MI-PAP

Dominik Soukup, Jiří Kadlec

26. 04. 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 reprezentaci vnitřních struktur algoritmu bylo použito dvourozměrné pole integerů. Pouze pro příznak uzavření uzlů bylo použito jednorozměrné pole boolů.

Pro celkovou 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 tohoto 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. Pro reprezentaci matice délek a matice předchůdců bylo použito dvourozměrné pole integerů Pseudokód 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 Vektorizace

Pro vektorizaci algoritmů byla použita automatická podpora vektorizace v kompilátorech *g++* a *icc*. K její implementaci byl primárné použit popis,

který je k dispozici v dokumentaci gcc.

2.1 Dijkstrův algoritmus

Struktura Dijkstrova algoritmu patří mezi špatně vektorizovatelnou. Téměř veškeré použité smyčky nepodporují formát g++ pro autovektorizaci. Jedinou smyčkou vhodnou k vektorizaci byla smyčka sloužící k inicializaci matic (vzdálenosti, sousednosti) Dijkstrova algoritmu. Oproti sekvenčnímu řešení však muselo dojít k její úpravě. Inicializační cyklus byl upraven tak, že se všechny prvky matic nastavily na stejnou počáteční hodnotu. Hodnota startovacího prvku byla nastavena až po počáteční inicializační smyčce. Dále z důvodu neznámého počtu interací byl tento počet nastaven do proměnné *tmp*, ještě před voláním inicializačních cyklů. Zároveň bylo nutné změnit datový typ pole s příznaky uzavřenosti uzlů. Místo datového typu boolean byl použit typ integer.

Autovektorizace překladače *g++* zvektorizovala pouze hlavní inicializační smyčku. Překladač *icc* zvektorizoval hlavní inicializační i smyčku pro incializaci pole příznaků uzavřenosti uzlů.

Naměřené výsledky se nachází v kapitole 4.

2.2 Floyd-Warshallovův algoritmus

Floyd-Warshalův algoritmus byl testován na čtyřech různých implementacích, kde bylo pozorováno zda se algoritmus podařilo zvektorizovat a zrychlit.

Popis jednotlivých algoritmů, měření a výsledků běhů budou popsány v jednotlivých kapitolách.

2.2.1 Klasický Floyd-Warshall

```
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]
```

Toto je implementace klasického algoritmu, jak bývá popisován téměř ve veškerých příručkách. Z vnořených cyklů je patrná datová závislost, která bránív provedení vektorizace.

2.2.2 Ternární operátor

```
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][j]); }
```

V tomto případě bylo vynecháno počítání matice předchůdců, aby bylo možné zvektorizovat výpočet matice sousednosti. Výpočet byl zvektorizován díky automatické podpoře kompilátoru g++, ale výsledná vektorizace nevedla k lepším běhovým časům.

2.2.3 min(a,b) metoda

```
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] );
}
```

V této variantě byl nahrazen ternární operátor z předchozího algoritmu za funkci *min(a,b)*, která vrací minimum ze dvou čísel. Pokus o vektorizaci byl neúspěšný na obou kompilátorech.

