Semestralni projekt NI-PDP 2020/2021:

Paralelni algoritmus pro reseni problemu SAJ: strelec a jezdec na sachovnici

Mykyta Boiko

magisterske studijum, FIT CVUT, Thakurova 9, 160 00 Praha 6

May 5, 2021

1 Definice problemu SAJ

V teto semestralni prace je resena uloha SAJ: strelec a jezdec na sachovnici.

1.1 Vstupni data

- **k** = prirozene cislo, **20>k>5**, reprezentujici delku strany sachovnice **S** o velikosti **kxk**
- **q** = prirozene cislo **q**$<k^2/2$ reprezentujici pocet rozmistenych figurek na sachovnici **S**
- **C[1..q]** = pole souradnic rozmistenych figurek na sachovnici **S**
- **I** = souradnice strelce na sachovnici **S**
- **J** = souradnice jezdce (kone) na sachovnici **S**

1.2 Pravidla

Na pocatku je na ctvercove sachovnici **S** rozmisteno textbfq figurek a 1 strelec a 1 jezdec. Tomuto rozmisteni figurek budeme rikat **pocatecni konfigurace**. Jeden **tah** je posun strelce podle sachovych pravidel ci posun jezdce podle sachovych pravidel. Prvni tah provadi strelec. Pokud strelec ci jezdec tahnou na policko obsazene nekterou figurkou, seberou ji. Strelec s jezdcem se museji pravidelne stridat v tazich (strelec musi tahnout, i kdyz na polickach jeho barvy zadne figurky nezbyvaji). Strelec nikdy nesebere jezdce a jezdec nikdy nesebere strelce, i kdyby tato moznost vznikla. **Cilem hry** je odstranit vsechny figurky pomoci **minimalniho poctu** tahu tak, aby se sachovnice dostala do **cilove konfigurace**, kdy na ni zustane samotny strelec a jezdec.

1.3 Vystup algoritmu

- Nejkratsi posloupnost stridavych tahu strelce a jezdce vedouci do cilove konfigurace.
- Posloupnost ma format **seznamu** souradnic policek, na ktere stridave tahne strelec a jezdec s oznacenim **hvezdickou** tech policek, kde doslo k odstraneni nektere figurky.

Vsechna mereni jsem provadel jak na svem pocitaci tak i na klasteru STAR. V teto zprave uvadim jenom vysledky a mereni ze STARu.

2 Popis sekvencniho algoritmu a jeho implementace

Definovany problem jsem resil rekurzivnim algoritmem **BB-DFS**, coz je algoritmus zalozeny na prohledavani stavoveho prostoru do hloubky, cilem ktereho je nalezeni pripustneho koncoveho stavu **s nejmensi cenou**.

Pridal jsem radu vylepseni a orezavani prostoru, ktere zrychluji vypocet. Nyni se probereme nektere dulezite aspekty moje implementace.

2.1 Implementacni aspekty

Prvni vec na kterou se podivame je trida reprezentujici sachovnice, resp. jednotlive stavy ve kterych se ona nachazi v prubehu behu algoritmu.

Source Code 1: Trida Chessboard

```
class Chessboard {
private:
    // list of all possible movement directions for bishop
    static const std::vector<Coord_t> BISHOP_MOVEMENT_DIRECTIONS;
    // list of all possible movement directions for knight
    static const std::vector<Coord_t> KNIGHT_MOVEMENT_DIRECTIONS;
    // 20>size>5; max_size = 19
    int8_t size;
    Coord_t knightCoord;
    Coord_t bishopCoord;
    // one dimensional boards coords representation; only the
     ↪  location of enemies is indicated as true value
    std::vector<bool> boardEnemiesCoords;
    // the number of enemies for the current state of the board
    int16_t actualEnemiesCnt;
    // enum represented whose move is next: BISHOP or KNIGHT
    ChessPiece whoOnTurn;
    // the set returned by calling next(whoOnTurn) can contain up
     ↪  to 2k-2 fields for the bishop or 8 for the knight,
     ↪  depending on  whoOnTurn and the occupancy of the
     ↪  chessboard by pieces
    std::vector<Coord_t> next() const;
    // the val() function evaluates fields from the set returned by
     ↪  next() function
    int8_t val(Coord_t &coord) const;

    std::vector<Coord_t> nextBishopPositions() const;
    std::vector<Coord_t> nextKnightPositions() const;
    bool isLeadToDiagonalWithAnyEnemy(Coord_t &coord) const;
```

```
25      // etc..
26  };
```

Druha nemene dulezita trida z impementacni casti je trida, ktera se stara o reseni problemu. Obsahuje samotny algoritm, provadi rekurzivni volani, porovnava nalezeny ceny atd.

Trida **Solver** nejperve prijme na vstup instanci problemu, coz je vyse zminovana trida **Chessboard**. V prvnim kroku se vygeneruje nejaky pocet startovnich reseni. Potom z techto stavu pousti prohledavani do hloubky. V dalsich verzich kodu pocet vygenerovanych stavu bude zaviset na poctu vlaken a procesu, ktere provadi vypocet.

