Semestrální projekt NI-PDP 2020/2021

Paralelní algoritmus pro řešení problému střelce a jezdce na šachovnici

Ondřej Schejbal

FIT ČVUT, Thákurova 9, 160 00 Praha 6

May 5, 2021

1 Definice problému a popis sekvenčního algoritmu

Za úkol bylo řešit problém střelce a jezdce na šachovnici (dále jen saj). K dispozici byla na vstupu zadána šachovnice. Na zadané šachovnici se kromě postavy jezdce a střelce vyskytovaly také postavy pěšců. Cílem této úlohy bylo najít nejkratší možnou posloupnost tahů, které vedou k zabrání všech pěšců. Na tahu se vždy střídal střelec s jezdcem. První tah vždy náležel střelci. V rámci zadaných dat byla k dispozici také informace o rozměrech zpracovávané čtvercové šachovnice a také tzv. maximální hloubka, což byla horní mez počtu kroků, která pomohla omezit prohledávaný stavový prostor problému.

Cílem celého projektu bylo implementovat řešení zadaného problému nejprve běžným sekvenčním algoritmem a následně na tento algoritmus aplikovat různé techniky, které běh programu paralelizují a následně pozorovat jejich efektivitu.

Výkony různých podob implementace budeme sledovat a porovnávat na 3 instancích problému. Jedná se o instance saj6, saj7 a saj12, které jsme dostali ve vzorových datech a jejichž velikost hrací plochy je 11x11, 11x11 resp. 13x13. Jedná se o instance, jejichž vzájemný čas běhu v sekvenčním algoritmu byl nejvíce odlišný, což doufám povede k možnostem pozorovat odlišnosti i v jednotlivých verzích implementace paralelizace.

Všechny naměřené hodnoty, které se v tomto reportu budou vyskytovat byly naměřeny na výpočetním serveru STAR, který nám dala fakulta k dispozici. Jednotlivé verze řešení byly vždy spuštěny 10x a prezentované hodnoty jsou vždy průměrnou hodnotou z těchto 10 běhů. Má implementace je napsána v jazyce C++ a všechny verze mého programu je vhodné kompilovat s optimalizací o3, konkrétně s použitím přepínačů *-O3 -funroll-loops*.

2 Popis sekvenčního algoritmu

V mé implementaci řešení zadaného problému jsem se rozhodl řešit zadaný problém za využití principu rekurze. Úkol si tedy rozděluji technikou rozděl-a-panuj na menší a menší podúlohy a následně v kořenech tohoto pomyslného stromu vyhodnocuji jednotlivé podoby řešení. Tento přístup bylo možné zvolit díky tomu, že jsme v instancích, které nám byly zpřístupněny vedoucími předmětu, měli k dispozici informaci o maximálním počtu kroků řešení. Kdyby jsme tuto informaci zadanou neměli, tak by problém byl neupočítatelně složitý a větvení rekurze bylo nekonečné a tento přístup bych k implementování řešení nemohl použít.

Ukončující podmínka rekurze je založena právě na porovnání aktuální hloubky stromu se zadaným maximem. Zde byl prostor pro aplikování jednoduché heuristiky, která hledání řešení velmi urychlila. Namísto porovnávání aktuální hloubky stromu s maximální hloubkou jsem k aktuální hloubce přičetl také počet pěšců, kteří ještě nebyli zajati na šachovnici. Myšlenka tohoto vylepšení je založena na tom, že pokud na šachovnici zbývá např. 5 pěšců, tak je k jejich zajmutí potřeba minimálně 5 kroků. Tedy pokud součet aktuální hloubky a počtu nezajatých pěšců je větší než maximální hloubka, rekurze se zastaví a dále nerozvětvuje.

Další heuristikou, která také přinesla velké zrychlení bylo snižování hranice maximální hloubky. V případě, že se v průběhu rekurzivního volání nalezlo validní řešení, jehož hloubka byla nižší než hodnota maximální hloubky, tak se hodnota maximální hloubky nastaví na hodnotu hloubky tohoto nalezeného řešení.

Ve svém řešení jsem zadanou šachovnici reprezentoval jako vector znaků a nejlepší nalezenou posloupnost kroků (řešení) jsem si uchovával ve vectoru jako globální proměnnou.

