

Semestrální projekt MI-PDP 2020/2021:

Paralelní algoritmus pro řešení problému prohledávání stavového prostoru

Martin Šafránek

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

zdrojové kódy:

<https://github.com/TaIos/ni-pdp-semesterka>

11. května 2021

1 Definice problému a popis sekvenčního algoritmu

Program řeší problém nalezení optimální posloupnosti tahů pro střelce a jezdce, která vede k sebrání všech pěšců rozmístěných na šachovnici. Jedná se o analogii problému obchodního cestujícího. Nalezení optimálního řešení je proto NP těžký úkol. Řešení v této práci používá bruteforce s heuristikami pro ořezávání stavového prostoru.

1.1 Popis vstupu

Příklad vstupu je uveden na obrázku 1. Obsahuje popořadě vždy

1. přirozené číslo k , reprezentující délku strany šachovnice S o velikosti $k \times k$,
2. horní mez délky optimální posloupnosti d_{max}^* ,
3. pole souřadnic rozmístěných figurek na šachovnici S .

```
11
22
-----
-----P---
P-----P--P-
---PJS--P--
----P-----
---P--P----
-----P----
-----P----
-----P----
-----PP----
-----
```

Obrázek 1: Příklad vstupních dat pro $k = 11$, $d_{max}^* = 22$. Střelec je označen S, jezdec J, pěšák P a prázdné políčko -.

1.2 Heuristiky

Sekvenční algoritmus používá dvě heuristiky pro pohyb střelce a koně.

Heuristika střelec. Z množiny možných políček, kam je možné střelce přemístit jsou preferována ty, která obsahují pěšce. Pokud takové políčko neexistuje, jsou preferována políčka s alespoň jedním pěšákem na diagonále. Jinak se pohyb střelce rozhodne náhodně.

Heuristika kůň. Z množiny možných políček, kam je možné koně přemístit jsou preferována ty, která obsahují pěšce. Pokud takové políčko neexistuje, jsou preferována políčka, z kterých kůň ve svém následujícím tahu může vzít pěšáka. Pokud ani takové políčko neexistuje, jsou preferována políčka, z kterých kůň v následujících dvou tazích může vzít pěšáka. Jinak se pohyb koně rozhodne náhodně.

1.3 Pseudokód

```
1 Function existujeLepšíŘešení(šachovnice S, nejlepší řešení S*):
2   if počet pěšáků S + počet tahů S ≥ délka S*
3     or počet tahů S + počet pěšáků S > maximální hloubka
4     or S* má minimální možný počet tahů then
5     | return True
6   else
7     | return False
8   end
9
10 Function sequence(šachovnice S, nejlepší řešení S*):
11   if existujeLepšíŘešení(S, S*) then
12     | return
13   end
14
15   if počet pěšáků S je 0 then
16     | aktualizujNejlepšíŘešení(S, S*)
17     | return
18   end
19
20   Tahy ← prázdný list
21   if je na tahu kůň then
22     | Tahy = HeuristikaKůň(S)
23   end
24
25   if je na tahu střelec then
26     | Tahy = HeuristikaStřelec(S)
27   end
28
29   forall t in Tahy do
30     | Snext = ProvedTah(S, t)
31     | sequence(Snext, S*)
32   end
33
```

Algoritmus 1: sekvenční

2 Popis paralelního algoritmu a jeho implementace v OpenMP - taskový paralelismus

Taskový paralelní algoritmus je naimplementován pomocí OpenMP. Hlavní rozdíl oproti sekvenčnímu algoritmu popsanému v je rozdělení úlohy prohledávání stavového prostoru na tasky. Task je základní jednotka, kterou je OpenMP schopno přidělit vláknu a provést tak výpočet. Pro zadanou úlohu task znamená šachovnici s pozicí všech figurek a historii tahů. Takto vytvořené tasky OpenMP přidává do svého taskpoolu, z kterého si je vlákna vyzvedávají a řeší. Dále všechny vlákna řešící tasky z taskpoolu sdílejí nejlepší řešení d_{best} . Heuristiky jsou totožné jako v podsekcí 1.1.

2.1 Konstanty a parametry pro škálování algoritmu

Taskový paralelní algoritmus implementovaný pomocí OpenMP umožňuje nastavení konstant, které ovlivní logiku funkce programu a tedy i výpočetní čas. Změněny byly pouze zde zmíněné konstanty. Jejich hodnota byla určena empiricky na vstupních datech. Nejedná se o optimální hodnoty, protože jejich nalezení je stejně těžký problém jako nalezení optimální cesty v původním problému.

