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 $O(nodes^3)$

2 Vektorizace

Pro vektorizaci algoritmů byla použita automatická podpora vektorizace v kompilátoru g++. Její popis 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 Dijktrova algoritmu (vzdálenosti, sousednosti). 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 inicializaci. Dále z důvodu neznámého počtu interací byl tento počet nastaven do proměnné *tmp*, ještě před voláním inicializačních cyklů.

Díky použité vektorizaci a zapnutým optimalizacím došlo ke snižení celkového výpočetního času. Náměřené výsledky jsou v kapitole ..

2.2 Floyd-Warshallovův algoritmus

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 je 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

3.3 Naměřené výsledky