VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

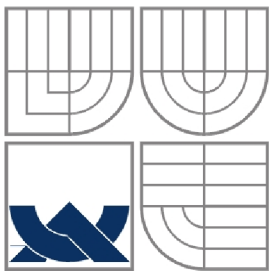
PROBLÉM OBCHODNÍHO CESTUJÍCÍHO - PARALELNÍ ŘEŠENÍ NA SMP (OPENMP)

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

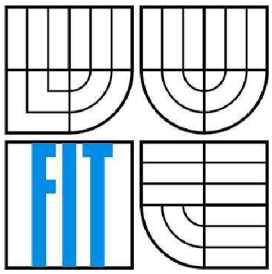
AUTOR PRÁCE
AUTHOR

MICHAL HRUŠKA

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PROBLÉM OBCHODNÍHO CESTUJÍCÍHO - PARALELNÍ ŘEŠENÍ NA SMP (OPENMP)

THE TRAVELING SALESMAN PROBLEM – PARALLEL SOLUTION USING SMP (OPENMP)

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

MICHAL HRUŠKA

VEDOUCÍ PRÁCE
SUPERVISOR

ING. KAŠPÁREK TOMÁŠ

BRNO 2009

Abstrakt

Práce porovná sériové a paralelní postupy řešení problému obchodního cestujícího. Používá některé známé heuristiky, které byly testovány po dobu několika desítek hodin. Algoritmy jsou porovnávány časovou a paměťovou složitostí. Bylo zjištěno, že paralelizace relativně malé části zdrojového kódu v případě ACO urychlí řešení způsobem, který odpovídá Amdahlově zákonu.

Klíčová slova

Paralelní řešení, problém obchodního cestujícího, OpenMP

Abstract

Thesis compares serial and parallel methods for solving travelling salesman problem. Some of the well-known heuristics are used. These have been tested for long hours. Time and space complexity are used as comparing measure. It was discovered that small part of ACO source code in parallel is going to speed up the code according to Amdahl's law.

Keywords

Parallel solution, traveling salesman problem, OpenMP

Citace

Hruška Michal: *Problém obchodního cestujícího - paralelní řešení na SMP (OpenMP)* Brno, 2009, bakalářská práce, FIT VUT v Brně.

Problém obchodního cestujícího - paralelní řešení na SMP (OpenMP)

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing.Tomáše Kašpárka.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Michal Hruška
20. května 2009

Poděkování

Děkuji vedoucímu práce Ing. Tomáši Kašpárkovi za cenné rady a náměty, které mně pomohly při zpracování této práce.

Obsah ..... strana

1. Úvod a cíl práce

Tato bakalářská práce se zabývá paralelním řešením problému obchodního cestujícího (anglicky Travelling Salesman Problem, dále jen TSP) na SMP. Úloha TSP patří mezi nejznámější NP-úplné úlohy. I přesto, že problém lze velmi jednoduše interpretovat a je znám již dvě století, stále neexistuje jeho účinné řešení.

První část této práce si klade za cíl seznámit s TSP obecně. Problém souvisí s hledáním hamiltonovských kružnic v grafech. Je zde načrtnut matematický základ a definice pomocí n měst. Zmíněno je také, že tento problém patří do skopiny NP-úplných úloh a některé varianty TSP.

Dále jsou probrány známé heuristické postupy, jak řešit tento problém. Jmenovitě je možné podívat se v obsahu na podkapitoly kapitoly č. 3.

Kapitola č.4 pak obsahuje postup práce spolu s některými použitými pomůckami.

Algoritmy byly testovány, zda jejich teoretické časové a paměťové složitosti odpovídají praxi. Výsledky tohoto snažení jsou pak v kapitole č. 5.

Práce obsahuje krátký úvod a seznámení se systémem OpenMP. Způsob práce s OpenMP a jeho základní popis je uveden v kapitole č. 6.

Hledání vhodných kritérií algoritmů pro paralelní zpracování je uvedeno v kapitole č. 7.

Součástí této práce je také implementace dvou vybraných řešení v závislosti na jejich vhodnosti pro systémy se sdílenou pamětí.

Cílem této práce je analýza efektivnosti a škálovatelnosti navrženého paralelního řešení v závislosti na velikosti vstupního problému a počtu využitých procesorů.

Tuto práci jsem si zvolil, protože mě zaujala problematika obchodního cestujícího.

V závěru je shrnutí dosažených výsledků.

2. O problému obecně

2.1 Definice problému obchodního cestujícího

Hledání řešení TSP souvisí s teorií grafů. Konkrétně pak s hledáním hamiltonovské kružnice v grafu.

a) Teorie grafů - hamiltonovské cykly

Problém obchodního cestujícího vyžaduje nalezení nejkratší cesty, která je určena k návštěvě souboru měst a návratu do počátečního bodu. Neorientovaná úloha může být uváděna v teorii grafů následovně. Nechť $G = (V,A)$ je graf, kde $V = \{v_1, ..., v_n\}$ je vrchol (nebo uzel) a $A = \{(v_i, v_j) | v_i, v_j \in V, i \neq j\}$ je hrana s nezápornými náklady (nebo vzdálenostmi) matice $C = (c_{ij})$ spojená s A. TSP spočívá v určení minima nákladů hamiltonovského cyklu v úloze grafu. (Pokorná, P., 2008)

Souměrné implicitní používání neorientované hrany, spíše než orientovaného oblouku, může být také výslovně stanovené vztahem $c_{ij} = c_{ji}$. Úloha TSP patří mezi NP-úplné úlohy.

b) Definice obchodního cestujícího pomocí n měst

TSP je možno prezentovat takto. Obchodní cestující má za úkol navštívit n různých měst a vrátit se do místa odkud vyšel. Je potřeba stanovit nejkratší spojení cest, které má obchodní cestující navštívit, aby bylo možné jasně určit, že daná trasa je nejkratší možná. Obchodník má navštívit každé město právě jednou.

Protože kterákoliv permutace n měst přináší možné řešení, musí být n! měst ohodnoceno ve vyhledávacím prostoru. (Pokorná, P., 2008).

n	Počet různých cest
3	1
5	12
7	360
10	181440
15	43589145600
20	60822550204416000
25	3102242008666200000000000
30	442088099686985000000000000000
35	14761639951980200000000000000000000000
40	1019894104059870000000000000000000000000000000
45	132913578739422000000000000000000000000000000000000000
50	3041409320171340000000000000000000000000000000000000000000000000

Tabulka 2.1 Složitost problému

2.2 Úlohy NP

NP je zkratka pro nedeterministický polynomiální čas. Je to množina problémů, které se řeší v polynomiálně omezeném čase. Tyto problémy lze popsat tak, že pro jejich vyřešení v polynomiálním čase by bylo zapotřebí při každém rozhodování vědět, kterou cestou se vydat, aby cesta byla nejkratší.

Dále lze o těchto úlohách obecně říci, že jejich řešení se obtížně hledá, avšak je možné snadno ověřit správnost řešení. Typickým příkladem je pak hledání kružnice v grafu, která prochází všemi body grafu (tzv. Hamiltonovská kružnice). Existují různé algoritmy pro její nalezení. Ověřovací mechanismus pak zjišťuje, jestli se jedná o kružnici a jestli prochází všemi body.

NP-úplné problémy jsou ty z nedeterministicky polynomiálních problémů, na které lze polynomiálně redukovat všechny ostatní problémy z NP. NP-úplné úlohy jsou tedy v podstatě nejtěžší úlohy z NP. Existuje otázka, zda se třída P = NP. K takové rovnosti by došlo v okamžiku, kdy by byl nalezen deterministický polynomiální algoritmus pro nějakou NP-úplnou úlohu, protože pak by bylo možné řešit všechny nedeterministicky polynomiální problémy v polynomiálním čase. Většina odborníků však předpokládá, že se P ≠ NP. Jedná se o jeden ze sedmi problémů tisíciletí, které vypsal Clay Mathematics Institute v roce 2000 a nabízí za vyřešení jeden milion dolarů.