V tabulce 1 je možné vidět naměřené hodnoty pro vybrané instance. Nejdůležitější je pro nás hodnota ve sloupci **Čas**, kterou budeme pozorovat i v dalších částech. Počet volání již v dalších částech neuvádím, protože se s každým spuštěním lišil. To je samozřejmě následkem paralelizace. Všechny hodnoty v tabulce jsou průměr nasbíraných hodnot z 10 měření.

Název instance	**Čas**	**Počet volání**
saj12	236.836	2042336142
saj6	59.887	502969403
saj7	146.307	1257866470

Tabulka 1: Naměřené hodnoty implementace sekvenčního řešení

3 Popis paralelního algoritmu v OpenMP - task paralelismus

Prvním technika paralelizování kódu byla za pomocí openMP knihovny, kde jsem využil principu task paralelismu. Task paralelismus spočívá v tom, že se úkol rozdělení na části (tasky) a ty jsou následně přiřazovány vláknům. Task paralelismus mi šel zprovoznit bez obtíží a stačilo k tomu vhodně označit paralelní bloky původního sekvenčního kódu pomocí vhodných direktiv.

Konkrétně bylo třeba udělat úpravu na 3 místech. Nejprve bylo třeba deklarovat blok, který chceme paralelizovat a deklarovat, aby první běh rekurze zpracovalo pouze 1 vlákno. Toho bylo docíleno pomocí direktivy **omp parallel**, resp. **omp single**, jak je možné vidět na úryvku kódu 1, kde *findBestPlay()* je rekurzivní funkce, jež provádí veškerou logiku algoritmu a je shodná s tou ze sekvenčního řešení. V části kódu, která těmto direktivám předchází jsou zadefinovány pouze proměnné, které obsahují informace, ke kterým by mělo mít přístup každé vlákno, a proto jsem nastavil výchozí chování proměnných jako *shared*. Chci aby vlákna měli k proměnným přístup a případné časově závislé chyby jsem si v kódu pohlídal, viz dále.

```
# pragma omp parallel default(shared)
{
    # pragma omp single
        findBestPlay(...);
}
```

Kód 1: Použití **parallel** a **single** direktiv v openMP - task paralelismu

Další nutnou úpravou bylo označit kritickou sekci, kterou byla část, kde se upravuje maximální hloubka prohledávaného řešení. Tato kritická sekce byla ošetřena pomocí direktivy **omp critical** viz úryvek kódu 2.

Dále se v mém kódu vyskytovala ještě jedna kritická sekce a tou byla inkrementace čítače počtu volání. To jsem ošetřil tím, že jsem použil direktivu **omp atomic**, která danou inkrementaci označila za atomickou operaci a tím jsem předešel případným problémům. Počítání iterací se ukázalo být méně významným při

```
if (remainingSoldiers <= 0) {
    if (currentDepth < MAX_DEPTH) {
        #pragma omp critical
        {
            if (currentDepth < MAX_DEPTH) {
                MAX_DEPTH = currentDepth;
                bestMoves = currentMoves;
            }
        }
    }
    return;
}
```

Kód 2: Použití **critical** direktivy v openMP - task paralelismu

paralelizovaném výpočtu, protože u jednotlivých běhů se počty volání očekávaně lišily pro každý běh. V dalších úpravách své implementace jsem tedy tuto část kódu vynechal úplně.

Poslední změnou, kterou bylo třeba pro korektní implementaci task paralelismu provést bylo podstromy přidělovat jednotlivým vláknům, neboli přidělit vláknům tzv. tasky. To jsem provedl direktivou **omp task**. V kódu 3 je možné vidět použití této direktivy. Práci jsem přidělil samostanému vláknu pouze v případech, kdy aktuální hloubka byla menší, nebo rovna polovině celkového počtu pěšců. Díky této podmínce vytvářím tasky pouze do určité hloubky podstromů. Vytváření tasků na všech úrovních vede k situaci, kdy se výhody paralelizace začnou ztrácet. Výpočty ve velkých hloubkách jsou často velmi krátké a převažuje tedy náročnost plánovače při přepínaní vláken oproti výpočetní náročnosti.

```
if (currentDepth <= SOLDIERS_CNT/2) {
    # pragma omp task
        findBestPlay(...);
} else {
    findBestPlay(...);
}
```

Kód 3: Použití **task** direktivy v openMP - task paralelismu

Program je kvůli openMP knihovně potřeba kompilovat navíc s přepínačem *-fopenmp*. Na vstupu program očekává počet vláken, které má použít k vykonání práce.

