Факултет техничких наука

Универзитет у Новом Саду

Рачунарски системи високих перформанси

Паралелно претраживање графа по ширини

Аутор:
Стефан Алексић

Индекс:
E2 42/2022

11. јануар 2023.

Сажетак

Тема овог рада јесте дистрибуирано односно паралелно претраживање графа по ширини. Мотивација за писање рада јесте пре свега истраживање начина за убрзање релативно основних операција над веома битном структуром података у рачунарству, односно графом. Такође, дубље истраживање проблема који се јавља у дистрибуираном и паралелном програмирању када се говори о претраживању графова. Наиме, због структуре која је, за недостатак бољих речи, непредвидива, тешко је добити прихватљиве перформансе при приступању нелинеарно складиштеним подацима. Кроз рад су анализиране методе за расподелу посла између процеса односно нити који врше саму претрагу, предности и мане оба приступа, као и постигнути резултати при њиховој примени. Закључено је да иако наизглед комплексно и мукотрпно, ипак је могуће постигнути неко убрзање кроз паралелни приступ у окружењима са дељеном меморијом. Нажалост, што се овог рада тиче, имплементација у оквиру окружења са дистрибуираном меморијом није показала позитивне резултате.

Садржај

Списак изворних кодова

Списак слика

Списак табела

Увод

Употреба апстракције уз помоћ графова за анализу и разумевање разних врста података добија све већи значај. Неки од примера података који могу да се апстрахују користећи графове подразумају: податке о интеракцијама на друштвеним мрежама, податке банкарских трасакција, податке о препоруци разних реклама корисницима апликација на основу њихових интеракција, комуникационих података попут електронске поште и телефонских мрежа, податке биолошких система и различитих облика релационих података генерално. Када се говори о вештачкој интелигенцији, апсолутно је неопходно увести неку врсту графа и примењивати разноврсне алгоритме над њим. Заједнички проблеми у математичкој области теорије графова и у областима примене укључују идентификацију и рангирање важних ентитета, откривање аномалија у обрасцима или изненадних промена у мрежама, проналажење чврсто повезаних кластера ентитета, итд. Решења ових проблема обично укључују класичне алгоритме за проблеми као што су: пребацивање графа у структуру стабла (уклањање циклуса), проналажење најкраћих путања, проналажење двоповезаних компоненти, упаривања, прорачуне засноване на протоку. Да бисмо задовољили потребе теоријске анализе графова за нове апликације које захтевају рад са структурама великих скупова података, од суштинског је значаја да убрзамо основне проблеме графова користећи актуелне паралелне системе.

У овом раду ће се детаљније разматрати проблем проналажења најкраћег пута, односно најмањег броја скокова, између два чвора у графу и то применом алгоритма за претрагу графа по ширини, познатог као *breadth-first traversal*. Биће имплементирани алгоритми паралелне верзије овог алгоритма, користећи системе са дистрибуираном и дељеном меморијом. Такође, биће дат и приказ резултата добијених примењујући ове имплементације на насумично генерисаним графовима.

У првом одељку ће се изнети теоријске основе графова, попут терминологија, основних идеја операција над њима, описом идеја претраживања истих и сл.. Такође, у оквиру овог одељка ће бити описани и разни начини репрезентације графа,

појединим од којих ће бити коришћени ради олакшане обраде у паралелном окружењу.

У оквиру другог одељка ће бити описани поједини приступи у паралелизацији претраживања графа. Поред описа, ово поглавље ће садржати и идеје за имплементирање сваког од појединих приступа, заједно са алгоритмима и описом истих.

Треће поглавље ће садржати имплементацију алгоритама за паралелно претраживање графа у дистрибуираним и системима са дељеном меморијом. Такође, на крају описа сваке од имплементација стојаће приказ резултата добијених кроз њихову примену на насумично генерисаним графовима.

На крају, у последњем одељку биће изнет и закључак самог рада.

Глава 1

Графови

1.1 Теоријске основе

Граф, посматрано из угла дискретне математике, односно прецизније теорије графова, представља структуру која се састоји од скупа објеката који међу собом могу имати специфичну повезаност, односно релацију. Сами објекти који се повезују се називају **теменима *(eng. vertex)***, док су везе између тих објеката **потези, односно ивице *(eng. edge)***. Посматрано из угла рачунарства, граф представља апстрактну структуру података која служи за само моделовање графова из математичке области теорије графова на рачунару.

Најједноставнија таксономија графова јесте на **усмерене** и **неусмерене** графове. **Усмерени графови** су сачињени од скупа темена и скупа **усмерених потега**, такође познатих и под називом **гране *(eng. arc)***, док се **неусмерени графови** дефинишу као уређеи пар скупа чворова и **неусмерених потега**, односно **линкова**.

За потег e_i који спаја темена x і y кажемо да је усмерен ако важи да је x директан следбеник y, односно y директан претходник x, док не важи обрнуто, да је y директан следбеник x, односно x директан претходник y. Математички то можемо представити као:

$$\mathrm{e}_i = (x, y) \neq (y, x) = e_j$$

За потег e_p који спаја темена s і t кажемо да је неусмерен ако важи да је s директан следбеник t, односно t директан претходник s, и да важи обрнуто, да је t директан

следбеник s, односно s директан претходник t. Математички то можемо представити као:

$$\mathrm{e}_p = (s, t) = (t, s) = e_q$$

Уколико зађемо дубље у природу графова, можемо дефинисати и путању између два чвора у графу. Путања представља секвенцу или низ потега, таквих да се одредишно теме i-ог потега поклапа са полазним чвором $i + 1$-ог потега. Ова секвенца за први потег има потег коме је полазни чвор заправо чвор од ког се започиње обилазак, док је последњи потег у секвенци, потег коме је одредишни чвор заправо чвор до ког се тражи пут. Овај низ у већини случаја је низ јединствених темена, а самим тим и низ јединствених потега. Код неусмерених графова ће због њихове природе важити да уколико постоји пут од чвора до чвора B, постојаће и пут од чвора B до чвора , док код усмерених графова ово неће бити случај.