2.2.4 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;
}
}
}
```

V této variantě bylo nahrazeno dvourozměrné pole za jednorozměrné. Při kompilování byla autovektorizace úspěšná pouze u kompilátoru *icc*. Ovšem k výslednému zrychlení doby běhu nedošlo.

2.2.5 Shrnutí

Jak výsledky ukazují 1, 2, tak z našeho měření pro kompilátor *g++* nejlépe vychází klasický Floyd-Warshall. I přes to, že varianta s ternárním operátorem

byla vektorizováná, tak její výsledky jsou pomalejší nebo stejně rychlé jako implementace, které se nepodařilo zvektorizovat. To bude pravděpodobně zapříčeněno datovou závislostí a přemazávání cache paměti.

Pro kompilátor *icc*, vychází obdobně klasický Floyd-Warshall nejlépe a zvektorizovaný algoritmus s jednorozměrným polem obdobně není rychlejší, kde příčinou bude datová závislost.

Během vykonávání práce byly objeveny odborné články, které se zabývají vektorizací Floyd-Warshallova algoritmu a odstranění datové závislosti. Jeden z těchto článku je k dispozici zde.

# vl./# h	1000	1500	2000	5000
Klasický	1.13062	3.83618	9.27812	136.591
Ternární op.	1.53532	5.2044	12.8748	196.535
Min(a,b)	1.53201	5.2404	12.7781	206.468
1D pole	2.30716	7.71098	18.4961	284.289

Tabulka 1: Tabulka vývoje času různých implementací vektorizace pro sekvečního řešení. Měřeno pro různý počet uzlů a vláken. Přeloženo překladačem g++ na CPU.

# vl./# h	1000	1500	2000	5000
Klasický	2.09091	6.99608	16.6729	N/A
Ternární op.	N/A	N/A	N/A	N/A
Min(a,b)	N/A	N/A	N/A	N/A
1D pole	N/A	N/A	N/A	N/A

Tabulka 2: Tabulka vývoje různých implementací vektorizace pro sekvečního řešení. Měřeno pro různý počet uzlů a vláken. Přeloženo překladačem icc na CPU.

3 Paralelizace - OpenMP

Pro paralelizaci algoritmů byla využito datového paralelismu. V této kapitole jsou popsány nutné úpravy, které bylo potřeba provést. V další části se nachází naměřené výsledky použitých variant algoritmů.

3.1 Dijkstrův algoritmus

Už při vektorizaci tohoto algorimu bylo zřejmé, že jeho struktura není příliš dobře paralelizovatelná. Jedinou možností paralelizace je přiřazení počátečních

uzlů každému vláknu. Vlákna následně pro daný počáteční uzel naleznou nejkratší cesty do všech ostatních uzlů.

V sekvečním algoritmu jsou jednotlivé počáteční uzly zadávány ve smyčce *for*. Z tohoto důvodu je pro tuto část vhodný datový paralelismus pomocí direktivy *paralel for*. Aby bylo možné využít veškeré funce knihovny OpenMP, byl kód sekvenčního řešení upraven z objektového do procedurálního stylu.

Jednotlivé instance jsou navzájem disjuktní, a proto je nejvhodnější použít *schedule(static)*. Veškeré globální proménné, vstupující do paralelní části, jsou sdíleny mezi vlákna. Na rozdíl od sekvenčního řešení, bylo přesunuto globální pole *closed* do funkce *dijkstra*. Pole *closed* obsahuje příznaky uzavřených uzlů a má tedy lokální význam pro každé vlákno. Všechny ostatní datové struktu jsou totožné jako u sekvenčního řešení.

Použitá paralelizace výrazně snížila celkový výpočetní čas. Naměřené výsledky jsou v následujících kapitolách.

3.2 Floyd-Warshallovův algoritmus

Implementace Floyd-Warshallova algoritmu je v podstatě jednoduchá, ale tělo nejvnitřnějšího *for* cyklu se dá přepsat několika způsoby. Na základě toho bylo testováno, který způsob běží nejrychleji a jestli se ho podaří úspěšně vektorizovat.

Pro měření hodnot byly použity stejné implementace jako tomu bylo u implementací v sekci Vektorizace 2. Na těchto algoritmech byla zjišťována jejich rychlost oproti různých vstůpům a počtu vláken.

Pro zajištění co nejmenší režie vytváření vláken jsou vlákna vytvářena před prvním *for* cyklem, čímž zapříčiníme, že se nám vlákna vytvoří pouze jedenkrát.

Pro paralerizaci jsme paralerizovali 2. *for* cyklus. První paralerizovat nelze, jelikož vždy musi doběhnout vnitřní dva. Paralerizace nejvnitřního *for* cyklu je možná, ale oproti paralerizaci druhého *for* cyklu by byla mnohem pomalejší, kvůli mnoha implicitním barierám na konci cyklu.

V tomto případě implicitních barier je *nodes*. Při vektorizaci nejvnitřnějšího *for* cyklu by jich bylo *nodes*nodes*, což by byla zbytečně velká režie navíc.

4 Naměřené výsledky první části

Tato kapitola obsahuje souhrn naměřených hodnot pro různé překladače, architektury a implementace algoritmů.

4.1 Dijkstrův algoritmus

Pro otestování implementace Dijkstrova algoritmu byly vygenerovány grafy o velikosti: 1000, 1500 a 2000 uzlů. Protože je doba běhu tohoto algoritmu

závislá i na počtu hran, měření bylo prováděno i pro různé hustoty. Grafy byly vygenerovány pomocí generátoru grafů. Nastavení ohodnocení jednotlivých hran bylo prováděno náhodně pomocí skriptu a jazyce *Bash*.

