Fakulta informačních technologií

Paralelní architektury počítačů

technická zpráva

Hledání nejkratších cest v grafu

Autoři:
Vojtěch Myslivec
Zdeněk Nový

15. dubna 2015

Abstrakt

Účelem této práce je sumarizovat výsledky měření řešení problému hledání nejkratších cest v grafu (NCG). Práce se zaměřuje na řešení problému Dijkstrovým a Floyd-Warshallovým algoritmem a porovnání sekvenční a několika paralelních implementací.

Klíčová slova Dijkstra, Floyd, Warshall, nejkratší cesty, NCG, OpenMP, Cuda

Obsah

1 Úvod

Tato práce se zabývá implementací dvou algoritmů hledání nejkratších cest v grafu. Jedná se o implementaci sekvenčním algoritmem, který je poté paralelizován pro procesor a pro grafickou kartu. Pro jednotlivé algoritmy je

provedeno měření, které si klade za cíl určit zrychlení paralelních algoritmů proti sekvenčnímu.

2 Hledání nejkratších cest v grafu

2.1 Definice

Hledání nejkratších cest v grafu je NP-úplná grafová úloha, jejímž cílem je nalézt v zadaném grafu nejkratší cesty mezi všemi možnými dvojicemi uzlů A a B [6].

2.2 Algoritmy

2.2.1 Dijkstrův algoritmus

Dijkstrův algoritmus slouží k nalezení všech nejkratších cest ze zadaného uzlu do všech ostatních uzlů grafu. Graf nesmí obsahovat hrany se zápornou délkou [4].

Princip Dijkstrův algoritmus je zobecněné prohledávání grafu do šířky, při kterém se vlna šíří na základě vzdálenosti od zdrojového uzlu. K uchovávání uzlů slouží prioritní fronta, která je řazena podle vzrůstající vzdálenosti od zdroje. V každém kroku algoritmu je vybrán uzel s nejmenší vzdáleností a pro každého souseda je vypočítána jeho vzdálenost od zdrojového uzlu [4].

2.2.2 Floyd-Warshallův algoritmus

Floyd-Warshallův algoritmus slouží k nalezení nejkratších cest mezi všemi dvojicemi uzlů v grafu. Graf může obsahovat hrany, ale nikoliv cykly, záporné délky [5].

Princip Floyd-Warhsallův algoritmus pracuje s maticí sousednosti, kde hrana je ohodnocena vahou. Na počátku tato matice obsahuje pouze vzdálenosti dvou uzlů, mezi kterými je vedena hrana. V každém kroku je vybrán jeden uzel jako prostředník. Prvek matice sousednosti se přepočítá, pokud je vzdálenost z počátečního do koncového uzlu kratší přes nového prostředníka než bez něj [5].

3 Sekvenční algoritmus

3.1 Společná implementace

Oba algoritmy vycházejí z obecného principu, který je popsán v kapitole 2.2.2. Algoritmy pracují s grafem, který je programu předložen jako soubor, ve kterém je graf ve formě matice sousednosti. Společnou částí je tedy načítání vstupu a jeho kontrola.

3.2 Dijkstrův algoritmus

Protože Dijkstrův algoritmus slouží k hledání nejkratších cest od jednoho zdrojového uzlu, je nutné jej spouštět pro každý uzel grafu. To zajišťuje funkce *dijkstraNtoN*.

3.2.1 Dijkstrův algoritmus z jednoho zdroje

Pro výpočet Dijkstrova algoritmu z jednoho zdrojového uzlu se alokují tři pole o velikosti počtu uzlů. V jednom je uložena vzdálenost daného uzlu od zdrojového, ve druhém předchozí uzel v nalezené nejkratší cestě. Třetí pole určuje, jestli je už uzel uzavřený pro výpočty.

Algoritmus prochází postupně, podle nejmenší vzdálenosti, všechny uzly, které se nacházejí v prioritní frontě. Z daného uzlu vypočítá pro každého svého souseda novou cestu, která by vedla přes uzel samotný a porovná ji s dosavadní vzdáleností souseda. Menší vzdálenost je zapsána do pole vzdáleností a algoritmus pokračuje.

Prioritní fronta Za účelem prioritní fronty byla implementována binární halda, kde složitost výběru minima je logaritmická, oproti nativní implementaci pomocí pole, kde je složitost výběru minima lineární.

