Факултет техничких наука

Универзитет у Новом Саду

Рачунарски системи високих перформанси

Оптимизација крушкаловог алгоритма за MST

Аутор:	*Индекс:*
Матија Максимовић	E2 33/2024
Урош Николовски	E2 38/2024
Мили Бован	E2 163/2024

5. август 2025.

Сажетак

Овај рад се бави паралелном имплементацијом и оптимизацијом Крушкаловог алгоритма за одређивање минималног разапињућег стабла (енгл. Minimum Spanning Tree, MST). Крушкалов алгоритам је похлепни алгоритам који итеративно гради MST додавањем ивица са најмањом тежином које не формирају циклус у графу. [2]

У раду су развијене две паралелне верзије алгоритма: једна која користи OpenMP библиотеку за вишејезгарне процесоре и друга која користи CUDA платформу за графичке процесоре (GPU). Циљ је био истражити потенцијал модерних паралелних архитектура за убрзање овог фундаменталног проблема.

Експериментални резултати показују значајна убрзања у односу на секвенцијалну верзију. Имплементација са OpenMP постигла је убрзање од 4.6 пута, док је CUDA имплементација постигла убрзање од 3.34 пута. Ови резултати демонстрирају да се Крушкалов алгоритам може ефикасно паралелизовати и да нуди запажене перформансе на савременом хардверу.

Садржај

Списак изворних кодова

1 Увод

Проблем одређивања минималног разапињућег стабла (енгл. *Minimum Spanning Tree*, MST) један је од фундаменталних проблема оптимизације у теорији графова и рачунарству. За дати повезан, неусмерен и тежински граф, циљ је пронаћи подграф који повезује све његове чворове у једно стабло, тако да је сума тежина свих ивица у том стаблу минимална. Овај проблем има широку примену у пракси, укључујући пројектовање ефикасних мрежа (телекомуникационих, рачунарских, транспортних), анализу биолошких података, као и алгоритме за кластеровање података.

Један од најпознатијих алгоритама за решавање овог проблема је Крушкалов алгоритам. То је похлепни алгоритам чија се методологија заснива на следећим корацима:

1. Креира се "шума" у којој је сваки чвор графа засебно стабло.
2. Све ивице из оригиналног графа сортирају се у неопадајућем редоследу према својим тежинама.
3. Пролази се кроз сортирану листу ивица. За сваку ивицу се проверава да ли повезује чворове који се већ налазе у различитим стаблима. Уколико повезује, та ивица се додаје у MST, а два стабла се спајају у једно. У супротном, додавање ивице би формирало циклус, па се она одбацује.
4. Алгоритам се завршава када све ивице буду обрађене или када MST садржи $V - 1$ ивица, где је V број чворова.

Временска комплексност Крушкаловог алгоритма је доминантно одређена кораком сортирања ивица. Уколико граф има V чворова и E ивица, комплексност сортирања је $O(E \log E)$. Операције провере циклова и спајања стабала, које се ефикасно имплементирају помоћу структуре података *Union-Find*, имају знатно мању комплексност, приближно $O(E \cdot \alpha(V))$, где је $\alpha(V)$ веома споро растућа инверзна Акерманова функција. Стога, корак сортирања представља главно "уско грло" и примарни је кандидат за оптимизацију.

У овом раду, фокус је стављен на паралелизацију управо овог најзахтевнијег корака. Представљена је паралелна имплементација Quicksort алгоритма са циљем да се убрза укупно време извршавања Крушкаловог алгоритма на модерним вишејезгарним процесорима.

2 Преглед постојећих решења

Проблем убрзања Крушкаловог алгоритма је предмет многих истраживања. Vitaly Osipov, Peter Sanders и Johannes Singler у свом раду [2] представљају Filter-Kruskal, модификовани алгоритам који избегава сортирање свих ивица. Њихов приступ

се заснива на филтрирању ивица које "очигледно" не могу бити део MST, чиме се смањује број ивица за сортирање.