4.1.1 Sekveční řešení

Sekvenční řešení testuje běh původního základního programu a upravené verze pro vektorizaci. Testy byle měřeny pro kompilátor *g++* 3, 4 a 5. Z hodnot je patrné, že zapnutím optimalizací překladače získáme dobré zrychlení (pro některé hodnoty až dvojnásobnné). Naopak zvektorizování kódu vede k mírnému zpomalení.

#uzlů/#hran	K=10	K=50	K=100	K=300	K=500	K=700
1000	5.92764	10.0172	13.0879	23.6305	33.2583	28.9448
1500	18.2387	29.6598	37.3034	N/A	86.7427	114.921
2000	N/A	N/A	99.8919	155.273	207.924	271.259

Tabulka 3: Tabulka vývoje času sekvečního řešení pro různý počet hran a počet uzlů. Přeloženo překladačem g++.

#uzlů/#hran	K=10	K=50	K=100	K=300	K=500	K=700
1000	2.90301	5.73726	8.46729	18.8166	29.1521	25.2372
1500	7.4949	16.7092	23.3007	N/A	71.8074	98.6267
2000	N/A	N/A	49.3614	92.1438	133.657	183.64

Tabulka 4: Tabulka vývoje času sekvečního řešení se zapnutými optimalizacemi pro různý počet hran a počet uzlů. Přeloženo překladačem g++

#uzlů/#hran	K=10	K=50	K=100	K=300	K=500	K=700
1000	3.11856	6.61403	9.63747	20.2483	31.0386	27.0293
1500	7.48844	19.1842	26.9179	N/A	77.0189	105.082
2000	N/A	N/A	56.9591	102.577	144.774	196.528

Tabulka 5: Tabulka vývoje času sekvečního řešení se zapnutými optimalizacemi a vektorizací pro různý počet hran a počet uzlů. Přeloženo překladačem g++.

4.1.2 Paralelní řešení - g++

Při měření časů pro paralelních běh byly dosažené výsledky podobné pro různé počty uzlů. Pro přehlednost tato kapitola obsahuje jen tabulku pro

2000 uzlů 6. Z tabulky je vidět, že pokud každé stálé snižování celkového času při větším počtu vlánek. Pro 24 vláken není vidět takový nárůst, protože na testovacím uzlu je reálně k dispozici jen 12 cpu a pro vyšší počet vláken se projevuje technologie hyper-threading.

#vl./#h	K=100	S	K=300	S	K=500	S	K=700	S
1	48.6147	1.0058	89.2398	1.0325	130.064	1.0276	179.398	1.0236
2	24.5003	2.0147	44.6241	2.0648	65.0491	2.0547	89.7604	2.0458
4	12.2412	4.0323	22.4773	4.0994	32.6038	4.0994	44.9285	4.08738
6	8.21695	6.0072	14.9652	6.1572	21.7674	6.1402	30.1173	6.0974
8	6.3049	7.8290	11.3962	8.0854	16.4875	8.1065	25.7099	7.1427
10	5.24213	9.4162	9.22468	9.9888	13.3494	10.0122	18.3008	10.0345
12	4.41145	11.1893	7.87597	11.6993	11.371	11.7541	15.2752	12.0221
24	3.56467	13.8473	5.56699	16.5518	7.55135	17.6997	9.93128	18.4910

Tabulka 6: Tabulka vývoje času paralelního řešení se zapnutými optimalizacemi pro 2000 uzlů a různý počet vláken a uzlů. Přeloženo překladačem g++.

4.1.3 Sekvenční řešení - icc pro cpu

Sekveční verze algoritmu byla zkompilována i pod překladačem *icc*. Výsledky se nachází v tabulkách 7, 8 a 9. Oproti *g++* má překladač *icc* rychlejší časy při kompilaci bez optimalizací. Při jejich zapnutí není zrychlení tak velké. Navíc výsledné časy jsou pomalejší než u *g++*. Po zvektorizování došlo ke stejnému drobnému zpomalení jako v předchozím případě.

#uzlů/#hran	K=10	K=50	K=100	K=300	K=500	K=700
1000	2.61094	7.02613	10.9689	21.9903	31.0827	106.61
1500	7.83539	19.1486	29.5759	N/A	82.767	114.921
2000	N/A	N/A	60.6437	117.838	161.389	209.85

Tabulka 7: Tabulka vývoje času sekvečního řešení pro různý počet hran a počet uzlů. Přeloženo překladačem icc pro cpu.

poč. uzlů/poč. hran	K=10	K=50	K=100	K=300	K=500	K=700
1000	2.46371	6.49127	10.1132	21.0518	30.1956	24.2999
1500	7.5094	17.8654	27.2618	N/A	79.5227	103.688
2000	N/A	N/A	55.916	109.236	152.893	201.916

Tabulka 8: Tabulka vývoje času sekvečního řešení se zapnutými optimalizacemi pro různý počet hran a počet uzlů. Přeloženo překladačem icc pro cpu.

poč. uzlů/poč. hran	K=10	K=50	K=100	K=300	K=500	K=700
1000	2.4832	6.49663	10.0885	21.0607	30.2059	24.2888
1500	7.49878	17.8578	27.2555	N/A	79.5125	103.685
2000	N/A	N/a	55.9737	109.257	152.921	201.925

Tabulka 9: Tabulka vývoje času sekvečního řešení se zapnutými optimalizacemi a vektorizací pro různý počet hran a počet uzlů. Přeloženo překladačem icc pro cpu.

4.1.4 Paralelní řešení - icc pro cpu

Ačkoli pro sekveční řešení dosahoval lepších časů překladač *g++*, při paralelizaci je mírně rychlejší, a tedy i efektivnější, překladač *icc*. Pokles doby běhu při růstu vláken je jinak obdobný. Získané výsledky se nachází v tabulce 10

#vl./#h	K=100	S	K=300	S	K=500	S	K=700	S
1	56.3744	0.9918	109.678	0.9959	152.553	1.0022	200.284	1.0081
2	28.056	1.9930	54.6426	1.9990	76.0376	2.0107	99.9976	2.0192
4	14.0388	3.9829	27.3568	3.9930	38.0303	4.0202	50.0288	4.0359
6	9.40213	5.9471	18.295	5.9708	25.4268	6.0130	33.4343	6.0391
8	7.04928	7.9321	13.7353	7.9529	19.0909	8.0086	25.2987	7.9812
10	5.77533	9.6818	11.0558	9.8804	15.4848	9.8737	20.9525	9.6368
12	4.84285	11.5460	9.28455	11.7653	12.8411	11.9065	16.9308	11.9259