Када говоримо о графу генерално, било који чвор графа нема ограничење у виду кардиналности скупа потега којима је повезан са осталим чворовима графа, као ни то до којих чворова може имати потеге. У таквом графу је могуће идентификовати циклусе, или петље. Петља представља путању у којој су почетни и одредишни чвор једнаки. Специфична ситуација код неусмерених графова је та да уколико постоји барем један потег, аутомарски постоји и циклус у том графу.

На супротној страни, можемо дефинисати и најкраћи пут између два чвора у графу, односно путању са најмањом кардиналношћу од свих могућих путања између два задата чвора у графу. Јасно је да оваква путања никада неће имати циклусе.

Дакле, проналажење најкраћег пута у графу може бити јако комплексан проблем, па самим тим има и веома широк спектар решења која се прилагођавају за различите случаје.

1.2 Репрезентација графа

Математички, граф можемо представити као уређен пар два скупа V и E који представљају скупове чворова и потега графа G респективно, односно $G(V, E)$. Оваква репрезентација је математички јасна, међутим у рачунарству би била јако непрактична. Из тог разлога постоји више начина за представљање графа.

1.2.1 Логичка репрезентација графа

Када се говори о матричној репрезентацији графа, углавном се мисли на квадратне матрице, које су, у зависности од кардиналности скупа потега , или ретко или густо поседнуте.

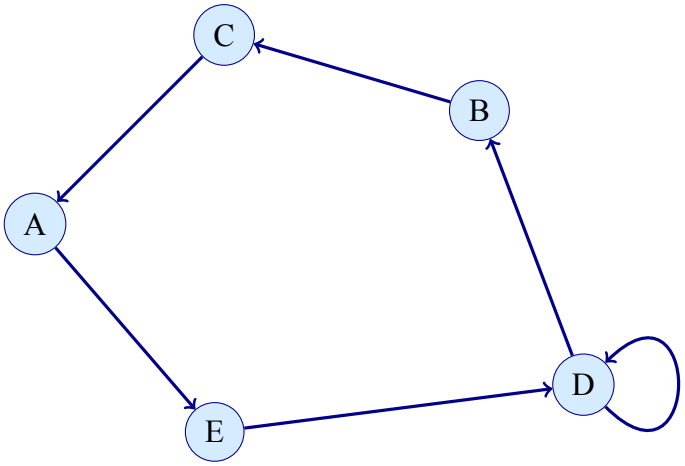
Узећи за пример граф $G = (V, E)$ где важи следеће:

$$V = \{A, B, C, D, E\}\ E = \{e1, e2, e3, e4, e5, e6\}$$

$$e1 = (A, B)\ e2 = (B, C)\ e3 = (C, D)$$

$$e4 = (D, E)\ e5 = (E, A)\ e6 = (E, E)$$

Овај граф можемо визуелно представити (1.1).



Слика 1.1: *Пример усмереног графа G*

Уколико бисмо желели да моделујемо овако нешто у оквиру програмског језика C, имали бисмо структуру чвора (1, linije 1-5), потега (1, linije 7-12) и графа (1, linija 14-19).

Структура чвора би се састојала из показивача на први потег у листи потега који полазе из тог чвора, показивача на тип чвора, који би представљао логичког следбеника у листи свих чворова графа и једне целобројне променљиве која би представљала вредност/идентификатор чвора.

```c
struct node{
    struct node* next;
    struct edge* neighbours;
    int value;
};

struct edge{
    struct edge* next;
    struct node* source;
    struct node* destination;
    int weight;
};

struct graph{
    struct node* nodes;
    struct edge* edges;
    int nodes_numb;
    int edges_numb;
};
```

Изворни код 1: *Репрезентација графа у C-у*

Структура потега би садржала два показивача на структуру типа чвора, један за одредишни, други за изворишни чвор, један показивач на тип потега, којим би се представљао логички потег следбеник тренутног потега у графу и евентуално целобројну промењиву која би представљала тежину самог потега (уколико говоримо о тежинским графовима).

На крају, сама структура графа би садржала два показивача, један показивач на тип чвор, који представља логички почетак листе свих чворова у графу и други показивач на тип потег, који би представљао логичку листу свих потега у графу.

Из овога можемо закључити да репрезентација графа може постати јако комплексна уколико кренемо логички да је имлементирамо у оквиру различитих програмских језика.

1.2.2 Матрична репрезентација графа

Над компликованим структурама је јако тешко имплементирати већ постојеће алгоритме, тако да је увек препоручљиво пребацити се на домен који је рачунару јако познат. У овом случају то су матрице (низови).

Када се говоримо о матричној репрезентацији графа, углавном се мисли на квадратне матрице, којима се означавају суседи, односно потези. Одавде такве матрице добијају назив матрице суседства, или матрице повезаности. У зависности од кардиналности скупа потега , ове матрице могу бити ретко или густо поседнуте.

Приступ за репрезентацију графа у матричном облику је одиста прост. За квадратну матрицу , елементи i-ог реда матрице ће представљати потеге од i-ог темена, док ће у пресеку са -ом колоном имати вредност 1, уколико постоји потег $_{ij} = (v_i, v_j)$, односно вредност 0 уколико не постоји потег између i-ог и j-ог темена. Наиме, уколико је кардиналност скупа чворова графа G једнака целобројној вредности n ($|V| = n$), онда би наша квадратна матрица A била димензија $n \times n$, или математички:

$$G(V,E) \wedge |V| = n \implies A \in M_{n\times n}$$

$$A = [a_{ij}]_{n\times n}, \quad a_{ij} = \begin{cases} 1, & if e_{ij} = (v_i, v_j) \in E \\ 0, & if e_{ij} = (v_i, v_j) \notin E \end{cases}$$

