Semestrální projekt MI-PDP 2019/2020:

Paralelní algoritmus pro řešení problému MHR

Petr Ančinec

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

Květen 1, 2020

1. Definice problému a popis sekvenčního algoritmu

Problém minimálního hranového řezu

n = přirozené číslo představující počet uzlů grafu G, 160 >= n >= 20
k = přirozené číslo řádu jednotek představující průměrný stupeň uzlu grafu G, n >= k >= 3
G(V,E) = jednoduchý souvislý neorientovaný hranově ohodnocený graf o n uzlech a průměrném stupni k Ohodnocení hran grafu jsou desetinná čísla z intervalu (0 … 1)
a = přirozené číslo, n/4 =< a =< n/2

Nalezněte **minimální hranový řez grafu rozděleného na části velikosti** *a* **a** *n-a*, rozdělení množiny uzlů *V* do dvou disjunktních podmnožin *X, Y* takových, že množina *X* obsahuje *a* uzlů, množina *Y* obsahuje *n-a* uzlů a součet ohodnocení všech hran *{u,v}*, kde *u* je z *X* a *v* je z *Y*, (čili velikost hranového řezu mezi *X* a *Y*), je minimální.

Sekvenční algoritmus

Vstup: Textový soubor, který má na prvním řádku hodnoty **n k a**, a poté následuje **n * k /2** trojic **u1 u2 v**, každá trojice reprezentuje neorientovanou hranu mezi uzly **u1** a **u2** s ohodnocením **v**.

Výstup: Bitmapa (vektor 1 a 0), která říká, jestli vrchol na indexu i spadá do množiny X nebo Y, následuje součet ohodnocení hran, který je minimální.

Struktury: Graf si držíme jako matici hran, tzn. položka matice na pozici **i,j** říká, jestli v grafu vede hrana z vrcholu **i** do vrcholu **j** a jaké má ohodnocení, jelikož jsou hrany neorientované, matice je symetrická. Bitmapa je reprezentována pomocí typu uint64_t.

Algoritmus: Problém je rekurzivní, nabízí se tedy velmi jednoduché rekurzivní řešení, kdy u bitmapy (např. 0010) vypočítáme cenu, v případě lepší ceny ji uložíme a pokračujeme rozvětvením na konfigurace kde do leva bitmapě přidáme 1 a 0, nové konfigurace tedy budou 00010 a 10010 a takto pokračujeme až projdeme všechny řešení. Abychom výpočet urychlili, používáme metodu větví a hranic (B&B) s následujícími ořezy:
1) pokud je cena konfigurace větší než nejlepší, nemá cenu pokračovat (cena se může jen zvýšit)
2) pokud je v množině X **a** vrcholů, nemá cenu přidávat do ní další vrcholy (X má obsahovat **a** vrcholů)
3) pokud je v množině Y **n-a** vrcholů, nemá cenu přidávat do ní další vrcholy (Y má obsahovat **n-a** vrcholů)

Jelikož počítání ceny konfigurace je poměrně dlouhé (naivní řešení potřebuje projít celou matici), cenu konfigurace vypočítáváme jako cenu předchozí konfigurace + vrchol, který jsme přidali do X nebo Y.

2. Popis paralelního algoritmu v OpenMP - task paralelismus

Algoritmus pro task paralelismus je jednoduché rozšíření sekvenční verze. Vstupy i výstupy zůstávají stejné, program má navíc jeden volitelný parametr, který udává, kolik vláken chceme k řešení problému použít. Pokud parametr není specifikovaný, program použije defaultní hodnotu z funkce omp_get_max_threads().
Po načtení počtu vláken, které chceme použít, algoritmus vypočte hloubku zanoření jako 4 + log2(počet vláken). Tento vzoreček byl navržený tak, aby každé vlákno mělo dostatek práce na rovnoměrné rozložení zátěže, aby se hloubka zanoření škálovala s počtem vláken a aby vlákna pracovali dlouho na jednom volání funkce, což minimalizuje overhead. Vzoreček byl poté ověřen experimentálně oproti jiným parametrům a vyšel jako nejlepší.