Hledání Hamiltonovské kružnice a tedy i TSP spadají do množiny NP-úplných problémů.

2.3 Varianty TSP

Symetrická úloha - TSP má symetrickou matici nákladů (graficky jde o neorintovaný graf)

Asymetrická úloha - TSP má nesymetrickou matici nákladů (graficky jde o orientovaný graf), existují různá ohodnocení vzdáleností mezi dvěma body (městy)

3. Algoritmy

Tato kapitola obsahuje přehled nejznámějších algoritmů, kterými lze řešit TSP. U každého algoritmu lze nalézt stručný popis, časovou a paměťovou složitost.

Časová i paměťová složitost je vyjádřena pomocí tzv. *asymptotické složitosti*. Tato složitost určuje, jak se bude náročnost algoritmu měnit v závislosti na změně velikosti vstupní informace. V tomto případě na změně velikosti vstupní instance měst.

V každé tabulce jsou uvedeny průměrné hodnoty z provedených experimentů s kódy jednotlivých algoritmů. Uvedená naměřená paměťová náročnost je vždy maximální naměřená hodnota.

Pro testování algoritmů bylo použito pět instancí měst z veřejné knihovny *TSPLIB*.

U provedených testů nebyl brán zřetel na to jestli algoritmus dává řešení optimální nebo blízké optimálnímu, jako spíše na časovou a paměťovou složitost.

Tabulky uvádí průměry naměřených časů, ve kterých algoritmus nalezl řešení. Dále je v nich uváděna maximální paměťová složitost algoritmů, která byla měřena pomocí linuxových příkazů příkazu *valgrind* a *top*.

Počet měst v jednotlivých TSPLIB souborech

	eil101	**gil262**	**rat783**	**rl1304**	**pr2392**	**rl5915**	**usa13509**
počet měst	101	262	783	1304	2392	5915	13509

Tabulka 3.1 Použité *TSPLIB* instance

3.1 Mravenčí kolonie

Anglicky *Ant Colony Optimalization(ACO).* Tento algoritmus je inspirován skutečnými živými mravenci. Tato metoda je založena na algoritmu rojové inteligence. Tu lze chápat jako systém decentralizovaných jednotek, které spolupracují navzájem a zároveň ovlivňují okolní prostředí. Jednotliví mravenci mezi sebou mohou komunikovat přímo nebo nepřímo působením na prostředí.

Mravence lze považovat za sociální druh. To lze specifikovat tak, že každý jedinec přispívá celku tak, že pracuje pro jeho zachování.

Při hledání potravy mravenci používají pro navigaci speciální feromon, který zanechávají na místech, kterým prošli. Tedy pokud byla potrava nalezena v určitém směru na určité cestě, pak je po této cestě zanechán tento feromon v odpovídajícím množství. Mravenci se při rozhodování řídí podle toho, jak velké množství feromonu je na které cestě a po určité době je pak zvolena tato cesta a je upřednostňována do té míry, že ostatní varianty zaniknou. To je umožněno tím, že feromon z cest postupem času mizí.

Podle výsledků uvedených v této zprávě nachází v porovnání s jinými algoritmy optimální řešení nebo řešení velmi kvalitní (Pokorná, P., 2008).

Bylo experimentováno s kódem ze zdroje (Stuetzle, T., 2004).

Časová složitost

$O(N)$

Paměťová složitost

$O(N^2)$

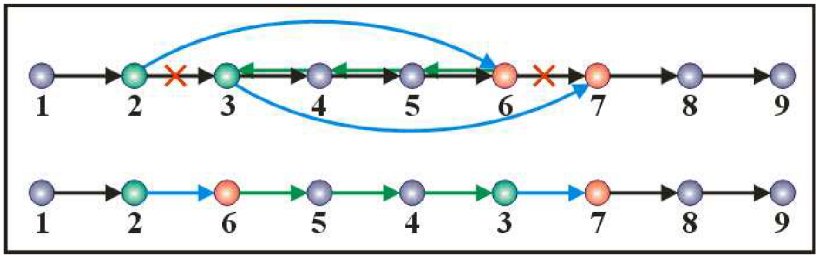
Časová a paměťová složitost

	eil101	**gil262**	**rat783**	**rl1304**	**pr2392**	**rl5915**	**usa13509**
čas (s)	0,02	0,07	0,47	1,3	4,12	26,5	
paměť (kB)	1560	4846	22420	50597	143629	725962	

Tabulka 3.2 ACO – výsledky

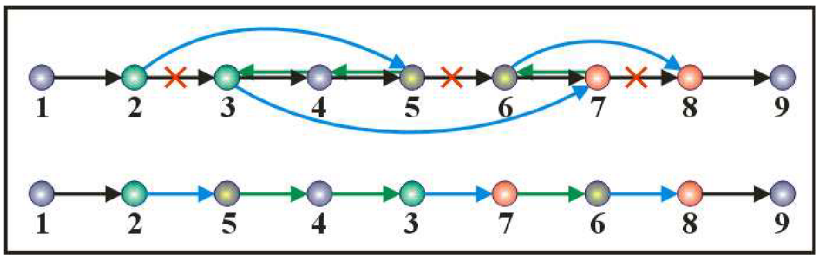
3.2 2-Opt a 3-Opt

Tyto algorimtmy spočívají v hledáních lokálních optimálních řešení. Tzv. dvoj- a troj-záměna hran.



Obrázek 1 Princip 2-Opt

Dvoj-záměna spočívá v nalezení takových dvou uzlů, mezi kterými existuje hrana a současně existuje hrana mezi nimi a jejich následníky (v našem případě se jedná o uzly 2,3). Jestliže jsou nalezeny dva takové uzly – dokončí se dvoj-záměna zobrazeným způsobem.



Obrázek 2 Princip 3-Opt

(obrázky č. 1 a č. 2 převzaty z
http://www.volny.cz/martin.smetana/skola/prace/paa/uloha5/tsp.html)

Troj-záměna hledá celkem tři uzly. Dle obrázku: hledáme takový uzel 2, ze kterého vede hrana do uzlu 5, který je v nalezeném řešení dál. Z následníka prvního uzlu, tedy z uzlu 3 musí vést hrana do uzlu 7 a konečně musí existovat hrana z následníka uzlu 5, uzel 6, ze kterého musí existovat hrana do následníka uzlu 7, tedy uzel 8.

Lokální prohledávání 2-Opt a 3-Opt je využíváno u většiny dalších zde prezentovaných algoritmů. Jako například LKH, simulovane žíhání, Tabu Search, genetické algoritmy.

Jedna z možností jak řešit TSP je pak spouštět lokální prohledávání iterovaně. Takovému přístupu se říká Iterated Local Search(ILS).
Bylo experimentováno s kódem ze zdroje (Paradiseo team, 2009).

Časová složitost

2-opt $O(N^2)$

3-opt $O(N^3)$

Paměťová složitost

2-opt $O(N^2)$

3-opt $O(N^3)$

Časová a paměťová složitost

	eil101	gil262	rat783	rl1304	pr2392	rl5915	usa13509
čas (s)	8,23	66,25	1238,77	5602,65	37480		
paměť (kB)	312	935	3116	9347	24926	46736	

Tabulka 3.3 ILS - výsledky

3.3 Tabu search

Metoda začíná s „počátečním“ - náhodným řešením. K nalezení optimálního řešení pak používá algoritmy pro vyhledávání lokálního optima 2-opt. K určení, zda je nalezená cesta nejlepší, se ověřuje délka celkové cesty, respektive zjišťuje se, zda je nalezená cesta kratší než ta předchozí. Hledání řešení pokračuje typicky do té doby, dokud se neprovede určitý počet běhů. Následně je vybráno to řešení, které je nejkratší.