Konstanta TASK_THRESHOLD. Pokud vlákno řeší instanci a délka její cesty je delší než TASK_THRESHOLD, nevytváří další OpenMP tasky a nepřidává je do taskpoolu. Zadanou instanci vyřeší použitím sekvenčního algoritmu

popsaném v sekci .

název	hodnota
TASK_THRESHOLD	6

Tabulka 1: Konstanty použité v OpenMP taskovém paralelismu.

2.2 Pseudokód

```
1 Function openMpTask(šachovnice  $S$ , nejlepší řešení  $S^*$ ):
2   if existujeLepšíŘešení( $S$ ,  $S^*$ ) then
3     | return
4   end
5
6   if počet pěšáků  $S$  je 0 then
7     | if existujeLepšíŘešení( $S$ ,  $S^*$ ) then
8       | #pragma omp critical
9       | if existujeLepšíŘešení( $S$ ,  $S^*$ ) then
10        | aktualizujNejlepšíŘešení( $S$ ,  $S^*$ )
11        | end
12      | end
13    end
14
15    return
16  end
17
18  Tahy  $\leftarrow$  prázdný list
19  if je na tahu kůň then
20    | Tahy = HeuristikaKůň( $S$ )
21  end
22
23  if je na tahu střelec then
24    | Tahy = HeuristikaStřelec( $S$ )
25  end
26
27  forall  $t$  in Tahy do
28    |  $S_{next}$  = ProvedTah( $S$ ,  $t$ )
29    | #pragma omp task firstprivate(...)
30    | openMpTask( $S_{next}$ ,  $S^*$ )
31  end
32
33
```

Algoritmus 2: OpenMP task

3 Popis paralelního algoritmu a jeho implementace v OpenMP - datový paralelismus

Datový paralelismus v OpenMP pracuje s datově nezávislými celky, které podle určené strategie přiděluje vláknům na zpracování. Nezávislý datový celek je pro zadanou úlohu šachovnice s pozicí všech figurek a historií tahů.

První krok je vygenerování datově nezávislých celků – to je provedeno před použitím OpenMP. Ty jsou následně najednou předány OpenMP. To je určenou strategií rozdělí mezi vlákna. Každé vlákno pak provádí sekvenční řešení problému popsané v sekci . Vlákna mezi sebou sdílejí pouze nejlepší řešení.

3.1 Konstanty a parametry pro škálování algoritmu

Prvním krokem před spuštěním OpenMP řešení je vytvoření datově nezávislých instancí. Ty se vytvoří použitím sekvečního algoritmu, viz sekce . Jejich počet je regulován konstantou `EPOCH_CNT`.

Parametrem OpenMP je konstanta `schedule`. Ta určuje politiku přidělování datově nezávislých instancí vláknům. Zde je použita hodnota `dynamic()` bez parametrů. To znamená, že pokud vlákno dokončí výpočet je mu přiřazena jedna další datově nezávislá instance k vyřešení.

Konstanta `EPOCH_CNT`. Určuje, kolik datově nezávislých instancí je vygenerováno. Pro každou epochu jsou provedeny všechny možné tahy buď koněm, nebo střelcem. Po vyčerpání všech epoch jsou stavy, do kterých se kůň a střelec dostali použity jako nezávislé instance.

Konstanta `schedule`. OpenMP konstanta, která nastavuje politiku přidělování datově nezávislých instancí vláknům.

název	hodnota
<code>EPOCH_CNT</code>	3
<code>schedule</code>	<code>dynamic()</code>

Tabulka 2: Konstanty použité v OpenMP datovém paralelismu

3.2 Pseudokód paralelního algoritmu — datový paralelismus

Porchetta andouille flank kielbasa. Tail biltong turducken porchetta burgdoggen ground round shoulder ham, hamburger bacon shankle landjaeger fatback pork belly doner. Sirloin doner venison shankle cow, hamburger flank sausage pork belly. Tenderloin venison pancetta corned beef tongue cow pork belly capicola ball tip salami short ribs. Sirloin rump andouille tail shank fatback bresaola.