V následujících obrázcích jsou vidět nutné úpravy kódu na task paralelismus, který používá parametry hloubky zanoření a počtu vláken. Algoritmus na začátku vytváří paralelní tasky, které jsou poté zpracovány poolem vláken. Jelikož je podmínka nastavená tak, aby se paralelní tasky vytvářeli pouze na začátku, každé vlákno počíta poměrně dlouhou rekurzivní větev a nevzniká tak velký overhead spojený s přepínáním kontextu apod.

```
void solve_start() noexcept {
    solve(0, 0);
    solve(0, 1);
```

```
void solve_start() noexcept {
    #pragma omp parallel num_threads(use_threads) {
        #pragma omp single {
            #pragma omp task {
                solve(0, 0);
            }
            #pragma omp task {
                solve(0, 1);
            }
        }
    }
```

```
solve(configuration, 0);
...
solve(configuration, 1);
```

```
#pragma omp task if (index <= max_depth) {
    solve(configuration, 0);
}
...
#pragma omp task if (index <= max_depth) {
    solve(configuration, 1);
}
```

Jelikož je na moderních procesorech zaručeno že lze data číst paralelně bez chyby, jediná kritická sekce musela být umístěna k bloku, který aktualizuje nejlepší cenu a konfiguraci, je to jediné místo, kde by více vláken zapisovalo do stejné proměnné. Navíc byl použít double-checked locking pattern, takže vlákna skoro nikdy nečekají na kritickou sekci.

3. Popis paralelního algoritmu v OpenMP - data paralelismus

Algoritmus pro data paralelismus vychází částečně z sekvenční verze. Vstupy, výstupy i parametry jsou stejné jako u task paralelismu. Místo hloubky zanoření se počítá počet konfigurací jako 64 * počet vláken. Vzoreček částečně vychází z task paralelizmu (počet stavů je 2^hloubka zanoření). Konstanta 2^6 (64) byla zvolena experimentálně. Stejně jako u task paralelismu potřebujeme dostatek stavů, aby došlo k rovnoměrnému rozložení úloh a zároveň nechceme stavů moc, aby vlákna dlouho počítali jeden stav a tím se minimalizoval overhead spojený s přepínáním výpočtu.

Úprava na datový paralelizmus je složitější. Využívá se fronty, která je na pozadí implementovaná jako pole fixní délky kvůli efektivnější paralelizaci for cyklu a lepšímu cachování. Na začátku programu do fronty pushneme sériově dostatečný počet konfigurací (64 * počet vláken) a poté paralelním for loopem vyřešíme každý stav fronty sekvenčně:

```
#pragma omp parallel for schedule(dynamic, 4)
for (uint64_t i = 0; i < queue.size(); ++i) {
    solve_seq(queue.at(i));
}
```

Sekvenční verze využívá stejné kritické sekce jako task paralelismus, a navíc první volání (solve_seq) předpočítá cenu konfigurace (u task paralelismu se tohle dělo s každým indexem, tady je potřeba tohle chování jednorázově nahradit).

U omp parallel for bylo zvoleno dynamické plánování, protože využíváme B&B a tím pádem může mít každá konfigurace odlišnou dobu výpočtu. Velikost bloku 4 vychází z toho, že máme 64 tasků na každé vlákno, což je vždy dělitelné 4, tudíž nezbyde žádný neúplný blok. Větší blok by neumožňoval dostatečné rozložení zátěže a menší bloky mají zbytečný overhead. Tyto parametry byly experimentálně ověřeny oproti jiným velikostem bloků a statickému plánování a vyšly jako nejlepší.

4. Popis paralelního algoritmu v MPI

MPI algoritmus vychází z datové OpenMP verze. Vstupy a výstupy jsou stejné, parametr pro počet vláken OpenMP paralelismu spolu se vzorečkem pro výpočet stavů pro datový paralelizmus je stejný. Stejný vzoreček je také použit pro počet úloh, které jsou poslány jednotlivým MPI slavům (64 * počet slavů). Vzoreček byl ověřen experimentálně a funguje stejně jako jiné, o hodně větší konstanty. Menší konstantu jsem zvolil pro případ, že by se stavy posílali po pomalé lince (cokoliv co není základní deska). Díky méně stavům není potřeba tolik přenosů, což bude fungovat dobře v případě, že je na lince zpoždění.

Pro komunikaci mezi slavem a masterem se používá struktura, která obsahuje konfiguraci, index a cenu. Význam se liší podle toho, kdo strukturu posílá. Pokud strukturu posílá master slavovi, konfigurace odpovídá tomu, co má slave vyřešit, index značí odkud se má konfigurace řešit a cena říká nejlepší možnou cenu, kterou master zná. To umožňuje poměrně efektivní B&B ořezávání (nejlepší cena se aktualizuje s každým problémem mezi slavem). Pokud strukturu slave posílá masterovi, konfigurace značí nejlepší nalezenou konfiguraci, cena je cenou této konfigurace a index obsahuje ID slava. Master pak ID slava využije k tomu, aby mu přímo poslal další práci.