Другачији приступ предлажу Haiming Li, Qiyang Xia и Yong Wang, који имплементирају *Two-branch Kruskal Algorithm* [1]. Њихова метода користи "пивот" вредност за поделу скупа ивица на два дела (ивице лакше и теже од пивота), након чега се Крушкалов алгоритам примењује паралелно на оба дела, а резултати се на крају спајају.

За разлику од наведених приступа који мењају логику селекције ивица, наш рад се фокусира на директно убрзање фундаменталног корака сортирања кроз паралелно извршавање, пружајући оптимизацију која се може применити на класичну верзију алгоритма.

3 Методологија

У овом поглављу ће бити детаљно описана методологија и имплементација решења у OpenMP и CUDA технологијама.

3.1 Генерисање графа и улазни параметри

Графови су генерисани насумично са 100 милиона ивица и 100 хиљада чворова и са максималном тежином од милион. У примеру 3.1 је приказана функција за генерисање ивица, где као параметре прослеђујемо број ивица, број чворова, те локацију низа где ће ивице бити смештене. Ивице су дефинисане као структура од 3 елемента: *почетни чвор, крајњи чвор, тежина*, где су почетни чвор и крајњи чвор тачке које спаја ивица, а тежина представља тежину те ивице. Након што смо генерисали ивице, можемо да пређемо на имплементацију алгоритма.

3.2 Секвенцијална имплементација

За алгоритам нам је потребно број ивица, број чворова, ивице, те резултујући низ који чине ивице које формирају MST.
Први корак алгоритма је сортирање ивица по тежини у растућем редоследу. За то се користи прилагођена верзија алгоритма *Quick Sort*, алгоритма за сортирање који се заснива на парадигми **подели и владај** (*Divide and Conquer*).

Класични *Quick Sort* алгоритам функционише у следећим корацима:

1. **Избор пивота**: Из низа се бира један елемент који ће бити основа за поређење (пивот).

2. **Партиционисање**: Сви елементи мањи од пивота премештају се лево од њега, а сви већи десно. Након тога, пивот је на својој коначној позицији.

```c
void create_edges(int num_of_edges, int num_of_nodes,
unsigned int edges[num_of_edges][3]) {
    srand(time(NULL));
    for (int i = 0; i < num_of_nodes - 1; i++) {
        edges[i][0] = i;
        edges[i][1] = i + 1;
        edges[i][2] = rand() % MAX_EDGE_WEIGHT + 1;
    }
    for (int i = num_of_nodes - 1; i < num_of_edges; i++) {
        edges[i][0] = rand() % num_of_nodes;
        edges[i][1] = rand() % num_of_nodes;
        while (edges[i][1] == edges[i][0]) {
            edges[i][1] = rand() % num_of_nodes;
        }
        edges[i][2] = rand() % MAX_EDGE_WEIGHT + 1;
    }
}
```

Изворни код 1: Креирање ивица

3. **Рекурзивна примена**: Алгоритам се рекурзивно примењује на леви и десни део низа (без пивота), док се низ не сортира у потпуности.

У најбољем и просечном случају, *Quick Sort* има временску сложеност $\mathcal{O}(n \log n)$, али у најгорем случају (на пример, ако је пивот увек највећи или најмањи елемент), сложеност се погоршава на $\mathcal{O}(n^2)$.

Да би се умањила вероватноћа најгорег случаја, наша имплементација користи напреднију стратегију избора пивота, као што можемо видети у примерима 3.2 и 3.2. Уместо да се пивот бира насумично или као средњи елемент, примењујемо следећи поступак:

- Формира се узорак од $\sqrt{n}$ првих елемената из тренутног подниза.
- Тај узорак се сортира *инсертион* методом по тежини ивица.
- Медијана овог узорка се користи као пивот.

