Semestrální projekt MI-PDP 2016/2017:

Paralelní algoritmus pro řešení problému maximálního bipartitiního podgrafu

Martin Melka

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

May 3, 2017

1 Definice problému a popis sekvenčního algoritmu

1.1 Definice problému

Řešení problému maximálního bipartitního podgrafu (MBG) znamená nalézt takový podgraf H daného grafu G, který je bipartitní a jeho počet hran je maximální, tj. neexistuje žádný další bipartitní podgraf grafu G, který by měl větší počet hran než H. Množinu hran grafu H označme F.

Graf je bipartitní právě tehdy, když lze všechny jeho uzly obarvit dvěma barvami.

Jelikož je vstupní graf souvislý, lze určit spodní mez počtu hran řešení. Tou bude $|F| = n - 1$, kde n je počet hran grafu. Toto řešení je dáno tím, že pro každý souvislý graf lze najít kostru, tedy takový podgraf, který je strom. Kostru pak stačí "zakořenit", uzly uspořádat do úrovní, jak je u stromů zvykem, a každou úroveň obarvit jinou barvou než tu předchozí.

1.2 Formát vstupních dat

Formát vstupních dat je daný zadáním:

- Na prvním řádku je číslo n, určující počet hran grafu G
- Na následujících n řádcích je vždy n číslic 0 nebo 1, které reprezentují matici sousednosti grafu – 1 znamená, že odpovídající uzly mezi sebou mají hranu, 0 že nemají.

1.3 Formát výstupních dat

Formát výstupních dat má následující strukturu:

```
Best bipartite graph edge count:
================
==   25 edges ==
================
Bipartite subsets:
0) 0  4  6  7  9  14  1  11  3
1) 2  8  10  12  13  15  5

Edges in bipartite subgraph:
0 <-> 2
```

```
0 <-> 8
(...)


Computation time:
13.0867
```

Kde:

- Ohraničená hodnota udává počet hran nalezeného grafu,
- množiny čísel pod `Bipartite subsets` ukazují, do které ze dvou množin ten který uzel patří (tj. kterými barvami jsou uzly obarveny),
- seznam dvojic čísel udává, mezi kterými uzly jsou ve výsledném grafu hrany,
- a nakonec je uveden čas v sekundách, jež byl potřeba pro dokončení výpočtu.

1.4 Popis sekvenčního algoritmu

Algoritmus řešící problém MBG sekvenčně využívá prohledávání do hloubky, případně do šířky. Detailní popis těchto metod není předmětem tohoto předmětu, jelikož už byly popsány dříve (BI-PA2, BI-ZUM, BI-EFA, BI-GRA).

K reprezentaci uzlů prohledávaného stavového prostoru jsem zvolil matici sousednosti. Z hlediska implementace to znamená to, že každý stav je reprezentován třídou `Graph`, jež si uchovává vlastní kopii dvourozměrného pole – matici sousednosti. Kromě toho má některé pomocné metody, například pro ověření, zda je tento graf bipartitní, spojitý, či nespojitý (metoda vrací hodnoty 1, 0, resp. -1 jako příznak těchto vlastností).

1.4.1 Generování stavů

V každém kroku DFS je nutné vygenerovat potomky řešeného stavu (grafu), které se vloží do zásobníku a budou dále zpracovávat. Algoritmus, kterým jsem se rozhodl tyto sousedy-potomky generovat zajišťuje, že množina vygenerovaných podgrafů bude disjunktní – tedy že žádné dva grafy nebudou totožné, což by v případě jednoduchého odebírání hran nastávalo.

Algoritmus generování funguje tak, že ze zadaného grafu postupně odebírá hrany, čímž generuje podgrafy. Hrany jsou odebírány následujícím způsobem – každý graf má indexy `startI` a `startJ` do matice sousednosti. Ty určují, od které pozice v matici dále budou odebírány hrany (tj. měněny jedičky na nuly). Na předchozí hrany algoritmus nesmí sahat.

Vygenerování množiny potomků stavu s grafem `G` pak znamená:

```
given graph G:

neighbors = []
foreach (i,j)>(startI,startJ):
  if edge(i,j) is present:
    new_graph = G.remove_edge(i,j)
    new_graph.startI = i
```

```
new_graph.startJ = j
neighbors.append(new_graph)
```

Tímto způsobem generuji stavy, které může mít dále smysl prohledávat.

1.4.2 Ořezávání stavového prostoru

Před samotným zpracováním vygenerovaných stavů je vhodné provést kontrolu, zda to má vůbec smysl:

- Pokud je graf bipartitní, nemá smysl prohledávat jeho potomky – budou mít určitě menší počet hran a tento graf bude tedy zaručeně lepší.
- Pokud má graf stejný nebo menší počet hran než dosud nejlepší nalezený graf, nemá smysl ho dále prohledávat.
- Pokud je vygenerovaný graf nespojitý, nemá smysl se s ním dále zabývat - zajímají mě pouze spojité grafy. Test spojitosti provádím v rámci testu bipartity, viz popis reprezentace uzlu výše.
- Pokud má graf méně hran než $|V| - 1$, nemá smysl se s ním zabývat, jelikož toto je spodní mez řešení.