Jedním z důvodů omezené efektivity začínat vždy s náhodným řešením je to, že nepočítá s možností, že se budou lokálně optimální řešení shlukovat, to znamená, pro jakékoliv dané lokální optimum pak bude platit, že může mít lepší řešení někde poblíž. Pokud by toto byla pravda, pak by bylo lepší začít znovu od řešení právě nalezeného, spíš než začínat znovu s úplně náhodným řešením. A to je podstata toho co Tabu search dělá. Obecná strategie je vždy udělat nejlepší nalezený pohyb, i když to znamená, že se právě nalezené řešení zhorší (tzv. *uphill* pohyb). Takže za předpokladu, že všichni sousedé právě nalezeného řešení jsou prověřeni při každém kroku, Tabu search alternuje mezi hledáním lokálního optima a identifikací nejlepšího sousedního řešení, které je pak použito jako počáteční řešení dalšího optimalizačního kroku. Pokud bychom ovšem prováděli pouze tento postup, hrozilo by podstatné nebezpečí, že tento nejlepší pohyb z tohoto „nejlepšího sousedního řešení“ by nás přivedlo zpátky k lokálnímu optimu, které jsme právě opustili nebo k některému nedávno navštívenému řešení. A to je místo, kde přichází na řadu tabu v Tabu search. Informace o nejposledněji navštívených řešeních se uchovávají v jednom nebo více tabu listech a tato informace je použita k diskvalifikaci nových pohybů, které by zmařili práci těchto právě provedených kroků (Johnson D. S., McGeoch L., A., 1997)
Bylo experimentováno s kódem ze zdroje (Paradiseo team, 2009).

Časová složitost

$O(N^3)$

Paměťová složitost

$O(N^3)$

Časová a paměťová složitost

	eil101	gil262	rat783	rl1304	pr2392	rl5915	usa13509
čas (s)	5,10	34,00	581,34	2246,47	15625,62	53922,32	
paměť (kB)	312	3116	6231	15579	46736	274183	

Tabulka 3.4 TS - výsledky

3.4 Lin-Kernighan

Tento algoritmus je stále považován za jeden z nejefektivnějších algoritmů pro řešení TSP. V podstatě se jedná o generalizaci algoritmů 2-opt a 3-opt. Konkrétně tak, že se specifikuje tzv. x-záměna, to znamená, že se zamění x hran v grafu namísto dvou nebo třech, jak je tomu u 2-opt a 3-opt. Při každém kroku algoritmu se pak rozhoduje kolik takovýchto záměn je potřeba udělat, aby byla výsledná cesta kratší.

Oproti algoritmu 3-opt poskytuje Lin-Kernighan kvalitnější řešení. Konkrétně se jedná průměrně o třetinu lepší řešení (Johnson D. S., McGeoch L., A., 1997).
Experimentováno s kódem ze zdroje (Helsgaun, K., 2009).

Časová složitost

$O(N^{2,2})$

Paměťová složitost

$O(N)$

Časová a paměťová složitost

	eil101	gil262	rat783	rl1304	pr2392	rl5915	usa13509
čas (s)	0	0,8	10,2	35,3	437,9	4287,5	
paměť (kB)	1591	2489	5250	11020	13901	27836	

Tabulka 3.5 LKH výsledky

3.5 Simulované žíhání

Simulované žíhání potřebuje nějaké počáteční řešení, které bude postupně vylepšovat, ideálně hamiltonovskou kružnici. Zatímco každé další kroky tabu search jsou řízeny striktně deterministicky, spoléhá simulované žíhání mnohem více na náhodu. Hlavní rozdíl od ostatních přístupů je, že simulované žíhání zkoumá další možné cesty v náhodném pořadí, taková cesta je přijata pokud je lepší nebo projde speciálním testem. Tento test zahrnuje kontrolní parametr „teplota“, který je v konkrétně v případě simulovaného žíhání postupně snižován. Pokud je zvolena cesta speciálním testem, znamená to v podstatě přijmutí horšího řešení než je to stávající. Čím se teplota snižuje, tím je méně a méně pravděpodobné, že by takové řešení bylo přijato.

Bylo experimentováno s kódem ze zdroje (Paradiseo team, 2009).

Časová složitost

$O(N^2)$

Paměťová složitost

$O(N^2)$

Časová a paměťová složitost

	eil101	**gil262**	**rat783**	**rl1304**	**pr2392**	**rl5915**	**usa13509**
čas (s)	0,9	0,97	1,28	1,3	1,98	4,83	13,32
paměť (kB)	312	935	3116	9347	24926	137092	713499

Tabulka 3.6 SA – výsledky

3.6 Genetické algoritmy

Jedná se o heuristické postupy. Principy evoluční biologie jsou základem hledání řešení. Pro hledání lepšího řešení se používají techniky napodobující evoluční procesy známé z biologie, jako například dědičnost, mutace, přirozený výběr, křížení.

Princip práce genetického algoritmu je postupná tvorba generací různých řešení daného problému. Při řešení se uchovává tzv. populace, jejíž každý jedinec představuje jedno řešení daného problému. Jak populace probíhá evolucí, řešení se zlepšují. Tradičně je řešení

reprezentováno binárními čísly, řetězci nul a jedniček, nicméně používají se i jiné reprezentace (strom, pole, matice, aj.).

V typickém modelu je populace na začátku simulace (v první generaci) složena z naprosto náhodných členů. V přechodu do nové generace je pro každého jedince spočtena tzv. *fitness* funkce, která vyjadřuje kvalitu řešení reprezentovaného tímto jedincem. Podle této kvality jsou stochasticky vybráni jedinci, kteří jsou modifikováni (pomocí mutací a křížení), čímž vznikne nová populace. Tento postup se iterativně opakuje, čímž se kvalita řešení v populaci postupně vylepšuje. Algoritmus se obvykle zastaví při dosažení postačující kvality řešení, případně po předem dané době (Wikipedia, 2009).

Při hledání zdrojů pro otestování tohoto algoritmu, bylo zjištěno, že je využíván široce k demonstračním a výukovým účelům. Z několika zkoušených programů, které řeší TSP pomocí GA se zdá, že se nepoužívají pro řešení rozsáhlejších problémů. Jedním z důvodů, který vede k tomuto závěru je, že většina těchto řešení nepodporuje větší vstupní instanci než 1000 měst. Přesto bylo provedeno několik testů. Na rozdíl od paměťové složitosti, která se změří velmi snadno, je určení časové ne zcela zřejmé. Důvodem je povaha algoritmu, který pracuje na základě postupného vývoje. Tedy se sám od sebe nezastaví a je na uživateli, kdy rozhodne, že uvedené řešení je pro něj postačující. U velmi malých instancí je zřejmé, kdy algoritmus nalezl řešení optimální nebo optimálnímu velmi podobné.

Bylo experimentováno s kódem ze zdroje (Lalena, M., 1995)

Časová složitost

$O(N^2)$

Paměťová složitost

$O(N^2)$

Časová a paměťová složitost

	eil101	**gil262**	**rat783**	**rl1304**	**pr2392**	**rl5915**	**usa13509**
čas (s)	240	1380,18	3420,58	6800			
paměť (kB)	62314	118397	302224	514093			

Tabulka 3.7 GA - výsledky

4. Materiál a metodika

V práci byla testována časová a paměťová složitost algoritmů pomocí vybraných počítačových programů z internetových zdrojů.