Program se dělí na 1 mastera a N slavů. Master na začátku vygeneruje dostatek stav stavů stejně jako v OpenMP verzi a každému slavovi pošle právě jeden stav na vyřešení (zároveň si drží počet stavů, které poslal slavům a nedostal je zpět). Poté se spustí smyčka, která probíhá, pokud počet stavů u slavů není 0. V této smyčce master čeká na výsledek pomocí funkce MPI_recv od jakéhokoliv slava. V momentě, kdy dostane výsledek, aktualizuje nejlepší cenu a konfiguraci a pošle slavovi další práci, pokud nějaká je (tím pádem se počet stavů u slavů nezmění a smyčka pokračuje). V momentě, kdy není práce, která by šla slavovi poslat, počet konfigurací u slavů se zmenší na 0 jakmile dorazí všechny řešení, smyčka se ukončí. Po ukončení smyčky master pošle všem slavům speciální zprávu pro ukončení, vypíše výsledek a sám se ukončí. Slave funguje na verzi datového paralelismu OpenMP. Dostane od mastera jednu konfiguraci, pro tu si vygeneruje dostatek menších stavů a řeší ji stejně jako datový paralelismus OpenMP. Po vyřešení ji pošle zpět masterovi.

Funkce MPI_recv bohužel využívá aktivního čekání. Master tedy na 100 % vytěžuje celé výpočetní vlákno i když provádí velmi málo práce. Pokud by MPI_recv byla navržena jako pasivní čekání (částečně jde zařídit přepínačem --mca mpi_yield_when_idle 1), je práce mastera zanedbatelná a jde ho tedy pustit na stejném CPU jako jednoho ze slavů.

```cpp
//Slave
while (true) {
    //Získej problém od mastera
    MPI_Recv(&p, 1, problem_type, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    //Pokud je to signál k ukončení, zkonči
    if (p.index == WORK_TERMINATE) { break; }
    //Nastav globální nejlepší cenu
    best_price = p.current_best_price;
    //Vyřeš problém pomocí OpenMP datového paralelismu
    solve_start(p.configuration, p.index);
    //Nahraj výsledek do struktury
    p.configuration = best_configuration;
    p.current_best_price = best_price;
    //Nahraj své ID do struktury
    p.index = cpu_rank;
    //Pošli výsledek masterovi
    MPI_Send(&p, 1, problem_type, 0, 0, MPI_COMM_WORLD);
}
```

```cpp
//Master - dokud je v oběhu práce
while(work_sent > 0) {
    //Přijmi řešení od slava
    MPI_Recv(&p, 1, problem_type, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    --work_sent;
    //Pokud je řešení globálně nejlepší, aktualizuj ho
    if (p.current_best_price < best_price) {
        best_price = p.current_best_price;
        best_configuration = p.configuration;
    }
    //Pokud je nějaká nevyřešená práce
    if (queue.size() > 0) {
        pair<uint64_t, int64_t> subproblem = queue.pop();
        single_problem s = {subproblem.first, subproblem.second, best_price};
        //Pošli ji slavovi
        MPI_Send(&s, 1, problem_type, p.index, 0, MPI_COMM_WORLD);
        //Zaeviduj
        ++work_sent;
    }
}
```

5. Naměřené výsledky a vyhodnocení

Sekvenční algoritmus

Data	mhr_37_15_17	mhr_37_15_18	mhr_38_15_18	mhr_38_20_15
Čas (ms)	271246	620216 *	690394 *	397043

* Data označená hvězdičkou lehce přesahují dobu výpočtu ve frontě PDP_serial, algoritmus používá na začátku funkci solve(0, 0); solve(0, 1);. Aby šly tyto data použít, byla spočtena nejdříve levá větev jako jeden program solve(0, 0), výsledek a konfigurace byly uloženy do souboru, poté byl spuštěn druhý program, který nahrál konfiguraci a výsledek a spustil solve(0, 1). Čas vznikl jako součet časů levé a pravé větve.