V tabulce 2 je možné vidět naměřené časy pro různé počty vláken. Časy jsou průměrnou hodnotou pro 10 spuštění dané instance problému. Můžeme pozorovat, že tato úprava přinesla výrazné zrychlení výpočtů u instancí saj6 a saj7 zatímco u větší instance byla v průměru pomalejší než sekvenční algoritmus.

U instancí saj6 a saj7 je vidět, že od pěti vláknech se již výpočetní čas nijak zásadně nezlepšoval, naopak někdy se dokonce mírně zhoršil. To je nejspíše způsobeno tím, že režie na organizaci vláken byla zbytečně velká oproti řešenému problému, takže nevynikly výhody paralelizace. U největší z instancí (saj12) se totiž tento jev neprojevil. Také je dobré zmínit, že pro instanci saj12 se nepodařilo na výpočetním clusteru STAR naměřit data pro 1 vlákno, protože doba běhu vždy přesáhla stanovený horní limit. Běh pro 1 vlákno byl tedy výrazně pomalejší než u sekvenčního algoritmu.

Název instance	Počet vláken	Čas
saj12	5	365.2275
saj12	10	377.5876
saj12	15	292.3617
saj12	20	244.8638
saj6	1	6.2333
saj6	5	6.2016
saj6	10	4.6299
saj6	15	5.4644
saj6	20	6.3161
saj7	1	3.6959
saj7	5	1.5261
saj7	10	2.1194
saj7	15	3.9072
saj7	20	0.9991

Tabulka 2: Naměřené doby běhu pro jednotlivé instance a různé počty vláken - openMP task paralelismus

4 Popis paralelního algoritmu v OpenMP - datový paralelismus

Dále jsem provedl opět paralelizaci za pomocí knihovny openMP. Tentokrát se ale jednalo o datový paralelismus, který spočívá ve vhodném přidělení jednotlivých podstromů vláknům v rámci paralelizace for cyklu. Klasická realizace spočívá v tom, že z počátečního řešení (počátečního stavu šachovnice) se připraví vhodný počet podstromů, které jsou následně přiřazovány jednotlivým vláknům dle vybraného rozvrhu (schedule).

Data jsem si připravil do vectoru za pomocí algoritmu BFS, kde jsem si připravil celkem deseti násobek počtu vláken stavů, které reprezentovaly začátky jednotlivých podstromů. Tento počet jsem vybral na základě experimentů, kde na vybraných datech přinášel tento počet nejlepší časové zrychlení. Některé tyto stavy již samozřejmě obsahují i částečně rozdělané řešení, což jsem zohlednil i ve své implementaci vhodnou datovou strukturou. Implementace datového paralelismu následně byla již bez obtíží. For cyklus zpracovávající vector těchto připravených částečných řešení jsem anotoval direktivou **omp parallel for**, jak je možné vidět v úryvku kódu 4. Dále bylo potřeba ošetřit kritickou sekci, kde jsem upravoval proměnnou maximální hloubky. To jsem provedl úplně stejně jako u task paralelismu, viz kód 2. Jednotlivá vlákna následně provádějí sekvenční výpočet pro přidělený podstrom.

```
vector<Solution> subRoots = getMovesBFS(threadCnt * 10, root);

# pragma omp parallel for schedule(dynamic)
for (int i = 0 ; i < subRoots.size() ; i++) {
    findBestPlay_seq(...);
}
```

Kód 4: Použití **parallel for** direktivy v openMP - datovém paralelismu

Pro datový paralelismus bylo třeba vybrat vhodný schedule, neboli přístup pro rozdělování jednotlivých prvků iterovaného vectoru vláknům. Vyzkoušel jsem různé kombinace možných přístupů, kde nejvhodnějším se

ukázal být **dynamic**, přidělující prvky iterace vláknům po jednom. Porovnání jednotlivých přístupů je možné vidět v tabulce 3.

