Факултет техничких наука

Универзитет у Новом Саду

Рачунарски системи високих перформанси

Проблем трговачког путника: Паралелна имплементација и анализа

Аутор:
Маја Благић

Индекс:
E2 83/2022

12. јануар 2023.

Сажетак

У овом раду ће се анализирати проблем трговачког путника и његово рјешење путем *brute force* алгоритма. Алгоритам ће бити паралелизован користећи *OpenMP* парадигму. На крају ће се анализирати и поредити времена извршавања секвенцијалног и паралелног алгоритма.

Садржај

Списак изворних кодова

Списак слика

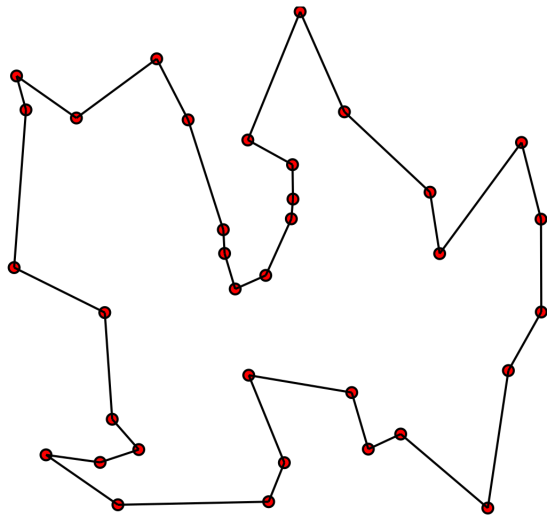
Списак табела

1 Увод

Проблем трговачког путника (енгл. *travelling salesman problem*) представља изазов проналажења најкраћег и најефикаснијег пута за скуп градова и растојања између њих. Формално речено за *N* раздаљина између сваког пара градова, треба пронаћи најкраћу руту која ће посјетити сваки град једном и вратити се у град из ког је почела. Метода *brute force* рјешава проблем тако што почетни град посматра као *град1* и затим генерише све пермутације преосталих *N*-1 градова и враћа пермутацију са најмањом цијеном путовања. Временска комплексности за рјешење је *O(N!)*. Циљ овог рада је паралелизација *brute force* алгоритма користећи *OpenMP* парадигму. [1] Репозиторијум на овом линку садржи приједлог паралелизације алгоритма. [1]

2 Проблем трговачког путника

Проблем трговачког путника поставља сљедеће питање: С обзиром на списак градова и даљина између сваког пара градова, који је најкраћи могући пут да се посјети сваки град тачно једном и да врати се у почетни град. Проблем представља НП-тешки (енгл. *NP-hard*) проблем [1] који је битан за теорijску рачунарску науку (енгл. *theoretical computer science*) [1]. Проблем је први пут формулисан 1930. године и један је од најинтензивнијих проблема у оптимизацији. Иако је проблем рачунски тежак, велики број хеуристика и алгоритама за рјешавање су познати, тако да су неки случаеви са десетинама хиљада градова ријешени. Када се проблем трговачког путника модификује постаје потпроблем у многим областима, као што је на прмјер ДНК секвенцирање (енгл. *DNA sequencing*) [1]. У овим аликацијама концепт града представља ДНК фрагмент, а концепт растојања између градова представља мјеру сличности између фрагмената. Описани проблем се такође појављује у астрономији када се посматрају многи извори и потрбено је оптимизовати вријеме помјерања телескопа између извора.



Слика 1: Рјешење проблема трговачког путника. Црна линија приказује најкраћи пут који повезује тачке.

3 Серијска имплементација brute force алгоритма за проблем трговачког путника

Алгортитам за проблем трговачког путника је поприлично једноставан јер представља обичну пермутацију градова са једним условом да први град мора увијек бити *град1*. Када се схвати да је свака рута представља једну пермутацију *brute force* алгоритам је сигурно подобно рјешење. Израчунаће се свака пермутација и цијена пута, а на крају ће се изабрати оптимално рјешење. Псеудокод за овај алгоритам се налази на слици 2.