24	3.3027	16.9303	5.61524	19.4534	8.14952	18.7609	9.96011	20.2724

Tabulka 10: Tabulka vývoje času paralelního řešení se zapnutými optimalizacemi pro 2000 uzlů a různý počet vláken a uzlů. Přeloženo překladačem icc pro cpu.

4.1.5 Paralelní řešení - icc pro Xeon Phi

Při návrhu programu bylo cílem vytvořit paralelní řešení pro klasická cpu. Vytvořený program byl pouze překompilován pro Xeon Phi. Z naměřených výsledků 11 je zřejmé využití velkého počtu vláken, a proto je doba výpočtu tak nízká. Při zvyšování počtu vláken už doba výpočtu dále neklesá, protože Dijkstrův algoritmus nebylo možné více vektorizovat.

#vl./#h	K=100	K=300	K=500	K=700
32	10.171	11.7799	12.9829	15.1481
61	6.2056	7.19268	7.85071	8.58323
122	3.8946	4.65457	5.04411	5.6166
183	3.09126	3.67682	4.03957	4.5518
244	2.79369	3.34279	3.69122	4.04843

Tabulka 11: Tabulka vývoje času paralelního řešení se zapnutými optimalizacemi pro 2000 uzlů a různý počet vláken a uzlů. Přeloženo překladačem icc pro Xeon Phi.

4.2 Floyd-Warshallovův algoritmus

4.2.1 Sekveční řešení

V této sekci bylo naměřeno zrychlení, jaké způsobují přepínače pro optimalizaci a paralelizaci pro kompilátory *g++* a *icc* pro implementaci využívající klasickou implementaci Floyd-Warshalla.

Z následujících tabulek 12, 13 lze vidět, že optimalizace měla obrovský vliv na rychlost programu. Oproti tomu, vektorizace neměla vůbec žádný, jelikož tato implementace nebyla zvektorizována.

parametry	K=1000	K=1500	K=2000
g++	10.653	35.8642	86.0735
g++ O3	1.11474	3.79906	9.19753
g++ O3 vekt	1.13062	3.83618	9.27812

Tabulka 12: Tabulka vývoje času sekvečního řešení pro různé parametry kompilátoru a různý počet uzlů. Přeloženo překladačem g++.

parametry	K=1000	K=1500	K=2000
g++	2.09111	6.99989	16.6813
g++ O3	2.09121	7.00416	16.6674
g++ O3 vekt	2.09091	6.99608	16.6729

Tabulka 13: Tabulka vývoje času sekvečního řešení pro různé parametry kompilátoru a různý počet uzlů. Přeloženo překladačem icc pro cpu.

4.2.2 Paralelní řešení - g++

V následující sekci byly naměřeny všechny čtyři implementace Floyd-Warshalla pro různě velké vstupní data a různý počet vláken. Měření probíhalo pod kompilátorem *g++* na cpu.

Z následujících tabulek lze vidět nádherné a mnohdy lineární z rychlení. Pro implementace využívající dvourozměrné pole nastal zvrat pro vstup pro 5000 uzlů, kde algoritmus přestal zrychlovat u 6-8 vlákna. Toto chování dáváme za vinu datové závislosti a častému přemazávání cache paměti.

Oproti tomu algoritmus využívající jednorozměrné pole zrychluje neustále i pro vyšší počet vláken pro vstup o 5000 uzlech. Avšak i díky této vlastnosti nedosahuje rychlosti jako Klasický FW využívající i 3x méně vláken.

Z tohoto měření vychází nejlépe klasický FW. Veškeré výsledky jsou shrnuty v tabulkách 14, 15, 16 a 17.

#vl./#h	K=1000	S	K=1500	S	K=2000	S	K=5000	S
1	1.11602	1.0130	3.79188	1.0116	9.18048	1.0106	136.591	N/A
2	0.564859	2.0015	1.84135	2.0833	4.34206	2.1368	75.3649	N/A
4	0.288087	3.9245	0.972909	3.9429	2.17533	4.2651	38.4821	N/A
6	0.196072	5.7663	0.670776	5.7190	1.49812	6.1931	27.2238	N/A
8	0.15429	7.3278	0.479784	7.9956	1.11342	8.3329	24.535	N/A
10	0.130064	8.6927	0.38863	9.8710	0.886158	10.4700	23.9531	N/A
12	0.185516	6.0944	0.514168	7.4609	0.932699	9.9476	23.6708	N/A
24	0.140569	8.0431	0.398891	9.6171	0.76749	12.0889	24.015	N/A

Tabulka 14: Tabulka vývoje času paralelního řešení se zapnutými optimalizacemi a vektorizací pro různý počet uzlů a různý počet vláken. Přeloženo překladačem g++ pro cpu.

#vl./#h	K=1000	S	K=1500	S	K=2000	S	K=5000	S
1	1.53532	0.7364	5.2044	0.7371	12.8748	0.7206	196.535	N/A
2	0.780031	1.4494	2.61042	1.4695	6.19777	1.4970	103.417	N/A
4	0.385762	2.9308	1.92644	1.9913	3.08789	3.0046	58.5137	N/A
6	0.261746	4.3195	1.28373	2.9883	2.12849	4.3590	53.6015	N/A
8	0.208339	5.4268	0.92291	4.1566	1.81371	5.1155	54.6599	N/A
10	0.178032	6.3506	0.773648	4.9585	1.50141	6.1796	54.4292	N/A
12	0.152006	7.4379	0.620251	6.1848	1.35568	6.8438	53.8458	N/A
24	0.2881	3.9244	0.57697	6.6488	1.19574	7.7593	54.2186	N/A

Tabulka 15: Tabulka vývoje času paralelního řešení s ternárním operátorem, zapnutými optimalizacemi a vektorizací. Měřeno pro různý počet uzlů a vláken. Přeloženo překladačem g++ pro cpu.

#vl./#h	K=1000	S	K=1500	S	K=2000	S	K=5000	S
1	1.53201	0.7379	5.2404	0.7320	12.7781	0.7260	206.468	N/A
2	0.777374	1.4544	2.58762	1.4825	6.18361	1.5004	102.925	N/A
4	0.394649	2.8648	1.31667	2.9135	3.13383	2.9606	60.1841	N/A
6	0.269766	4.1911	0.885247	4.3334	2.15547	4.3044	54.0843	N/A
8	0.205032	5.5143	0.778462	4.9278	1.57456	5.8925	54.8656	N/A
10	0.175406	6.4457	0.615799	6.2295	1.3035	7.1178	55.8123	N/A
12	0.151809	7.4476	0.622568	6.1618	1.24625	7.4448	54.2548	N/A