На пример, уколико бисмо желели да граф са слике 1.1 представимо као квадратну матрицу, добили бисмо следећу матрицу суседства :

$$V = \{\, A,\, B,\, C,\, D,\, E \,\}$$

$$E = \{\, e(A,B),\, e(B,C),\, e(C,D),\, e(D,E),\, e(E,E) \,\}$$

$$A = \begin{bmatrix} e(A,A) \notin E & e(A,B) \in E & e(A,C) \notin E & e(A,D) \notin E & e(A,E) \notin E \\ e(B,A) \notin E & e(B,B) \notin E & e(B,C) \in E & e(B,D) \notin E & e(B,E) \notin E \\ e(C,A) \notin E & e(C,B) \notin E & e(C,C) \notin E & e(C,D) \in E & e(C,E) \notin E \\ e(D,A) \notin E & e(D,B) \notin E & e(D,C) \notin E & e(D,D) \notin E & e(D,E) \in E \\ e(E,A) \in E & e(E,B) \notin E & e(E,C) \notin E & e(E,D) \notin E & e(E,E) \in E \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

С обзиром да се ради о слабо повезаном графу, $|E| << |V|^2$, добијамо ретко поседнуту матрицу суседства, која је препуна нула. Оваква репрезентација је сада подложна матричним алгоритмима, међутим меморијски беспотребно заузума огроман простор.

1.2.3 Репрезентација графа назубљеном матрицом

У претходом делу смо видели како изгледа представљање графа матрицом суседства. Могли смо да закључимо да за поједине случаје (слабо повезане графове), оваква репрезентација није најпогоднија, зато што имамо огроман број 0, који нам не говори ништа специјално. Шта би се десило уколико бисмо само уклонили нуле из матрице?

За дати пример графа 1.1, матрица суседства има следећи изглед:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Када бисмо уклонили нуле из матрице добили бисмо:

$$A = \begin{bmatrix} 1 & \\ 1 & \\ 1 & \\ 1 & \\ 1 & 1 \end{bmatrix}$$

Овде се сада јављају два проблема:

1. Редови немају исти број колона (темена немају исти број суседа),
2. Шта нам сад значе јединице, ако су поравнане налево?

Први проблем можемо решити линеаризацијом матрице A на вектор a, који је сада максималне димензије $n \times n$, где је $n = |V|$. Вектор a бележи елементе a_{ij} из матрице суседства који су различити од нуле, односно за које важи $e(v_i, v_j) \in E$.

Како бисмо могли да бележимо померање у оквиру вектора a уводимо још један вектор d, који ће бити димензије n, а чије вредности ће ићи у опсегу $[1-n^2)$. Елемент вектора d_i ће представљати индекс у оквиру вектора a од кога креће листа суседа за теме v_i. Математички:

$$d(i) = d(i-1) + |E_{v_i}| \quad \wedge a_0 = 1$$

где је

$$E_{v_i} = \{e(v_i, v_j) \mid \forall v_j \in V \wedge e(v_i, v_j) \in E\} \implies E_{v_i} \subset E$$

Односно сви потези из скупа E који за полазни потег имају потег v_i. Како бисмо могли да водимо евиденцију и о томе колико укупно потега има у графу, додајемо на крај вектора d елемент који представља суму $d(n) + |E_{v_n}|$, односно укупан број потега у графу. Сада је вектор d димензија $n+1$.

Сада ће наши вектори a и d на основу матрице A изгледати:

$$a = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$
$$d = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 6 \end{bmatrix}$$

Остало нам је само још да дефинишемо шта која вредност у вектору значи. Знамо да је вредност $d(i)$ индекс у оквиру вектора a од којег почињу суседи темена v_i. Уколико бисмо уместо јединица заправо бележили суседе, онда би вектор a имао изглед:

$$a = \begin{bmatrix} B & C & D & E & A & E \end{bmatrix}$$

Пресликавањем ових темена функцијом:

$$F(v_i) = i, \quad \forall i \in [1, |V|]$$

Добијамо конични облик вектора a:

$$a = \begin{bmatrix} 2 & 3 & 4 & 5 & 1 & 5 \end{bmatrix}$$

Овим смо постигли, да слабо повезани граф G представимо на веома једноставан начин, који је веома меморијски ефикасан и у позадини користи нозове, те је меморијски приступ максимизован.

1.3 Претраживање графа

Претраживање графа представља процес посећивања *(eng. visit)* сваког чвора у графу где је посета генерализација неке од операција читања или модификације датог чвора. У зависности од редоследа посећивања, може се дефинисати широка таксономија алгоритама за претраживање графа.

Сваки од алгоритама за претраживање графа има за циљ избегавање циклуса у графу, а специјална врста обиласка графа јесте проблем проналажења најкраћег пута (пута са најмањом ценом, уколико се ради о тежинским графовима).

1.3.1 Алгоритми обиласка графа

Најпознатији алгоритми за обилазак графа јесу облизака графа по дубини и обилазак графа по ширини. Идеја иза ова два алгоритма јесте да се у прослеђеном графу G и са почетним чвором v обиђу сви чворови који имају директне или индиректне потеге од почетног чвора и то без понављања (циклуса).

Алгоритам обиласка графа по дубини

Претрага по дубини *(eng. Depth-First Search - DFS)* је алгоритам за прелазак преко коначног графа, односно графа са коначним бројем темена/потега. Идеја иза *DFS* алгоритма јесте да се у свакој итерацији прво посети чворове потомке чвора који се обрађује, па тек онда његове суседе. Односно прелази дубину било ког одређеног пута пре него што истражи његову ширину. Стек, или као структура података, или својство рекурзије, се генерално користи приликом имплементације овог алгоритма.

Алгоритам почиње са изабраним *"коренским"* теменом. Затим итеративно прелази са тренутног темена на његове суседе, који су непосећени, све док више не може да пронађе непосећено теме на које би прешао са своје тренутне локације. Алгоритам се затим враћа дуж претходно посећених темена, све док не пронађе теме повезано са следећом неистраженом територијом. Затим ће наставити новом путањом као и раније, враћајући се уназад када наиђе на ћорсокак. Алгоритам се завршава тек када се врати преко првобитног *"коренског"* темена из првог корака, односно одакле је кренуо. Псеудокод овог алгоритма приказан је у оквиру исечка 1.2.