Оваквим избором пивота значајно се умањује вероватноћа лоше поделе елемената, јер се пивот приближава реалној медијани низа. Ово у просеку побољшава баланс између леве и десне стране током партиционисања, чиме се ефективно умањује дубина рекурзије и стабилизује сложеност ка $\mathcal{O}(n \log n)$ чак и на неравномерно распоређеним подацима.

```
void sequential_quick_sort(int start_idx, int end_idx,
    unsigned int edges[][3]) { //Both indexes are inclusive
    int len = end_idx - start_idx + 1;
    if (len <= 1) { return; }
    if (len == 2) {
        if (edges[start_idx][2] > edges[end_idx][2]){
            SWAP_UINT3(edges[start_idx], edges[end_idx]);
        }
        return;
    }
    //Picking a pivot
    int slen = (int) sqrt(len);

    unsigned int temp[slen][2];
    temp[0][0] = edges[start_idx][2];
    temp[0][1] = start_idx;
    for (unsigned int i = 1; i < slen; i++){
        unsigned int j = 0;
        unsigned int holder[2];
        holder[0] = edges[start_idx + i][2];
        holder[1] = start_idx + i;
        while (j < i && temp[j][0] < holder[0]){ j++; }
        while (j <= i) {
            SWAP_UINT2(holder, temp[j]);
            j++;
        }
    }
    unsigned int pivot_val = temp[(int) slen / 2][0];
    int pivot_idx = temp[(int) slen / 2][1];

    ...
}
```

Изворни код 2: *Quick Sort* имплементација први део

```
    ...

    //Switch so that pivot is the last element
    SWAP_UINT3(edges[pivot_idx], edges[end_idx]);

    int left_idx = start_idx, right_idx = end_idx - 1;
    while (1 > 0) {
        while (left_idx < right_idx &&
            edges[left_idx][2] < pivot_val) {
            left_idx++;
        }
        while (left_idx < right_idx &&
            edges[right_idx][2] >= pivot_val) {
            right_idx--;
        }

        if (left_idx >= right_idx) {
            if (right_idx == end_idx - 1 &&
                edges[right_idx][2] < pivot_val){
                //if pivot is greater than all the other elements,
                    move right_idx, and break
                right_idx += 1;
                break;
            }
            SWAP_UINT3(edges[end_idx], edges[right_idx]);
            break;
        } else {
            SWAP_UINT3(edges[left_idx], edges[right_idx]);
            left_idx++;
        }
    }
    sequential_quick_sort(start_idx, right_idx - 1, edges);
    sequential_quick_sort(right_idx + 1, end_idx, edges);

}
```

Изворни код 3: *Quick Sort* имплементација други део

Поред тога, наша имплементација садржи и следеће оптимизације:

- Ако је подниз дужине 1 или 2, он се директно сортира без рекурзије.
- Ако је пивот већ највећи елемент у поднизу, он се само премешта на крај, без непотребног партиционисања.

Ова побољшања чине имплементацију ефикаснијом како у времену, тако и у меморији, задржавајући добру перформансу и за велике скупове података.

Даље у алгоритму иницијализујемо структуру *Union-Find* која служи за праћење повезаности чворова, односно да ли додавање нове гране прави циклус. На почетку је сваки чвор посебна компонента. Након тога пролазимо кроз све ивице од најмање до највеће, по тежини, и за сваку проналазимо корене скупова којима припадају два чвора. Уколико су корени исти, то значи да су оба чвора већ у истој повезаној компоненти, а уколико нису то значи да не формирају циклус. Када одредимо ивицу која има најмању тежину и она не формира циклус, два скупа у *Union-Find*-у спајамо, повећавамо укупну тежину те додајемо ивицу у MST. У примеру 3.2 видимо комплетну секвенцијалну имплементацију Крушкаловог алгоритма.

```c
int sequential_kruskal(int num_of_edges, int num_of_nodes,
unsigned int edges[][3], unsigned int result_edges[][3]) {
    sequential_quick_sort(0, num_of_edges- 1, edges);

    unsigned int *parent = (unsigned int *) malloc(num_of_nodes
    * sizeof(unsigned int));
    unsigned int *rank = (unsigned int *)malloc(num_of_nodes
    * sizeof(unsigned int));

    make_union_find(parent, rank, num_of_nodes);

    unsigned int min_cost = 0;
    int l = 0;

    for (unsigned int i = 0; i < num_of_edges; i++) {
        unsigned int v1 = find_set(parent, edges[i][0]);
        unsigned int v2 = find_set(parent, edges[i][1]);
        unsigned int wt = edges[i][2];

        if (v1 != v2) {
            union_set(v1, v2, parent, rank);
            min_cost += wt;
            result_edges[l][0] = edges[i][0];
            result_edges[l][1] = edges[i][1];
            result_edges[l][2] = edges[i][2];
            l += 1;
            if (l == num_of_nodes) {
                break;
            }
        }
    }
    free(parent);
    free(rank);
    return min_cost;
}
```