24	0.245878	4.5982	0.462049	8.3025	1.11153	8.3471	54.4386	N/A

Tabulka 16: Tabulka vývoje času paralelního řešení s použitím operace minimum, zapnutými optimalizacemi a vektorizací. Měřeno pro různý počet uzlů a vláken. Přeloženo překladačem g++ pro cpu.

# vl./# h	K=1000	S	K=1500	S	K=2000	S	K=5000	S
1	2.30716	0.4900	7.71098	0.49749	18.4961	0.5016	284.289	N/A
2	1.16078	0.9740	3.86552	0.9924	9.06954	1.0229	147.518	N/A
4	0.588729	1.9204	1.94404	1.9733	4.55258	2.0379	74.0468	N/A
6	0.39564	2.8576	1.3027	2.9447	3.0499	3.04210	49.8403	N/A
8	0.303309	3.7276	0.985183	3.8938	2.37103	3.9131	37.6339	N/A
10	0.249257	4.5359	0.793701	4.8332	1.88205	4.9297	33.8847	N/A
12	0.237044	4.7696	0.738506	5.1945	1.63522	5.6739	45.7191	N/A
24	0.224236	5.0420	0.69695	5.5042	1.53367	6.0496	23.668	N/A

Tabulka 17: Tabulka vývoje času paralelního řešení s použitím 1D pole, zapnutými optimalizacemi a vektorizací. Měřeno pro různý počet uzlů a vláken. Přeloženo překladačem g++ pro cpu.

4.2.3 Paralelní řešení - icc pro cpu

V následující sekci byly naměřeny všechny čtyři implementace Floyd-Warshalla pro různě velké vstupní data a různý počet vláken. Měření probíhalo pod kompilátorem *icc* na cpu.

Následující měření je velmi podobné jako měření pod kompilátor g++ pro cpu. Veškeré výsledky jsou shrnuty v tabulkách 18, 19, 20 a 21.

# vl./# h	K=1000	S	K=1500	S	K=2000	S	K=5000	S
1	1.18724	1.7611	4.26375	1.6408	9.78776	1.7034	145.714	N/A
2	0.604722	3.4576	1.96683	3.5570	4.53606	3.6756	79.6401	N/A
4	0.307858	6.7918	1.02545	6.8224	2.32599	7.1680	40.7008	N/A
6	0.217842	9.5982	0.683377	10.2375	1.6555	10.0712	24.953	N/A
8	0.288806	7.2398	0.516303	13.5503	1.33139	12.5229	24.953	N/A
10	0.233989	8.9359	0.571045	12.2513	1.18771	14.0379	24.0903	N/A
12	0.199814	10.4642	0.69606	10.0509	1.20063	13.8867	24.0191	N/A
24	0.273615	7.6417	0.749615	9.3328	1.04454	15.9619	23.8775	N/A

Tabulka 18: Tabulka vývoje času paralelního řešení se zapnutými optimalizacemi a vektorizací pro různý počet uzlů a různý počet vláken. Přeloženo překladačem icc pro cpu.

# vl./# h	K=1000	S	K=1500	S	K=2000	S	K=5000	S
1	1.63663	1.2775	5.68406	1.2308	10.766342	1.5486	208.844	N/A
2	0.840402	2.4879	2.79016	2.5074	6.62211	2.5177	108.829	N/A
4	0.427177	4.8947	1.42125	4.9224	3.3204	5.0213	60.0722	N/A
6	0.329771	6.3404	0.961846	7.2735	2.26049	7.3757	53.715	N/A
8	0.311656	6.7090	0.828294	8.4463	1.95655	8.5215	53.8724	N/A
10	0.342779	6.0998	0.865386	8.0843	1.37308	12.1427	53.8636	N/A
12	0.345857	6.04559	0.782279	8.9432	1.45729	11.4410	54.6911	N/A
24	0.362607	5.7663	0.766342	9.1291	1.28627	12.9622	54.5179	N/A

Tabulka 19: Tabulka vývoje času paralelního řešení s ternárním operátorem, zapnutými optimalizacemi a vektorizací. Měřeno pro různý počet uzlů a vláken. Přeloženo překladačem icc pro cpu.

# vl./# h	K=1000	S	K=1500	S	K=2000	S	K=5000	S
1	1.66482	1.2559	6.00194	1.1656	15.044	1.1082	218.948	N/A
2	0.838843	2.4926	3.25277	2.1508	7.99549	2.0852	107.715	N/A
4	0.431354	4.8473	1.52402	4.5905	3.37797	4.9357	60.0681	N/A
6	0.285453	7.3248	1.11465	6.2764	2.28065	7.3105	53.4797	N/A
8	0.22869	9.1429	1.05606	6.6246	1.93363	8.6225	53.8425	N/A
10	0.355444	5.8825	1.19275	5.8655	1.63224	10.2147	54.0836	N/A
12	0.299708	6.9764	1.2015	5.8227	1.64468	10.1374	53.9701	N/A
24	0.533674	3.9179	0.928473	7.5350	1.46989	11.3429	42.4176	N/A

Tabulka 20: Tabulka vývoje času paralelního řešení s použitím operace minimum, zapnutými optimalizacemi a vektorizací. Měřeno pro různý počet uzlů a vláken. Přeloženo překladačem icc pro cpu.

# vl./# h	K=1000	S	K=1500	S	K=2000	S	K=5000	S
1	1.60057	1.3063	5.3707	1.3026	13.8527	1.2035	207.931	N/A
2	0.805904	2.5944	2.69189	2.5989	6.97983	2.3887	109.997	N/A
4	0.424144	4.9297	1.36367	5.1303	3.39435	4.9119	56.3528	N/A
6	0.355205	5.8864	0.977089	7.1601	2.62519	6.3511	49.9928	N/A
8	0.401894	5.2026	1.3385	5.2268	3.22977	5.1622	56.3645	N/A
10	0.324551	6.4424	1.09821	6.3704	2.61639	6.3724	48.7085	N/A
12	0.355205	5.8864	0.941953	7.4272	2.2372	7.4525	N/A	N/A
24	0.621728	3.3630	1.20774	5.7927	2.71977	6.1302	N/A	N/A

Tabulka 21: Tabulka vývoje času paralelního řešení s použitím 1D pole, zapnutými optimalizacemi a vektorizací. Měřeno pro různý počet uzlů a vláken. Přeloženo překladačem icc pro cpu.

4.2.4 Paralelní řešení icc pro Xeon Phi

V následující sekci byly naměřeny všechny čtyři implementace Floyd-Warshalla pro různě velké vstupní data a různý počet vláken. Měření probíhalo pod kompilátorem *icc* na procesoru Xeon Phi.

Z následujících tabulek 22, 23, 24 a 25. lze vidět, že srovnání různých výsledků různých implementací je velice podobné. Oproti CPU jednotce zde nevychází nejrychleji klasický FW, ale zbývající tři implementace, které mají velice podobné výsledky.