Алгоритам 1 Обилазак графа по дубини

```
Улаз: Граф G, полазно теме v
1:  procedure depthFirstSearch(G, v)
2:      create stack S
3:      push(S, v)
4:      mark v
5:      while S is not empty do
6:          w ← pop(S)
7:          visit w
8:          for all edges e in incidentEdges(G, w) do
9:              x ← adjacentVertex(w, e)
10:             if x is not marked then
11:                 mark x
12:                 push(S, x)
```

Слика 1.2: *Псеудокод - Обилазак графа по дубини*

Временска комплексност алгоритма претраге графа по дубини је у најгорем случају $\mathcal{O}(|V| + |E|)$. Јасно је да је разлог овога то што сваки чвор и потег морају бити посећени барем једном. Уколико говоримо о потпуно повезаним графовима, тада важи да је $|E| = |V|^2$, па је онда временска комплексност таквих графова у најгорем случају $\mathcal{O}(|V|+|V|^2)$. Што се просторне комплесности тиче, она је $\mathcal{O}(d)$, где је d максимална дубина графа. Ово је јасно, јер тада на стек морају да се нађу сви чворови кроз које се прошло да би се стигло до најдубљег темена. Овај алгоритам је погодан када немамо предубоко стабло, као и то када нас не интересује да ли је крајњи чвор близу почетног чвора.

Алгоритам обиласка графа по ширини

Претрага у ширину *(eng. Breadth-First Search - BFS)* је алгоритам за претрагу коначног графа. Почиње од

"коренског" темена и истражује све чворове на тренутној дубини пре него што пређе на чворове на следећем нивоу дубине. Додатна меморија, обично структура података типа ред *(eng. queue)*, потребна је за праћење следбенских чворова који су наишли, али још нису истражени.

На пример, у шаховској завршници, машински алгоритам за шах може да изгради стабло игре од тренутне позиције применом свих могућих потеза и користи претрагу у ширину да пронађе позицију за победу за беле. Имплицитна стабла (као

што су стабла игре или друга стабла за решавање проблема) могу бити бесконачне величине. Претрага по ширини ће гарантовано пронаћи стање решења, уколико оно постоји. Псеудокод алгоритма претраге по ширини је дат у оквиру исечка 1.3.

Алгоритам 2 Обилазак графа по ширини

```
Улаз: Граф G, полазно теме v
1:  procedure breadthFirstSearch(G, v)
2:      create queue Q
3:      enqueue(Q, v)
4:      mark v
5:      while Q is not empty do
6:          w ← dequeue(Q)
7:          visit w
8:          for all edges e in adjacentEdges(w) do
9:              x ← adjacentVertex(w, e)
10:             if x is not marked then
11:                 mark x
12:                 enqueue(Q, x)
```

Слика 1.3: *Псеудокод - Обилазак графа по ширини*

Временска комплексност може бити дата као $\mathcal{O}(|V| + |E|)$, с обзиром да ће у најгорем могућем случају свако теме и потег бити посећени барем једном. Битно је напоменути да кардиналност скупа потега $|E|$ може да варира између $\mathcal{O}(1)$ и $\mathcal{O}(|V|^2)$, у зависности од тога колико је граф повезан. Тако да временска комплексност алгоритма са слике 1.3 у најгорем могућем случају, за потпуно повезан граф може достићи $\mathcal{O}(|V| + |V|^2)$. Што се просторне комплексности тиче, она достиже максимум са $\mathcal{O}(b^d)$, где је b максимални фактор гранања, а d представља максималну дубину графа. Овај алгоритам очигледно треба избегавати уколико је меморија ограничавајући фактор, као и то када желимо да тражени чвор буде најближи, односно има најмање чворова преко којих мора да се иде, почетном чвору.

Глава 2

Паралелно претраживање графа

Претраживање података је привукло велику пажњу последњих година захваљујући расту потражње за техникама које омогућују претрагу података великих размера у важним областима као што су геномика, астрофизика и национална безбедност. Претраживање графова игра важну улогу у анализи великих скупова података у многим случајевима који у некој мери имају податаке који се доводе у међусобну релацију, јер се такви подаци често представљају у облику графова, као што су семантички графови. Претрага по ширини је од посебног значаја међу различитим методама претраживања графова и користи се у многим апликацијама. Природа односа између два темена у семантичком графу, на пример, може бити одређена најкраћим путем између њих помоћу претраге по ширини.

Претраживање веома великих графова са милијардама темена и потега представља изазове углавном због огромног простора за претрагу који намећу овакви графови. Посебно, често је немогуће складиштити тако велике графиконе у радној меморији једног рачунара. Ово чини традиционалне паралелне алгоритме за претрагу по ширини засноване на *PRAM*-у неупотребљивим и захтевају дистрибуиране паралелне алгоритме где се израчунавање пребацује на процесор који поседује податке. Очигледно, скалабилност дистрибуираног *BFS* алгоритма за врло велике графове постаје критично питање, будући да се потражња за локалном меморијом и међупроцесорском комуникацијом повећава како се повећава величина графа.

У овом раду је анализирана скалабилна и ефикасна дистрибуирана *BFS* шема која је способна да рукује графовима са милијардама темена и потега. Такође, алгоритам је имплементиран у програмском језику *C* и тестиран над насумично генерисаним графовима који су представљени назубљеним матрицама.

2.1 Паралелни *BFS* са дистрибуираном меморијом

У овом одељку је имплементиран и анализиран дистрибуирани *BFS* алгоритам са једнодимензионим партиционисањем *(eng. one dimentional partitioning)*. Предложени алгоритам је *BFS* алгоритам синхронизован по нивоима који напредује ниво по ниво, почевши од изворног чвора, где је ниво темена дефинисан као његов граф удаљености од извора. У наставку је са P означен број процесора, n означава број темена у насумичном графу G, а k означава просечан степен. P процесори су мапирани у дводимензионални логички процесорски низ, док R и C означавају траке редова и колона процесорског низа, респективно. Разматрни су само усмерени графови.