Název instance	dynamic, 1	dynamic, 2	dynamic, 3	guided
saj6	2.625	2.679	4.769	3.906
saj7	0.060	0.164	4.100	3.164
saj12	97.151	113.431	118.395	425.549

Tabulka 3: Porovnání doby běhu pro různé hodnoty parametru schedule

V tabulce 4 je možné vidět porovnání naměřených časů běhu pro vybrané instance s různým počtem přidělených vláken. Opět se jedná o průměr z 10 běhů mé implementace. Můžeme pozorovat, že při přidělení pouze jednoho vlákna je výpočet výrazně pomalejší než běh sekvenčního algoritmu. S rostoucím počtem přidělených vláken je možné u každé instance pozorovat snižující se časovou náročnost. Je možné vidět, že pro největší instanci (saj12) je zlepšení výrazné pro každé zvýšení počtu vláken zatímco u saj6 a saj7 je výrazné časové zlepšení při nasazení 5 vláken, ale další navyšování počtu vláken již výslednou časovou náročnost nezlepšuje v takové míře. Můžeme tedy považovat přidání dalších vláken v takovém případě za zbytečné, protože nenese skoro žádné zlepšení vzhledem k navýšení počtu vláken.

Vzhledem k použití knihovny openMP je i tuto verzi potřeba kompilovat s přepínačem *-fopenmp*. Na vstupu program také očekává počet vláken.

Název instance	Počet vláken	Čas
saj12	1	347.1333
saj12	5	193.3745
saj12	10	141.5383
saj12	15	90.3697
saj12	20	79.1904
saj6	1	88.7293
saj6	5	4.9187
saj6	10	2.7228
saj6	15	2.3538
saj6	20	2.1387
saj7	1	218.272
saj7	5	0.8337
saj7	10	1.1007
saj7	15	1.2358
saj7	20	0.1667

Tabulka 4: Naměřené doby běhu pro jednotlivé instance a různé počty vláken - openMP datový paralelismus

5 Popis paralelního algoritmu v MPI

Poslední úpravou bylo vyzkoušet si paralelizovat program tak, aby byl funknčí i pro jednotlivé výpočetní jednotky, které nesdílí paměť. To bylo provedeno rozšířením implementace openMP datového paralelismu pomocí knihovny **MPI**.

MPI umožňuje zprostředkovat posílání zpráv mezi jednotlivými výpočetními jednotkami. Hlavním cílem této části implementace bylo správně zprovoznit komunikaci a celou spolupráci jednotlivých výpočetních jednotek společně se správným posíláním si dat obsahujících úkoly, řešení, či jiné instrukce. To jsem zrealizoval přístupem master-slave. Jedna výpočetní jednotka slouží pro řízení komunikace, rozdělování úkolů a synchronizací mezi pracujícími vlákny. Všechny ostatní budou tzv. sluhové a musí tedy umět přijímat úkoly a zpracovávat je. Má implementace těchto 2 stran je přiblížena v následujících podsekcích.

5.1 Popis implementace master vlákna

Master proces je nejdůležitější jednotkou ke správnému doběhnutí celého programu. V master vlákně jsem si nejprve vygeneroval vector podstromů pro zadanou instanci problému. To jsem provedl pomocí algoritmu BFS úplně stejně, jako jsem generoval tyto stavy v implementaci datového paralelismu pod openMP. Následně jsem pomocí blokujícího volání **MPI_Send** rozeslal do každého vlákna jednoduchou zprávu obsahující strukturu se základními informacemi o řešené instanci. Posílám informaci o rozměrech šachovnice, počtu pěšců a zjištěné maximální hloubce.