Source Code 2: Funkce solve v tride Solver

```
1   void Solver::solve() {
2       // ... start time recording ... //
3       this->clock.startTime();
4
5       // Ala BFS with DEPTH=1 (in the future will be moved into a
        ↪  separate function returning the queue of pairs(Move,
        ↪  Chessboard_state))
6       for(auto &nextPriorPosition: this->problemToSolve-
        ↪  >getChessboard()->getNextOrderedByVal())
        ↪  {
7           auto temporaryMovingHistory = solveDFSChessboard(*this-
            ↪  >problemToSolve->getChessboard(), nextPriorPosition,
            ↪  1);
8
9           if (temporaryMovingHistory.empty()) continue;
10
11          // was found better solution, so update target result
            ↪  history
12          if (temporaryMovingHistory.size() <
            ↪  resultMovingHistory.size() ||
            ↪  resultMovingHistory.empty()) {
13              resultMovingHistory =
                ↪  std::move(temporaryMovingHistory);
14          }
15          // was found the best possible solution
16          if (this->isFoundBestPossibleSolution()) break;
17      }
18
19      // ... stop time recording, save solution ... //
20  }
```

Rekurzivni prohledavani prostoru a jeho nasledujici orezavani se provadi ve funci **solveDF-**

SChessboard. Nejlepsi nalezena cena v danem kroku rekurzivniho volani se porovna s globalni aktualne nejlepsi cenou. V pripade, kdyz je mensi, tak se updatuje globalni nejlepsi cena a dojde k navraceni, v opacnem pripade, nic se updatovat nebude a stejne dojde k navraceni z dane vetvi.

Za zminku take stoji, ze pridal jsem do te funkce par optimalizacnich podminek/triku.

- K navratu taky dojde kdyz soucet aktualni hloubky(/pocet jiz provedenych tahu na sachovnice) a poctu zbyvajicich nepratelu v aktualnim stavu sachovnice je vetsi nebo se rovna globalni aktualne nalazene nejlepsi cene.

```
curDepth + board.getActualEnemiesCnt() >= this->curBestSolutionDepth
```

- Pri expandovani dalsich stavu pro rekurzivni prohledavani jsem pridal **Fast Forward Check**. Myslenka je takova, ze v podstate je to uplne to same, co je v bodu vys, ale kontrola se provadi o krok dopredu. Kdyz soucet aktualni hloubky + 1 a poctu nepratelu v pristim kroku je vetsi nebo roven globalni aktualne nejlepsi nalezene cene, tak pak je rekurzivni volani funkce pro tento stav preskoceno, nebot v teto vetvi lepsi reseni nenajdeme.

```
(curDepth + 1) + board.getFastForwardEnemiesCntForPos(nextPriorPosition) >
>curBestSolutionDepth
```

- Stejne tak, skoro vsude je pridana podminka, ktera kontroluje, jestli jiz bylo nalezeno nejlepsi mozne reseni, coz je pocet nepratelu na sachovnice na zacatku. Je treba rici, ze tato podminka neni pro dane datove vstupy nikdy splnena. Tak ci onak, algoritmus musi prohledat skoro cely stavovy prostor pro nalezeni reseni.

Samotna rekurzivni funkce prohledavajici do hloubky bez sikovneho orizavani zabirala prilis dlouhou dobu. Zminene optimalizacni triky moc zrychluji beh programu.

Source Code 3: Funkce solveDFSChessboard v tride Solver

```
1  /**
2   * Recursive function that trying to find the shortest sequence of
    ↪  moves to remove all enemies pieces on board
3   * @param board indicates chessboard state for current depth
4   * @param posToMove
5   * @param curDepth
6   * @return
7   */
8  std::list<Coord_t> Solver::solveDFSChessboard(Chessboard board,
    ↪  Coord_t &posToMove, int16_t curDepth) {
9      // increment counter
10     this->recursiveCallsCnt++;
```

```
11      // update chessboard state
12      board.moveActiveChessPieceToPos(posToMove);
13
14      // if the number of enemies went to zero & count of
        ↪  steps/price is best ever => update curBestSolutionDepth
15      if (board.getActualEnemiesCnt() == 0 &&
        ↪  (this->curBestSolutionDepth > curDepth)) {
16          this->curBestSolutionDepth = curDepth;
17          return {posToMove};
18      }
19
20      // return empty list in cases:
21      // - is found best possible solution or
22      // - conditions for min actual depth are not met
23      if (curDepth + board.getActualEnemiesCnt() >=
        ↪  this->curBestSolutionDepth ||
        ↪  this->isFoundBestPossibleSolution()) return {};
24
25      std::list<Coord_t> resultMovingHistory;
26
27      for(auto &nextPriorPosition: board.getNextOrderedByVal()) {
28          // do fast forward check; comment out this line to see
            ↪  clear recursive behavior of DFS
29          if ((curDepth + 1) +
            ↪  board.getFastForwardEnemiesCntForPos(nextPriorPosition)
            ↪  >= this->curBestSolutionDepth) {
30              continue;
31          }
32
33          auto temporaryMovingHistory = solveDFSChessboard(board,
            ↪  nextPriorPosition, curDepth+1);
34
35          if (temporaryMovingHistory.empty()) continue;
36
37          // better solution was found, so update target result
            ↪  moving history
38          if (temporaryMovingHistory.size() <
            ↪  resultMovingHistory.size() ||
            ↪  resultMovingHistory.empty()) {
39              resultMovingHistory =
                ↪  std::move(temporaryMovingHistory);
40          }
```

```
41          if (this->isFoundBestPossibleSolution()) break;
42      }
43
44      // add resultMovingHistory
45      if (!resultMovingHistory.empty()) {
46          // prepend the posToMove to the beginning of the
            ↪  resultMovingHistory
47          resultMovingHistory.push_front(posToMove);
48      }
49
50      return resultMovingHistory;
51  }
```

2.2 Vysledky STAR VS Referencni

Vsechny casy jsou namereny na STARu a zprumerovany ze 3 behu. Vetsina vysledku je mnohem lepsi nez referencni: plati to jak pro cas, tak i pro pocet volani.