Leberkas ham bacon, pastrami turducken pork belly cupim salami kielbasa doner. Turkey cupim meatball capicola jowl cow shank chicken drumstick kevin salami swine pork belly. Drumstick leberkas corned beef beef short loin boudin. Turkey strip steak bacon, ball tip sirloin pork loin pork.

3.3 Pseudokód

```
1 Function openMpData(šachovnice  $S$ , nejlepší řešení  $S^*$ ):
2   Instance  $\leftarrow$  vygenerujInstance( $S$ )
3   #pragma omp parallel for ...
4   forall  $S_{gen}$  in Instance do
5     | sequenceAlgoritmusSOmpCriticalProUpdate( $S_{gen}$ ,  $S^*$ )
6   end
7
8
```

Algoritmus 3: OpenMP data

4 Popis paralelního algoritmu a jeho implementace v MPI

Řešení s použitím MPI se skládá ze dvou částí. První je datový OpenMP paralelismus, viz sekce 3.3. Druhou část tvoří MPI. Ten má za úkol řídit a distribuovat výpočet na několika výpočetních uzlech.

MPI proces začíná tím, že si master vlákno identickým způsobem jako v sekci s datovým paralelismem³³ vygeneruje datově nezávislé instance. Ty pak serializuje a společně s globálním nejlepším řešením je pošle přes MPI interface slavům. Každý z nich pomocí datového paralelismu popsaného v sekci 3.3 vyřeší přijaté řešení a odešle ho zpět master vláknu. Pak požádá master vlákno o další instanci k vyřešení. Pokud master vlákno dojdou instance k vyřešení, rozešle slavům zprávu o ukončení výpočtu.

4.1 Konstanty a parametry pro škálování algoritmu

Protože MPI využívá pro řešení tasků datový OpenMP paralelismus popsaný v sekci 3.3, jsou zde uvedeny pouze MPI konstanty.

TODO nastavení počtu vypocetnich jader

4.2 Pseudokód

```
1 Function mainMPI(šachovnice S, nejlepší řešení S*):
2   if master vlákno then
3     Instance  $\leftarrow$  vygenerujInstance(S)
4     for  $i \leftarrow 1 \dots$  počet slave procesů do
5       | MPI_Send(slavei, Instance[i], S*)
6     end
7
8     alive  $\leftarrow$  počet slave procesů
9     while True do
10      | if MPI_Iprobe(...) then
11      |   Sslave, idslave  $\leftarrow$  MPI_Recv(...)
12      |   if not existujeLepšíŘešení(Sslave, S*) then
13      |     | aktualizujNejlepšíŘešení(Sslave, S*)
14      |   end
15
16      | if zvýbá nějaká nevyřešená instance S' v Instance then
17      |   | MPI_Send(idslave, S', S*)
18      | else
19      |   | MPI_Send(idslave, ukončující token)
20      |   | alive  $\leftarrow$  alive - 1
21      | end
22
23      | if alive = 0 then
24      |   | break
25      | end
26
27    end
28
29  end
30
31  else if slave vlákno then
32    while True do
33      | if MPI_Iprobe(...) then
34      |   | S', S*, token  $\leftarrow$  MPI_Recv(...)
35      |   | if token = práce then
36      |   |   | Sslave  $\leftarrow$  openMpData(S', S*)
37      |   |   | MPI_Send(idmaster, Sslave)
38      |   | else if token = konec then
39      |   |   | break
40      |   | end
41      | end
42
43    end
44
45  end
46
47 end
48
```

Algoritmus 4: MPI

5 Naměřené výsledky a vyhodnocení

1. Zvolte tři instance problému s takovou velikostí vstupních dat, pro které má sekvencní algoritmus časovou složitost mezi 1 a 10 minutami. Pro měření času potřebný na čtení dat z disku a uložení na disk neuvazujte a zakomentujte ladici tisky, logy, zprávy a výstupy.

instance	sekvenční	OpenMP task	OpenMP data	MPI
saj7	272			
saj10	103			
saj12	489			

Tabulka 3: Naměřené výsledky. Jednotky jsou uvedené v sekundách. Konstanta p je počet použitých výpočetních jader.

2. Merte paralelní čas při použití $i = 2, \dots, 60$ výpočetních jader.
3. Tabulková a případně graficky zpracované naměřené hodnoty časové složitosti měrných instancí běhu programu s popisem instancí dat. Z naměřených dat sestavte grafy zrychlení $S(n, p)$.
4. Analýza a hodnocení vlastností paralelního programu, zvláště jeho efektivnosti a skalovatelnosti, případně popis zjištěného superlineárního zrychlení.