Byly použity různé algoritmy z různých internetových zdrojů. Hlavním kritériem výběru byla potencionální využitelnost algoritmu pro daný typ řešení. I přesto, že byly zdrojové kódy z různých zdrojů, byla jejich funkčnost dostačující.

Pokud to bylo z časových důvodů možné, pak byl každý algoritmus spuštěn na dané *TSPLIB* instanci 10x a poté byl vypočítán průměr z těchto hodnot.

Pokud to nebylo možné, tzn. čas zpracování byl příliš dlouhý, pak byl program spuštěn alespoň několikrát, tak, aby bylo možné formulovat alespoň přibližné závěry. Pokud totiž program opakovaně běžel delší dobu a odchylka délky zpracování byla opakovaně minimální – byla přijata hypotéza, že při dalším spuštění budou výsledky obdobné.

Výsledky testování jsou uvedeny jednotlivě u každého algoritmu - pro daný algoritmus a v následující kapitole pak souhrnně. Souhrnné výsledky – uvedené v závěru – umožňují porovnání algoritmů.

Byl zpracován popis způsobu práce s OpenMP a jeho základní chování (kapitola č. 6).

Dále práce obsahuje hledání vhodných kritérií algoritmů pro paralelní zpracování.

V průběhu práce byly použité programy doplněny o OpenMP direktivy, které umožnily paralelní zpracování programu. Často nebylo zcela zřejmé, do kterých míst kódu by bylo vhodné vložit požadované direktivy, což vyžadovalo podrobnější studium zdrojových kódů.

Po vložení direktiv byly sledovány efekty, které změny přinesly.

Použitý hardware

1. Všechny testy byly provedeny na procesoru Intel(R) Pentium(R) D CPU 2,8 GHz s 3 GB RAM.

Použitý software

1. Použitý operační systém byl Kubuntu 7.04 „Feisty Fawn“
2. Použitý překladač gcc, g++ (verze 4.2.4 (Ubuntu 4.2.4-1ubuntu3)
3. Pro otestování genetických algoritmů byl použit C# - kód spuštěný v linuxu pomocí MonoDevelop.
4. Linx profiler pro zjištění bližších informací o použitém kódu – gprof.

5. Vyhodnocení algoritmů

Cílem popisu algoritmů a experimentování s nimi bylo provést toto vyhodnocení a zjistit, který z nich bude vhodný pro implementaci v systému se sdílenou pamětí.

Prázdná místa ve výsledcích jsou testy, které nebylo možné provést. V případě ACO se jedná o omezení použitého kódu, který nepřijímá města větší než 6000. U ostatních případů není nutné další instance testovat, protože je již zjevné, jakým směrem se bude křivka ubírat a z časových důvodů nebylo možné tyto testy zcela dokončit.

Časová a paměťová složitost

	Časová složitost	**Paměťová složitost**
ACO	$O(N)$	$O(N^2)$
ILS	$O(N^2)$ - $O(N^3)$	$O(N^2)$ - $O(N^3)$
TS	$O(N^3)$	$O(N^3)$
LKH	$O(N^{2.2})$	$O(N)$
SA	$O(N^2)$	$O(N^2)$
GA	$O(N^2)$	$O(N^2)$

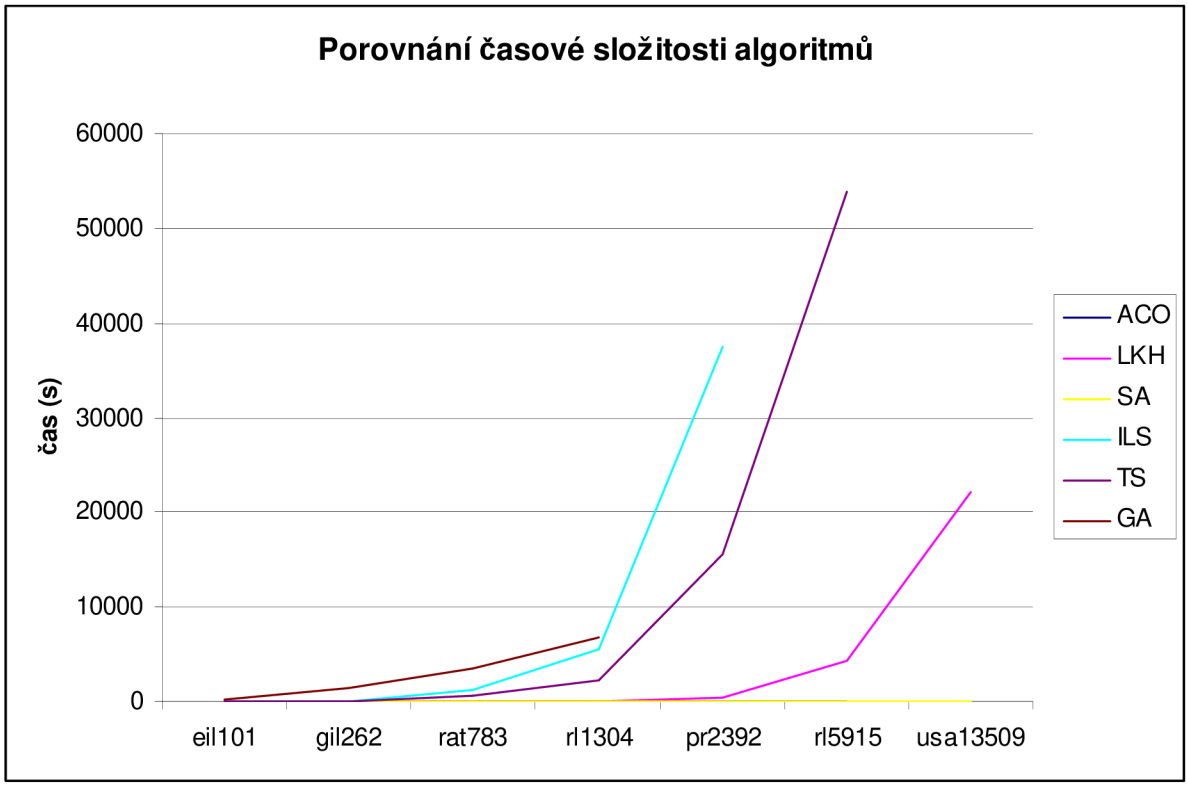
Tabulka 5.1 Porovnání složitostí algoritmů

Porovnání časové složitosti (sekundy)

	eil101	**gil262**	**rat783**	**rl1304**	**pr2392**	**rl5915**	**usa13509**
ACO	0,02	0,08	0,47	1,3	4,12	26,5	
ILS	8,23	66,25	1238,77	5602,65	37480		
TS	5,1	34,00	581,32	2246,47	15625,62	53922,32	
LKH	0	0,8	10,2	35,3	437,9	4287,5	22072,27
SA	0,9	0,97	1,28	1,3	1,98	4,83	13,32
GA	240	1380,18	3420,5	6800			

Tabulka 5.2 Empirické výsledky časových složitostí

Tabulky č. 5.1 a 5. 2 demonstrují korespondenci nastudované teorie s naměřenými empirickými výsledky.