Následuje hlavní cyklus, jehož úkolem je obsluhovat slave procesy, tedy posílat jim úkoly a přijímat od nich výsledky. To jsem implementoval pomocí while cyklu tak, že vždy na začátku cyklu volám blokující příkaz **MPI_Probe**, který se odblokuje v případě, že ze slave procesu přichází nějaká zpráva. Příkaz načte informace o zprávě aniž by ji přijal. Následně je zkoumám flag příchozí zprávy, který indikuje o který typ zprávy se jedná následně přijme daný typ zprávy. Pokud se jedná o flag indikující, že vlákno je připraveno k práci, tak vláknu práci pošle. Dalším typem flagu, který může přijít je informace o tom, že je práce hotova. V takovém případě master vlákno přijme vypracované řešení, zkontroluje jeho validitu, případně si ho uloží jako doposud nejlepší řešení a následně vláknu pošle další práci.

Posílání práce jsem implementoval pomocí 3 zpráv, ve kterých se postupně pošlou všechny potřebné informace do pracovního vlákna. Součástí poslední zprávy je i informace o tom, jaká je aktuální hodnota maximální hloubky. Tuto hodnotu si master vlákno aktualizuje vždy při aktualizaci nejlepšího řešení. Pracující vlákna tak mají možnost doběhnout rychleji díky této aktualizované hodnotě.

Tento cyklus pokračuje tak dlouho, dokud master z vectoru nepošle všechny připravené podstromy k řešení a následně nedokončí svou práci všechna slave vlákna. Jakmile tato situace nastane, tak vyšle všem slave vláknům zprávu o tom, že je již všechna práce hotova. Vše výše popsané je možné vidět v kódu 5.

5.2 Popis implementace slave vlákna

Slave procesy vykonávají následující instrukce. Každý slave nejprve přijme od mastera strukturu se základními informacemi o řešené instanci. Následně vyšle zprávu master vláknu o tom, že je připraven pracovat. Následně spustí nekonečný while cyklus, ve kterém vždy na začátku zavolá blokující **MPI_Probe** a podobně jako v master vláknu, tak jakmile se toto volání odblokuje, tak vykoná sadu instrukcí, závisející na flagu příchozí zprávy.

Od master vlákna následně mohou přicházet zprávy 2 typů. Buďto zpráva obsahující první část práce. V

```
vector<Solution> subRoots = getMovesBFS(processCount * 10, startingSol);

// initial information push to slaves
InitialInfoForSlave initialInfo{BOARD_SIZE, MAX_DEPTH, SOLDIERS_CNT};
for (int workerRank = 1 ; workerRank < processCount ; workerRank++) {
    MPI_Send(&initialInfo, ..., workerRank, TAG_M_INITIAL_INFO, MPI_COMM_WORLD);
}

vector<BoardMove> receivedSolution(MAX_DEPTH, BoardMove(-1));
while (i < subRoots.size() || workingSlaves > 0) {
    MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    if (status.MPI_TAG == TAG_S_WORK_DONE) {
        workingSlaves--;
        MPI_Recv(&receivedSolution[0], ..., status.MPI_SOURCE, TAG_S_WORK_DONE, ...);
        // compare received solution with current best
        short foundSolMovesCnt = isSolutionValid(receivedSolution);
        if (foundSolMovesCnt > -1 && foundSolMovesCnt < MAX_DEPTH) {
            bestMoves = receivedSolution;
            MAX_DEPTH = foundSolMovesCnt;
        }
    } else {
        MPI_Recv(nullptr, ..., status.MPI_SOURCE, TAG_S_READY_TO_SERVE, ...);
    }

    if ((status.MPI_TAG == TAG_S_WORK_DONE || status.MPI_TAG == TAG_S_READY_TO_SERVE)
        && i < subRoots.size()) {
        // send work info (in 3 messages)
        MPI_Send(boardState, ..., status.MPI_SOURCE, TAG_M_WORK_PART_1, MPI_COMM_WORLD);

        MPI_Send(moves, ..., status.MPI_SOURCE, TAG_M_WORK_PART_2, MPI_COMM_WORLD);

        WorkPart3 wp3{playerOnTurn, bishopPosition,
        knightPosition, movesCnt, remainingSoldierCnt, MAX_DEPTH};
        MPI_Send(&wp3, ..., status.MPI_SOURCE, TAG_M_WORK_PART_3, MPI_COMM_WORLD);
        workingSlaves++;i++;
    }
}

// inform slaves that work is done
for (int workerRank = 1 ; workerRank < processCount ; workerRank++) {
    MPI_Send(nullptr, 0, MPI_BYTE, workerRank, TAG_M_ALL_WORK_DONE, MPI_COMM_WORLD);
}
```