Изворни код 4: Сортирање ивица

```
void run_parallel_quick_sort(int start_idx, int end_idx,
    unsigned int edges[][3]) {
# pragma omp parallel num_threads(NUM_OF_THREADS)
#pragma omp single
{
    parallel_quick_sort(start_idx, end_idx, edges);
}
}
```

Изворни код 5: Паралелна имплементација сортирања

3.3 Паралелна имплементација

Паралелна имплементација је имплементирана у *OpenMP* и *CUDA* технологијама.

3.3.1 OpenMP

OpenMP (*Open Multi-Processing*) је API за паралелно програмирање на системима са више процесора (CPU језгара). Омогућава једноставно и декларативно увођење паралелизма у *C*, *C++* и *Fortran* програме коришћењем директива (*pragma*), библиотечких рутина и променљивих окружења које упућују компајлер како да извршава одређене делове кода паралелно. Заснива се на концепту **дељене меморије** Најчешће се користи за убрзавање петљи и секција кода које се могу независно извршавати, уз минималне измене у постојећем секвенцијалном коду. Паралелизација Крушкаловог алгоритма огледа се у паралелизацији корака сортирања ивица, односно у нашем случају, паралелизацији *Quick Sort* алгоритма, док су остали кораци имплементирани идентично као и у секвенцијалној имплементацији.

Прво иницијализујемо паралелно окружење да се покреће Директивом *pragma omp single* осигуравамо да само једна од покренутих нити покрене функцију за сортирање. У функцији за сортирање проверавамо да ли је низ премали (мањи од 1024), уколико јесте, не креира се паралелни задатак, јер би се тиме постигло додатно оптерећење (*overhead*), већ се извршава секвенцијално, док уколико је довољно велики низ, онда креирамо паралелни задатак. У примеру 3.3.1

3.3.2 CUDA

CUDA (*Compute Unified Device Architecture*) је развојна платформа коју је развила компанија *NVIDIA*, омогућавајући директно програмирање и коришћење графичких процесора (GPU) за опште намене. Уз *CUDA*, програмери могу писати паралелни код

на *C*, *C++* или *Python* језику који се извршава на GPU језгрима, што значајно убрзава израчунавања која захтевају висок степен паралелизма, као што су научне симулације, машинско учење и обрада великих података. Слично *OpenMP* имплементацији, паралелизација помоћу CUDA технологије је такође фокусирана на корак сортирања ивица, док се остатак Крускаловог алгоритма извршава секвенцијално на централном процесору (CPU). За сортирање на графичкој картици (GPU), имплементирана су два различита приступа.

Приступ 1: Коришћење Thrust библиотеке Први приступ користи *Thrust*, CUDA библиотеку високог нивоа која нуди готове паралелне алгоритме. Процес се одвија у следећим корацима:

1. **Припрема података на хосту (CPU):** Оригинални низ ивица edges[][3] се "распакује" у одвојене векторе за чворове и тежине. Овај корак је додатно убрзан коришћењем OpenMP паралелизма.

2. **Трансфер података на уређај (GPU):** Припремљени вектори се копирају из меморије хоста у меморију графичке картице.

3. **Сортирање на уређају:** Позива се функција thrust::sort_by_key. Она користи низ тежина као кључеве за сортирање и паралелно преуређује низове чворова на основу тих кључева.

4. **Трансфер резултата назад на хост:** Сортирани подаци се враћају са графичке картице на CPU.

5. **Реконструкција низа:** Сортирани вектори се поново спајају у оригинални edges[][3] формат.

Овај приступ је приказан у примеру 6.

Приступ 2: Рекурзивни Quick Sort кернел Други приступ је имплементација *Quick Sort* алгоритма директно у CUDA C++, користећи CUDA Dynamic Parallelism (CDP). Ова технологија омогућава да један CUDA кернел (функција која се извршава на GPU) покреће друге кернеле.