Algorithm 1 Brute Force Serial TSP

Input: $city\_list$, $cost\_matrix$
$optimal\_cost \leftarrow \infty$
$optimal\_path \leftarrow null$
$city\_list \leftarrow list\ of\ cities\ excluding\ city_1$
while $next\_permutation(city\_list)$ **do**
  $temp\_cost \leftarrow get\_path\_cost(city\_list, cost\_matrix)$
  {Cost of travelling cities in order of cities in $city\_list$}
  if $temp\_cost < optimal\_cost$ **then**
    $optimal\_cost \leftarrow temp\_cost$
    $optimal\_path \leftarrow city\_list$ {with $city_1$ appended at start & end}
  end if
end while
Output: $optimal\_path$, $optimal\_cost$

Слика 2: Псеудокод brute force алгоритма

Функција `next_permutation(...)` провјерава да ли постоји још могућих пермутација градова. Када се *град1* узме за почетну тачку број пермутација постаје *N-1* а временска комплексност функције постаје *O(N-1)*. Од важности је и метода `get_path_ cost(...)` која рачуна цијену пута за сваку пермутације. Функција пролази кроз низ N градова како би се израчунала укупна цијена, те је њена временска комплексност *O(N)*. На крају се закључује да је формула времена извршавања алгоритма $O((N-1)! \times N) = O(N!)$. Имплементација описаног алгоритма у c++ програмском језику је приказана на изворном коду 1

```cpp
#include<bits/stdc++.h>
#include <ctime>
#include <chrono>

vector<int> tsp_serial(vector<vector<int>>&matrix)
{
    int n = matrix.size();

    int optimal_value = INF;
    vector<int>ans;

    vector<int>nodes;
    for(int i=1;i<n;i++)nodes.push_back(i);

    do
    {
        vector<int>temp = nodes;
        temp.push_back(0);

        temp.insert(temp.begin(),0);
        int val = find_path_cost(matrix,temp);
        if(val<optimal_value)
        {
            optimal_value = val;
            ans = temp;
        }

    }while(next_permutation(nodes.begin(),nodes.end()));

    return ans;
}
```

Изворни код 1: Имплементажија псеудо кода у c++ програмском језику

4 Паралелизација brute force алгоритма за проблем трговачког путника

У примјеру приказаном на изворном коду 2 креиран је паралелан блок. Блок садржи приватну промјенљиву `nodes` те свака нит има своју локалну инстанцу поменуте промјенљиве, док су дијељене промјенљиве `ans` и `optimal_value` заједничке за нити. Како бисмо паралелизовали *brute force* алгоритам, све пермутације градова су равномјерно распоређене између нити. У случају да број нити не подјели број пермутација без остатка, тај остатак ће бити додијељен посљедњој нити за обраду. Како би се осигурало да почетне пермутације буду другачије, оне ће се креирати у зависности од `id` нити путем методе `nth_permutation()`. Затим, остатак пермутација за сваку нит ће се израчунати у `do while` петљи. За сваку пермутацију се рачуна цијена, а затим се она пореди са дијељеном промјенљивом `optimal_value`. Ако нова пермутација има мању цијену од тренутне оптималне онда ће њена вриједност постати нова оптимална. Описани процес поређења и ажурирања цијене се дешава у критичној секцији. На овај начин осигурано је да само једна нит може да чита и по потреби ажурира `optimal_value`, те неће доћи до трке за подацима(енгл. *data race*)[1]. Поступак је исти за ажурирање дијењене промјенљиве `ans` која памти оптималну пермутацију.

```
vector<int> tsp_omp(vector<vector<int>>&matrix)
{
    int n = matrix.size();
    int optimal_value = INF;
    vector<int>ans;
    vector<int>nodes;
    long int k=fact[n-1];

    #pragma omp parallel private(nodes) shared(ans,optimal_value)
    {
    for(int i=1;i<n;i++)nodes.push_back(i);
    int num = omp_get_num_threads();
    int id = omp_get_thread_num();

    long int iter_per_thread= k/num;
    int extra = k%num;
    if (id<extra) {
        nodes = nth_permutation(nodes,(id)*(iter_per_thread+1));
        iter_per_thread=iter_per_thread+1;
    }
    else {
        nodes = nth_permutation(nodes,(id)*iter_per_thread+extra);
    }
    int i=0;
    do
    {
        vector<int>temp = nodes;
        temp.push_back(0);
        temp.insert(temp.begin(),0);
        int val = find_path_cost(matrix,temp);
        #pragma omp critical
        {
        if(val<optimal_value){
            optimal_value = val;
            ans = temp;
            }
        }
        i++;
        next_permutation(nodes.begin(),nodes.end());
    }while(i<iter_per_thread);
    }
    return ans;
}
```