3.3 Floyd-Warshallův algoritmus

Floyd-Warshallův algoritmus obsahuje tři vnořené for cykly a funguje na principu popsaném v 2.2.2. Jako datové struktury používá čtyři pole o velikosti počtu uzlů. Pro každý uzel si algoritmus udržuje aktuální vzdálenosti ke všem uzlům a navíc vzdálenosti z předchozí iterace. Další dvě pole obsahují předchozí uzel v nalezené cestě.

3.4 Implementace

Aktuální implementace je k nahlédnutí i ke stažení na adrese `https://github.com/VojtechMyslivec/PAP-NCG`. Sekvenční algoritmus se nachází ve složce *01_sekvencni*.

4 Paralelní algoritmus pomocí knihovny OpenMP

4.1 Dijkstrův algoritmus

Paralelizace algoritmu spočívá v paralelizaci cyklu, který prochází jednotlivé uzly a pro ně řeší problém hledání nejkratší cesty v grafu z jednoho počátečního uzlu. Každé vlákno tedy zpracovává jeden uzel jako počáteční a z něj hledá nejkratší cesty do všech ostatních uzlů.

4.1.1 Paměťové struktury

Každé vlákno dostane ukazatel na strukturu grafu. Protože všechna vlákna používají strukturu grafu pouze ke čtení, nedochází při přístupu k této struktuře k žádným problémům.

Každé vlákno si vytvoří jeden objekt, ve kterém si alokuje vlastní pole vzdáleností a předchůdců, které používá pro své výpočty. Tyto struktury jsou po ukončení funkce vlákna dealokovány společně s objektem.

4.1.2 Úprava algoritmu

Z důvodu paralelizace algoritmu bylo nutné upravit použitou prioritní frontu. V sekvenčním řešení byla použita implementace pomocí binární haldy 3.2.1. Z důvodu paralelizace výběru minima z fronty je pro paralelní řešení výhodnější použít implementaci polem.

4.1.3 Vektorizace

Pomocí přepínačů optimalizace *-O3* a podpory vektorových sad *-msse4.2* kompilátoru *gcc* byla zapnuta podpora vektorizace cyklů [3]. Pro záznam o pokusech vektorizace byl použit přepínač *-ftree-vectorizer-verbose=n*, kde za *n* byly dosazeny 1, 3, 5, kde čím vyšší číslo, tím podrobnější informace [3].

Obrázek 1: Úspěšně vektorizovaný cyklus.

```
dijkstra.cpp:38: note: LOOP VECTORIZED.
dijkstra.cpp:28: note: vectorized 1 loops in function.

for ( unsigned j = 0; j < pocetUzlu; j++ ) {
        vzdalenostM[i][j] = DIJKSTRA_NEKONECNO;
        predchudceM[i][j] = DIJKSTRA_NEDEFINOVANO;
}

dijkstra.cpp:99: note: not vectorized: number of iterations cannot be co
dijkstra.cpp:99: note: bad loop form.
dijkstra.cpp:82: note: vectorized 0 loops in function.
```

Obrázek 2: Cyklus, který se nepodařilo vektorizovat.

```
dijkstra.cpp:99: note: not vectorized: number of iterations cannot be co
dijkstra.cpp:99: note: bad loop form.
dijkstra.cpp:82: note: vectorized 0 loops in function.

for ( unsigned i~= 0 ; i~< pocetUzlu ; i++ ) {
        vzdalenostM[idUzlu][i] = vzdalenost[i];
        predchudceM[idUzlu][i] = predchudce[i];
}
```

Obrázek 3: Upravený cyklus, aby mohl být vektorizován.

```
Vectorizing loop at dijkstra.cpp:99
dijkstra.cpp:99: note: LOOP VECTORIZED.
dijkstra.cpp:82: note: vectorized 1 loops in function.

unsigned tmp = pocetUzlu;
for ( unsigned i~= 0 ; i~< tmp ; i++ ) {
        vzdalenostM[idUzlu][i] = vzdalenost[i];
        predchudceM[idUzlu][i] = predchudce[i];
}
```

Původní stav Na obrázcích 1 a 2 je znázorněn příklad jednoho cyklu, který je byl vektorizován a druhý cyklus, který se nepodařilo vektorizovat z důvodu nespočitatelného počtu iterací.

Optimalizace Výpis 3 dokazuje úspěšnou úpravu cyklu, který se, díky malé změně zdrojového kódu, podařilo vektorizovat.