# vl./# h	1000	1500	2000	5000
61	0.549392	1.11292	2.43986	36.7248
122	0.425091	0.868312	2.01094	24.9441
183	0.453759	0.925265	1.87424	24.1287
244	0.48025	0.917716	1.70123	20.7968

Tabulka 22: Tabulka vývoje času paralelního řešení se zapnutými optimalizacemi a vektorizací pro různý počet uzlů a různý počet vláken. Přeloženo překladačem icc pro Xeon Phi.

# vl./# h	1000	1500	2000	5000
61	0.341478	0.906301	2.02982	28.98
122	0.467893	0.854509	1.65097	22.0668
183	0.446723	0.884041	1.66929	21.8362
244	0.453767	0.841075	1.54657	18.9269

Tabulka 23: Tabulka vývoje času paralelního řešení s ternárním operátorem, zapnutými optimalizacemi a vektorizací. Měřeno pro různý počet uzlů a vláken. Přeloženo překladačem icc pro Xeon Phi.

# vl./# h	1000	1500	2000	5000
61	0.416301	0.981006	2.08319	29.1444
122	0.434135	0.938226	1.74136	22.314
183	0.454789	0.918957	1.65643	21.5983
244	0.482584	0.863016	1.58683	19.1723

Tabulka 24: Tabulka vývoje času paralelního řešení s použitím operace minimum, zapnutými optimalizacemi a vektorizací. Měřeno pro různý počet uzlů a vláken. Přeloženo překladačem icc pro Xeon Phi.

# vl./# h	1000	1500	2000	5000
61	0.418698	1.14185	2.35745	29.9558
122	0.425456	0.961513	1.77961	22.2691
183	0.448976	0.874202	1.68529	21.4681
244	0.456448	0.820163	1.53792	20.9832

Tabulka 25: Tabulka vývoje času paralelního řešení s použitím 1D pole, zapnutými optimalizacemi a vektorizací. Měřeno pro různý počet uzlů a vláken. Přeloženo překladačem icc pro Xeon Phi.

5 Zhodnocení první části

5.1 Dijkstrův algoritmus

Z provedených testů vyplývá, že pro sekvenční řešení je nelepší použít překladač *g++* s jeho optimalizacemi. Pro paralelní řešení je vhodnější použít kompilátor *icc* (pro cpu i Xeon Phi).

Při porovnání cpu a Xeon Phi je rychlejší Xeon Phi díky velkému počtu vláken. Z naměřených výsledků se jako ideální počet vláken zdá 24 pro cpu a 122 pro Xeon Phi.

Bez ohledu na finance je doporučenou architekturou Xeon Phi.

5.2 Floyd-Warshallovův algoritmus

Měření vyšlo velice zajímavě, v závislosti na velikosti vstupní instance vychází lepé buď Xeon Phi nebo CPU jednotka. Pro instance 5000 začíná lépe vycházet Xeon Phi pro implementaci s jednorozměrným polem. Pro velikost instance 2000 a méně vychází lépe kompilace na CPU jednotce. Z toho vyplývá, že přibližně mezi instancemi 2000 a 5000 je hranice, od které bude vycházet lepší čas na Xeon Phi místo na CPU.

Kompilování pod *icc* na CPU vychází velmi podobně pro vyšší počet vláken jako pro kompilaci pod *g++*. U sekvenčního řešení nebo pro nižší počet vláken lépe vychází kompilátor *g++*.

Výsledky měřené pod GPU získavají linenarní zrychlení zhruba do 8-10 vlákna v závislosti na typu implementace, pro vyšší počet vláken to zrychluje velmi pomalu nebo dokonce i občas zpomaluje. Jelikož ani jedna implementace neobsahuje kritické sekce, které by mohli průběh zpomalovat, tak se zdá, že zpomalení způsubují implicitní bariéry a datové závislosti při, které se přemazává cache pamět častěji pro vyšší počet vláken.

Jelikož existuje hranice, při které každá implementace přestává zrychlovat na Xeon Phi a CPU jednotce, tak je zbytečné investovat do vyššiho počtu vláken na CPU. U CPU je lineární zrychlení zhruba do 8-10 vlákna. Na Xeon Phi lineární zrychlení není, ale se zvyšujícím počtem vláken jde vidět alespoň drobné zrychlení, avšak toto zrychlení, od 122 vláken, už není nijak velké.

Při výběru řešení se bude muset brát ohled na to, jestli se budou měřit instance nad 5000 uzlů a více, kde dominuje Xeon Phi, nebo pod 5000 uzlů, kde vychází lépe CPU.

6 OpenACC

OpenACC je vysokoúrovňový programovací model pro akcelerátory, který se svým použitím se velmi podobá OpenMP. Obecně pro využití akcelerátorů je potřeba odstranit rekurzi a postarat se správu paměti akcelerátoru a

CPU. V následující kapitole jsou popsány nutné úpravy, které bylo potřeba provést při implementaci OpenACC. Dále je vyhodnocena doba běhu pro různě velké instance.

6.1 Dijkstrův algoritmus

Dijkstrův algoritmus neobsahuje žádné rekurzivní volání, a proto nebyly potřeba žadné úpravy tohoto typu. Pro uplatnění paralelizace lze využít pouze smyčky *for*, ve které se volá řešící funkce vždy pro různý počáteční uzel. Podobně jako u OpenMP byla zde použita direktiva *#pragma acc parallel loop*. Z výpisů překladače a nalezených materiálů vyplývala nutnost použití direktivy *#pragma acc routine*, které umožňuje volat danou funkci z akcelerátoru. Dále bylo potřeba vytvořit používané datové struktury v paměti akcelerátoru. Pro tyto potřeby byla použita direktiva *#pragma acc declare create*.

I přes všechny tyto úpravy stále nebylo možné program přeložit z důvodu vnitřní chyby kompilátoru.

