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`

výsledky měření:
`https://github.com/TaIos/ni-pdp-vysledky-mereni-star`

12. května 2021

1 Definice problému a popis sekvenčního algoritmu

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

1.1 Popis vstupu a výstupu

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.

Výstup obsahuje optimální posloupnost tahů pro sebrání všech pěšců na šachovnici. Příklad výsputu je na obrázku 2.

```
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 `-`.

```
4,4 *
2,1
3,3
1,3 *
4,2 *
```

Obrázek 2: Příklad výstupu. Každá řádka obsahuje souřadnice na šachovnici, kam se v daném tahu kůň nebo střelec přesunul. Hvězdička značí, že při daném tahu byl sebrán pěšec.

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 Naměřené sekvenčí časy

instance	**doba běhu** [m]
saj7	1.727
saj10	8.163
saj12	4.542

Tabulka 1: Sekvenční algoritmus – doba běhu v minutách.

1.4 Pseudokód sekvenčního algoritmu

```
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 sekci 1 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 pomocí měření. 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 1.

název	hodnota
`TASK_THRESHOLD`	4

Tabulka 2: 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
13         end
14
15         return
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         if počet tahů S > TASK_THRESHOLD then
30             openMpTask(S_next, S*)
31         else
32             #pragma omp task firstprivate(...)
33             openMpTask(S_next, S*)
34         end
35
36     end
37
38
```

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 1. 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 mírně upraveného sekvečního algoritmu 1. 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 3: Konstanty použité v OpenMP datovém paralelismu.

3.2 Pseudokód paralelního algoritmu — datový paralelismus

Datový paralelismus používá sekvenční algoritmus 1. Přidává navíc synchronizaci mezi vlákny pro přepisování nejlepšího řešení S^*. Ta je identická jako u taskového OpenMP algoritmu 2.

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 38. Druhou část tvoří MPI. To má za úkol řídit a distribuovat výpočet na několika výpočetních uzlech.

MPI řešení začíná tím, že si master proces identickým způsobem jako v sekci s datovým paralelismem vygeneruje datově nezávislé instance. Ty pak serializuje a společně s globálním nejlepším řešením je pošle přes MPI interface slave procesům. Každý z nich pomocí datového paralelismu vyřeší přijmuté řešení a odešlě ho zpět master procesu. Pak požádá master proces o další instanci k vyřešení. Pokud master vláknu nemá už žádné instance k vyřešení, pošle slave procesu ukončující zprávu. Více viz algoritmus 4.

4.1 Konstanty a parametry pro škálování algoritmu

Použité konstanty jsou shodné jako v sekci 38, protože MPI využívá pro řešení instancí OpenMP datový paralelismus. Pro MPI nebylo potřeba nastavovat žádné konstanty, které ovlivní rychlost nalezení řešení.

4.2 Pseudokód