4.2 Floyd-Warshallův algoritmus

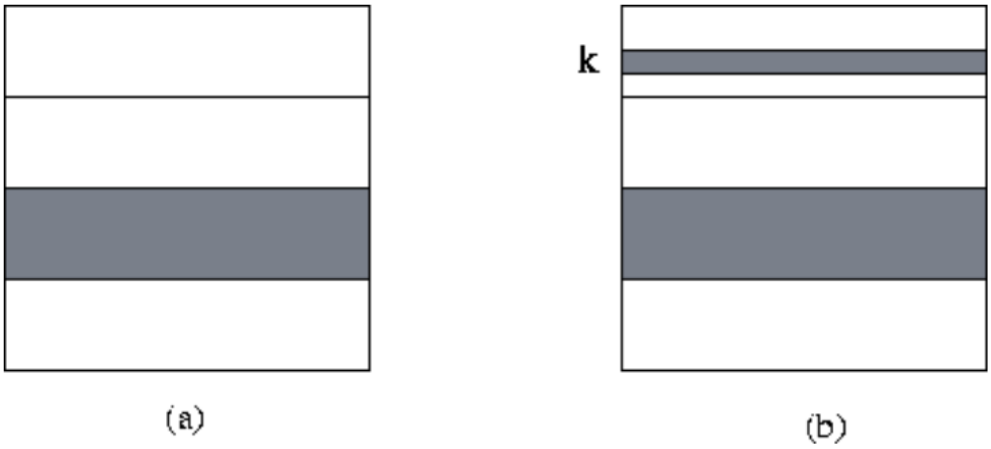
Díky třem vnořeným sekvenčního algoritmu existuje několik možností, jak algoritmus paralelizovat.

4.2.1 První varianta

První variantou je algoritmus paralelizovat pouze v jednom cyklu, který definuje, která řádka je právě zpracovávána. Přidělená data jednomu vláknu zobrazuje obrázek 4. Algoritmus zapisuje pouze do přidělených sloupců, tedy řádků zobrazených v části *a*. Řádek k v části b využívají všechna vlákna pouze ke čtení, proto zde nedochází ke konfliktům. Tato varianta je implementována.

4.2.2 Druhá varianta

Druhou variantou, jak problém paralelizovat je použít původní variantu a přidat paralelizaci zároveň ve vnitřním cyklu, který prochází jednotlivé sloupce matice. V takovém případě by jednomu vláknu byl přidělen jeden nebo více



Obrázek 4: Ukázka dat přidělených jednomu vláknu při paralelizaci jednoho cyklu [1].

necelých řádků ohraničených sloupci. Tato varianta se jeví vhodnější pouze při velkém počtu dostupných vláken, proto není v naší implementaci použita.

4.2.3 Vektorizace

Z výpisu 5 je patrné, že v algoritmu Floyd-Warshall je vektorizován pouze jeden cyklus. Z důvodu jednoduchosti algoritmu a absence jednoduchých *for* cyklů se nepodařilo zvektorizovat žádný další cyklus.

4.3 Měření

Na obou algoritmech bylo provedeno měření, které si klade za cíl analyzovat čas, zrychlení a efektivitu použitého paralelního algoritmu. Měření bylo prováděno na hustých grafech, kde při generování grafů byla použita pravděpodobnost 0.5, že mezi dvěmi uzly existuje hrana.

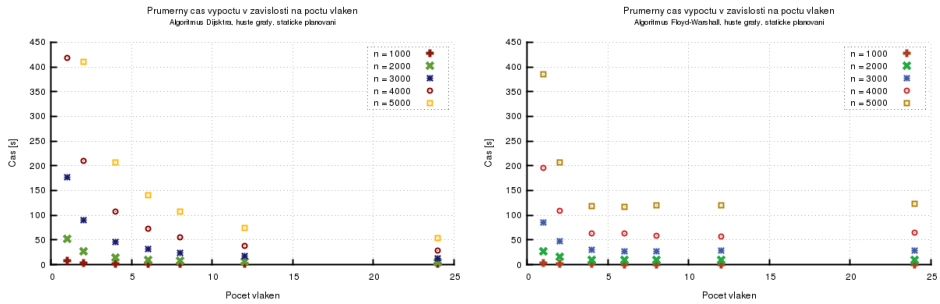
4.3.1 Testovací data

Jako testovací data byly vygenerovány grafy s počtem uzlů 1000, 2000, 3000, 4000, 5000. Měření probíhalo na serveru `star2.fit.cvut.cz` na stroji *gpu-02* pro počet vláken 1, 2, 4, 6, 8, 12, 24.