2.1.1 Дистрибуирани *BFS* алгоритам са једнодимензионим партиционисањем

Једнодимензионо партиционисање графа подразумева расподелу темена графа, тако да сваки чвор и сви потези који произилазе из тог чвора припадају само једном прецесору [1]. Скуп темена који припадају једном процесору q ће даље бити називан скуп *локалних темена*. У наставку је приказа илустрација једнодимензионог партиционисања темена графа на P партиција коришћењем матрице суседства A, која је подељена тако да су локална темена за неки процесор q континуална.

$$\left[\begin{array}{c} A_1 \\ \hline A_2 \\ \hline \vdots \\ \hline A_P \end{array}\right]$$

Индекси подматрица суседства A_i представљају индикатор процесора коме је додељена матрица. Потези који произилазе од темена v формирају листу потега, која је представљена листом темена у оквиру j-ог реда матрице суседства A. Како би партиционисање било што балансираније, сваки процесор треба да добије прибли-

[1] Претпоставља се да се један процес извршава на једном прецесору.

жно једнак број темена и потега који од тих темена произилазе. У оквиру исечка 2.1 је дат псеудокод алгоритма за дистрибуирано претраживање графа са једнодимензионим партиционисањем, почевши од чвора v_s. У оквиру алгоритма, свако теме v_i бива означено са својим нивоом, у оквиру низа $L_{v_s}(v_i)$, што означава растојање чвора v_s од чвора v_i. Низ L_{v_s} је дистрибуиран у складу са дистрибуцијом темена, тако да процесор P_i поседује растојања почетног темена v_s до својих локалних темена $\{ v_{(i,0)}, v_{(i,1)}, ..., v_{(i,m)} \}$ где је $m = \frac{|V|}{|P|}$.

Алгоритам 3 Паралелни *BFS* са једнодимензионим партиционисањем

Улаз: Граф $G(V, E)$, полазно теме s

Излаз: Низ d чији елементи представљају број скокова од почетног темена s до темена $v_i \in V$

1: **procedure** bfsDistributedSearch($G(V, E)$, s)
2: Initialize $L_s(v) \leftarrow \begin{cases} \inf, & v \neq s \\ 0, & v = s \end{cases}$
3: **for** $level \leftarrow 0$ *to* ∞ **do**
4: $F \leftarrow \{v | L_s(v) = level\}$, скуп локалних темена за процес са нивоом $level$
5: **if** F *is empty* за све процесе **then**
6: Терминирај спољашњу петљу свим процесима
7: $N \leftarrow$ {суседна темена из скупа F (нису нужно локална темена за тренутни процес)}
8: **for all** *processes* q **do**
9: $N_q \leftarrow$ {темена из N која припадају процесу q}
10: **Send** N_q *to process* q
11: **Receive** $\widehat{N_q}$ *from process* q
12: $\widehat{N} \leftarrow \cup_q \widehat{N_q}$ ($\widehat{N_q}$ је могуће да има дупликате)
13: **for** $v \in \widehat{N}$ *and* $L_s(v) = \infty$ **do**
14: $L_s(v) \leftarrow level + 1$
15: **return** L_s

Слика 2.1: *Псеудокод дистрибуираног BFS алгоритма са једнодимензионим партиционисањем*

Алгоритам функционише на следећи начин. На сваком нивоу (дубине графа), сваки од процесора има скуп F који представља скуп локалних темена који се налазе на растојању *level* од почетног чвора. Листа потега која се добија од сваког од темена из скупа F бива унификована у скуп потега N који представља скуп сусед-

них темена. Поједина темена у оквиру скупа N неће бити из скупа локалних темена који је додељен процесору. За ова темена, врши се размена са процесором q, коме се темена која њему припадају шаљу, а темена која припадају тренутном процесору примају од истог процесора q. Сваки од процесора врши ову размену темена и формира коначан скуп $\widehat{N}$, скуп темена до којих се стигло у тренутном нивоу, а који су из скупа локалних темена за тренутни процесор. Могуће је да је процесор означио поједина темена из $\widehat{N}$ као посећена, што значи да их сада више неће разматрати.

2.1.2 Дистрибуирани *BFS* алгоритам са дводимензионим партиционисањем

Дводимензионо партиционисање графа јесте партиционисање потега, тако да сваки потег припада тачно једном процесору. Поред тога, скуп темена графа је такође подељен, тако да свако теме припада само једном процесору. Процесор складишти поједине потеге који су везани за његова локална темена, као и поједине потеге који нису везани за његова локална темена. Овакво партиционисање је приказано у оквиру матрице суседства A, која је организована тако да су темена које поседује један процесор суседна.

$$\left[\begin{array}{cccc}
A^{(1)}_{1,1} & A^{(1)}_{1,2} & \dots & A^{(1)}_{1,C} \\
A^{(1)}_{2,1} & A^{(1)}_{2,2} & \dots & A^{(1)}_{2,C} \\
\vdots & \vdots & \ddots & \vdots \\
A^{(1)}_{R,1} & A^{(1)}_{R,2} & \dots & A^{(1)}_{R,C} \\
\multicolumn{4}{c}{\vdots} \\
A^{(C)}_{1,1} & A^{(C)}_{1,2} & \dots & A^{(C)}_{1,C} \\
A^{(C)}_{2,1} & A^{(C)}_{2,2} & \dots & A^{(C)}_{2,C} \\
\vdots & \vdots & \ddots & \vdots \\
A^{(C)}_{R,1} & A^{(C)}_{R,2} & \dots & A^{(C)}_{R,C}
\end{array}\right]$$

Овде, партиционисање је извршено за $P = R \cdot C$ процесора, који су логички организовани у матрицу процесора $R \times C$. Надаље ће се користити термини *ред-процесора* и *колона-процесора* како би се означиле подгрупе процесора у процесорској

мрежи. У оквиру дводимензионог партиционисања, матрица суседства је подељена на $R \cdot C$ блокове редова и C блокове колона. Нотација $A_{i,j}^{(*)}$ означава блок који припада процесору који у мрежи процесора има индексе (i, j). Сваки од процесора има C блокова. Како би се извршило партиционисање темена, процесор $P_{(i,j)}$ добија темена која одговарају блоку редова $(j-1) \cdot R + i$. Како би партиционисање било избалансирано, сваки од процесора би требало да добије по једнак број темена и потега. Једнодимензионо партиционосање се може свести на случај дводимензионог партиционисања када је $R = 1$ или $C = 1$.