1. **Трансфер података:** Целокупан низ ивица се копира са CPU на GPU меморију.

2. **Покретање главног кернела:** Са хоста се покреће само једна инстанца кернела cdp_simple_quicksort.

```
void gpu_sort(unsigned int (*edges)[3], int numEdges) {
thrust::host_vector

thrust::device_vector<unsigned int> dev_weights
    = host_edge_weights;
thrust::device_vector<unsigned int> dev_nodes1
    = host_nodes1;
// ... (copy other vectors) ...

thrust::sort_by_key(dev_weights.begin(),
    dev_weights.end(), dev_nodes1.begin());
// ... (sort other vectors similarly) ...


thrust::copy(dev_weights.begin(),
    dev_weights.end(), host_edge_weights.begin());
// ... (copy other vectors) ...

// ... (OMP loop to reassemble the edges array) ...
}
```

Изворни код 6: Паралелно сортирање помоћу Thrust библиотеке

3. **Рекурзивно паралелно сортирање:** Унутар кернела, након партиционисања низа, покрећу се два нова кернела за сортирање леве и десне партиције. Ово се ради асинхроно коришћењем CUDA токова (*streams*), што је аналогно *OpenMP task* моделу.

4. **Хибридна оптимизација:** Да би се избегло додатно оптерећење за веома мале поднизове, уколико је дужина подниза мања од дефинисаног прага (*SELECTION_SORT*), кернел прелази на једноставнији *Selection Sort* алгоритам уместо да покреће нове кернеле.

5. **Трансфер резултата:** Након што се заврши целокупно рекурзивно сортирање, финални низ се копира назад на CPU.

Пример рекурзивног кернела дат је у 7.

```
__global__
void cdp_simple_quicksort(unsigned int data[][3],
unsigned int left, unsigned int right, int depth) {
if( depth >= MAX_DEPTH || right - left <= SELECTION_SORT ){
    selection_sort(data, left, right);
    return;
}
// ... (Partitioning logic) ...
if (left < nright) {
    cudaStream_t s;
    cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
    cdp_simple_quicksort<<<1, 1, 0, s>>>(data, left,
        nright, depth + 1);
    cudaStreamDestroy(s);
}
if (nleft < right) {
    cudaStream_t s1;
    cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
    cdp_simple_quicksort<<<1, 1, 0, s1>>>(data, nleft,
        right, depth + 1);
    cudaStreamDestroy(s1);
}
}
```

Изворни код 7: Рекурзивни Quick Sort кернел са CUDA Dynamic Parallelism

4 Закључак

Овај рад се бавио проблемом оптимизације Крушкаловог алгоритма за одређивање минималног разапињућег стабла (MCT), са фокусом на паралелизацију најзахтевнијег корака – сортирања ивица. Имплементирана су решења заснована на OpenMP и CUDA технологијама, чиме је истражен потенцијал модерних паралелних архитектура за убрзање овог класичног проблема.

Резултати су показали да је предложени приступ веома успешан. OpenMP имплементација на вишејезгарним процесорима постигла је убрзање од 4.6 пута, док је CUDA имплементација на графичким процесорима остварила убрзање од 3.3 пута у поређењу са секвенцијалном верзијом. Ови резултати јасно потврђују да се паралелизацијом може значајно смањити време потребно за решавање MCT проблема на великим графовима и демонстрирају ефикасност обе тестиране паралелне парадигме.

Простор за даља унапређења постоји у виду хибридних приступа. Комбиновање представљене паралелне имплементације сортирања са напредним техникама, као што су филтрирање ивица из *Filter-Kruskal* алгоритма [2] или партиционисање из *Two-Branch Kruskal* методе [1], представља обећавајући правац за будућа истраживања која би могла довести до још значајнијих убрзања.

Библиографија

[1] Haiming Li, Qiyang Xia, and Yong Wang. Research and improvement of kruskal algorithm, Sep 2017.

[2] Vitaly Osipov, Peter Sanders, and Johannes Singler. The filter-kruskal minimum spanning tree algorithm. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 52--61. SIAM, 2009.