Kód 5: Pseudokód implementace master vlákna v MPI

takovém případě proces přijme i zbylé 2 zprávy obsahující zbývající informace o úkolu k zpracování a vypracuje řešení. Vypracuje ho pomocí openMP implementace datového paralelismu, který byl rozebírán v sekci 4. Jakmile nalezne výsledek, tak ho odešle do master vlákna. Druhým druhem zprávy, kterou může slave vlákno obdržet je informace o tom, že je již všechna práce hotova. V takovém případě ukončí while cyklus a skončí.

Implementace pod MPI se kompiluje pomocí mpi překladače. Já jsem kompiloval kód pod překladačem *mpi++*. Následně se program spustí pomocí příkazu *mpirun -np x a.out*, kde *x* je požadovaný počet výpočetních jednotek.

V tabulce 5 je možné pozorovat naměřené doby běhu pro vybrané instance pro 3, resp. 4 výpočetní uzly a různé počty vláken. Pro méně výpočetních jednotek neměla úloha vzhledem k implementovanému master-slave principu smysl. Výsledky nejsou dle mého názoru tak dobré, jak bych očekával. Narůstající počet vláken vždy zmenšuje dobu výpočtu, ale u instancí saj6 a saj7 jsou časy výpočtu s 3 výpočetními jednotami shodné s časy když pracovali 4 výpočetní jednotky. Pouze u instance saj12 je možné zaznamenat zlepšení u vyššího počtu výpočetních jednotek. Vyplývá z toho, že při větších instancích úkolu je přidání výpočetních jednotek vhodné, ale u těch menších je to zbytečný výkon, který nepřináší výrazné zlepšení.

Dále je dobré upozornit na to, že naměřené časy jsou sice lepší než pro sekvenční běh, ale jsou řádově horší než naměřené hodnoty pro taskový a datový paralelismus z minulých sekcí. Toto je velmi nečekané zjištení, neboť tato implementace měla datový paralelismus ještě zrychlit za pomoci více výpočetních jednotek. To se mu ovšem povedlo pouze pro největší vybranou instanci, tedy pro saj12. Vyplývá z toho, že vypracovaná úprava má smysl pouze pro velké instance problému. Pro malé instance nejspíše naopak brzdí výpočty komunikace mezi výpočetními jednotkami.

```
// initial info request
InitialInfoForSlave initialInfo;
MPI_Recv(&initialInfo, ...);
// process initial info, prepare necessary data

// inform master that I am ready to serve
MPI_Send(nullptr, ..., 0, TAG_S_READY_TO_SERVE, MPI_COMM_WORLD);

while(true) {
    MPI_Probe(0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    if (status.MPI_TAG == TAG_M_ALL_WORK_DONE) {
        MPI_Recv(nullptr, ..., 0, TAG_M_ALL_WORK_DONE, ...);
        break;
    }

    // receive work from master (in 3 messages)
    if (status.MPI_TAG == TAG_M_WORK_PART_1) {
        vector<char> boardState(BOARD_SIZE * BOARD_SIZE);
        MPI_Recv(&boardState[0], ..., 0, TAG_M_WORK_PART_1, ...);

        vector<BoardMove> moves(initialInfo.maxDepth, BoardMove(-1));
        MPI_Recv(&moves[0], ..., 0, TAG_M_WORK_PART_2, ...);

        WorkPart3 wp3;
        MPI_Recv(&wp3, ..., 0, TAG_M_WORK_PART_3, ...);

        // update global best sol depth
        MAX_DEPTH = wp3.bestMaxDepth;

        // process received data and prepare them for work

        // find solution for received subRoot
        findBestPlay_parallel(...);
        MPI_Send(&bestMoves[0], ..., 0, TAG_S_WORK_DONE, MPI_COMM_WORLD);
    }
}
```