За дводимензионо партиционисање, претпоставља се да је листа потега датог темена једна колона у оквиру матрице суседства. На основу тога, сваки блок у дводимензионом партиционисању садржи парцијалну листу суседа. Када се примени овакво партиционисање, сваки процесор има скуп F који представља скуп локалних темена процесора $P_{(i,j)}$ који се налазе на нивоу *level* у односу на почетно теме v_s. Посматрањем темена v_i из скупа F, процесор власник овог темена шаље поруке осталим процесорима у својој колони процесора информацију да је стигао до чвора v_i, како би му они вратили листу потега која садржи ово теме. Оваква комуникација се назива операцијом *експанзије (eng. expand)*. Парцијалне листе потега сваког од чворора се претражују како би се формирао скуп N који представља скуп чворова за следећи ниво дубине. Темена из N су даље разврстана тако да се деле у подскупе N_q који се размењују са процесором P_q у оквиру истог реда мреже процесора. Оваква комуникација се назива операцијом *склапања (eng. fold)*. Предност дводимензионог партиционисања је то што процесорска комуникација у оквиру реда процесора и колоне процесора захтева R и C процесора, респективно, док је у једнодимензионом партиционисању при свакој комуникацији укључен сваки процесор. У оквиру алгоритма 2.2.

У операције експанзије, процесори шаљу индексе граничних темена (локална темена која се налазе на тренутној дубини, односно на дубини $level$) осталим процесима. За густо поседнуте матрице, а у појединим случајима и за ретко поседнуте матрице), ова операција је традиционално имплементирана као $all - gather$ групна операција, с обзиром да сви индекси које поседује процесор морају бити послати. За секвенцијални BFS, ово је еквивалентно скупу граничних темена у глобалу. Ова операција нажалост није скалабилна када се повећава број процесора.

За слабо повезане графове, веће перформансе би се добиле, уколико би се слала само теман која су гранична (на раздаљини $level$ од почетног темена v_s) и то да се шаљу само оним процесорима који имају непразну листу парцијалних потега која одговарају тим теменима. Ова операција може бити имплементирана као колективна $all - to - all$ операција или алтернативно као већи број позива $broadcast$ операци-

Алгоритам 4 Паралелни *BFS* са дводимензионим партиционисањем

Улаз: Граф $G(V, E)$, полазно теме s

Излаз: Низ d чији елементи представљају број скокова од почетног темена s до темена $v_i \in V$

1: **procedure** bfsDistributedSearch($G(V, E)$, s)
2: Initialize $L_s(v) \leftarrow \begin{cases} \text{inf}, & v \neq s \\ 0, & v = s \end{cases}$
3: **for** $level \leftarrow 0$ to ∞ **do**
4: $F \leftarrow \{v | L_s(v) = level\}$, скуп локалних темена за процес са нивоом $level$
5: **if** F is $empty$ за све процесе **then**
6: Терминирај спољашњу петљу свим процесима
7: **for all** *processors q* у колони процесорске мреже **do**
8: **Send** F to $processor$ q
9: **Receive** $\widehat{F_q}$ $from$ $processor$ q (скупови F_q су дисјунктни)
10: $\widehat{F} \leftarrow \cup_q \widehat{F_q}$
11: $N \leftarrow \{$суседна темена из скупа $\widehat{F}$ користећи локалну листу потега (нису нужно локална темена за тренутни процес)$\}$
12: **for all** $processes$ q у реду процесорске мреже **do**
13: $N_q \leftarrow \{$темена из N која припадају процесу $q\}$
14: **Send** N_q to $process$ q
15: **Receive** $\widehat{N_q}$ $from$ $process$ q
16: $\widehat{N} \leftarrow \cup_q \widehat{N_q}$ ($\widehat{N_q}$ је могуће да има дупликате)
17: **for** $v \in \widehat{N}$ and $L_s(v) = \infty$ **do**
18: $L_s(v) \leftarrow level + 1$
19: **return** L_s

Слика 2.2: *Псеудокод дистрибуираног BFS алгоритма са дводимензионим партиционисањем*

је. У случају дводимензионог партиционисања, сваки процесор мора да складишти информацију у листи потега других процесора у оквиру своје колоне у процесорској мрежи. Меморија неопходна за ове податке је пропорцинална броју локалних темена процесора, те је и скалабилна.

Операција склапања је традиционално имплементирана за густо поседнуте матрице, као $all - to - all$ комуникација. Алтернатива јесте да се имплементира опера-

ција склапања као операција $reduce - scatter$. У том случају, сваки процесор прима $\widehat{N}$ директно и линија 16 у алгоритму 2.2 није неопходна. Операција редукције, која се јавља у фази редукције ове операције је операција унија скупа и елиминише дупликате.

2.2 Паралелни *BFS* алгоритам са дељеном меморијом

У поређењу са паралелним *BFS* алгоритмом са дистрибуираном меморијом, дељена меморија обезбеђује већи меморијски пропусни опсег и ниже кашњење. Пошто сви процесори заједно деле меморију, сви имају директан приступ њој. Дакле, нема потребе за интерпроцесну комуникацију, која је неопходна да би дистрибуирана меморија добила податке из удаљене локалне меморије. На овај начин се избегавају додатни трошкови размене порука.

Међутим, показало се да су број темена у сваком новоу и број суседа сваког од темена веома неправилни, што доводи до веома неправилних приступа меморији и расподеле рада у *BFS*-у. У паралелном *BFS*-у, ова специфичност смањује предности паралелизације због неуравнотеженог оптерећења. Као резултат тога, веома је важно да паралелни *BFS* на дељеној меморији буде избалансиран. Штавише, истраживање локације података такође може убрзати процес паралелизације.