6 Závěr

Cílem předmětu bylo si na jednoduchém úkolu prohledávání stavového prostoru v šachovnici vyzkoušet metody pro nalezení optimálního řešení. Nejdříve jsem implementoval jednoduché sekvenční řešení. To jsem dále paralelizoval s použitím OpenMP. Přitom jsem se naučil jak s OpenMP zacházet a dva způsoby paralelizace – datová a tasková. Použití OpenMP mi nedělalo větší problémy, protože se stačí pouze zamyslet a chytře do sekvenčního kódu doplnit pár direktiv případně dopsat jednu/dvě funkce. Navíc se pod OpenMP dá kód rozumně debugovat. Větší problém jsem měl s MPI. Pro jeho použití jsem musel doplnit a přepsat značnou část fungujícího OpenMP kódu. Nejvíce práce na MPI mi zabrala serializace/deserializace instancí a řešení deadlocků při posílání.

7 Spuštění a překlad

Skripty ve výpisech 2 a 3 automatizují překlad a spuštění OpenMP a MPI řešení na svazku STAR. Pro sekvenční úlohu jsem skript nevytvářel, protože rychlejší bylo ruční měření a překlad. Kompletní skripty i s konfiguračními soubory jsou volně dostupná na <https://github.com/TaIos/ni-pdp-vysledky-mereni-star>. Pro stručnost uvádím tabulku 4, která zobrazuje pouze příkazy, které skripty používají pro překlad.

řešení	příkaz
sekvenční	<code>g++ --std=c++11 -O3 -funroll-loops</code>
OpenMP	<code>g++ --std=c++11 -O3 -funroll-loops -fopenmp</code>
MPI	<code>mpicxx --std=c++11 -lm -O3 -funroll-loops -fopenmp</code>

Tabulka 4: Kompilační příkazy používané skripty.

```

CPP_PROGRAM_TEMPLATE='main.template.cpp'
RUN_SCRIPT_TEMPLATE='../serial_job.template.sh'
CPP_COMPILE="g++"
CPP_FLAGS="--std=c++11 -O3 -funroll-loops -fopenmp"
QRUN_CMD="qrun2 20c 1 pdp_serial"
DATA_PATH="/home/saframa6/ni-pdp-semesteralka/data"

createDirectory() {
    if [ ! -d ${1} ]
    then
        mkdir -p ${1}
    fi
}

cd ${1:-$(pwd)}
OUT_DIR="out$(find . -mindepth 1 -maxdepth 1 | sed 's/^\./\./g' | grep -P '^out\d*$' | wc -l)"

createDirectory ${OUT_DIR}

INSTANCES=(7 10 12) # saj instance id
PROCNUMS=(1 2 4 6 8 10 16 20) # number of openmp cores
for INSTANCE in ${INSTANCES[*]}
do
    for PROCNUM in ${PROCNUMS[*]}
    do
        WORKDIR=$(realpath "${OUT_DIR}/saj${INSTANCE}-p${PROCNUM}")
        createDirectory ${WORKDIR}

        CPP_PROGRAM=$(realpath "${WORKDIR}/main.cpp")
        EXE_PROGRAM=$(realpath "${WORKDIR}/run.out")
        RUN_SCRIPT=$(realpath "${WORKDIR}/openmp-job-saj${INSTANCE}-p${PROCNUM}.sh")
        STDERR=$(realpath ${WORKDIR}/stderr)
        STDOUT=$(realpath ${WORKDIR}/stdout)
        touch ${STDERR} ${STDOUT}

        echo $WORKDIR
        echo -e "\tCPP program: ${CPP_PROGRAM}"
        echo -e "\tEXE program: ${EXE_PROGRAM}"
        echo -e "\tRUN script: ${RUN_SCRIPT}"

        sed "s/{PROCNUM}/${PROCNUM}/g" ${CPP_PROGRAM_TEMPLATE} > ${CPP_PROGRAM}
        echo -e "\tCOMPILE: ${CPP_COMPILE} ${CPP_FLAGS} ${CPP_PROGRAM} -o ${EXE_PROGRAM}"
        ${CPP_COMPILE} ${CPP_FLAGS} ${CPP_PROGRAM} -o ${EXE_PROGRAM}

        sed "
            s/{EXE_PROGRAM}/${EXE_PROGRAM}|g;
            s/{ARGUMENTS}/${DATA_PATH}/saj${INSTANCE}.txt|g;
            s/{STDOUT}/${STDOUT}|g;
            s/{STDERR}/${STDERR}|g;
            " ${RUN_SCRIPT_TEMPLATE} > ${RUN_SCRIPT}
        echo -e "\tQRUN: ${QRUN_CMD} ${RUN_SCRIPT}"

        ${QRUN_CMD} ${RUN_SCRIPT}
        echo "=====
done
done

echo "DONE"
exit 0