OpenMP algoritmy

mhr_37_15_17	2	4	6	8	10	14	20
Task (ms)	200394	103427	75683	56885	43501	36434	27825
Data (ms)	145724	75105	57780	41066	34511	27144	23292

mhr_37_15_18	2	4	6	8	10	14	20
Task (ms)	384826	198392	143416	108677	89126	72039	51999
Data (ms)	329941	170073	111977	90289	73641	55963	42087

mhr_38_15_18	2	4	6	8	10	14	20
Task (ms)	412576	208209	150557	112690	98841	72807	52377
Data (ms)	325844	166275	119751	90052	72053	57083	41078

mhr_38_20_15	2	4	6	8	10	14	20
Task (ms)	265497	138280	92084	74698	58923	47889	36660
Data (ms)	220025	112909	72581	61098	48822	39355	28196

MPI algoritmus (1 master, 2 slave)

mhr_37_15_17	4 (2x2)	8 (4x2)	12 (6x2)	16 (8x2)	20 (10x2)	28 (14x2)	40 (20x2)
MPI (ms)	76096	42414	30380	24543	19504	19583	19439

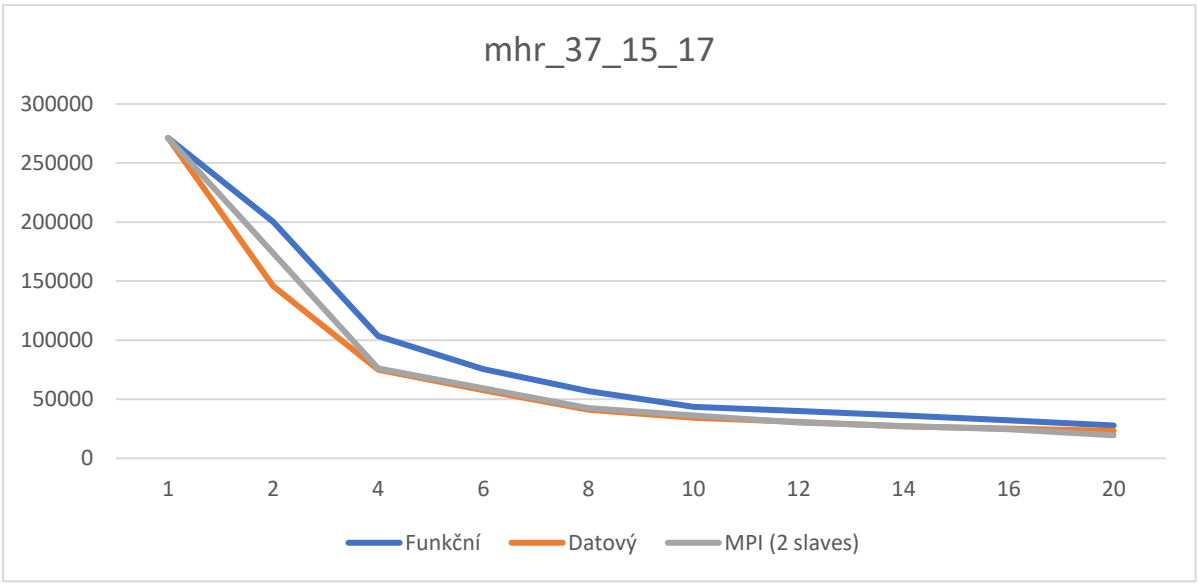
mhr_37_15_18	4 (2x2)	8 (4x2)	12 (6x2)	16 (8x2)	20 (10x2)	28 (14x2)	40 (20x2)
MPI (ms)	143369	80264	57278	46467	36743	36878	36654

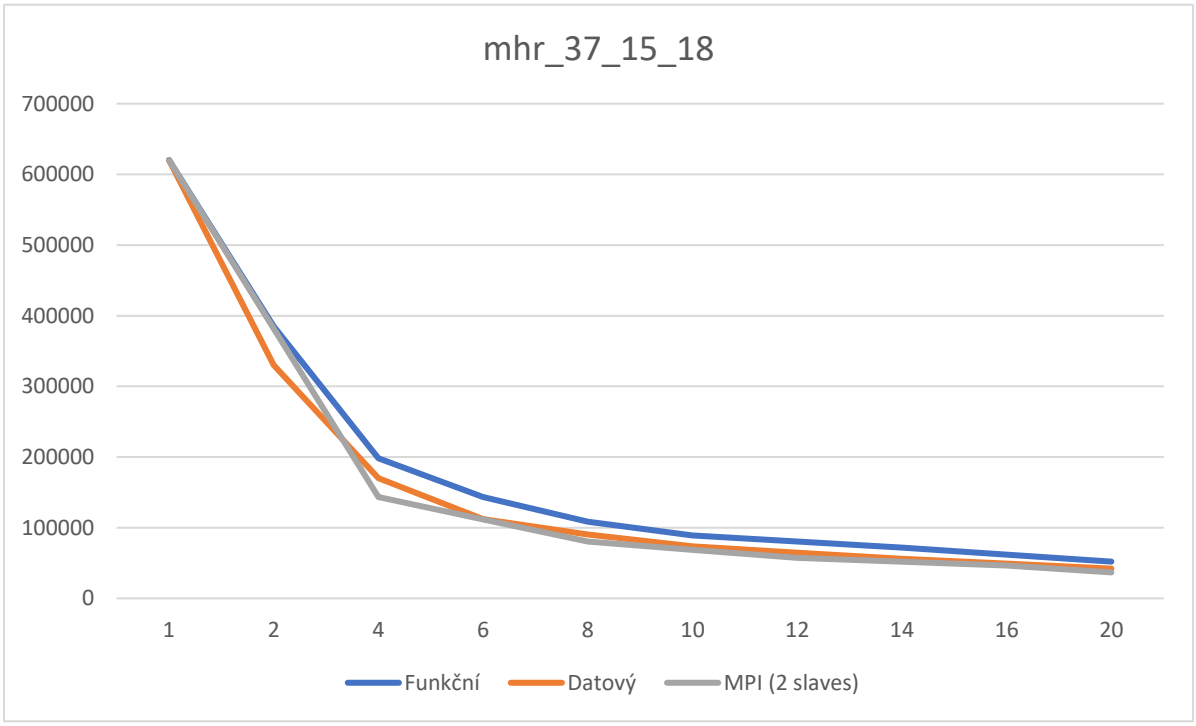
mhr_38_15_18	4 (2x2)	8 (4x2)	12 (6x2)	16 (8x2)	20 (10x2)	28 (14x2)	40 (20x2)
MPI (ms)	153005	85886	61771	49671	39171	39499	39013