Многи паралелни *BFS* алгоритми на дељеној меморији могу се поделити у два типа:

1. Приступ усредсређен на контејнере и
2. Приступ усредсређен на темена

У оквиру приступа усредсређеним на контејнер, креирају се две структуре података за складиштење тренутних граничних и следећих граничних темена. Скуп следећих граничних темена се пребацује на скуп тренутних граничних темена на последњем кораку сваке итерације. Постоји компромис између цене за синхронизацију и локације података према месту где се подаци чувају. Ове две структуре података могу се држати у сваком ентитету за обраду (као што је нит) који подржава локализацију података, али захтева додатне механизме за балансирање оптерећења. Алтернативно, они могу бити глобални како би обезбедили имплицитно балансирање оптерећења, где се специјалне структуре података користе за истовремени приступ ентитета за обраду. Али тада ће ти ентитети за обраду радити истовремено и потребно је више напора за синхронизацију.

Поред тога, може се оптимизовати организација података контејнера. Типична структура података у серијском *BFS*-у и неком паралелном *BFS*-у је *FIFO* ред, јер је једноставан и брз, с обзиром да операција уметања и брисања кошта само константно време.

Друга алтернатива је структура врће. Операција уметања у врећу траје $\mathcal{O}(\log n)$ времена у најгорем случају, док је потребно само константно амортизовано време које је брзо као и *FIFO*. Штавише, уједињење двеју врћа траје $\mathcal{O}(\log n)$ времена где је n број елемената у мањој врћи. Операција раздвајања двеју врћа такође траје $\mathcal{O}(\log n)$ времена. Уз помоћ структуре врћа, одређени број темена (према параметру грануларности) се чува у једној врћи и структура врће постаје основни паралелни ентитет. Штавише, редуктор се може комбиновати са структуром врће за паралелно уписивање темена и њихово ефикасно прелажење.

Приступ усредсређен на темена третира теме као паралелни ентитет, што омогућава паралелну итерацију. Свако теме је додељено паралелном ентитету. Овај приступ усредсређен на темена може добро да функционише само ако је дубина графа веома мала. Дубина графа у *BFS*-у је дефинисана као максимално растојање било ког темена у графу до изворног темена. Према томе, приступ усредсређен на темена је веома погодан за *GPU*-ове ако је свака нит мапирана на тачно једно теме.

Глава 3

Имплементација

Претходно представљени алгоритми [2.1, 2.2] су у овом раду имплементарани у програмском језику *C* коришћењем алата *OpenMPI* и *OpenMP* за олакшање паралелног програмирања у окружењу са дистрибуираном односно паралелном меморијом респективно. На крају овог поглавља ће бити наведени и резултати добијени овим имплементацијама.

3.1 Креирање насумичног графа

Имплементација функције која врши креирање насумичног графа је приказана на исечку 2. Фукција `void generate_random_graph(...)` прима параметре:

1. *long long** *graph* - показивач на низ целобројних вредности у оквиру кога ће бити бележени сегменти са суседима за одговарајуће теме,
2. *long long** *degrees* - низ целебројних вредности који ће бележити помераје у оквиру низа *graph*, како би се знало који сегмент низа припада ком темену,
3. *long long vertex_numb* - целобројна променљива која представља број чворова у графу,
4. *long long max_degrees* - целобројна променљива која представља максимални број потега који једно теме може да има и

5. *long long min_degrees* - целобројна променљива која представља минимални број потега који једно теме може да има.

На почетку (у оквиру линија 179 и 180) се решавају граничне вредности променљивих *max_degrees* и *min_degrees*, док се у оквиру линије 181 врши иницијализација целобројне промељиве *start* на вредност 0, што ће бити померај првог темена у графу и кроз сваку итерацију ће се вредност ове промељиве акумулирати и бележити у низ помераја *degrees*. То се дешава у линијама 183-187, где се у петљи генерише насумична вредност у опсегу $[max\_degrees - min\_degrees]$ и додаје на претходну вредност променљиве *start*.

У линији 189 се врши алоцирање меморије за сам низ потега *graph*, док се у линијама 192-197 врши и сама иницијализација, односно генерисање насумичних потега. Кроз две for петља се пролази кроз низ *graph*, читају вредности у оквиру низа *degrees* и генеришу уникатне вредности у опсегу $[0 - vertex\_numb)$. Фуникција `void reset_options(...)` врши иницијализацију низа на опсег могућих вредности, док функција `long long get_unique_random(...)` из низа извлачи елемент са насумичне позиције, тај елемент враћа, а на његово место убацује последњи елемент из низа, што за резултат има избацивање резултујуће вредности из низа.

```
160 void reset_options(long long *options, long long size){
161     for(long long i = 0; i < size; i++)
162         options[i] = i;
163 }
164
165 long long get_unique_random(long long *options, long long maximum){
166     long long cursor = rand() % maximum,
167     tmp = options[cursor];
168     options[cursor] = options[maximum - 1];
169     return options[maximum - 1] = tmp;
170 }
171
172 void generate_random_graph(
173     long long** graph,
174     long long* degrees,
175     long long vertex_numb,
176     long long max_degrees,
177     long long min_degrees
178 ){
179     long long max_deg = vertex_numb < max_degrees ? vertex_numb : max_degrees,
180         min_deg = (max_deg > min_degrees) ? min_degrees : max_deg,
181         start = 0;
182
183     for(long long i = 0; i < vertex_numb; i++){
184         degrees[i] = start;
185         start += rand() % (max_deg - min_deg + 1) + min_deg;
186     }
187     degrees[vertex_numb] = start;
188
189     long long *G = (long long*) malloc(sizeof(long long) * start),
190         *options = (long long*) malloc(sizeof(long long) * vertex_numb);
191
192     for(long long i = 0; i < vertex_numb; i++){
193         reset_options(options, vertex_numb);
194         long long range = degrees[i + 1] - degrees[i];
195         for(long long j = 0; j < range; j++)
196             G[degrees[i] + j] = get_unique_random(options, vertex_numb - j);
197     }
198
199     *graph = G;
200
201     free(options);
202 }
```

Изворни код 2: *Генерисање насумичног графа*

3.2 Секвенцијална имплементација *BFS* алгоритма

Што се секвенцијалне имплементације алгоритма за претрагу по ширини тиче, он је дат у оквиру исечка 3. Функција `void bfs_seq(...)` прима параметре:

1. *long long* graph* - низ целобројних вредности који садржи сегменте за суседе одговарајућих темена,
2. *long long* degrees* - низ целобројних вредности који садржи померање у оквиру низа *graph*,
3. *long long vertex_numb* - целобројна вредност која представља број потега у графу, односно димензију низа *degrees* (*vertex_numb* + 1),

4. *long long start* - целобројна вредност која представља индекс почетног чвора у оквиру низа *degrees* и

5. *long long* distance* - низ целобројних вредности у оквиру којих ће бити бележено растојање од почетног чвора $start$ до чвора v_i представљено у виду броја суседа који деле ова два чвора.