Vysledky STAR				
Instance	Cas [s]	Referencni cas [s]	Pocet volani	Referencni pocet volani
saj1	0.000	0.0	6.000e+00	750
saj2	2.433	0.6	8.085e+05	14e+6
saj3	0.071	0.01	2.039e+05	300e+3
saj4	5.978	0.4	3.399e+06	8e+6
saj5	0.067	12	1.114e+04	193e+6
saj6	4.373	31	1.105e+06	502e+6
saj7	0.470	67	7.173e+04	1.2e+9
saj8	123.472	1170	3.849e+07	19e+9
saj9	6.068	6.3	2.947e+06	89e+6
saj10	71.627	36	1.513e+07	460e+6
saj11	0.597	0.2	2.271e+05	3e+6
saj12	238.843	72	8.878e+07	1.2e+9

3 Popis paralelniho algoritmu a jeho implementace v OpenMP - taskovy paralelismus

K paralelizace sekvencniho algoritmu se vyuziva framework **OpenMP**. Rozlisujeme dva pristupy k paralelizace:

- **OpenMP Task** - funkcni paralelismus
- **OpenMP Data** - iteracni datovy paralelismus

V teto kapitole jde o funkcnim paralelismu. U funkcniho paralelismu se namisto puvodniho rekurzivniho volani funkce vytvareji ulohy pro OpenMP. Princip je takovy, ze vytvarena uloha je vlozena do frony uloh a ceka na vyzvednuti/zpracovani nekterym volnym vlaknem. Co se tyce implementacni casti, tak kod pro OpenMP task paralelismus je skoro stejny jako sekvencni. Pribylo jen par drobnych zmen. Ted telo funkce **solve()** vypada nasledovne:

Source Code 4: Telo funkce solve() v tride Solver pro taskovy paralelismus

```
/**
 * Main function responsible for processing and solving problems
 */
void Solver::solve() {
    // .. start time recording ..//
    // do BFS expandation, returns  queue of
     ↪  pairs(Chessboard_state, nextMove)
    expandBFS();

    // number of thread can be set whether OMP_NUM_THREADS (env
     ↪  var) or num_threads(8)
    #pragma omp parallel default(none)
    {
        #pragma omp single
        {

            while (!tasksQueue.empty())
            {
                auto task = std::move(tasksQueue.front());
                tasksQueue.pop();

                auto board = task.first;
                auto posToMove = task.second;
                #pragma omp task default(none) firstprivate(board,
                 ↪  posToMove)
                {
                    solveDFSChessboard(board, posToMove);
                }

                // was found the best possible solution
                if (isFoundBestPossibleSolution()) break;
            }
        }
    }

```

```
33      // ... stop time recording and save solution ...//
34  }
```

Co se vlastne zmenilo: expandovani stavu pro reseni v rekurzivni funce se premistilo do zvlastni funkce **expandBFS()**. Pri spousteni vypoctu se zase zavola rekurzivni funkce, ale jiz v paralelnim bloku. V sekci #**pragma omp single** veskere vypocty se provadi jenom v jednom vlaknu: pripravujeme stavy pro rekurzivni reseni. Pak nasleduje sekce #**pragma omp task**, ve ktere se prave provadi paralelizace kodu.

Pri paralelizace rekurzivni funkce pribylo par dulezitych zmen. Prvni a nejdeluzitejsi je, ze aktualizace nejlepsi ceny byla premistena do kriticke sekce.

Source Code 5: Kriticka sekce pri aktualizace nejlepsiho reseni

```
1       if (board.getActualEnemiesCnt() == 0 && (curUpperBoundDepth >
        ↪  board.getMovingHistorySize())) {
2           #pragma omp critical
3           {
4               if (curUpperBoundDepth > board.getMovingHistorySize())
                ↪  {
5                   curUpperBoundDepth = board.getMovingHistorySize();
6                   curBestSolutionMovingHistory =
                    ↪  board.getMovingHistory();
7               }
8           }
9           return;
10      }
```

Taky jsem navic zavedl konstantu poctu kroku: zbyvajici pocet kroku pred prepnutim na zpracovani jednim vlaknem, t.j. sekvencne.

Source Code 6: Podminka na prepnuti na sekvencne zpracovani

```
1           // Let's define 4 as the number of steps:
2           // - remaining to the curUpperBoundDepth before switching
            ↪  to one thread/sequence processing
3           if (curUpperBoundDepth - board.getMovingHistorySize() <=
            ↪  4) {
4               solveDFSChessboard(board, nextPriorPosition);
5           } else {
6               #pragma omp task default(none) firstprivate(board,
                ↪  nextPriorPosition)
7               {
8                   solveDFSChessboard(board, nextPriorPosition);
9               }
10          }
```

Tenhle trik se docela zrychlil beh celeho programu. Timto se vlastne vyhneme zbytecnemu overheadu pro vytvarni vlaken.