1.5 Naměřené časy

Časy naměřené na vzorových datech na lokálním stroji (frekvence CPU 2.4GHz) jsou následující:

- `graph10_5.txt` – 1020 ms
- `graph17_3.txt` – 10 ms
- `graph20_3.txt` – 120 ms
- `graph25_3.txt` – 130 ms
- `graph14_4.txt` – 1650 ms

2 Popis paralelního algoritmu a jeho implementace v OpenMP

Paralelní algoritmus v OpenMP, tj. řešení se sdílenou pamětí, jsem řešil dvěma způsoby – pomocí task paralelismu a datového paralelismu.

Task paralelismus jsem řešil jednoduše vytvářením nového `OMP Task` při každém rekurzivním volání DFS funkce. Až na toto tvoření tasků a uzamykání aktualizace nejlepšího řešení do kritické sekce, bylo toto řešení totožné se sekvenčním.

Datový paralelismus byl implementačně o něco náročnější než *Task paralelismus*. Pro jeho korektní běh bylo nejprve nutné vygenerovat dostatečně velkou množinu počátečních (disjunktních) stavů, nad kterou poté budou OpenMP vlákna iterovat. Generování počátečních stavů jsem extrahoval do samostatné funkce, která ale vnitřně funguje velmi podobně běžnému BFS algoritmu. Prohledávání probíhá stejně jako u "ostrého" řešení s tím, že ve chvíli, kdy velikost fronty grafů čekajících na zpracování odpovídá zadanému parametru, algoritmus na její konec vloží nedozpracovaný graf (abych nepřišel o žádný ze stavů) a celou frontu vrátí jako výsledek

funkce. Nad touto frontou grafů pak standardně pomocí `#pragma omp parallel for` spustím požadovaný počet vláken.

Tím, že vlákna mají disjunktní množinu počátečních stavů, které jsou na sobě nezávislé, nemusí na sebe nijak čekat (vyjma kritické sekce při nalezení nového maxima).

Počet generovaných stavů jsem empiricky stanovil na 50násobek počtu vláken, ale ideální hodnota tohoto parametru silně závisela na vstupních datech.

2.1 Příkazová řádka

Kompilace a spuštění programu v příkazové řádce se provede následující sekvencí příkazů:

```
$ cd project
$ make clean && make
$ ./solver <input-graph> [seq|omp|mpi] [num_threads]
```

Druhý parametr programu udává:

- **seq** – sekvenční běh.
- **omp** – běh pouze v OpenMP (nepoužívají se volání MPI).
- **mpi** – běh OpenMP + MPI (viz dále).

V případě použití parametrů `omp` nebo `mpi` je ještě nutné udat parametr `num_threads`, který určuje, kolik vláken se má na každém výpočetním uzlu spustit.

3 Popis paralelního algoritmu a jeho implementace v MPI

Paralelní algoritmus v MPI umožňuje počítat distribuovaně na více výpočetních uzlech. Jedná se o algoritmus s distribuovanou pamětí – každý z uzlů má svou vlastní a je třeba řešit jejich synchronizaci.

Výpočet maxima na každém z uzlů probíhá totožně jako v předchozím případě algoritmu OpenMP. Rozdíl je v tom kdy a s jakými parametry se tyto OpenMP výpočty spouštějí. To je řízeno komunikací uzlů v MPI. V této části popíšu způsob, jakým tato komunikace probíhá.

Obecný popis algoritmu je takový, že Master proces na začátku výpočtu vygeneruje počáteční stavy. Slave procesy si je od Mastera postupně odebírají a sami je u sebe řeší v OpenMP. Mastera notifikují o nově nalezených maximech a zároveň jsou od něj informováni o nových maximech, nalezených jinými Slavy. Master sám počítá, pokud nejsou žádné Slave procesy, které by potřebovaly novou práci.