```
1  Function mainMPI(šachovnice S, nejlepší řešení S*):
2      if master vlákno then
3          Instance ← vygenerujInstance(S)
4          for i ← 1 ... počet slave procesů do
5              MPI_Send(slave_i, Instance[i], S*)
6          end
7
8          alive ← počet slave procesů
9          while True do
10             if MPI_Iprobe(...) then
11                 S_slave, id_slave ← MPI_Recv(...)
12                 if not existujeLepšíŘešení(S_slave, S*) then
13                     aktualizujNejlepšíŘešení(S_slave, S*)
14                 end
15
16                 if zvýbá nějaká nevyřešená instance S' v Instance then
17                     MPI_Send(id_slave, S', S*)
18                 else
19                     MPI_Send(id_slave, ukončující token)
20                     alive ← 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 ← MPI_Recv(...)
35                 if token = práce then
36                     S_slave ← openMpData(S', S*)
37                     MPI_Send(id_master, S_slave)
38                 else if token = konec then
39                     break
40                 end
41
42             end
43
44         end
45
46     end
47
48
```

Algoritmus 4: MPI

5 Vyhodnocení naměřených výsledků

Všechny tabulky s naměřenými daty jsou z důvodu jejich velikosti v příloze 8. Zde je jejich diskuze a vyhodnocení. Měření probíhala v době, kdy na svazku STAR bylo ve frontě průměrně 100 naplánovaných běhů.

Zrychlení. Z tabulek 8 je patrné, že v několika případech nastalo superlineární zrychlení. To se projevilo nejvýrazněji u OpenMP task. Důvodem může být plánovač, který OpenMP tasky z poolu přiřazuje vláknům. Přiřazování se s největší pravděpodobností děje nedeterministicky. Může tak nastat, že oproti sekvenčnímu řešení, které je vždy deterministické, dojde k výraznému prořezání stavového prostoru hned při prvních vyřešených istancí. Naopak horší než lineární zrychlení dosáhly všechny MPI implementace. Důvodem je pravděpodobně moje implementace, kdy slave proces komunikuje a tedy i updatuje nejlepší řešení pouze pokud kompletně prohledal přidělený stavový prostor.

Efektivita. Efektivita výpočtu v tabulkách 9 kopíruje anomálii superlineárního zrychlení – u OpenMP je efektivita větší než 1, což by nemělo nastávat. Důvodem je pravděpodobně zase datová závislost úlohy. Naopak u MPI je efektivita velice nízká a důvod kopíruje zrychlení – implementace komunikace mezi MPI procesy.

Škálovatelnost. Ideální škálovatelnost úlohy by byla taková, že při zvyšování výpočetního výkonu klesá výpočetní čas. U OpenMP task, viz tabulka 5, toto lze pozorovat se zvyšováním počtu výpočetních jader. Zajímavá je ale instance `saj10`, kde pro 16, 20 jader výkon klesl. Toto by mohlo být způsobeno zvýšenou režií na přepínání kontextu. Stejná chování ale nezle pozorovat pro další instance. Pokud bych měl čas, udělal bych více měření a způměroval bych výsledky. Z jednoho měření v tomto případě nic vyvodit nedokážu. Škálovatelnost u OpenMP data, viz tabulka 6, se podobá taskové – bylo by potřeba více měření. Škálovatelnost MPI, viz tabulka 7, je prokazatelná. Pro zvyšující se počet výpočetních jader a node se snižuje čas. Nejedná se ale o výrazný rozdíl.

6 Spuštění a překlad

Pro automatické testování OpenMP a MPI jsem vytvořil dva `bash` skripty. Každý z nich upraví zdrojový kód řešení podle potřeby a překompiluje ho. Dále vygeneruje qrun skript, který spustí na svazku STAR. Výsledky uloží na předem definované místo. Pro zjednodušení uvádím tabulku 4, která ukazuje kompilační příkazy a přepínače použité jednotlivými skripty.