Toto reseni jsem si pro mereni nevybral, jelikoz OpenMP datovy paralelismus byl o neco trochu rychlejsi.

4 Popis paralelniho algoritmu a jeho implementace v OpenMP - datovy paralelismus

4.1 Popis algoritmu pro datovy paralelismus

Datovy paralelismus uz predstavuje odlisny druh pristupu k paralelnimu rozdeleni prace. Zase puvodni kod sekvencniho reseni se trochu menil.

Stejna jak u taskoveho paralelismu na zacatku sekvencne pomoci **BFS** se vygeneruje startovni mnozina stavu/konfigurace sachovnice. Ale ted uz je dulezite, aby pocet vygenerovanych stavu byl minimalne stejne velky jako pocet vlaken. V mem pridade, velikost startovni mnoziny stavu je zvolena jako pocet vlaken vynasobeny nejakou konstantou. Konstantu jsem nastavil na 22.

Potom v paralelnim for cyklu rozdeluje prace jednotlivym valknum. Jednotliva reseni jsou provadena sekvencne. Jak planovac jsem zvolil **auto**, prestoze jsem zkousel vsechny mozne varianty. Pri porovnani s ostanimi, **auto** ukazal nejlepsi vysledky. Stoji jeste za zminku, ze pokud volime planovac jiny nez auto (coz znamena, ze zodpovednost vyberu planovacu je kompilatoru), tak dava taky smysl nastavit velikost chunku, coz je pocet ukolu deleno poctem vlaken.

```
schedule(dynamic, chunk_size)
```

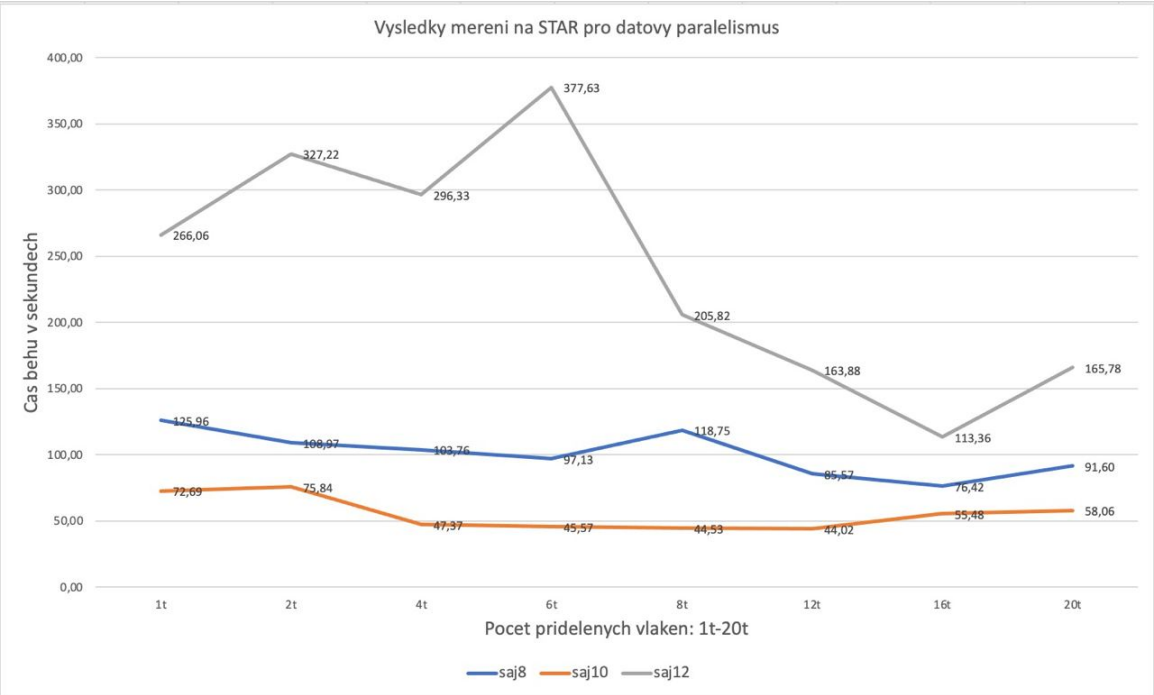
Source Code 7: Funkce solve v tride Solver pro datovy paralelismus

```
1  /**
2   * Main function responsible for processing and solving problems
3   */
4  void Solver::solve() {
5      // ...start time recording...//
6      // IMPORTANT NOTICE: number of thread can be set whether
       ↪  OMP_NUM_THREADS (env var) or num_threads(8)
7
8      // number of tasks to expand = MAX_NUMBER_OF_THREADS *
       ↪  CONSTANT (for example 22)
9
10     size_t numberOfTasksToExpand = omp_get_num_threads() * 22;
11     // do BFS expansion, returns  queue Chessboard states
12     expandBFS(numberOfTasksToExpand);
13     // move all task from queue to vector
```

```
14      std::vector<Chessboard> tasksVector(tasksQueue.begin(),
        ↪  tasksQueue.end());
15      tasksQueue.clear();
16
17      // chunk size is tasks count divided by number of threads:
        ↪  chunk_size = (int)tasksVector.size() /
        ↪  omp_get_max_threads()
18      // schedule(dynamic, chunk_size)
19      #pragma omp parallel for shared(tasksVector) default(none)
        ↪  schedule(auto)
20      for (size_t indx = 0; indx < tasksVector.size(); indx++) {
21          // Debug section
22          // printf("Thread %d (i = %zu)\n", omp_get_thread_num(),
            ↪  indx);
23
24          if (!isFoundBestPossibleSolution()) {
25              solveDFSChessboard(tasksVector[indx]);
26          }
27      }
28
29      // ...stop time recording and save solution...///
30  }
```