```

Obrázek 2: Bash skript tester-openmp.sh pro spuštění a otestování OpenMP řešení na svazku STAR.


```

CPP_PROGRAM_TEMPLATE='main.template.cpp'
RUN_SCRIPT_TEMPLATE='parallel_job.template.sh'
CPP_COMPILE="mpicxx"
CPP_FLAGS="--std=c++11 -lm -O3 -funroll-loops -fopenmp"
QRUN_CMD_TEMPLATE="qrun2 20c {NODENUM} pdp_long" # pdp_fast/pdp_long
DATA_PATH="/home/saframa6/ni-pdp-sememstralka/data"

createDirectory() {
    if [ ! -d ${1} ]
    then
        mkdir -p ${1}
    fi
}

cd ${1:-$(pwd)}
OUT_DIR="out$(find . -mindepth 1 -maxdepth 1 | sed 's/^\./\.\.\.\./g' | grep -P '^out\d*$' | wc -l)"

createDirectory ${OUT_DIR}

INSTANCES=(7 10 12) # saj instance id
PROCNUMS=(6 8 12 16 20) # number of openmp cores
NODENUMS=(3 4) # total number of MPI nodes
for INSTANCE in ${INSTANCES[*]}
do
    for PROCNUM in ${PROCNUMS[*]}
    do
        for NODENUM in ${NODENUMS[*]}
        do
            WORKDIR=$(realpath "${OUT_DIR}/saj${INSTANCE}-n${NODENUM}-p${PROCNUM}")
            createDirectory ${WORKDIR}

            CPP_PROGRAM=$(realpath "${WORKDIR}/main.cpp")
            EXE_PROGRAM=$(realpath "${WORKDIR}/run.out")
            RUN_SCRIPT=$(realpath "${WORKDIR}/mpi-job-saj${INSTANCE}-n${NODENUM}-p${PROCNUM}.sh")
            STDERR=$(realpath ${WORKDIR}/stderr)
            STDOUT=$(realpath ${WORKDIR}/stdout)
            touch ${STDERR} ${STDOUT}

            echo $WORKDIR
            echo -e "\tCPP program: ${CPP_PROGRAM}"
            echo -e "\tEXE program: ${EXE_PROGRAM}"
            echo -e "\tRUN script: ${RUN_SCRIPT}"

            QRUN_CMD=$(sed "s/{NODENUM}/{NODENUM}/g" <<< ${QRUN_CMD_TEMPLATE})

            sed "s/{PROCNUM}/${PROCNUM}/g" ${CPP_PROGRAM_TEMPLATE} > ${CPP_PROGRAM}

            echo -e "\tCOMPILE: ${CPP_COMPILE} ${CPP_FLAGS} ${CPP_PROGRAM} -o ${EXE_PROGRAM}"
            ${CPP_COMPILE} ${CPP_FLAGS} ${CPP_PROGRAM} -o ${EXE_PROGRAM}

            sed "
                s|{EXE_PROGRAM}|$EXE_PROGRAM|g;
                s|{ARGUMENTS}|$DATA_PATH/saj$INSTANCE.txt|g;
                s|{STDOUT}|$STDOUT|g;
                s|{STDERR}|$STDERR|g;
                " ${RUN_SCRIPT_TEMPLATE} > ${RUN_SCRIPT}
            echo -e "\tQRUN: ${QRUN_CMD} ${RUN_SCRIPT}"

            ${QRUN_CMD} ${RUN_SCRIPT}
            echo "=====
done
done
done

echo "DONE"
exit 0

```

8 Literatura

Tenderloin pork belly ham leberkas doner rump. Filet mignon beef pastrami pork belly drumstick. Beef ribs filet mignon porchetta pork turducken spare ribs tri-tip corned beef strip steak turkey capicola. Venison hamburger ball tip, buffalo fatback pork alcatra doner pork belly.