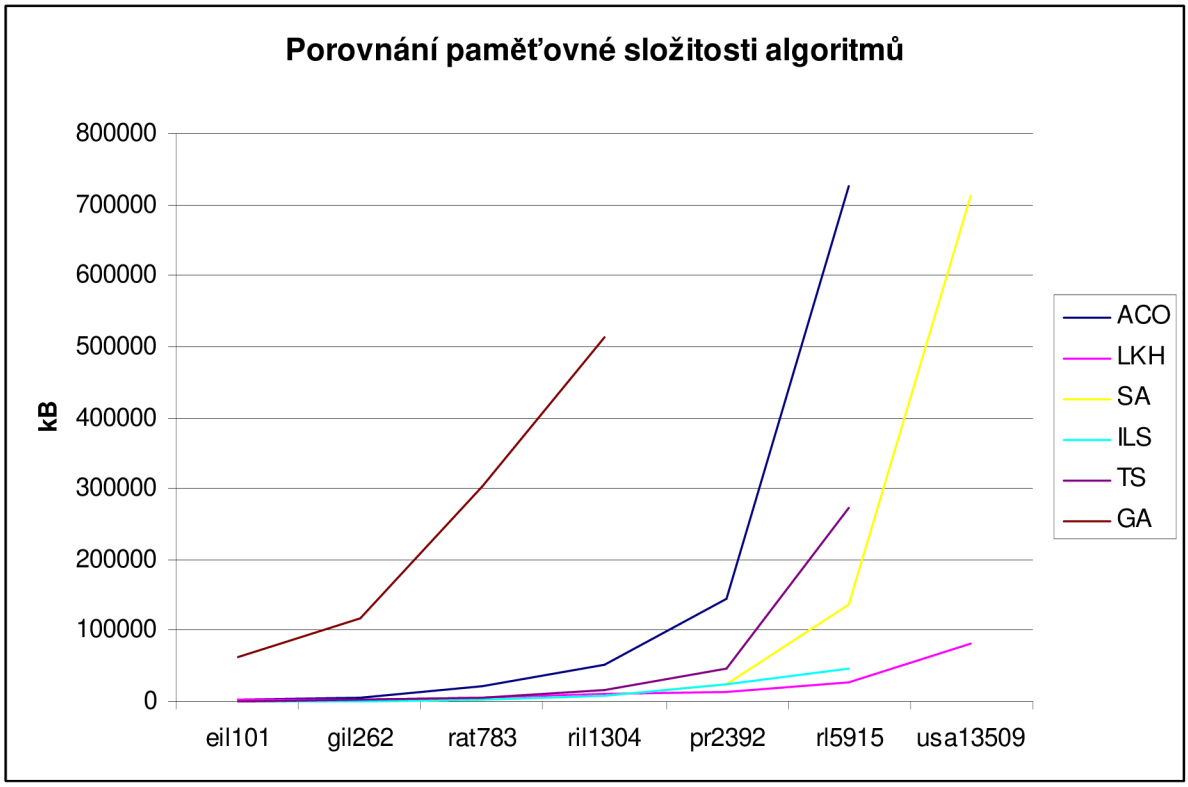
Graf 5.1 Grafické znázornění hodnot z tabulky č. 5.2

Porovnání paměťové složitosti (kB)

	eil101	**gil262**	**rat783**	**rl1304**	**pr2392**	**rl5915**	**usa13509**
ACO	1560	4846	22420	50597	143629	725962	
ILS	312	935	3116	9347	24926	46736	
TS	312	3116	6231	15579	46736	274183	
LKH	1591	2489	5250	11020	13901	27836	81454
SA	312	935	3116	9347	24926	137092	713499
GA	62314	118397	302224	514093			

Tabulka 5.3 Empirické výsledky paměťových složitostí

Z tabulky 5.3 vyplývá, že ani u paměťové složitosti použitých algoritmů nebyly zjištěny zásadní nesrovnalosti.

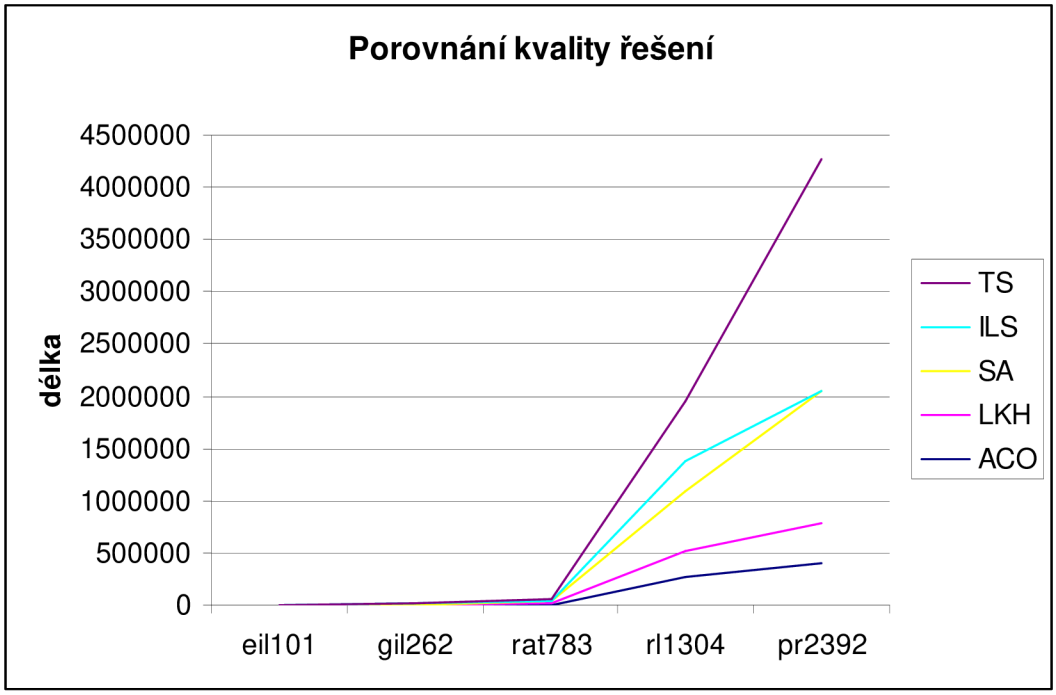


Graf 5.2 Grafické znázornění hodnot z tabulky č. 5.3

Porovnání kvality nalezeného řešení

	eil101	**gil262**	**rat783**	**rl1304**	**pr2392**	**rl5915**	**usa13509**
ACO	649,5	2483,9	9357,5	269977,9	408469,1	608120	
ILS	657,6	2574	9803	283048			
TS	679	2652,7	9902,5	567439	2216550		
LKH	629	2379,7	8806,5	252948	378942	568272	
SA	693	2681,7	13122	573082,9	1258511	5377522	

Tabulka 5.4 Empirické výsledky nalezené kvality řešení



Graf 5.3 Grafické znázornění hodnot z tabulky č. 5.4

Nalezení kvalitního řešení je jedním z palčivých témat problému obchodního cestujícího a proto si nedovolím vynechat, alespoň srovnání nalezených cest. Bystré oko si povšimne, že ve srovnání chybí problém rl5915. Za prvé, u všech algoritmů nebylo nalezeno řešení a za druhé algoritmus SA má pro tuto instanci tak velké číslo, že při jeho použití zkresluje celý graf nežádoucím směrem. Toto porovnání také neobsahuje GA, které nebylo bohužel možné v rámci testovaného zdrojového kódu správně ověřit.

U každého algoritmu, který řeší problém obchodního cestujícího stojí za úvahu jeho případné použití v praxi, které bude v extrémech vypadat zhruba následovně

1. Vrtání děr do destičky, ve které budou vždy na jiném místě a jen je potřeba je tam nějak udělat a rychle jít na další
   (algoritmus musí hledat řešení v řádu sekund, minut)
2. případ kdy se hledá vzor pro výrobu něčeho co bude podle tohoto vzoru vyráběno stejným způsobem stále stejně a jen jednou za dlouhou dobu bude potřeba vzor přepočítat
   (algoritmus může hledat řešení v řádu hodin, dnů, týdnů)

Z tohoto pohledu na věc tabulka porovnání časových složitostí mluví za vše, pokud příliš neuvažujeme kvalitu řešení.