Obrázek 5: Úspěšně vektorizovaný cyklus.

```
floydWarshall.cpp:90: note: vectorizing stmts using SLP.BASIC BLOCK VEC
floydWarshall.cpp:90: note: basic block vectorized using SLP

for ( unsigned k~= 0; k~< tmp; k++ ) {
        unsigned i;
        #pragma omp parallel for private( i, novaVzdalenost ) shared( d
dchudcePredchozi, predchudceAktualni )
        for ( i~= 0; i~< tmp; i++ ) {
                for ( unsigned j = 0; j < tmp; j++ ) {
```



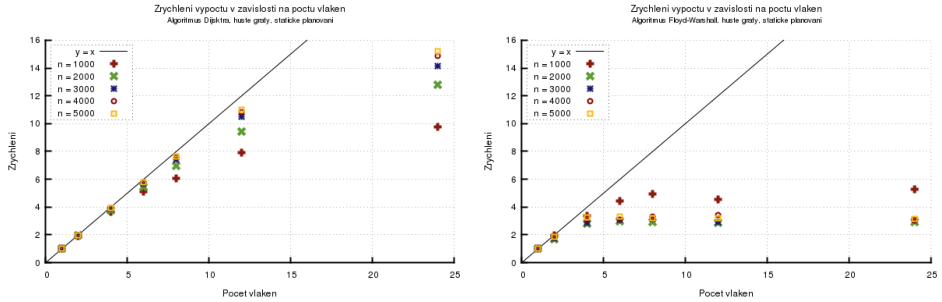
Obrázek 6: Závislost průměrného času výpočtu v závislosti na počtu vláken za použití statického plánování.

4.3.2 Výsledky

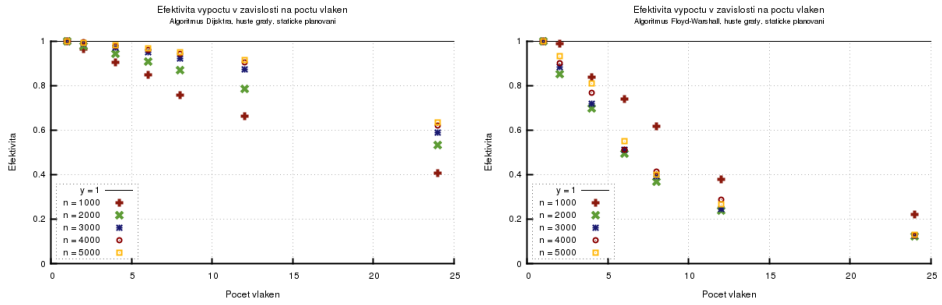
Grafy 6, 7 a 8 ukazují výsledky měření z pohledu času, zrychlení a efektivity.

Poměr rychlosti algoritmů v závislosti na hustotě grafu Graf 9 porovnává výpočetní časy při použití statického plánování v závislosti na hustotě grafu. Z grafů vyplývá, že oba algoritmy jsou téměř nezávislé na hustotě grafu.

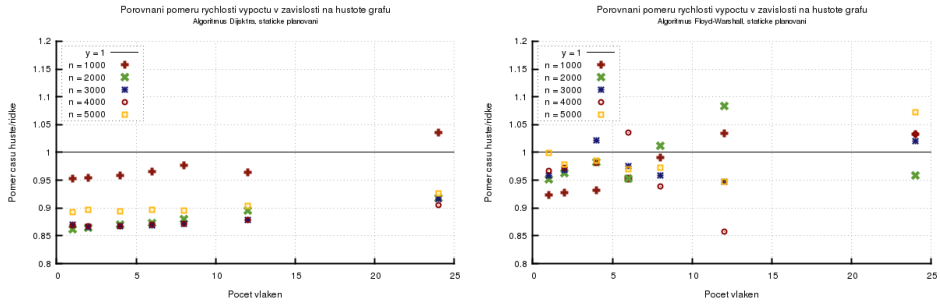
Poměr rychlosti algoritmů v závislosti na plánování Graf 10 zobrazuje poměr výpočetních časů v závislosti na použitém plánování. V porovnání byla použita plánování statické a dynamické. Z grafů je patrné, že plánování ovlivní čas výpočtu pouze nevýznamně a není tedy podstatné, jaké plánování