4.2 Vysledky STAR

Na STARu jsem meril cas pro vsechny vstupni soubory z rozsahu saj1-12. Vysledky jsem meril na **1, 2, 4, 6, 8, 12, 16, 20 vlaknech**. V diagramu niz ukazu jen vystupy casu pro nejnarocnejsi vstupy pri sekvencnim zpracovani. Zvolil jsem tri nejnarocnejsi ulohy/vstupy: **saj8, saj10, saj12**.



Na prvni pohled na diagram nejde rict, ze vysledky moc dobre. Navic pro vstup **saj12** pri 6 vlaknech se program skoro dvakrat zpomalil. Ale kdyz se podivame obecne, tak urcite se to zrychlelo.

S jistotou mohu rici, ze bychom nemeli ocekavat velky narust. Podle meho nazoru to muze byt zpusobeno tim, ze sekvencni program byl naimplementovany pro dane zadane vstupy az moc dobre a prave spravne bylo zvolene poradi smeru pohybu pro strelce a jezdce. Pokud bych to poradi mel jinak, tak by sekvencni program bezel o neco dele, stejne tak i datovy paralelismus by ukazal lepsi casovy prirust.

Source Code 8: Seznamy vsech moznych smeru pohybu pro strelce a jezdce

```
const std::vector<Coord_t> Chessboard::BISHOP_MOVEMENT_DIRECTIONS
    =
        std::vector<Coord_t>({
                                    Coord_t(-1, -1),
                                    Coord_t(-1, 1),
                                    Coord_t(1, -1),
                                    Coord_t(1, 1)
                                });

const std::vector<Coord_t> Chessboard::KNIGHT_MOVEMENT_DIRECTIONS
    =
        std::vector<Coord_t>({
                                    Coord_t(-2, -1),
```

```
12                                  Coord_t(-1, -2),
13                                  Coord_t(1, 2),
14                                  Coord_t(1, -2),
15                                  Coord_t(2, -1),
16                                  Coord_t(2, 1),
17                                  Coord_t(-1, 2),
18                                  Coord_t(-2, 1)
19                          });
```

Co jeste muzeme udelat, tak porovnat nejlepsi vysledky z datoveho paralelismu s sekvencnim resenim. Nejlepsi vysledky jsem ziskal, kdyz jsem to spoustel na 16 vlaknech. Takze to porovname:

OpenMP Data Vs Sekvencni reseni			
Instance	OpenMP 16 Threads cas [s]	Zrychleni (proti sekvencnimu)	Sekvencni cas [s]
saj8	76.42	**1.6x**	123.472
saj10	55.48	**1.3x**	71.627
saj12	113.36	**2.1x**	238.843

Vidime, ze nejvetsi zrychleni bylo pro vstup **saj12**, a to je az **2.1x**. Muzeme dojit k zaveru, ze datovy paralelismus funguje rychleji, i kdyz nepozorujeme silny narust.

5 Popis paralelniho algoritmu a jeho implementace v MPI

Paralelni algoritmus pro system s distribuovanou pameti je implementovan jako **Master-Slave** s pomoci knihovny **MPI**, ktera zajistuje meziprocesovou komunikace, tedy vypocet problemu je mezi jednotlive CPU. Jeden proces ma funkci mastera, ostatni jsou slave procesy. Komunikace se probiha prostrednictvim zprav. Kazdy vypocetni uzel muze posilat ruzne typy zprav. Adresatem zpavy muze byk jak nejaky konkretni uzel, tak i skupina uzlu. Pro kominkace mezi procesy jsou pouzity blokujici volani: funkce **MPI_Send** a **MPI_Recv**.

Pro paralelizace na urovni jednoho procesu je pouzit pristup **OpenMP Data**.

5.1 Master

Master ridi vsechny pracujici procesy a prideluji jim praci. Je treba poznamenat, ze prave v moji implementaci *master vyuziva jen jedno vlaknu a sam nikdy nepracuje*, a jen praci prideluje, dostava vysledky od slavu a aktualizuje globalni aktualne nejlepsi reseni. Master se zabyva nasledujicimi vecmi:

1. Stejne jako v datovem paralelismu, na zacatku predpocita startovni mnozinu stavu a ty nasledne distribuuje. Pocet stavu na expondovani je roven poctu slavu vynasobeny nejakou konstantou (coz je vetsi nez pocet slavu).