Shrnutí

Algoritmy ACO a SA se za testovaných podmínek ukázaly jako velmi rychlé.

LKH, jak vyplývá z prostudovaných materiálů, je jeden z nejlepších algoritmů pro řešení TSP. Empirické výsledky potvrzují, že se jedná o střední cestu mezi algoritmy, co se týká časové a paměťové náročnosti.

ILS se zdá o něco náročnější než bylo předpokládáno. Z několika zdrojů je na něj odkazováno jako na velmi rychlé a kvalitní výsledky poskytující řešení. V testovaných případech má relativně malou spotřebu paměti, ale už velmi brzy velkou časovou náročnost.

Tabu Search se ukázalo také jako velmi časově náročné, i když podobně jako ILS náročnost na paměť není nijak extrémní.

Genetické algoritmy se ukázaly již velmi brzy velmi náročné na použitou paměť a pro větší instance nepoužitelné.

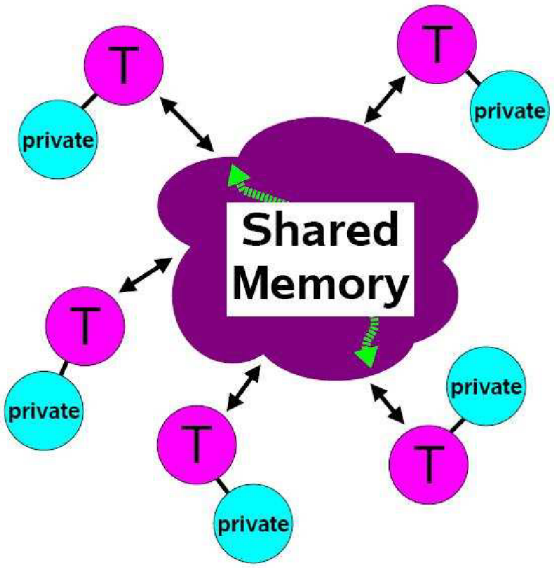
6. OpenMP

Náměty pro tuto kapitolu byly čerpány ze zdroje (Ruud van der Pas, 2005)(Seung-Jai M., 2008).

OpenMP (Open Multi Processing) je jednou z odpovědí na moderní vícejádrové technologie, se kterými přichází paralelní zpracování dat a práce se sdílenou pamětí.

Je to v podstatě standard pro práci se systémy se sdílenou pamětí tzv. SMP.

Princip systému se sdílenou pamětí může být následující:



- data mohou být sdílená (shared) nebo soukromá (private)
- sdílená data jsou přístupná všem vláknům
- soukromá data jsou přístupná pouze vláknům, které je vlastní
- přenosy dat jsou viditelné pro programátora

Obrázek 6.1 Princip systému se sdílenou pamětí

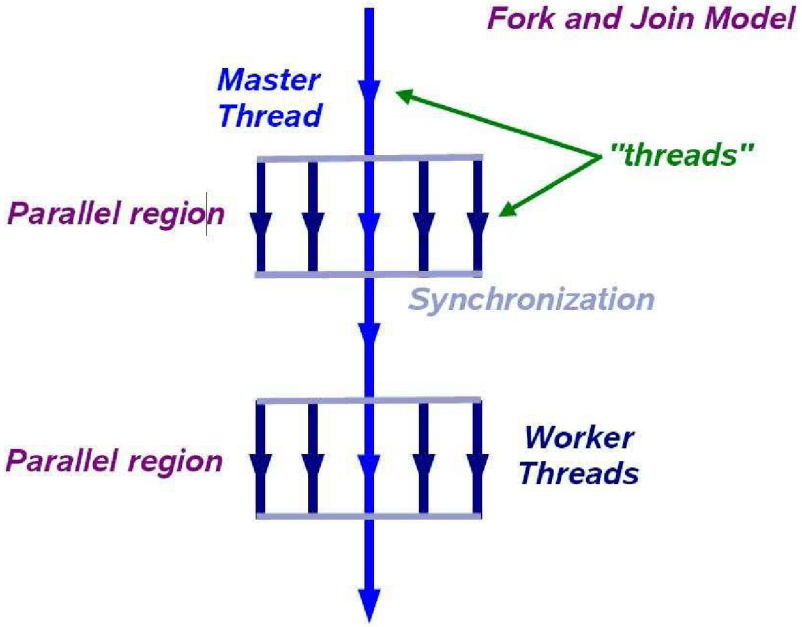
Podporované programovacími jazyky jsou C, C++ a Fortran. (OpenMP, 2009)

OpenMP je především používané pro paralelizaci cyklů. Postup je potom takový, najít nejnáročnější cyklus a ten rozdělit mezi vlákna.

Při použití OpenMP je nutné použít hlavičkový soubor *omp.h*.

Princip paralelizace

Pro paralelizaci je využívána práce s vlákny, způsobem *fork and join* demonstrovaným obrázkem.



Obrázek 6.2 Fork and Join Model

(obrázky č. 6.1 a č. 6.2 převzaty z http://www.nic.uoregon.edu/iwomp2005/iwomp2005_tutorial_openmp_rvdp.pdf)

Při používání OpenMP direktiv se podporuje strukturované programování

- Strukturovaný blok je takový blok kódu, který má jeden vstup na začátku a jeden výstup na konci
- Jediné větvení, které je povoleno je *STOP* v případě Fortranu a *exit()* v C, C++

Hlavní část tvoří direktivy pro překladač.

- Pro C, C++ mají direktivy tvar

  #pragma omp *construct [clause [clause]…]*
- Pro Fortran, jedna z těchto variant

  C$OMP construct [clause [clause]…]

  !$OMP construct [clause [clause]…]

  *$OMP construct [clause [clause]…]

Pro práci s OpenMP lze využít tyto typy konstrukcí

1. *parallel* – vyznačení oblasti, která má být provedena paralelně
2. *for/DO, sections, ..* – pro sdílení práce
3. *shared, private, …* - nastavení práce s daty

4. *barrier, flush,…* - synchronizace
5. *omp_get_num_threads(), …* - runtime funkce a proměnné prostředí

Příklad typického použití OpenMP pro sdílení práce *for* cyklu

Seriový program

```
void main()
{
 int i, k, N=1000;
 double A[N], B[N], C[N];
 for (i=0; i<N; i++) {
  A[i] = B[i] + k*C[i]
 }
}
```

Paralelní program

```
#include “omp.h”
void main()
{
 int i, k, N=1000;
 double A[N], B[N], C[N];
#pragma omp parallel for
 for (i=0; i<N; i++) {
  A[i] = B[i] + k*C[i];
 }
}
```

Shrnutí

OpenMP je velmi vhodné pro psaní paralelních programů. Poskytuje kompaktní prostředí a přitom velmi použitelné pro programování systémů se sdílenou pamětí. Do sériově psaných programů mohou být přidány OpenMP direktivy, ve kterých zůstává původní kód téměř nezměněn. Takto napsané programy jsou přenosné na široké spektrum systémů.

7. Výběr algoritmů pro paralelní zpracování

V této kapitole je uveden zjednodušený průběh jednotlivých algoritmů v bodech, protože z tohoto porovnání se nejlépe určí, který bude nejvhodnější pro paralelní zpracování.

ACO

1. Inicializace
2. Konstrukce

   For každý mravenec k (aktuálně ve stavu t) *do*

   repeat

   vybrat z pravděpodobnosti stav *k* přesunutí *do*

   přidat vybraný pohyb ke *k*-tému mravenčímu tabu seznamu.
3. Aktualizace cesty

   For každý pohyb mravence *do*

   vypočítat změny feromonu

   aktualizovat cestu matice

   end for
4. Ukončovací podmínka

   If not (konec testování) *goto* bod 2.