Všechny výsledky testování pro OpenMP implementaci byly vygenerovány jedním spuštěním skriptu tester-openmp.sh. Obdobně pro MPI skriptem tester-mpi.sh. Pro sekvenční řešení stačilo ruční spouštění.

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 automatizačními skripty.

7 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.

8 Literatura

Pro vypracování tohoto reportu jsem vycházel z vlastních měření na výpočetním svazku STAR dostupného na adrese `star.fit.cvut.cz` a materiálů předmětu NI-PDP dostupných na `https://courses.fit.cvut.cz/`.

A Naměřené výsledky

instance	**#jader**	**doba běhu** [m]
saj7	1	0.251
	2	0.311
	4	0.017
	6	0.075
	8	0.047
	10	0.019
	16	0.004
	20	0.006
saj10	1	2.199
	2	3.321
	4	2.077
	6	1.695
	8	1.428
	10	1.078
	16	1.439
	20	1.624
saj12	1	9.555
	2	∞
	4	6.303
	6	8.426
	8	8.113
	10	5.946
	16	∞
	20	5.654

Tabulka 5: OpenMP task algoritmus – doba běhu v minutách.

instance	**#jader**	**doba běhu** [m]
saj7	1	5.207
	2	4.935
	4	2.283
	6	0.944
	8	0.775
	10	0.005
	16	0.007
	20	0.004
saj10	1	2.032
	2	2.948
	4	2.133
	6	1.817
	8	1.667
	10	1.642
	16	1.686
	20	1.368
saj12	1	9.610
	2	8.232
	4	6.927
	6	5.880
	8	6.409
	10	6.695
	16	4.788
	20	8.986

Tabulka 6: OpenMP data algoritmus – doba běhu v minutách.

instance	#jader	#node	doba běhu [m]
saj7	6	3	1.891
	6	4	1.630
	8	3	1.518
	8	4	1.318
	12	3	1.316
	12	4	1.210
	16	3	1.294
	16	4	1.141
	20	3	1.269
	20	4	1.210
saj10	6	3	0.912
	6	4	0.583
	8	3	0.787
	8	4	0.542
	12	3	0.789
	12	4	0.502
	16	3	0.713
	16	4	0.520
	20	3	0.746
	20	4	0.521
saj12	6	3	4.111
	6	4	3.578
	8	3	3.409
	8	4	3.054
	12	3	3.267
	12	4	2.638
	16	3	3.021
	16	4	2.783
	20	3	2.703
	20	4	2.849

Tabulka 7: MPI algoritmus – doba běhu v minutách.

instance / p	1	2	4	6	8	10	16	20
saj7	18.094	14.589	259.065	60.309	96.200	227.303	1098.935	686.489
saj10	0.785	0.520	0.832	1.019	1.210	1.601	1.201	1.064
saj12	0.854	nan	1.295	0.969	1.006	1.373	nan	1.444

(a) OpenMP task – zrychlení

instance / p	p=1	p=2	4	6	8	10	16	20
saj7	0.872	0.920	0.920	0.920	5.857	787.676	644.293	644.293
saj10	0.850	0.586	0.810	0.951	1.036	1.052	1.025	1.263
saj12	0.849	0.849	1.178	1.178	1.178	1.178	1.178	1.178

(b) OpenMP data – zrychlení

instance / (p, n)	(6,3)	(6,4)	(8,3)	(8,4)	12,3)	(12,4)	(16,3)	(16,4)	(20,3)	(20,4)
saj7	2.402	2.787	2.992	3.445	3.450	3.752	3.509	3.978	3.577	3.754
saj10	1.894	2.960	2.193	3.187	2.190	3.440	2.420	3.321	2.313	3.313
saj12	1.986	2.281	2.394	2.673	2.499	3.094	2.702	2.933	3.020	2.865

(c) MPI – zrychlení

Tabulka 8: Naměřené zrychlení. Počet výpočetních vláken je označen jako p, počet uzlů jako n (pouze u MPI). Barevně zvýrazněné je superlineární zrychlení.

instance / p	1	2	4	6	8	10	16	20
saj7	18.094	7.294	64.766	10.051	12.025	22.730	68.683	34.324
saj10	0.785	0.260	0.208	0.170	0.151	0.160	0.075	0.053
saj12	0.854	nan	0.324	0.161	0.126	0.137	nan	0.072

(a) OpenMP task – efektivita

instance / p	1	2	4	6	8	10	16	20
saj7	0.872	0.460	0.497	0.802	0.732	78.768	40.268	45.575
saj10	0.850	0.293	0.202	0.158	0.130	0.105	0.064	0.063
saj12	0.849	0.496	0.295	0.231	0.159	0.122	0.107	0.045

(b) OpenMP data – efektivita

instance / (p,n)	(6,3)	(6,4)	(8,3)	(8,4)	12,3)	(12,4)	(16,3)	(16,4)	(20,3)	(20,4)
saj7	0.133	0.116	0.125	0.108	0.096	0.078	0.073	0.062	0.060	0.047
saj10	0.105	0.123	0.091	0.100	0.061	0.072	0.050	0.052	0.039	0.041
saj12	0.110	0.095	0.100	0.084	0.069	0.064	0.056	0.046	0.050	0.036

(c) MPI – efektivita

Tabulka 9: Naměřená efektivita. Počet výpočetních vláken je označen jako p, počet uzlů jako n (pouze u MPI). Barevně zvýrazněná je efektivita větší než 1.