```
PGCC-S-0000-Internal compiler error. Call in OpenACC
   region to support routine - _mp_malloc
   (dijkstra_dynamic.cpp: 121)
PGCC-W-0155-External and Static variables are not
   supported in acc routine-(dijkstra_dynamic.cpp:350)
PGCC/x86 Linux 16.10-0: compilation completed
   with severe errors
```

Stejného výsledku bylo dosaženo i při statickém alokování datových struktur.

Jediného funkčního řešení bylo dosaženo při paralelizování inicializační smyčky. Avšak tato optimalizace neměla pozitivní dopad na výslednou dobu běhu a je tedy zbytečná.

6.2 Floyd-Warshallovův algoritmus

Floyd-Warshallovův algoritmus neobsahuje žádné rekuzivní části, a proto nebylo potřeba odstraňování rekurze. Pro co největší využití potenciálu akcelerátorů je potřeba odstranit veškeré datové závisloti. Implementované řešení vychází z tohto článku, kde je matice postupně rozdělena do několika datově nezávislých částí.

Každou datově nezávislou část zpracovávájí 3 nebo 4 *for* cykly a je tedy možné využít všechny skupiny paralelizace - *gang*, *worker* a *vector*. Konkrétní velikosti jednotlivých skupin byly určeny experimentálně.

Veškeré výpočty jsou prováděny nad maticí délek. Při vytváření datových struktur na akcelátoru stačilo využít direktivu *copy*.

Naměřené doby výpočtu jsou v následující kapitole.

7 Naměřené výsledky druhé části

Tato kapitola obsahuje souhrn naměřených hodnot pro Floyd-Warshallův algoritmus při různě velkých instancích. Pro Dijktrův algoritmus nebylo prováděno žádné měření, protože jej nebylo možné zkompilovat s použitím *OpenACC*.

7.1 Floyd-Warshallovův algoritmus

čas/poč. uzlů	1000	1500	2000	5000
Wall	5.32283	7.57735	9.84589	47.7259
CPU	3.63387	4.88906	8.1469	47.5611

Tabulka 26: Tabulka vývoje času pro různě velké instance. Přeloženo překladačem pg++ pro technologii OpenACC.

8 Zhodnocení druhé části

8.1 Dijkstrův algoritmus

Dijktrův algoritmus patří mezi špatně paralelizovatelné. Jedinou možností je paralelizace jeho volání pro různé počáteční uzlu. Technologie *OpenACC* umožňuje paralelizaci takových kódů pomocí direktivy *#pragma acc routine*. Nicméně i po provedených úpravách nebylo možné kód přeložit.

Z důvodu své struktury, není Dijkstrův algoritmus vhodný pro paralelizaci na *GPU*. Vhodným způsobem pro paralelizaci je *OpenMP*.

8.2 Floyd-Warshallovův algoritmus

U Floyd-Warshallova algoritmu se úspěšně povedlo nasadit technologii *OpenACC*. Výsledné výpočetní časy jsou, oproti očekávání, poměrně pomalé. V nejlepším případě bylo dosaženo času odpovídajícímu běhu na 4 *CPU* vláknech.

Hlavní příčinou je vysokoúrovňový přístup *OpenACC*, který neumožňuje efektivně pracovat s jednotlivými vlákny na *GPU*. Další příčinou mohou být kompilátorové optimalizace, které nejsou dostupné pro *GPU*.

9 Cuda

CUDA je architektura a programový model pro provádění paralelních výpočtů na grafických kartách od společnosti NVIDIA. Protože se zaměruje pouze na určité typy grafickým karet, je zde mnohem více optimalizovaná, na rozdíl od

obecnějších technologií. Pro dobře paralelizovatelné algoritmy má obrovský potenciál pro výraznější zkracování doby výpočtu.

9.1 Floyd-Warshallovův algoritmus

V této kapitole se nachází popis nutných úprav algoritmu. Při paralelizaci bylo využito dosažených zkušeností z implementace technologií *OpenMP* a *OpenACC*. Použitá implementace vychází ze základního Floyd-Warshallova algoritmu.

9.1.1 Úpravy paralelního algoritmu

Obecně pro provádění výpočtu na grafických kartách je dobré odstranit veškeré rekurzivní části programu. V případě technologie *Cuda* je ale možné v určitých případech rekurzivní části ponechat. Implementace Floyd-Warshallova algoritmu neobsahuje žádné rekurzivní části, a proto tento problém nebylo potřeba řešit.

Dalším problémem je přenos datových struktur mezi pamětí *CPU* a *GPU*. Floyd-Warshall při výpočtu používá matici délek a předchůdců. Pro alokaci 2D pole Cuda nabízí funkci *cudaMallocPitch*. Tato funkce ovšem nealokuje pole přesné velikosti a v některých případech zabere více paměti. Díky tomu, šířka řádku může být o něco větší a je nutné dopočítávat přesnou pozici přes parametr *pitch*, který vrací délku řádku. Z důvodu přehlednější implementace bylo, při využití mapovací funkce, místo 2D pole použito pole 1D. Veškeré výpočty matice délek a předchůdců se provádějí na *GPU* jednotce, a proto stačí vstupní matice přenést pouze jednou před zahájením výpočtu na *GPU* a po skončení zpět.

9.1.2 Paměť

Vstupní matice je rozdělena do stejně velkých čtvercových bloků, jejichž velikost je určena parametrem BLOCK_SIZE. Z důvodu stejně velkých bloků může, při nevhodném rozměru matice, dojít k přístupu mimo alokované pole. Technologie Cuda nekontroluje přístupy k paměti, a proto může docházet k neočekávaným výsledkům. Vytvořená implementace s tímto problémem počítá a při přístupu do paměti se kontrolují hodnoty indexů.

Dalším omezením je počet vytvářených vláken v rámci jednoho bloku. Tento počet je limitován. V testovacím prostředí je tento 1024. Z tohoto důvodu nemůže být velikost bloku větší než 32.

Rozdělení do bloků, jejich mapování a volání cílové metody odpovídá následující kód:

