Semestrální projekt MI-PDP 2020/2021:

Paralelní algoritmus pro řesení problému prohledávání stavového prostoru

Martin Šafránek

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

zdrojové kódy:
`https://github.com/TaIos/ni-pdp-semestralka`

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

název	hodnota
`EPOCH_CNT`	3
`schedule`	dynamic()

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 ← vygenerujInstance(S)
3 |   #pragma omp parallel for ...
4 |   forall S_gen in Instance do
5 |   |   sequenceAlgorgitmusSOmpCriticalProUpdate(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 33. 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 33 vyřeší přijmuté řešení a odešlě ho zpět master vláknu. Pak požádá master vlákno o další instanci k vyřešení. Pokud master vláknu dojdou instance k vyřešení, rozešlě 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 33, jsou zde uvedeny pouze MPI konstanty.

TODO nastaveni poctu vypocetnich jader

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.

4.2 Pseudokód

```
   input  : k × k pole, mez d*_max
   output: optimální posloupnost tahů
 1 if abc or def then
 2 else
 3 |  ;
 4 end
 5 while While condition do
 6 |  instructions;
 7 |  if condition then
 8 |  |  instructions1;
 9 |  |  instructions2;
10 |  else
11 |  |  instructions3;
12 |  end
13 end
```

Algoritmus 4: MPI paralelismus

5 Naměřené výsledky a vyhodnocení

1. Zvolte tri instance problemu s takovou velikosti vstupnich dat, pro ktere ma sekvencni algoritmus casovou slozitost mezi 1 a 10 minutami. Pro mereni cas potrebny na cteni dat z disku a ulozeni na disk neuvazujte a zakomentujte ladici tisky, logy, zpravy a vystupy.

2. Merte paralelni cas pri pouziti $i = 2, \cdot, 60$ vypocetnich jader.

3. Tabulkova a pripadne graficky zpracovane namerene hodnoty casove slozitosti měernych instanci behu programu s popisem instanci dat. Z namerenych dat sestavte grafy zrychleni $S(n, p)$.

4. Analyza a hodnoceni vlastnosti paralelniho programu, zvlaste jeho efektivnosti a skalovatelnosti, pripadne popis zjisteneho superlinearniho zrychleni.

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 sekvenčného 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	`g++ --std=c++11 -O3 -funroll-loops`
OpenMP	`g++ --std=c++11 -O3 -funroll-loops -fopenmp`
MPI	`mpicxx --std=c++11 -lm -O3 -funroll-loops -fopenmp`

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-semestralka/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.

```bash
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-semestralka/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
```


Obrázek 3: Bash skript tester-mpi.sh pro spuštění a otestování MPI řešení na svazku STAR.

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.