ILS

1. Vytvoř počáteční řešení

   s_0 = vytvořit počáteční řešení
2. Optimalizuj počáteční řešení lokálním prohledáváním

   s* = spustit lokální prohledávání (s_0)
3. Hledej lepší řešení, dokud neplatí ukončující podmínka

   repeat

   s’ = najít odchylku(s*, history)

   s*‘= spustit lokální prohledávání(s‘)

   s* = akceptační kritérium (s*, s*‘, history)

   until (ukončovací podmínka)

pozn. s*… nalezené optimální řešení

TS

1. vygeneruj počátečení řešení
2. Hledej lepší řešení následujícím způsobem

while (ještě není ukončovací podmínka) *do*

zjisti N(s) (Neighbourhood)

zjisti T(s,k) (Tabu)

zjisti A(s,k) (Aspirant)

vyber nejlepší s' N(s,k) = {N(s) - T(s,k)}+A(s,k)

zapamatuj si s' pokud je lepší než předchozí řešení s* = s'.

end while

LKH

1. Vygenerovat náhodnou počáteční cestu s*
2. Vyber na cestě nevyzkoušenou možnost, pokud to není možné, jdi na bod 5
3. Najdi vhodné body pro následnou záměnu, tím je získána cesta s'
4. Pokud s' je lepší cesta než s_0, dosaď s* = s' a jdi na bod 2.
5. Pokud nebylo nalezeno žádné zlepšení pak přichází na řadu *backtracing*
6. Když algoritmus prověří všechny možnosti a nenalezne žádné nové zlepšení, pak algoritmus končí. Pokud je to potřeba, je možné vygenerovat nové náhodné počáteční řešení a jít na bod 1.

(pozn. všechny cesty sou považovány za nevyzkoušené, jakmile je nalezeno zlepšující řešení)

SA

1. Inicializovat stavy, teplotu, počáteční řešení
2. Hledat lepší řešení

   while (zbývá čas) *and* (řešení není dost dobré) *do*

   Vyber náhodného souseda

   Vypočítej jeho teplotu

   if (je lepší) *then*

   ulož zlepšení

   if (vydat se tímto směrem) *then*

   změň stav

   end while
3. Vrátit nalezené řešení

   return (nalezené řešení)

GA

1. Inicializace – vygeneruj náhodnou populaci *n* chrmomozomů
2. Fitness – vypočítej *fitness* funkci *f(x)* každého chromozomu *x* v populaci
3. Nová populace – Vytvoř novou populaci opakováním následujících kroku, dokud není vytvořena nová populace
   a. Výběr – vyber dva rodičovské chromozomy z populace podle *fitness* funkce
   b. Křížení – s určitou pravděpodobností křížit tyto rodiče a vytvořit jejich potomka. Pokud ke křížení nedojde, potomek je identická kopie rodičů.
   c. Mutace – s určitou pravděpodobností zmutuj v každé části chromozomu potomka
   d. Přijetí – umísti potomka v nové populaci
4. Nahrazení – Použij novou generaci pro další běh algoritmu
5. Ukončující podmínka
   a. pokud je naplněna skonči a vrať nejlepší řešení v současné populaci
   b. jinak jdi na bod 2

Shrnutí

Po tomto srovnání je jasné, že na paralelní zpracování se hodí algoritmy, které přirozeně provádějí nějaké hromadné akce, jako například konstrukce a především aktualizace cesty v případě ACO algoritmu, kdy se pro celou načtenou instanci počítají změny uloženého feromonu.

Dalším takto přirozeně vhodným je GA, kde se zejména výpočet nové populace jeví jako velmi vhodné pro paralelní zpracování.

V ostatních případech je vždy zlepšení více méně lokální a paralelizace se vysloveně nenabízí.

Nakonec byly vybrány pro paralelní zpracování algoritmy ACO a LKH, tedy jeden algoritmus, který je z principu vhodný a jeden z druhé skupiny.

8. Implementace

Tato kapitola popisuje, jakým způsobem bylo postupováno při implementaci OpenMP direktiv do seriového kódu.

Při výběru, kterou část kódu paralelizovat, bylo přihlédnuto také k tomu, jakou zprávu o něm podal program *gprof*. Jedná se o linuxový profiler. Tedy program pro zjištění, které části kódu jsou nejpoužívanější, respektive vykonávají se nejdelší dobu. Výsledky jsou k dispozici na přiloženém CD.

8.1 ACO

Inicializace

Po prozkoumání kódu programem ***gprof*** bylo zjištěno, že nejvíce času tráví program v bodě 1 -tedy inicializací. První myšlenkou tedy bylo, zkusit paralelně inicializaci. Program je pravděpodobně také proto navržen tak, že provede inicializaci jednou a poté může být hledáno řešení na tomto základě několikrát.

I přesto, že je v hlavní funkci, která provádí inicializaci - cyklus, který se dostatečně nabízí pro paralelizaci, nebylo ho možné paralelně zpracovat tak, aby se chování programu výrazně neprodloužilo. Po mnoha testech a úpravách zdrojového kódu bylo nutné dojít k závěru, že inicializaci ACO, alespoň v rámci tohoto kódu nebude možné provést.

Posledním ujištěním byl fakt, že jednoduchá paralelní nic nevykonávající konstrukce způsobila téměř 2x delší běh programu.

Konstrukce

Přes maximální snahu o paralelizaci kódu a opakované pokusy – nebylo nalezeno optimální řešení, které by bylo uspokojivé z hlediska časového přínosu řešení problému.

Aktualizace cesty

V tomto případě byla paralelizace poměrně velmi snadná a způsobila, že výsledný běh programu byl značně urychlen (více – viz. v další kapitole).

Zejména zajímavé stojí povšimnutí klauzule *schedule* pro *OpenMP* direktivu *for*, která obyčejně vypadá zhruba takto:

schedule(třída[,parametr])

možné třídy: static, dynamic, guided, runtime

parametr: určí rozdělení práce mezi procesory

Přičemž nastavení této klauzule zásadním způsobem ovlivní výpočetní rychlost algoritmu.

8.2 LKH

V tomto případě – po opakovaných pokusech – nebyla paralelizace úspěšná. Jedním z hlavních důvodů v tomto případě je větvení algoritmu v kritických místech funkce, která hledá další zlepšující řešení. Jak byl zmíněno výše, takovéto větvení OpenMP nepodporuje.

Paralelizaci toho algoritmu, který je principielně spíše sekvenční lze teoreticky řešit dvěma způsoby:

1. Rozdělit vstupní problém určitým způsobem a podproblémy pak řešit paralelně
2. Spustit algoritmus x-krát paralelně, prakticky bez jakéhokoliv zásahu do kódu, potom tedy smyslem není urychlení algoritmu, ale nalezení lepšího řešení rychleji

První řešení by se jistě dalo nějakým způsobem zařídit. Nakonec bylo rozhodnuto, že řešení pokud by mělo být provedeno kvalitně, vyžádalo by si čas na samostatnou bakalářskou práci.

Druhý způsob je velmi triviální a prakticky nevyžaduje žádný zásah do kódu. I když by se toto řešení pomocí OpenMP dalo udělat, bude snažší spustit prostě program 2x a nechat procesory ať si práci rozdělí samy.

9. Analýza navrženého řešení

Tato kapitola obsahuje analýzu, navrženého paralelního řešení. Snaží se najít odpověď na otázku zda je nějakým způsobem lepší než původní seriové řešení. Analýza spočívá ve spuštění programu sériově a paralelně na počítači s více jádry.