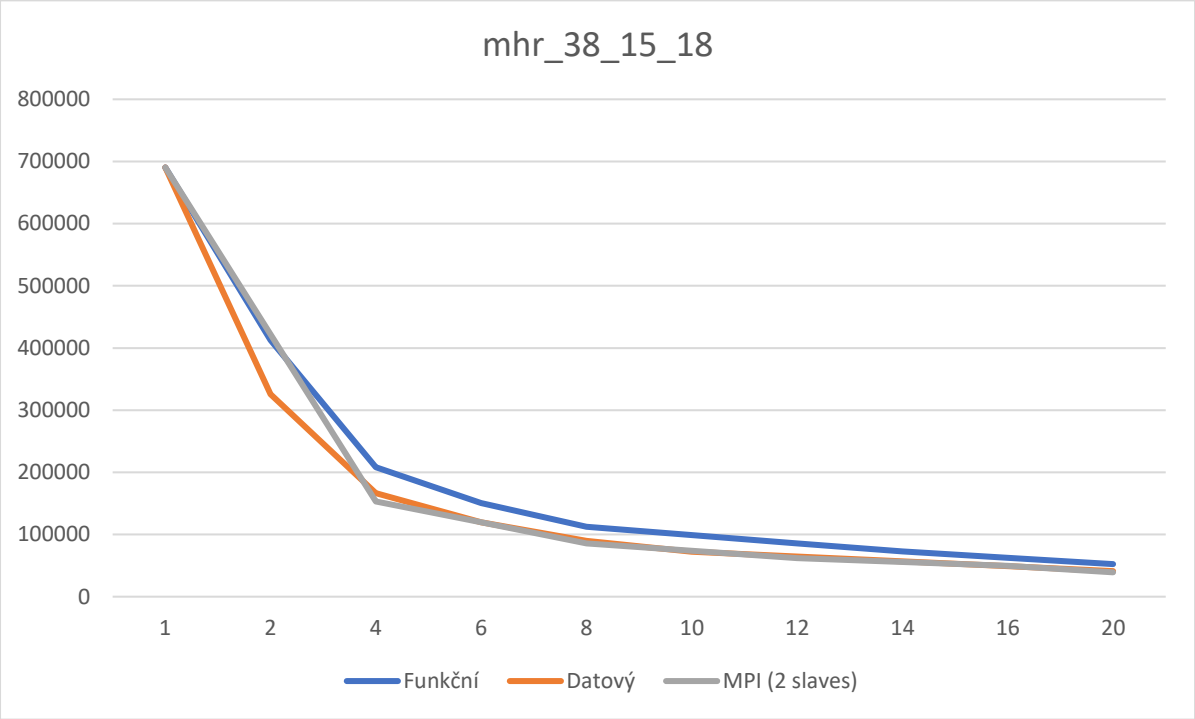
mhr_38_20_15	4 (2x2)	8 (4x2)	12 (6x2)	16 (8x2)	20 (10x2)	28 (14x2)	40 (20x2)
MPI (ms)	96058	54104	38143	31756	24471	24372	24133