```
dim3 dimGrid((nodes + BLOCK_SIZE - 1) / BLOCK_SIZE,
             (nodes + BLOCK_SIZE - 1) / BLOCK_SIZE);
```

```
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

for ( int i = 0; i < nodes; i++)
{
  GPU_FloydWarshall<<< dimGrid, dimBlock>>>
       (i, deviceDistanceMatrix, devicePathMatrix, nodes);
  cudaThreadSynchronize();
}
```

9.1.3 Výpočet

Průběh výpočtu je v tomto případě jednoduchý a přímočarý. Každé vlákno dostane dvojici indexů i a j s cílem relaxovat hranu *i,j*. Zároveň je využito efektivní indexace vláken v rámci bloků.

10 Naměřené výsledky třetí části

Tato kapitola obsahuje výsledky měření pro dostupné instance grafů. Zároveň obsahuje porovnání s architekturami použitými v předchozích kapitolách.

poč. uzlů	cuda	S sek	S openmp
1000	0.061053	18.2585	2.3024
1500	0.167120	22.7325	2.3868
2000	0.329809	27.8874	2.3270
5000	4.117568	N/A	5.8323
7500	13.656274	N/A	N/A

Tabulka 27: Tabulka vývoje času pro různě velké instance včetně porovnání nejlepších řešení ostatních použitých architektur. Přeloženo překladačem nvcc pro technologii Cuda.

11 Zhodnocení třetí části

Technologie *Cuda* se ukázala jako velmi vhodná pro paralelizaci Floyd-Warshallova algoritmu. Díky využití prostředků, které *Cuda* nabízí, bylo možné zrychlit výpočet téměr 6 krát oproti nejrychlejšímu času u *OpenMP*, kterého bylo dosaženo při použití 24 vláken. Při srovnání se sekvenční verzí bylo dosaženo zrychlení téměř 28. Průběh implementace byl bezproblémový a vše šlo podle očekávání. Na rozdíl od *OpenACC* je programovací model v *Cudě* velmi dobře implementován a nabízí více možností.

12 Závěr

Cílem této práce bylo otestovat chování Dijsktrova a Floyd-Warshallova algoritmu na ruzných výpočetních architekturách. Zajímavé bylo pozorovat naměřené doby běhu pro různé výpočetní modely i nutné změny v daných algoritmech. Dijsktrův algoritmus je navžený pro hledání nejkratších cest z jednoho uzlu do ostatních. Pro sekveční postup je tento přístup vhodný a v porovnání s ostatními algoritmy dosahuje nejlepších výsledků. Úkolem této práce bylo hledání nejkratších cest mezi všemi uzly. V tomto případě je se jako lepší ukázal Floyd-Warshallův algoritmus.

Po navržení sekvenčních řešení bylo úkolem zrychlovat výpočet pomocí paralelizace a vektorizace. V případě vektorizace je potřeba odstranit datové závisloti a upravit přepínače kompilátoru. I přes všechny pokusy nebylo ani pro jeden z algoritmů dosaženo výraznějšího zrychlení. U Dijktrova algoritmu bylo možné vektorizovat pouze inicializaci datových struktur, ale její dopad na výkon není téměř žádný. U Floyd-Warshallova algoritmu byly testovány různé implementace, ale ani jedna z nich plně neodstranila datovou závislost.

Výraznějšího zrychlení bylo dosaženo až u paralelizace pomocí *OpenMP*. Pro Dijkstrův algoritmus bylo dosaženo maximálně 20 násobného zrychlení oproti sekvenční verzi. Při porovnání kompilátorů (*icc, g++*) byl pro větší počty vlánek mírně lepší *icc*. Stejný kód byl spuštěn i na akcelátoru *Xeon Phi*, který dokázal výpočet zrychlit ještě přibližně 5 krát. Podobných výsledků bylo dosaženo i pro Floyd-Warshallův algoritmus. Zde paralelní řešení dosáhlo, při použití překladače *icc*, přibližně 15 násobné zrychlení oproti sekvenční verzi. *Xeon Phi* dokázál výpočet zrychlit ještě dvojnásobně, ale pouze pro instance nad 5000 uzlů. Pro nižší instance se časy shodovali s výsledky na *CPU*.

Dalším krokem byla paralelizace na grafických kartách. Prvním pokusem byla technologie *OpenACC*. U Dijkstrova algoritmu se ukázalo, že pomocí této technologie není možné program zkompilovat tak, aby bylo dosaženo nějakého zrychlení. Kvůli obtížnému paralelizování tohoto algoritmu nebylo prováděno další testování na grafických kartách. Algoritmus Floyd-Warshall se pomocí *OpenACC* podařilo zkompilovat, ale výsledné zrychlení nebylo příliš velké. Důvod tohoto výsledků je vysokoúrovňový návrh *OpenACC*.

Veškerá očekávání splnila až technologie *Cuda*. Díky jejímu zaměření na grafické karty *NVDIA* a možnosti efektivní práce s vlákny bylo dosaženo téměř 6 násobného zrychlení oproti *OpenMP* výsledku.

Dle získaných výsledků je pro Dijktrův algoritmus nejlepší paralelizace pomocí *OpenMP* za použití kompilátoru *icc* a akcelerátoru *Xeon Phi*. Pro algoritmus Floyd-Warshall je nejvhodnější paralelizovat výpočty na architektuře *Cuda*. V této implementaci byla paralelizována základní verze tohoto algoritmu. Dalšího zrychlení je možné dosáhnout při efektivním dlaždicovém zpracování.