2. Pak zacne rozesilat zpravy slavum. Vypada to nasledovne: najde nejaky **volny** slave proces, posle jemu zpravu a oznaci ten slave jako **busy**. Zatimco fronta se startovnimi stavy neni prazdna master bude zkouset posilat zpravy slavum. Pokud vsechny slavy jsou **busy**, tak zkusi pockat na odpoved od nejakeho slavu. Az dostane odpoved, tak ji zpracuje (pripadne aktualizuje globalni nejlepsi reseni), pak tento slave oznaci jako **volny** a odesle jemu jednu ze zbyvajicich zprav, ktera navic bude obsahovat aktualni globalni nejlepsi reseni, cimz se da pomoct slavu oriznout prohledavaci prostor.

3. Az se fronta se startovnimi stavy vyprazdni, tak pote ve smycce master bude cekat na odpovedi s resenimi od zbyvajicih **busy** slave procesu. Po prijmuti odpovedi se provede porovnani s jiz nejlepsim nalezenym resenim a pripadne jeho aktualizace.

4. Pokud jiz bylo nalezene nejlepsi mozne reseni nebo vsechny slave procesy jsou volne a master uz nema pro nich zadnou dalsi prace, tak rozesle vsem zpravu o ukonceni.

Source Code 9: Zjednoduseny kod masteru

```
1         // do BFS expansion, returns  queue Chessboard states; cnt
           ↪  of tasks to expand: number_of_slaves * constant (for
           ↪  example 2)
2         std::deque<Chessboard> tasksQueue =
           ↪  expandBFS(*problemToSolve->getChessboard(),
           ↪  (number_of_processes - 1) * 2);
3
4        // ... ///
5
6         // send all generated tasks to free (not busy) SLAVES
7         while (!tasksQueue.empty()) {
8             auto freeSlaveIter = std::find(freeSlaves.begin(),
               ↪  freeSlaves.end(), true);
9             if (freeSlaveIter != freeSlaves.end()) {
10                auto freeSlaveNum = freeSlaveIter -
                   ↪  freeSlaves.begin() + 1;
11
12                // unwrap new task
13                auto boardState = tasksQueue.front();
14                tasksQueue.pop_front();
15
16                // ... serialize board to MPI message //
17
18                // send serialized board to slave and set him at
                   ↪  vector as busy
```

```
19                  MPI_Send(&serializedBoard, sizeof(struct
                     ↪  MpiBoardMessage), MPI_PACKED, freeSlaveNum, 0,
                     ↪  MPI_COMM_WORLD);
20                  // mark this slave as busy
21                  freeSlaves[freeSlaveNum - 1] = false;
22
23              } else {
24                  MpiSlaveSolutionMessage slaveResponseMessage =
                     ↪  MpiSlaveSolutionMessage();
25                  MPI_Status status;
26                  // blocking receive
27                  MPI_Recv(&slaveResponseMessage, sizeof(struct
                     ↪  MpiSlaveSolutionMessage), MPI_PACKED,
                     ↪  MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
28                           &status);
29
30                  // mark this slave as free
31                  freeSlaves[status.MPI_SOURCE-1] = true;
32                  cntOfResponsesToWait--;
33
34                  // ... deserialize and try to update current
                     ↪  solution //
35              }
36          }
```

5.2 Slave

Slave proces vlastne prijima startovni konfigurace k reseni od master procesu.

1. Prijme zpravu/startovni konfigurace od mastera. Pokud to neni zprava o ukonceni behu, tak jde do bodu 2.
2. Ze zpravy vytahne a si nastavi cenu aktualne nejlepsiho reseni. Pak postup je stejny jako v datovem paralelismu.
3. Z toho stavu, ktery on dostane po deserializaci zpravy, slave si zase naexpanduje frontu stavu. Pocet vygenerovanych stavu se bude rovnat poctu vlaken daneho slavu vynasobeny nejakou konstantou.
4. Provede vypocet pomoci datoveho paralelismu a pote posle masterovi zpravu bud obsahujici nejlepsi nalezene reseni nebo zpravu, ze takove reseni se nepodarilo najit.
5. Pokud dostane zpravu od mastera o konci vypoctu, tak ukonci program.

Jak jiz bylo zmineno drive, ze master s kazdou zpravou posila slavu taky dosud nejlepsi nalezene reseni, podle ktereho pote slave muze vice orezavat prostor.

Source Code 10: Zjednoduseny kod slavu

```
while (true) {
            // blocking receive
            MPI_Recv(&masterMessage, sizeof(struct
             ↪  MpiBoardMessage), MPI_PACKED, 0, 0,
             ↪  MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            if (!masterMessage.keepRunningSlave) {
                break;
            }

            // ... propagation of new upper bound to slaves and
             ↪  message deserislization //

            size_t numberOfTasksToExpand = omp_get_num_threads() *
             ↪  22;
            std::deque<Chessboard> tasksQueue =
             ↪  expandBFS(boardState, numberOfTasksToExpand,
             ↪  false);

            // ... move all task from queue to vector ///

            // chunk size is tasks count divided by number of
             ↪  threads: chunk_size = (int)tasksVector.size() /
             ↪  omp_get_max_threads()
            // schedule(dynamic, chunk_size)
            #pragma omp parallel for shared(tasksVector)
             ↪  default(none) schedule(auto)
            for (size_t indx = 0; indx < tasksVector.size();
             ↪  indx++) {
                if (!isFoundBestPossibleSolution()) {
                    solveDFSChessboard(tasksVector[indx]);
                }
            }

            // ... wrapping solution to message //
            // blocking send
```

```
27              MPI_Send(&solutionMessage, sizeof(struct
                ↪  MpiSlaveSolutionMessage), MPI_PACKED, 0, 0,
                ↪  MPI_COMM_WORLD);
28          }
```