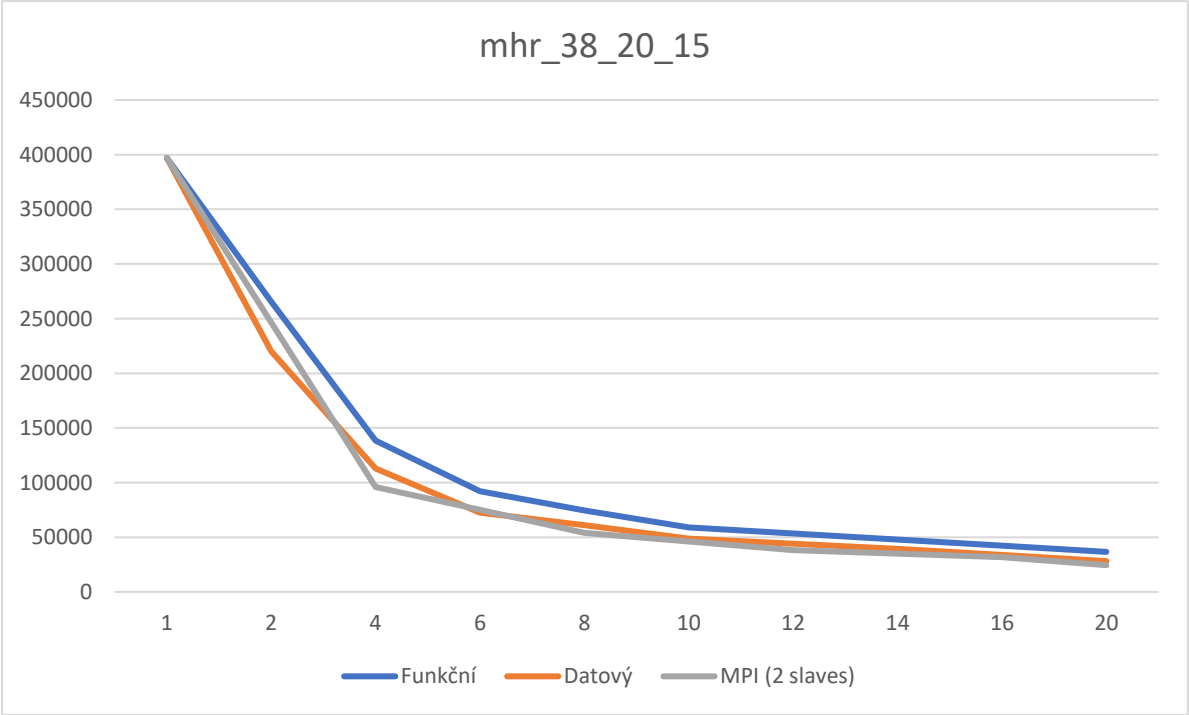
Sloupce 10x2, 14x2 a 20x2 obsahují přibližně stejné časy. Kvůli nastavení Staru nebylo možné získat více jak 10 výpočetních vláken pro OpenMP na slava. Testováno pomocí funkce omp_get_max_threads(). V grafech budu předpokládat největší výpočetní výkon 10x2 (20), což odpovídá OpenMP verzi. Pro OpenMP verzi se mi podařilo vynutit 20 výpočetních vláken, ale pouze ve frontě PDP_long a PDP_fast. PDP_serial nikdy nešla přes 10.