Kód 6: Pseudokód implementace slave vlákna v MPI

Název instance	Počet výp. uzlů	Počet vláken	Čas
saj12	3	5	122.726
saj12	3	10	71.7923
saj12	3	15	63.2767
saj12	3	20	55.1936
saj12	4	5	125.5993
saj12	4	10	57.3432
saj12	4	15	49.2835
saj12	4	20	44.7386
saj6	3	5	50.8584
saj6	3	10	33.3417
saj6	3	15	27.7284
saj6	3	20	26.3768
saj6	4	5	52.4282
saj6	4	10	32.224
saj6	4	15	28.1418
saj6	4	20	26.1661
saj7	3	5	132.5276
saj7	3	10	84.0055
saj7	3	15	68.5804
saj7	3	20	64.9596
saj7	4	5	133.1156
saj7	4	10	84.4819
saj7	4	15	69.882
saj7	4	20	63.2993

Tabulka 5: Naměřené doby běhu pro jednotlivé instance a různé počty vláken - paralelismus pod MPI

6 Hodnocení naměřených výsledků

Jednotlivá pozorování k naměřeným hodnotám je vždy možné najít v jednotlivých sekcí. Nyní se zaměříme na vypočtení zrychlení $S(n,p)$ a efektivity $E(n,p)$, Tyto hodnoty byly vypočítány pomocí následujících vzorců.

$$S(n,p) = \frac{\text{čas sekvenčního běhu}}{\text{čas paralelního běhu}} \tag{1}$$

$$E(n,p) = \frac{\text{čas sekvenčního běhu}}{p \cdot \text{čas paralelního běhu}} \tag{2}$$

Jednotlivé tabulky s vypočtenými hodnotami jsou níže. Můžeme pozorovat, že velikost zrychlení se pro jednotlivé druhy paralelizace značně liší. Jde tedy rozlišit jejich účinnost pro různě velké instance. V následujících odstavcích přiblížím pozorované vlastnosti pro jednotlivé úpravy.

U taskového paralelismu v openMP můžeme pozorovat, že výpočet největší instance se i pro vyšší počet vláken pouze přiblížil rychlosti sekvenčního řešení. Je to překvapující výsledek, ale zřejmě je to způsobeno danou podobou instance, ne pouze její velikostí. U instancí saj6 a saj7 je naopak zrychlení výrazné. U saj6 je vidět, že optimální počet vláken je 10 a přidání dalších vláken již větší zrychlení nepřinese. Naopak režie hodnotu zrychlení snížila. U instance saj7 se situace vyvíjela podobně jako u saj6. Konkrétně se od 5 vláken zrychlení zmenšovalo, ale pro 20 vláken se zase výrazně zvýšilo. Task paralelismus si tedy obecně dobře poradí se zrychlením pro menší a lehčí instance problému.

U datového paralelismu je situace s většími instancemi lepší než u taskového paralelismu, ovšem zrychlení je pořád velmi malé oproti množství přidaných vláken oproti sekvenčnímu řešení. Je vidět, že instance saj12 je velmi odolná vůči paralelizaci a ukázala se být méně vhodnou pro pozorování zrychlení. U instance saj6 je hezky vidět, že se zrychlení zvyšuje s rostoucím počtem vláken narozdíl od taskového paralelismu. U instance saj7 datový paralelismus přinesl dokonce superlineární zrychlení.

U datového paralelismu s MPI rozšířením je lineární řešení stabilně u všech pozorovaných instancí shodné, což je dobré vzhledem k tomu, že instance saj12 se při ostatních zpracováních vždy odlišovala. Indikuje to, že je algoritmus velmi robustní vůči podobě zadání. Vyšší počet vláken hodnotu zrychlení zvyšuje, ale ne tak výrazně, aby se dalo považovat za přínosné. Je možné vidět, že rozdíl mezi hodnotami pro 3 a 4 výpočetní uzly je zanedbatelný, což je překvapující. Myslím, že MPI implementace má potenciál přinést ještě vyšší zrychlení, které se mi nepovedlo implementovat. Myslím, že hlavním zdržením v mé implementaci by mohlo být množství posílaných zpráv, kde se mi nepovedlo velké množství dat správně zabalit a poslat v jedné zprávě, ale namísto toho jsem posílal 3 zprávy s menším payloadem.

