MI-PAP

Dominik Soukup, Jiří Kadlec

01. 03. 2017

Obsah

1 Implementace sekvenčního algoritmu

1.1 Popis Dijkstrova algoritmu

Algoritmus je možné chápat jako zobecněné prohledávání do šířky. Po provedení nám dává řešení nejkratších cest z jednoho počátečního uzlu do všech ostatních. Předpokladem algoritmu je, že žádná hrana není záporně ohodnocena. V této implementaci jsou navíc všechny hrany ohodnoceny kladně.

Pro složitost vytvořeného algoritmu platí: Počáteční inicializace použitých struktur *O(nodes)*. Délka hlavního cyklu závisí na počtu uzlů. V rámci tohoto cyklu se vybírá následující uzel a zpracují se jeho potomci. Pro výběr uzlů byla použita prioritní fronta. Celková složitost je tedy $O(nodes.log(nodes))$. Následně se zpracovávají všechny potomci zvoleného uzlu. Při změně ceny je potřeba vložit uzel do prioritní fronty. Složitost této časti je $O(edges.log(nodes))$. Nakonec je nutné počítat s tím, že se výpočet spouští z každého uzlu. Celková složitost implementovaného Dijsktova algoritmu je

$O(nodes * (nodes + nodes.log(nodes) + edges.log(nodes)))$

1.2 Popis Floyd-Warshallova algoritmu

Výsledkem tohto algoritmu jsou nejkratší cesty mezi všemi páry uzlů.

Pro složitost vytvořeného algoritmu platí: Počáteční inicializace použitých struktur *O(nodes)*. Následně se provádějí 3 vnořené cykly. Celková složitost je tedy $O(nodes^3)$

Floyd-Warshallův algoritmus díky třem vnořeným cyklům a jednoduché podmínce je implementačně jednoduchý a pseudokod pro nalezení nejkratších cest vypadá násedovně:

```
for k in 1 to n do
 for i in 1 to n do
   for j in 1 to n do
      if D[i][j] > D[i][k] + D[k][j] then
         D[i][j] = D[i][k] + D[k][j]
         P[i][j] = P[k][j]
```

kde matice **D** je matice délek a matice **P** je matice předchůdců.

2 OpenMP pro x86 a Xeon Phi)

2.1 Dijk

2.2 Floyd-Warshall

Implementace floyd-warshalova algoritmu je v podstatě jednoduchá, ale tělo nejvnitřnějšího for cyklu se dá přepsat několika způsoby. Na základě toho

jsme zkoušeli, který způsob běží nejrychleji a jestli se nám podařilo vektorizovat tento algoritmus.

Jednotlivé implementace a jejich zhodnocení si popíšeme v nasledujících podkapitolách.

2.2.1 Klasický Floyd-Warshall

Implementace vypadá následovně:

```
for ( int k = 0; k < n; ++k)
  for ( int i = 0; i < n; ++i)
    for ( int j = 0; j < n; ++j) {
      if (D[i][j] > D[i][k] + D[k][j]) {
        D[i][j] = D[i][k] + D[k][j];
        P[i][j] = P[k][j];
    }
```

Toto řešení není vektorizované a následné časy pro x86 a Xeon Phi jsou následovné.

vlákna\uzlů	1000	1500	2000
1	1.11602	3.79188	9.18048
2	0.564859	1.84135	4.34206
4	0.288087	0.972909	2.17533
6	0.196072	0.670776	1.49812
8	0.15429	0.479784	1.11342
10	0.130064	0.38863	0.886158
12	0.185516	0.514168	0.932699
24	0.140569	0.398891	0.76749

vlákna\	1000	1500	2000
61	0.937226	1.11005	2.49671
122	0.662043	0.9634	1.1.9717
183	0.457726	0.973209	1.90224
244	0.485168	0.886052	1.70336