3.1 Master proces

Master proces je proces s MPI rankem 0. Na začátku výpočtu standardním způsobem uvedeným výše vygeneruje počáteční grafy. Empiricky jsem jejich počet stanovil jako 80násobek počtu MPI procesů. V tuto chvíli Slave procesy nic nedělají. Po vygenerování stavů Master přejde do své výpočetní smyčky, která je zjednodušeně pseudokódem zapsána takto:

```
while not done:
  if message_pending:
    if message_is(slave_needs_work):
      if initial_graphs.size() > 0:
        send_work(slave_id, initial_graphs.pop())
      else:
        send_no_more_work(slave_id)
        if all_slaves_received_no_more_work:
          done = true

    elif message_is(slave_finished_computing):
      candidate_graph = receive_result(slave_id)
      if candidate_graph > current_best:
        current_best = candidate_graph

  else:  # No pending messages -> start computing on Master node
    if initial_graphs.size() > 0:
      work_graph = initial_graphs.pop()
      computeOpenMP(work_graph, current_best)
```

Smyčka se ukončí, pokud byly všechny grafy zpracovány a všem Slave procesům odeslána zpráva `no_more_work`. Po jejím skončení je nalezeno nejlepší řešení, protože byl prohledán celý (ořezaný) stavový prostor.

3.2 Slave proces

Slave proces je komplementární k Masteru. Jeho výpočetní smyčka vypadá takto:

```
while not done:
  work = request_work()
  if work is NO_MORE_WORK:
    break  # No more work to be done, quit loop

  my_best = computeOpenMP(work.graph, work.master_best)
  send_result(my_best)
```

Zde je smyčka jednodušší, Slave jen dokola žádá o práci a počítá. Spolu s prací Master vždy pošle i doposud nejlepší řešení, které si Slave uloží a použije pro ořezání stavového prostoru.

3.3 Formát MPI zpráv

Pro komunikaci v MPI používám hlavně jeden typ zprávy - Graf. Ta obsahuje zakódovaný objekt grafu a metadata. Kromě tohoto typu zprávy pak jen zprávy obsahující příznak (`tag`) – to jsou zprávy žádosti o práci a oznámení o tom, že žádná práce již není k dispozici.

Formát zprávy s grafem má tento formát:

- `int[5]`:

1. Počet hran nejlepšího zatím nalezeného řešení (používá se jen při zasílání grafu od Master pro Slave proces)
2. Počet uzlů posílaného grafu
3. `startI`
4. `startJ`
5. Počet hran posílaného grafu (šlo by spočítat z poslané matice, ale tímto se šetří výpočetní čas)

- `bool[n]`, kde `n` je počet uzlů grafu – první řádek matice sousednosti
- `bool[n]` – druhý řádek matice sousednosti
- ...

3.4 Příkazová řádka

Příkazy pro program jsou popsané v sekci výše. Pokud chci program spustit ve více procesech (pro MPI komunikaci), lze použít tuto konstrukci:

```
$ mpirun -n <process_count> ./solver input/graph mpi <threads_per_process>
```

4 Naměřené výsledky a hodnocení

4.1 Vstupní data

Běh algoritmu jsem měřil na třech vstupních grafech, těmi byly:

- `graph14_5` - vzorový vstup, graf o 14 uzlech.
- `custom_14_5_10` - vygenerovaný graf s parametry -n 14 -k 5, graf o 14 uzlech.
- `custom_16_5` - vygenerovaný graf s parametry -n 16 -k 5, graf o 16 uzlech.

Rozdělení výpočetních jader je uvedeno v tabulce 1. Naměřené časy v těchto konfiguracích jsou detailně rozepsány v tabulce 2 a zobrazeny v grafu 1.

4.2 Parametry škálování

Parametry pro škálování výpočtu jsou následující:

- **Počet vygenerovaných grafů na jedno vlákno** (v rámci výpočtu OpenMP) – 50
- **Počet vygenerovaných grafů pro každý MPI proces** – 80

Table 1: Rozdělení výpočetních jader

Počet jader	Počet vláken	Počet MPI procesů
1	1	1
2	2	1
4	4	1
8	4	2
16	4	4
24	4	6
32	8	4
60	10	6

Table 2: Čas výpočtu algoritmu

	Čas výpočtu [s]		
Počet jader	Graf 1	Graf 2	Graf 3
1	423,91	386,853	1309,21
2	403,442	720,561	491,596
4	389,849	732,664	604,743
8	767,365	418,172	>1800
16			
24			
32			
60			

4.3 Výsledky

Z naměřených dat je vidět, že paralelizace s výše uvedenými parametry (počty grafů na vlákno a proces) je neefektivní. Nejen, že je sublineární, v některých případech je dokonce pomalejší, než sekvenční řešení. Nejjednodušším způsobem optimalizace pro 1-4 výpočetní jádra by bylo spuštění programu jen s OpenMP. Algoritmus, který byl použit pro naměření těchto výsledků, vždy běžel v MPI módu – v případě jediného MPI uzlu tedy celý výpočet probíhal jen v Master MPI procesu a v algoritmu se zbytečně volaly MPI metody `MPI_Iprobe`.

Při 8 jádrech jsou již použity dva MPI uzly, každý se 4 vlákny. V grafech 1 a 3 došlo k poměrně znatelnému zhoršení výpočetní doby, v grafu 2 se doba zlepšila, ale stále byla delší než sekvenční řešení.

4.4 Možnosti zlepšení

Jak již jsem zmínil výše, velkou roli v době výpočtu hrály parametry pro počet generovaných počátečních grafů v závislosti na počtu vláken a procesů. Hodnoty použité pro výpočet jsem experimentálně stanovil na základě výpočtů na svém počítači, kde byl MPI paralelismus simulován pouze na jednom CPU se dvěma vlákny.

Pro zlepšení získaných časů by tedy bylo vhodné experimentovat s nastaveními těchto parametrů, ale tyto experimenty není v dobu odevzdávání zprávy možné uskutečnit, protože fronta čekajících výpočtů na clusteru STAR je neúnosně velká. Nezbyl mi tak dostatek času na optimalizaci časů pro dané vstupní grafy.

Samotný algoritmus komunikace mezi MPI procesy rovněž poskytuje prostor pro zlepšení. Poté, co Slave proces dokončí výpočet zadaného grafu, odešle o tom Master procesu informaci. Master by mohl ihned zaslat další práci Slave procesu, ale místo toho čeká, až o ni Slave sám požádá.

Další možností pro experimentování by byla úprava Master procesu tak, aby vůbec nepočítal. Mám podezření, že zejména při vyšším počtu MPI uzlů Slave procesy zbytečně čekají na dokončení výpočtu.

5 Závěr

Algoritmus, který jsem naimplementoval v rámci semestrální práce předmětu MI-PDP je korektní a věřím, že po úpravě komunikačního protokolu a experimentování se škálovacími parametry, což jsem popsal výše, se dostanu na lepší časy a lepší zrychlení.

Figure 1: Graf rychlosti výpočtu

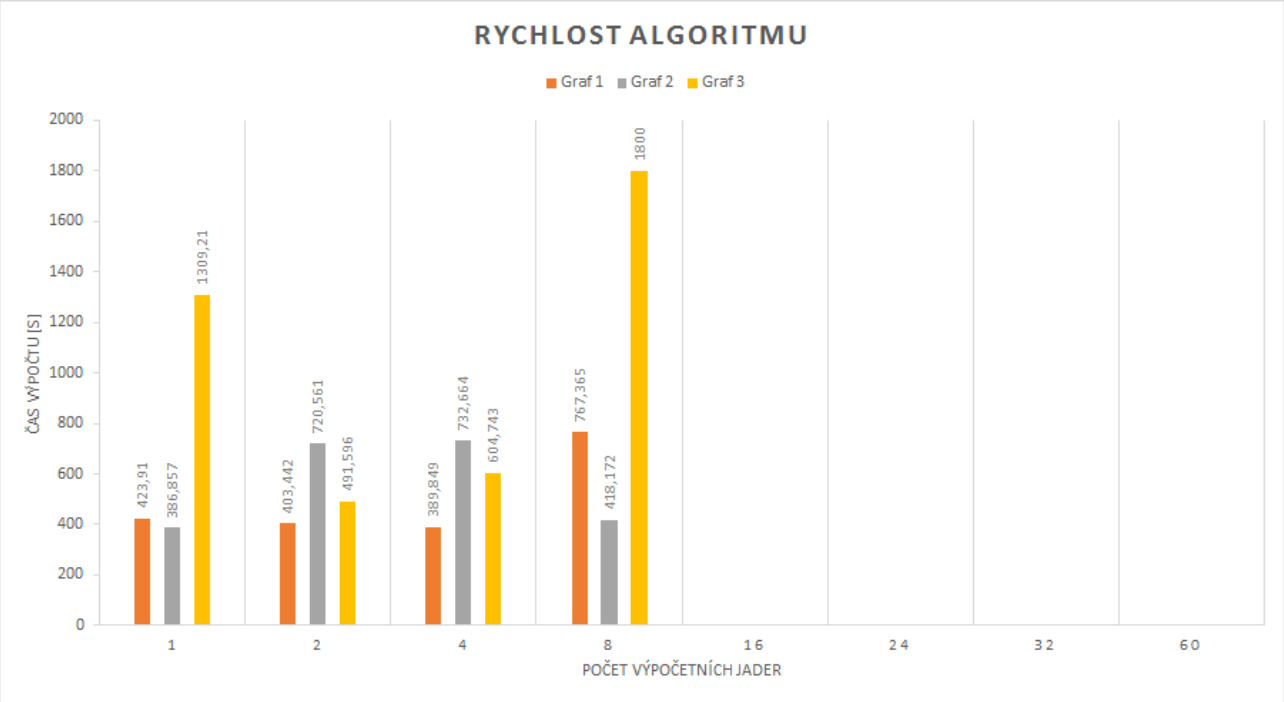

Figure 2: Graf zrychlení výpočtu