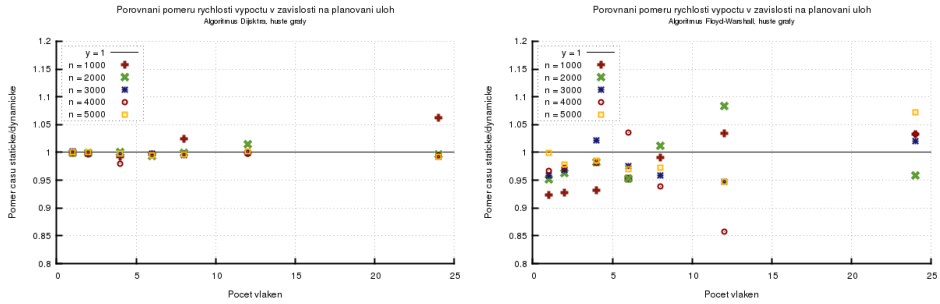
Obrázek 7: Závislost zrychlení paralelního algoritmu oproti sekvenčnímu v závislosti na počtu vláken za použití statického plánování.



Obrázek 8: Závislost efektivity algoritmu v závislosti na počtu vláken za použití statického plánování.



Obrázek 9: Porovnání poměru rychlostí výpočtu v závislosti na hustotě grafu při použití statického plánování.



Obrázek 10: Porovnání poměru rychlostí výpočtu v závislosti na použitém plánování.

je v algoritmu použito.

Statické plánování Při statickém plánování se iterace cyklu rovnoměrně rozdělí po blocích mezi všechna vlákna před začátkem provádění cyklu [2].

Dynamické plánování Dynamickým plánováním se rozumí rozdělení jednotlivých iterací vláknům po jedné iteraci. Vždy když vlákno dokončí iteraci, je mu přidělena další iterace [2].

4.3.3 Analýza

Dijsktra U výpočetního času je z grafů 6 je patrná exponenciální závislost, kdy se čas pro větší počet vláken téměř nezkracuje. Zrychlení, které je zobrazeno v grafu 7 na počátku stoupá téměř lineárně a teprve pro velký počet

vláken se zrychlení zmírňuje a křivka dostává logaritmický tvar. Z výše uvedeného vyplývá efektivita, která je zobrazena v grafu 8.

Floyd-Warshallův algoritmus U Floyd-Warshallova algoritmu se výpočetní čas pro počty vláken větší než 2 téměř nezkracuje. Zrychlení je tedy patrné pouze při použití 2 případně 4 vláknech. Z výše uvedené plyne, že efektivita paralelního algoritmu velmi rychle klesá.

4.3.4 Zhodnocení

Efektivita Dijkstrova algoritmu s přidáváním vláken pomalu klesá a například pro 24 vláken dosahuje hodnoty 0.5. Naproti tomu u Floyd-Warshallova algoritmu klesá efektivita mnohem rychleji a na hodnotě 0.5 se nachází už pro 6 vláken.

Výrazně lépe ve prospěch Dijkstrova paralelního algoritmu vycházejí i ostatní ukazatele – zrychlení a čas výpočtu.

Výsledky Floyd-Warshallova algoritmu mohou být způsobeny opakovaným vytvářením a rušením vláken. V každém vnějším cyklu se vytvoří daný počet vláken, zpracuje jeden uzel a všechna tato vytvořená vlákna se opět ukončí. Tedy za běhu algoritmu se vytváří a ruší *pocet_uzlu * vlaken*, kde parametr *vlaken* je počet najednou vytvářených paralelních vláken.

Reference

[1] Foster, I.: Case Study: Shortest-Path Algorithms. 1995, [cit. 2015-04-13].
URL http://www.mcs.anl.gov/~itf/dbpp/text/node35.html

[2] Šimeček, I.: Technologie OpenMP. 2014, [cit. 2015-04-14].
URL https://edux.fit.cvut.cz/courses/MI-PAP/_media/lectures/omp.pdf

[3] Šimeček, I.; Šoch, M.: Použití vektorizace v C/C++. 2015, [cit. 2015-04-13].
URL https://edux.fit.cvut.cz/courses/MI-PAP/_media/lectures/vektorizace.pdf

[4] Mička, P.: Dijkstrův algoritmus. [cit. 2015-04-08].
URL http://www.algoritmy.net/article/5108/Dijkstruv-algoritmus

[5] Mička, P.: Floyd-Warshallův algoritmus. [cit. 2015-04-08].
URL http://www.algoritmy.net/article/5207/Floyd-Warshalluv-algoritmus

[6] Mička, P.: Problém nejkratší cesty. [cit. 2015-04-08].
URL http://www.algoritmy.net/article/36597/Nejkratsi-cesta