2.2.2 Ternární operátor

V tomto algoritmu počítáme pouze s maticí délek a matici předchůdců vynecháváme.

```
for ( int k = 0; k < n; ++k)
  for ( int i = 0; i < n; ++i)
   for ( int j = 0; j < n; ++j) {
      D[i][j] = ( D[i][j] > D[i][k] + D[k][j] ? D[i][k] + D[k][j] : D[i]
   }
```

vlákna\uzlů	1000	1500	2000
1	1.53532	.5.2044	12.8748
2	0.780031	2.61042	6.19777
4	0.385762	1.92644	3.08789
6	0.261746	1.28373	2.12849
8	0.208339	0.92291	1.81371
10	0.178032	0.773648	1.50141
12	0.152006	0.620251	1.35568
24	0.2881	0.57697	1.19574

vlákna\uzlů	1000	1500	2000
61	0.758129	1.67294	3.90219
122	0.622142	1.62244	3.20099
183	0.632614	1.51672	3.06845
244	0.669848	1.41912	2.76457

2.2.3 min(a,b) metoda

Nejaky text.

```
for ( int k = 0; k < n; ++k)
  for ( int i = 0; i < n; ++i)
    for ( int j = 0; j < n; ++j) {
      D[i][j] = min( D[i][k] + D[k][j] ,  D[i][j] );
}
```

vlákna\uzlů	1000	1500	2000
1	1.53201	5.2404	12.7781
2	0.777374	2.58762	6.18361
4	0.394649	1.31667	3.13383
6	0.269766	0.885247	2.15547
8	0.205032	0.778462	1.57456
10	0.175406	0.615799	1.3035
12	0.151809	0.622568	1.24625
24	0.245878	0.462049	1.11153

vlákna\uzlů	1000	1500	2000
61	0.84409	2.05163	4.28279
122	0.673217	1.59254	3.06231
183	0.66181	1.5369	2.92522
244	0.668723	1.43921	2.78747

2.2.4 1D pole

Vyuziti 1d pole...

```
void NCG::FloydWarshall() {
```

```
for ( int k = 0; k < n; k++)
 for (int i = 0; i < n; i++)
   fw_inner(k, i);
}

void NCG::fw_inner(int k, int i) {
int d_ik = FWDistanceMatrix[i * nodes + k];
for (int j = 0; j < nodes; j++) {
 int d_kj = FWDistanceMatrix[k * nodes + j];
 int t = d_ik + d_kj;
 int ij = i * nodes + j;
 int d_ij = FWDistanceMatrix[ij];
 if (t < d_ij)
   FWDistanceMatrix[ij] = t;
}


}
}
```

vlákna\uzlů	1000	1500	2000
1	2.30716	7.71098	18.4961
2	1.16078	3.86552	9.06954
4	0.588729	1.94404	4.55258
6	0.39564	1.3027	3.0499
8	0.303309	0.985183	2.37103
10	0.249257	0.793701	1.88205
12	0.237044	0.738506	1.63522
24	0.224236	0.69695	1.53367

icc: LOOP BEGIN at fw1DArray.cpp(108,5) Peeled loop for vectorization, Multiversioned v1 remark #15301: PEEL LOOP WAS VECTORIZED LOOP END

vlákna\uzlů	1000	1500	2000
61	0.418698	1.14185	2.35745
122	0.425456	0.961513	1.77961
183	0.448976	0.874202	1.68529
244	0.456448	0.820163	1.53792

Asi i poynamenat, ze kvuli vektorizaci pro mensi hodnoty a vice jader nam to bude kvuli tomu zpomalovat...

2.2.5 Shrnutí

Srovnání všechn hodnot pro g++ pro pocet uzlu 2000.

vlákna\algoritmus	classic	ternarni operator	min(a,b)	1D pole
1	9.18048	12.8748	12.7781	18.4961
2	4.34206	6.19777	6.18361	9.06954
4	2.17533	3.08789	3.13383	4.55258
6	1.49812	2.12849	2.15547	3.0499
8	1.11342	1.81371	1.57456	2.37103
10	0.886158	1.50141	1.3035	1.88205
12	0.932699	1.35568	1.24625	1.63522
24	0.76749	1.19574	1.11153	1.53367

		classic	ternarni operator	min(a,b)	1D pole
icc pro 2000	61	2.49671	3.90219	4.28279	2.35745
	122	1.9717	3.20099	3.06231	1.77961
	183	1.90224	3.06845	2.92522	1.68529
	244	1.70336	2.76457	2.78747	1.53792

2.2.6 Dodatek k vektorizaci

mozna zminit ze existuji metody blocked and tiled, a ze jsou mozne dohledat treba tu:

a dokonce nepochopitelny a desne slozity kod je dostupny na: