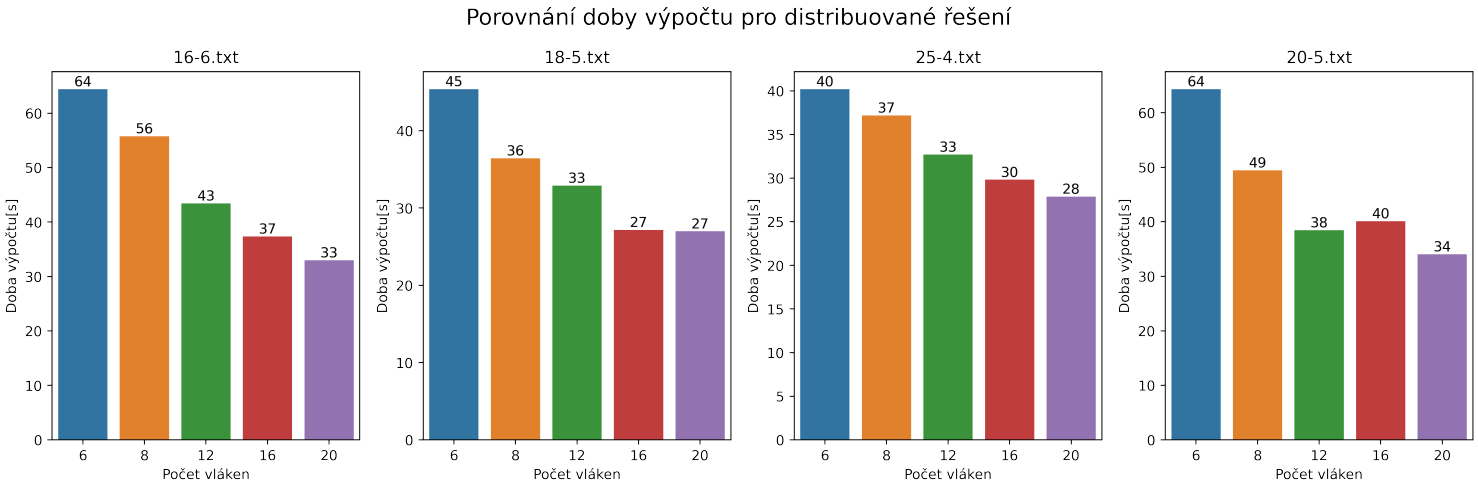
Obrázek 2: Porovnání doby výpočtu pro řešení pomocí taskového paralelismu

Pro datový paralelismus byla zvolena hodnota **5** maximální hloubky, kde se nacházejí počáteční stavy. Také byla vybrána na základě zkušeností získaných během semestru. Je vidět (viz 3) stejná závislost jako v předchozím grafu, kdy s počtem vláken klesá doba výpočtu, úhel poklesu je však mnohem strmější. Překvapivě je implementace algoritmu s jedním vláknem rychlejší než sekvenční, což je pravděpodobně způsobeno přípravou počátečních stavů. Vyhledávací prostor je prohledán tak, že při rozdělení prostoru je větší šance, že se nejlepší stav nachází mezi prvními prohledávanými stavy, protože většina nejlepších stavů obsahuje téměř všechny hrany grafu. Průběh poklesu je tentokrát na posledních 3 grafech výrazně odlišný. Nejrychlejší doba běhu je asi **čtyřikrát** rychlejší než sekvenční implementace, a tedy **dvakrát** rychlejší než taskové paralelní řešení. Po použití 20 vláken se přidání dalších nezdá být o tolik efektivnější.



Obrázek 3: Porovnání doby výpočtu pro řešení pomocí datového paralelismu

Nakonec byly porovnány výsledky distribuovaného řešení. Maximální hloubka pro master proces byla zvolena **5** a pro slave procesy **6**. Dokonce i většina běhů se 6 vlákny je rychlejší než nejrychlejší běh řešení pomocí datového paralelismu (viz 4). Doba výpočtu se neustále zkracuje a zhruba o polovinu se sníží od běhu se 6 vlákny k běhu s 20. Ve srovnání s čistě paralelními řešeními se zdá, že má ještě větší potenciál běžet rychleji, pokud budou přidány nové výpočetní uzly nebo jádra. Nejrychlejší běh každého grafu je asi **desetkrát** rychlejší než sekvenční, a proto asi **dvakrát až třikrát** rychlejší než implementace pomocí datového paralelismu.



Obrázek 4: Porovnání doby výpočtu pro distribuované řešení

Celkově se zdá, že implementace fungují dobře, protože algoritmus se s každou další implementací zrychloval. V případě sekvenčního algoritmu by mohl běžet rychleji, pokud by se našel lepší způsob ořezávaní stavového prostoru, což by pravděpodobně urychlilo všechny následující implementace. Realizace taskového paralelismu mohla být provedena jen o něco čitelněji. V případě datového paralelismu je diskutabilní, zda použít samostatnou třídu pro sběr počátečních stavů, protože dělá kód složitější. Místo DFS by stálo za zvážení použití prohledávání do šířky (BFS), jelikož dává algoritmu větší kontrolu nad počtem shromážděných stavů.

V případě distribuované implementace bylo vyvinuto úsilí o navázání komunikace mezi slave procesy, aby byl nejlepší stav aktuálnější. Implementace však měla horší výsledky než jednodušší verze. Ukázalo se, že je obtížné rozdělit paralelní část na dvě, z nichž jedna by sloužila pouze pro komunikaci a druhá pro výpočet. OpenMP prostě nebylo takto navrženo. Při zpětném pohledu na kód, nebylo třeba posílat graf a explorer pomocí instance `setting`, mohl to provést samotný podřízený proces.

7 Závěr

Vezmeme-li tyto maličkosti stranou, zdá se, že implementace byly provedeny kvalitně. Bylo dobře, že díky jednoduchosti algoritmu bylo možné věnovat více úsilí samotnému zrychlování a seznámení se s OpenMP a MPI. Další ulehčení bylo, když jsme na začátku semestru obdrželi video s nápovědou, jak implementovat sekvenční algoritmus. Dokumentace zadání byla dobře napsaná a pomohla s implementací algoritmu. Testovací instance byly připraveny správně, byly snadno dostupné a užitečné ve fázi ladění. Dokumentace pro generátor grafů byla dobře zpracovaná a užitečná ve fázi hodnocení projektu. Detaily skriptů a informace o tom, jak je zkompilovat a spouštět naše projekty, byly také provedeny dobře, ale možná si zaslouží svou vlastní sekci na webu s názvem STAR. Další skvělá věc byla, že si k nám cvičící při každém cvičení sedl a konzultoval s námi naše projekty. Celkově měl člověk možnost naučit se něco nového a soustředit se na to nejdůležitější díky kvalitně zpracované dokumentaci.