Název instance	Počet vláken	S(n,p)	E(n,p)
saj12	5	0.6485	0.1297
saj12	10	0.6272	0.0627
saj12	15	0.8101	0.054
saj12	20	0.9672	0.0484
saj6	1	9.6076	9.6076
saj6	5	9.6567	1.9313
saj6	10	12.9348	1.2935
saj6	15	10.9595	0.7306
saj6	20	9.4816	0.4741
saj7	1	39.5863	39.5863
saj7	5	95.8699	19.174
saj7	10	69.0323	6.9032
saj7	15	37.4455	2.4964
saj7	20	146.4388	7.3219

Tabulka 6: Paralelní zrychlení $S(n, p)$ a efektivita $E(n, p)$ pro openMP - task paralelismus

Název instance	Počet vláken	S(n,p)	E(n,p)
saj12	1	0.6823	0.6823
saj12	5	1.2248	0.245
saj12	10	1.6733	0.1673
saj12	15	2.6207	0.1747
saj12	20	2.9907	0.1495
saj6	1	0.6749	0.6749
saj6	5	12.1754	2.4351
saj6	10	21.9946	2.1995
saj6	15	25.4427	1.6962
saj6	20	28.0016	1.4001
saj7	1	0.6703	0.6703
saj7	5	175.4912	35.0982
saj7	10	132.9218	13.2922
saj7	15	118.3905	7.8927
saj7	20	877.6665	43.8833

Tabulka 7: Paralelní zrychlení $S(n, p)$ a efektivita $E(n, p)$ pro openMP - datový paralelismus

Název instance	Počet výp. uzlů	Počet vláken	S(n,p)	E(n,p)
saj12	3	5	1.9298	0.6433
saj12	3	10	3.2989	1.0996
saj12	3	15	3.7429	1.2476
saj12	3	20	4.291	1.4303
saj12	4	5	1.8856	0.4714
saj12	4	10	4.1301	1.0325
saj12	4	15	4.8056	1.2014
saj12	4	20	5.2938	1.3234
saj6	3	5	1.1775	0.3925
saj6	3	10	1.7962	0.5987
saj6	3	15	2.1598	0.7199
saj6	3	20	2.2704	0.7568
saj6	4	5	1.1423	0.2856
saj6	4	10	1.8585	0.4646
saj6	4	15	2.128	0.532
saj6	4	20	2.2887	0.5722
saj7	3	5	1.104	0.368
saj7	3	10	1.7416	0.5805
saj7	3	15	2.1334	0.7111
saj7	3	20	2.2523	0.7508
saj7	4	5	1.0991	0.2748
saj7	4	10	1.7318	0.433
saj7	4	15	2.0936	0.5234
saj7	4	20	2.3114	0.5778

Tabulka 8: Paralelní zrychlení $S(n, p)$ a efektivita $E(n, p)$ pro implementaci pod MPI

7 Závěr

Celkově mě práce na semestrální práci bavila. Přišlo mi, že rozdělení do vybraných iterací bylo velmi praktické. Na STARu jsem si i osvěžil psaní scriptů, což také oceňuji jako velký přínos.

Svou implementaci hodnotím kladně, sekvenční řešení dobíhalo na úlohách z courses v zhruba srovnatelných časech jako referenční řešení. Taskový a datový paralelismus v openMP se mi povedlo zprovoznit bez chyb, ale narazil jsme na to, že jsem vybral nevhodnou instanci, která nebyla moc přívětivá k paralelizaci. U MPI implementace již tak dobré pocity ze své implementace nemám. Zaměřil jsem se na to, aby řešení bylo přehledné a dobře jsem korigoval posílání zpráv, ale nakonec se mi nepovedlo zprovoznit posílání složitější struktury dat a byl jsem nucen data posílat po jednodušších celcích ve více zprávách. Zde vidím prostor pro zlepšení, který se také zřejmě promítl do naměřených hodnot. Také jsem si rozšířil povědomí o možnostech paralelizace a také toho, že paralelizace nutně neznamená zrychlení programu.

8 Zdroje

- `https://courses.fit.cvut.cz/NI-PDP`
- `https://gitlab.fit.cvut.cz/schejond/ni-pdp`