Doba výpočtu (ms) v závislosti na počtu jader









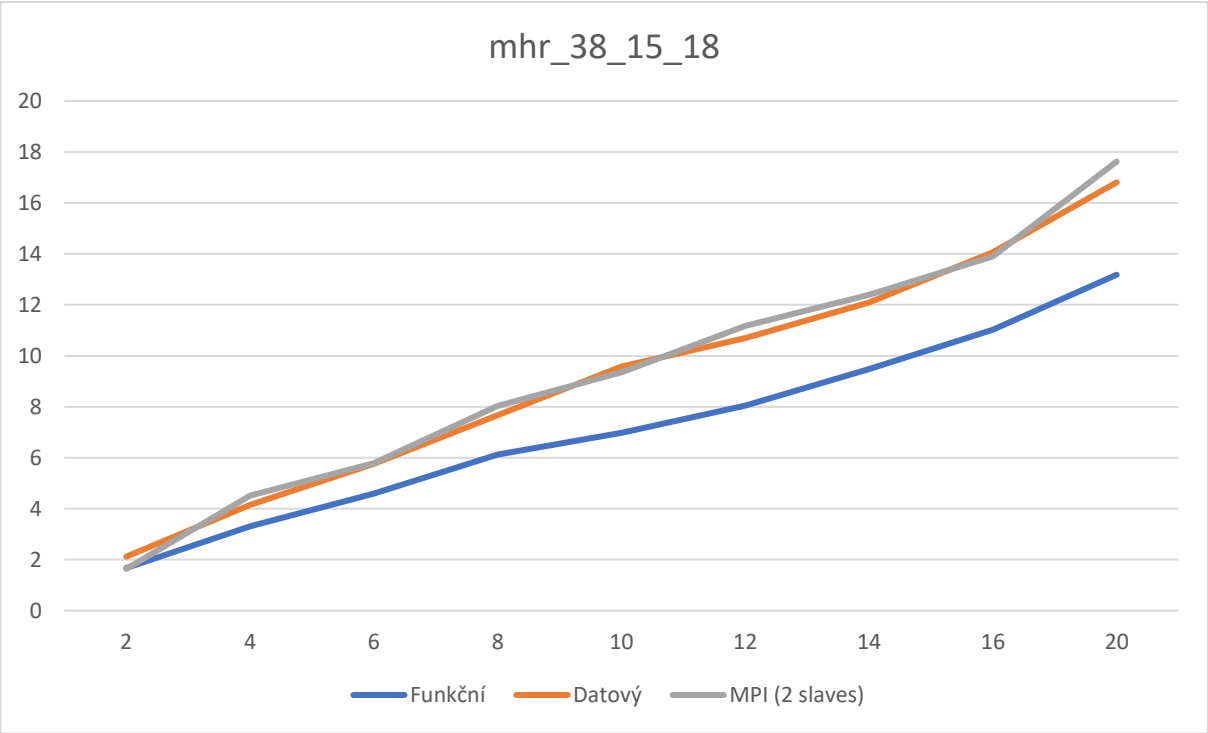
Zrychlení

mhr_37_15_17	2	4	6	8	10	12	14	16	20
Funkční	1.3536	2.6226	3.5840	4.7683	6.2354	6.7867	7.4449	8.4423	9.7483
Datový	1.8614	3.6116	4.6945	6.6051	7.8597	8.7988	9.9929	10.7560	11.6455
MPI (2 slaves)	1.5618	3.5645	4.5776	6.3952	7.4524	8.9284	9.8773	11.0519	13.9072

mhr_37_15_18	2	4	6	8	10	12	14	16	20
Funkční	1.6117	3.1262	4.3246	5.7070	6.9589	7.6967	8.6094	10.0004	11.9275
Datový	1.8798	3.6468	5.5388	6.8692	8.4222	9.5709	11.0826	12.6510	14.7365
MPI (2 slaves)	1.6245	4.3260	5.5467	7.7272	9.0186	10.8282	11.9565	13.3475	16.8798

mhr_38_15_18	2	4	6	8	10	12	14	16	20
Funkční	1.6734	3.3159	4.5856	6.1265	6.9849	8.0443	9.4825	11.0301	13.1812
Datový	2.1188	4.1521	5.7652	7.6666	9.5818	10.6925	12.0946	14.0666	16.8069
MPI (2 slaves)	1.6372	4.5122	5.7800	8.0385	9.3513	11.1767	12.3902	13.8993	17.6251

mhr_38_20_15	2	4	6	8	10	12	14	16	20
Funkční	1.4955	2.8713	4.3117	5.3153	6.7383	7.4344	8.2909	9.3920	10.8304
Datový	1.8045	3.5165	5.4703	6.4985	8.1325	9.0056	10.0888	11.7554	14.0815
MPI (2 slaves)	1.6104	4.1334	5.2882	7.3385	8.6083	10.4093	11.3605	12.5029	16.2250



Efektivita

mhr_37_15_17	2	4	6	8	10	12	14	16	20
Funkční	0.6768	0.6556	0.5973	0.5960	0.6235	0.5656	0.5318	0.5276	0.4874
Datový	0.9307	0.9029	0.7824	0.8256	0.7860	0.7332	0.7138	0.6723	0.5823
MPI (2 slaves)	0.7809	0.8911	0.7629	0.7994	0.7452	0.7440	0.7055	0.6907	0.6954

mhr_37_15_18	2	4	6	8	10	12	14	16	20
Funkční	0.8058	0.7816	0.7208	0.7134	0.6959	0.6414	0.6150	0.6250	0.5964
Datový	0.9399	0.9117	0.9231	0.8587	0.8422	0.7976	0.7916	0.7907	0.7368
MPI (2 slaves)	0.8122	1.0815	0.9245	0.9659	0.9019	0.9023	0.8540	0.8342	0.8440