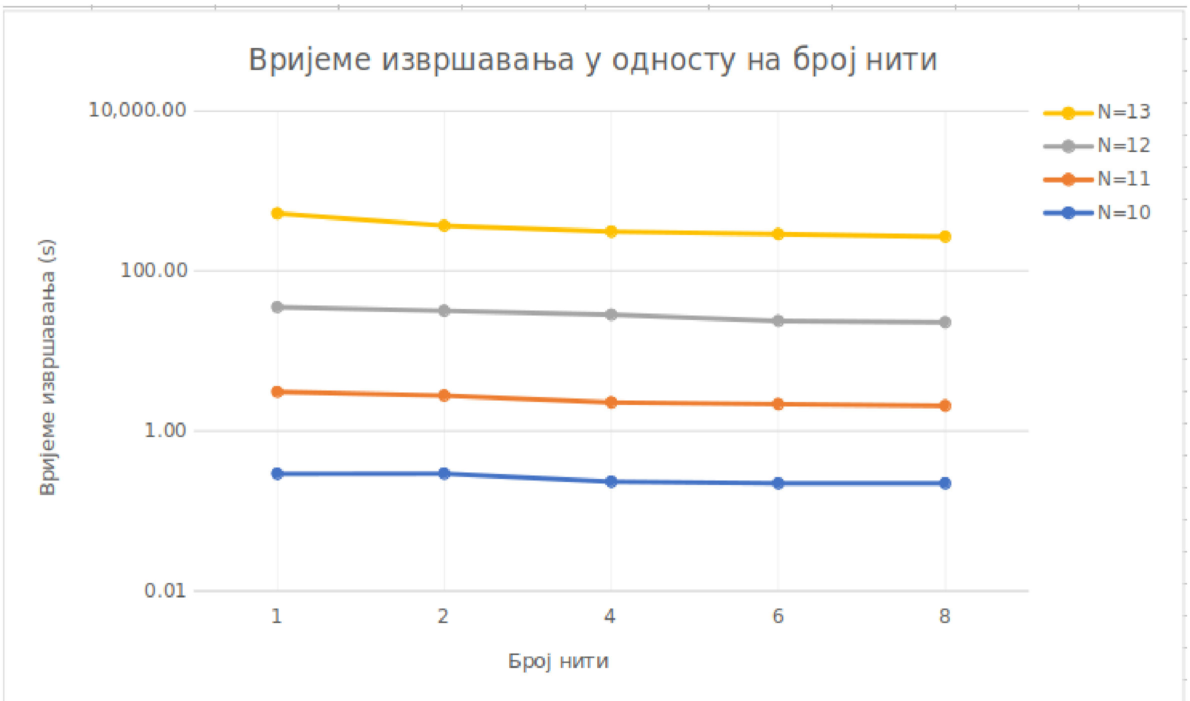

Изворни код 2: Имплементажија паралелног кода у c++ програмском језику

5 Анализа рјешења

Подаци добијени извршавањем паралелног алгоритма записани су у табели 1. У складу са њима график на слици 3 илуструје вријеме потребно за извршавање алгоритма и убрзасе које се постиже паралелном имплементацијом. Примјећује се да се вријеме извршавања паралеленог алгоритма смањује са повећањем броја нити које обрађују посао. Разлика у времену извршавања између два града је велика јер зависи од *N!*. Убрзање се повећава са порастом броја нити али тренд остаје сличан за све вриједности *N*.

N градова	1 нит	2 нити	4 нити	6 нити	8 нити
10	0.291	0.29224	0.23175	0.22148	0.2214
11	2.78114	2.46985	2.03499	1.93982	1.84333
12	31.96113	28.87815	26.0269	21.50501	20.6756
13	486.28221	335.2356	281.055	264.12	244.12

Табела 1: Измјерено вријеме извршавања алгоритма у односу на број нити и N градова.



Слика 3: Графички приказ времена извршавања алгоритма у односу на број нити и N градова.

6 Закључак

У овом раду описан је проблем трговачког путника као и начин за његово рјешавање путем *brute force* алгоритма. Основни фокус рада је био на имплементацији паралелног кода помоћу *OpenMP* парадигме. Описано рјешење је знатно убрзало алгоритам, а побољшане перформансе су приказане табеларно и путем графика. Рјешење би се могло доатно унаприједити комбинацијом *OpenMP* и *MPI* парадигми. Осим тога, могло би се проширити да проучава више разлитчитих хеуристика и алгоритама за рјешавање проблема трговачког путника. Новонастало рјешење би показало значај одабира алгоритма и паралелизације алгоритма за развој софтверског рјешења.

Библиографија

[1] Help on BibTeX entry types. http://nwalsh.com/tex/texhelp/bibtx-7.html. Accessed: 2015-03-12.