5.3 Implementacni detaily

V teto sekce je nutne zminit jak vypadaji samotne zpravy. V kodu niz je ukazano, jak vypada zprava, kterou posila master vsem slavum. Jako datove typy byly pouzite jenom primitvy a 1D pole. Stejne tak teto zprava se posila slavum, kdyz master chce ukoncit vypocet.

Source Code 11: MpiBoardMessage

```
1  struct MpiBoardMessage {
2      int8_t size;
3      int8_t knightCoord[2];
4      int8_t bishopCoord[2];
5      bool boardEnemiesCoords[BOARD_MAX_SIZE];
6      int16_t actualEnemiesCnt;
7      ChessPiece whoOnTurn;
8      int8_t posToMove[2];
9      int16_t movingHistorySize;
10     // moving history for X, Y coordinates
11     int8_t movingHistory_X[UPPER_BOUND_STEPS_CNT];
12     int8_t movingHistory_Y[UPPER_BOUND_STEPS_CNT];
13     int16_t currentUpperBound = UPPER_BOUND_STEPS_CNT;
14     bool keepRunningSlave = true;
15 };
```

Zprava-odpoved od slavu masteru vypada nasledovne:

Source Code 12: MpiSlaveSolutionMessage

```
1  struct MpiSlaveSolutionMessage {
2      // price aka movingHistorySize
3      int16_t price;
4      int16_t movingHistorySize;
5      // moving history for X, Y coordinates
6      int8_t movingHistory_X[UPPER_BOUND_STEPS_CNT];
7      int8_t movingHistory_Y[UPPER_BOUND_STEPS_CNT];
8      int recCallsCnt = 0;
9  };
```

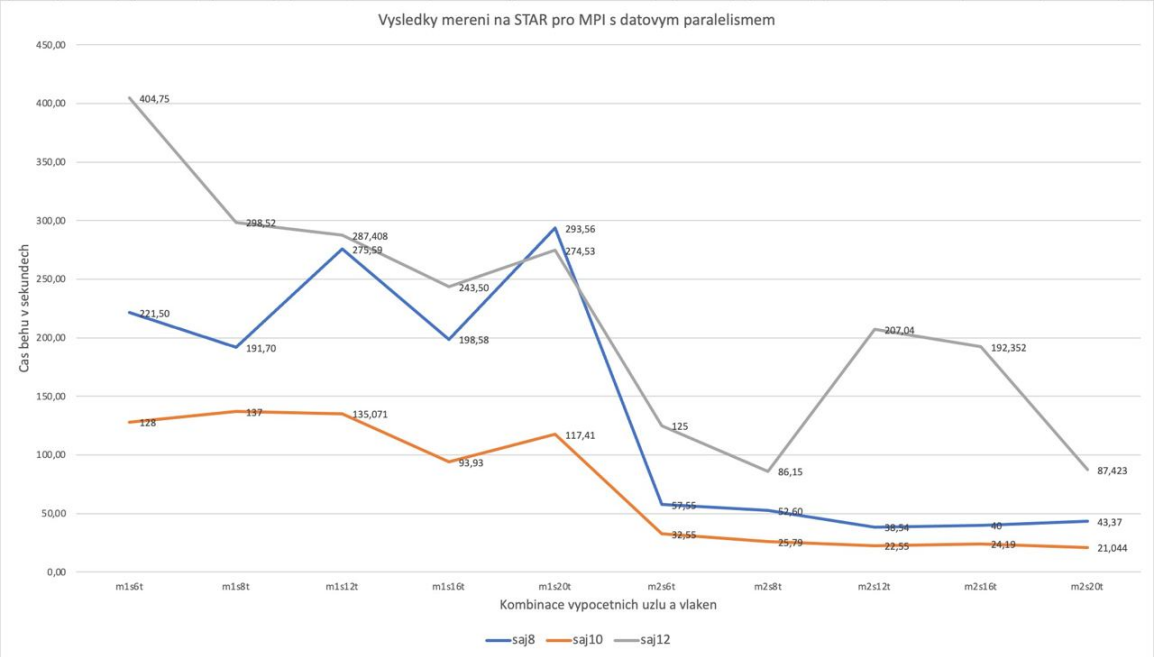
Obsahuje nejlepsi nalezenou cenu a prislusnou historii pohybu na sachovnici.