mhr_38_15_18	2	4	6	8	10	12	14	16	20
Funkční	0.8367	0.8290	0.7643	0.7658	0.6985	0.6704	0.6773	0.6894	0.6591
Datový	1.0594	1.0380	0.9609	0.9583	0.9582	0.8910	0.8639	0.8792	0.8403
MPI (2 slaves)	0.8186	1.1281	0.9633	1.0048	0.9351	0.9314	0.8850	0.8687	0.8813

mhr_38_20_15	2	4	6	8	10	12	14	16	20
Funkční	0.7477	0.7178	0.7186	0.6644	0.6738	0.6195	0.5922	0.5870	0.5415
Datový	0.9022	0.8791	0.9117	0.8123	0.8132	0.7504	0.7206	0.7347	0.7040
MPI (2 slaves)	0.8051	1.0333	0.8813	0.9173	0.8608	0.8674	0.8114	0.7814	0.8112

Zhodnocení

Dle očekávání je task paralelismus nejhorší, rekurze je vždy pomalejší než cyklus. Navíc volání funkce a paralelismus nad funkcemi má větší overhead (vlákno musí nahrát funkci, data na zásobník apod…). Pro stejný počet pracujících vláken jsem očekával, že datový paralelismus bude lepší než MPI, protože MPI je v podstatě datový paralelismus s dalším overheadem (posílání konfigurací přes ethernet). Myslím si, že MPI vychází lépe čistě díky nastavení Star serveru, MPI nemusí sdílet procesor s jiným uživatelem. Na 6-jádrovém desktopu doma mi datový paralelizmus vycházel cca o 20% lépe než MPI. MPI by navíc neměl tak efektivně ořezávat, protože nejlepší cena se aktualizuje pouze když přichází nová konfigurace, kdežto datový paralelizmus má vždy k dispozici globální nejlepší cenu.

Dle očekávání efektivita algoritmu s počtem jader klesá. Důvodem je větší overhead, častější konflikty v kritické sekci, a hlavně menší cache hit rate. Zrychlení mi připadne v pořádku, na několika málo místech je superlineární. Tento problém vznikl nejpravděpodobněji tím, že ke konci semestru byl Star přetížený, a nedalo se úlohy spouštět vícekrát a časy průměrovat. Další nepřesnosti zrychlení vznikly v důsledku nastavení serveru, přetížení apod., opět by šlo odstranit průměrováním časů měření, na což nebyl čas.

Při testech na desktopu se zrychlení datového a MPI paralelizmu blížilo k 1/p (oproti mé sekvenční verzi). Efektivita by šla zlepšit návrhem lepšího sekvenčního řešení, z něhož vychází všechny další algoritmy. Jelikož OpenMP algoritmy mají pouze jednu malou kritickou sekci, do které skoro nikdy nepřistupují zároveň a MPI nemá v rámci Master-Slave žádné kritické sekce nebo jiná čekání, jediné další zlepšení by mohlo být z lepšího plánování / posílání tasků.

MPI verze pro 1x master a Nx slave využívá pouze N-1 procesorů, protože master je v MPI (funkce Recv) implementovaný pomocí aktivního čekání a celý procesor na kterém běží bude nevyužitý. Tento problém jsem vyřešil tak, že jsem v MPI nastavil pomocí přepínače yield-when-idle pasivní čekání a pro X jader jsem program spouštěl na -np X+1. Jelikož master 99 % času spal, jádro, které sdílel se slavem bylo plně využito k výpočtům. Na Staru ovšem nelze do fronty přidávat program s parametrem -np > 3, a proto tyto data chybí. Univerzálnější řešení by bylo pomocí OpenMP implementovat zbylých N-1 vláken procesoru jako dalšího slava.

6. Závěr

Projekt mi připadl velmi zajímavý, v C++ programuji velmi často a doteď jsem znal pouze paralelizmus ze standartní knihovny (který je mimochodem od C++17 velmi dobrý). S OpenMP a MPI se pracuje velmi jednoduše. Připadlo mi, že je velmi těžké udělat někde chybu (time/data race). Jednotlivé úlohy jsou dobře poskládané, takže stačí udělat pár drobných změn a nemusí se přepisovat celý kód. Jediné, co mě zaskočilo je, že datový paralelizmus jsme měli dělat jako for loop nad frontou, což mi strašně připomíná jenom jinou verzi task paralelismu. Datový paralelismus jsem si představoval spíš jako vícevláknové počítání nad jedním polem / maticí / proměnnou, jak to bylo např. v OSY.