Použitý hardware byl popsán v kapitole č. 4 - viz Materiál a metodika.

Amdahlův zákon

Zrychlíme-li p% kódu s-krát, pak zrychlení bude

shora omezeno výrazem 100/[(100-p) + p/s]. (Furst, J., 2006)

	eil101	**gil262**	**rat783**	**rl1304**	**pr2392**	**rl5915**
ACO – sériově	0,0226	0,0777	0,4693	1,3	4,1222	26,5014
ACO – 2 jádra	0,0272	0,826	0,4505	1,22	3,771	24,41
rozdíl v %	-20,35%	-6,31 %	4,01 %	6,19 %	8,52 %	7,90 %

Tabulka 9.1 Porovnání sériového a paralelního spuštění

Z tabulky 9.1 vyplývá, že při použití paralelizace dochází k projevení Amdahlova zákona.

Graf 9.1 Grafické znázornění hodnot z tabulky č. 9.1

Shrnutí

Je nutné se zamyslet nad tím, zda navržené řešení je prospěšné nebo není. Je také potřeba si uvědomit, že tento algoritmus operuje v testovaných podmínkách velmi rychle. Maximálně v řádů desítek sekund. Pak lze očekávat, že zlepšení paralelizací nebude v řádu sekund až tak viditelný, jako by byl pro ještě větší instanci. Proto je zde uveden procentuelní rozdíl výsledků, který lépe vystihuje situaci. Jinými slovy, dobře je zde patrná účinnost Amdahlova zákona.

Důvod proč není otestována větší instance je už předem daný fakt použitého zdrojového kódu, který už na začátku neumožňoval větší vstup než 6000 měst. Toto omezení je pravděpodobně proto, že algoritmus má značné paměťové nároky. Jakým způsobem by se vyvíjela křivka paměťové zátěže lze ostatně dobře vidět v kapitole č. 5.

Režie a synchronizace paralelního zpracování při použití OpenMP direktiv je obecně velmi malá až zanedbatelná. Ale stojí za povšimnutí, že algoritmus je výrazně zpomalen paralelními konstrukcemi pokud je zadána velmi malá instance.

10. Závěr

Hlavním cílem této práce bylo naimplementovat známá paralelní řešení problému obchodního cestujícího a zjistit jak se budou chovat na systémech se sdílenou pamětí.

Už při vybírání algoritmů bylo zjevné, že algoritmus ACO bude ze svého principu dobře naimplementovaeltný pomocí OpenMP, alespoň co se týče funkce pro aktualizaci feromonových stop. A tak se také stalo.

Pokud by mělo být uvažováno zlepšit do budoucna chod tohoto programu, pak by možná stálo výhledově za zmínku paralelně zpracovat incializaci, která zabírá algoritmu většinu času. To v případě, že by měl být program zpouštěn mnohokrát za sebou s jinou instancí. Dalším potenciálním rozšířením je rozšířit program takovým způsobem, aby bylo možné zadat větší vstupní problém než 6000 měst.

Algoritmus LKH byl vybrán jako reprezentant skupiny algoritmů, které jsou svou povahou spíše sekvenční. Tedy ne přirozeně vhodné pro paralelní zpracování. Stalo se tak, že použitý sériový kód nebylo možné v daném časovém rozmezí paralelně zpracovat, protože taková úprava by si vyžádala více času než je rámec bakalářské práce. Protože je LKH jeden z nejlepších algoritmů pro řešení problému obchodního cestujícího, tak by stálo zde více než jinde, pokusit se tento problém v rámci další práce kvalitně paralelně zpracovat.

Nechtěným vedlejším produktem této práce je pěkné srovnání algoritmů řešících problém obchodního cestujícího. Zvláště pak jejich časové a paměťové složitosti. A to včetně teoretických poznatků i empirických výsledků, které tuto teorii dokládají.

OpenMP se ukázalo jako velmi jednoduché a přitom obecně velmi mocné. Je to právě tato jednoduchost, která je jedním z důvodů pro výsledky dosažené v této práci. Tedy výborné pro paralelizaci jednodušších cyklů. Prakticky nepoužitelné v případě, že je potřeba paralelně zpracovat hodně se větvící zdrojový kód.

Tuto práci jsem zpracoval rád. Dozvěděl jsem se mnohem více o různých postupech řešení problému obchodního cestujícího, porovnání těchto postupů a obohatil se o způsoby paralelně zpracovat sériově napsaný program - pomocí OpenMP direktiv.

Literatura

[1] NEŠETŘIL, J. *Teorie grafů*. Praha : SNTL, 1979.

[2] PLESNÍK, J. *Grafové algoritmy*. Bratislava : Veda, vydavateľstvo Slovenskej akadémie vied, 1983.

Elektronické zdroje

[1] FURST, Jiří. Úvod do OpenMP. *XXX* [online]. 2006 [cit. 2009-04-15]. Dostupný z WWW: <www.civ.cvut.cz/ESF/info/run1.php?did=44>.

[2] JOHNSON, David S., MCGEOCH, Lyle A. *The Traveling Salesman Problem: A Case Study in Local Optimization.* [online]. 1997 [cit. 2009-04-15]. Dostupný z WWW: <www.research.att.com/~dsj/papers/TSPchapter.ps>.

[3] HELSGAUN, Keld. *An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic* [online]. 2008 , September 2008 [cit. 2009-04-15]. Dostupný z WWW: <http://akira.ruc.dk/~keld/research/LKH/>.

[4] LALENA, Michael. *Traveling Salesman Problem Using Genetic Algorithms* [online]. 1995 [cit. 2009-04-15]. Dostupný z WWW: <http://www.lalena.com/AI/Tsp/>.

[5] PARADISEO TEAM. *Paradiseo* [online]. 2009 [cit. 2009-04-15]. Dostupný z WWW: <http://paradiseo.gforge.inria.fr/>.

[6] POKORNÁ, Petra. *Problém obchodního cestujícího pomocí metody Mravenčí kolonie*. [s.l.], 2008. 44 s. Univerzita Pardubic, Fakulta ekonomicko-správní. Vedoucí bakalářské práce Ing. Jan Panuš. Dostupný z WWW: <https://dspace.upce.cz:8443/dspace/bitstream/10195/29491/1/PokornaP_Problem%20obchodniho_JP_2008.pdf>.

[7] RUUD, van der Pas. *An Introduction Into OpenMP*. [online]. 2005 [cit. 2009-04-15]. Dostupný z WWW: <http://www.nic.uoregon.edu/iwomp2005/iwomp2005_tutorial_openmp_rvdp.pdf>.

[8] STUETZLE, Thomas. *ACO algorithms for the TSP* [online]. 2004 [cit. 2009-05-14]. Dostupný z WWW: <http://www.aco-metaheuristic.org/aco-code>.

[9] SEUNG-JAI, Min. *OpenMP Tutorial*. [online]. 2008 [cit. 2009-04-15]. Dostupný z WWW: <www.cri.purdue.edu/files/OpenMP.ppt>.

[10] SMETANA, Martin. *TSP - Problém obchodního cestujícího* [online]. 2003 [cit. 2009-04-15]. Dostupný z WWW: <http://www.volny.cz/martin.smetana/skola/prace/paa/uloha5/tsp.html>.

[11] *Wikipedia* [online]. 2009 , 20 May 2009, at 15:48 (UTC) [cit. 2009-04-25]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Genetic_algorithm>.

Obsah CD

1. Programy použité pro testování algoritmů
2. Upravené programy pro paralelní zpracování
3. Výsledky profileru *gprof*
4. Naměřené hodnoty při testování časové složitosti