5.4 Vysledky STAR

Stejne jako u datoveho paralelismu, na STARu jsem spoustel ulohy: **saj8, saj10, saj12**. Reseni merim na 2 a 3 vypocetnych uzlach, kde kazdy ma k dizpozici az 20 jader. Spoustelo se to pro takovou kombinaci procesu a vlaken:

- **1 master a 1 slave** (2 vypocetni uzly): 6, 8, 12, 16, 20 vlaken (dale se bude oznacovat m1s6/8/12/16/20t)
- **1 master a 2 slavy** (3 vypocetni uzly): 6, 8, 12, 16, 20 vlaken (dale se bude oznacovat m2s6/8/12/16/20t)

Jeste jednou pripomenu, ze master se nezavyba vypocty a pracuje jenom na jednom vlaknu. Takze veskera paralelizace se provadi jenom ve slave procesech. Ted uz se muzeme klidne podivat na vysledky v nasledujicim diagramu.



Na diagramu vidime, ze pro konfigrace 1 master, 1 slave a nejaky pocet vlaken se skoro nedochazi ke zrychleni. Na zacatku i dokonce pozorujeme vysledky horsi nez u datoveho a sekvencniho pristupu. Vubec jinou situaci pozorujeme pro konfiguraci 1 master a 2 slavy. Jiz na pocatenci kofiguraci se 6 vlakny na jednom slavu dochazi ke vyraznemu zrychleni oproti datovemu paralelismu. Pri nejvetsim moznem poctu vlaken, coz je 20 per slave, i dokonce vidime nejlepsi vysledky za celou dobu testovani programu. Z toho plyne, ze zvyseni poctu jader ma primy vliv na rychlost. Jeste jedina vec, ktera stoji za zminku je, ze pro ulohu **saj12** se cas behu zacina vyrazne zhorsovat pri koniguracich **m2s12t** a **m2s16t** oproti predchozim konfiguracim. Avsak pri konfiguraci **m2s20t** pro vstup **saj12** byl ziskan nejlepsi cas behu.

Ted porovname nejlepsi vysledky, coz je kofigurace **m2s20t**, se sekvencnimi vysledky a zjistime jestli skutecne doslo ke zrychleni a o kolik.

MPI OpenMP Data Vs Sekvencni reseni			
Instance	MPI m2s20t cas [s]	Zrychleni (proti sekvencnimu)	Sekvencni cas [s]
saj8	43.37	**2.8x**	123.472
saj10	21.044	**3.4x**	71.627
saj12	87,423	**2.7x**	238.843

Vidime, ze nejvetsi prirust ryhlosti je pro ukol **saj10** a je o **3.4x** rychlejsi oproti sekvencnimu.

Ted stejne tak jeste porovname s vysledky pro datovy paralelismus.

MPI OpenMP Data Vs OpenMP Data			
Instance	MPI m2s20t cas [s]	Zrychleni (proti OpenMP Data)	OpenMP data 16t cas [s]
saj8	43.37	**1.8x**	76.42
saj10	21.044	**2.6x**	55.48
saj12	87.423	**1.3x**	113.36

Pri porovnani s OpenMP data, taky vidime, ze nejvetsi prirust ryhlosti je pro ukol **saj10** a je o **2.6x**.

6 Vyhodnoceni

Nejlepsi casy vypoctu pro jednotliva reseni a vstupy			
Instance	Sekvencni [s]	OpenMP Data [s]	MPI + OpenMP Data [s]
saj8	123.472	76.42	43.37
saj10	71.627	55.48	21.044
saj12	238.843	113.36	87.423

Testovani se ukazalo, ze program skaluje, ale obcas vubec ne idealne. Hodne veci by se dalo zlepsit. Minimalne co ted na me napada, co by slo zefektivnit, tak v **MPI** implementaci taky pouzivat mastera pri vypoctech, aby skutecne slo pouzit vsech 60 dostupnych vlaken (20 vlaken per proces). Taky by slo predelat propagaci lepsiho nalezeneho reseni od masteru ke slavum kdykoliv kdyz master ziska nove lepsi reseni. Ted v moji implementaci plati, ze master posila nove lepsi reseni slavu jen spolu s novym stavem na zpracovani. Pridanim techto zmen by urcite slo zrychlit beh programu. Dale by bylo vhodne komunikovat i mezi slavy navzajem: sdilet nejlepsi reseni a mit moznost orezavat prostor jeste efektivnejsi.

7 Zaver

Celkove bych semestralni praci hodnotil kladne. Da se rict, ze vysledky splnily ocekovani. Pro zadany problem jsem implementoval sekvencni reseni, reseni task paralelismem, reseni

datovym paralelismem a pomoci MPI. Programovani nebylo az tak narocne. Nejvic casu jsem stravil pri implementaci MPI casti, ale to byla pro me asi nejzaimavejsi cast. Seznamil jsem se s vypocetnim clusterem STAR, kde jsem porovnaval casy behu jednotlivych instance.

Jak jsem jiz zminoval drive, v moji implementaci by slo nektere veci zlepsit, co by vedlo na zrychleni celeho programu. Tohle jsem poradne rozepsal v predchozi sekce. Prijde mi, ze je velmi zajimova myslenka, ze obcas muzeme se vyhnout zlepsovani algoritmu, pripadne hardwaru a staci jen vyuzit nevyuzitych jader v nasich procesorech, aby program bezel o mnoho rychleji. Samozrejme, ze nevzdy takovy pristup dava smysl a je funkcni, obcas i dokonce muze prinest hromadu dalsich problem. Myslim si, ze znalosti ziskane z tohoto predmetu budou pro me uzitecne v me dalsi pracovni kariere.