На почетку, у оквиру линија 36-38, се низ у којем се бележе растојања иницијализује за сваки чвор v_i на бесконачност (највећа целобројна вредност), док је растојање до почетног чвора постављена на вредност 0. Након овога се, у оквиру линија 40 и 41 врши алоцирање меморије за низове fs и ns (*fs - frontier set; ns - next frontier set*). У низ fs се убацује индекс почетног чвора *start* (линија 43), а затим се у линијама 45, 46 и 47 врши иницијализација променљивих *level*, *count1* и *count2* респективно. Променљива *level* бележи тренутни ниво, односно дубину графа која се обрађује у итерацији, *count1* бележи дужину низа *fs*, док *count2* бележи дужину низа *ns*.

У линијама 49 до 65 се налази петља која итерира све док низ *fs* није празан, односно промељива *count1* има вредност већу од нуле. У линијама 51 до 60 се налази петља којом се итерирају елементи низа *fs*. Сваки елемент бива смештен у променљиву *node* која се у линијама 53-59 користи за рачунање помераја у оквиру низа *degrees* и поново у петљи итерира сегмент у низу *graph*, одакле се добијају суседи чвора *node* односно променљива *neighbour* за сваку итерацију, која када се употреби као индекс у окивур низа *distance* и очита вредност која је *INFINITY*, следи упис вредности *level* $+ 1$ на њено место као и убацивање самог индекса у низ *ns* који ће се итерирати у следећем нивоу.

Као последње инструкције у $while$ петљи у оквиру линија 61-64 врши се замена низова fs и ns, ажурирање променљивих које бележе њихову величину, као и инкрементирање вредности променљиве *level*. У линијама 67 и 68 се врши деалоцирање низова ns и fs респективно.

```
29  void bfs_seq(
30      long long* graph,
31      long long* degrees,
32      long long vertex_numb,
33      long long start,
34      long long* distance
35  ){
36      for(long long i = 0; i < vertex_numb; i++)
37          distance[i] = INFINITY;
38      distance[start] = 0;
39
40      long long *fs = (long long*) malloc(sizeof(long long) * vertex_numb),
41          *ns = (long long*) malloc(sizeof(long long) * vertex_numb);
42
43      fs[0] = start;
44
45      long long level = 0,
46          count1 = 1,
47          count2 = 0;
48
49      while(count1 > 0)
50      {
51          for(long long i = 0; i < count1; i++){
52              long long node = fs[i];
53              for(long long j = degrees[node]; j < degrees[node + 1]; j++){
54                  long long neighbour = graph[j];
55                  if(distance[neighbour] == INFINITY){
56                      distance[neighbour] = level + 1;
57                      ns[count2++] = neighbour;
58                  }
59              }
60          }
61          count1 = count2;
62          count2 = 0;
63          swap(&fs, &ns);
64          level++;
65      }
66
67      free(ns);
68      free(fs);
69  }
```

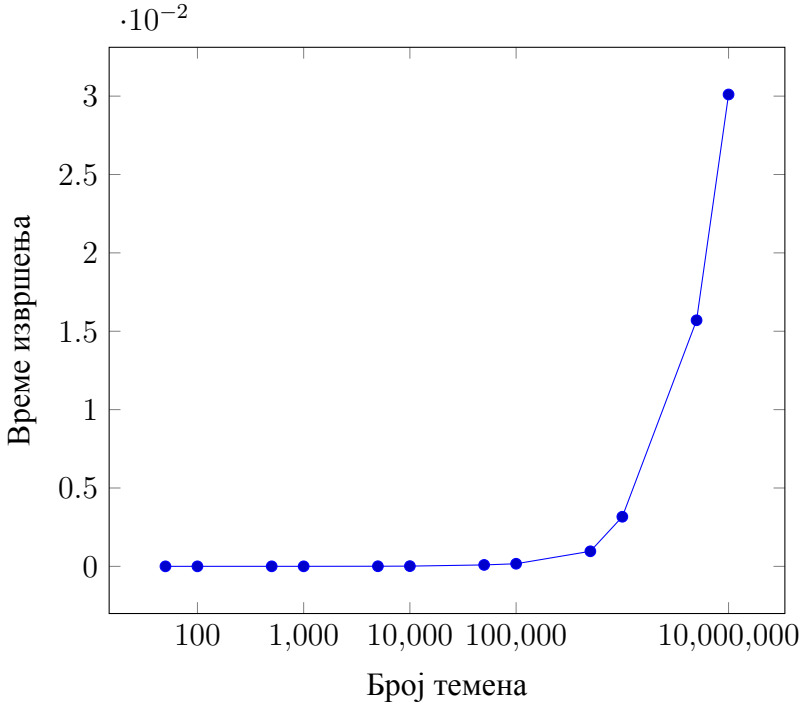
Изворни код 3: *Секвенцијални BFS*

3.2.1 Перформансе имеплементацираног алгоритма

Дијаграм који илуструје однос броја чворова и времена потребног за проналажење растојања до досежних чворова из почетног чвора је приказа на слици 3.1. Подаци су приказани и у оквиру табеле 3.1.

Број темена	Максималан број потега	Минималан број потега	Време извршавања
50	10	1	0.000002
100	10	1	0.000002
500	10	1	0.000003
1000	10	1	0.000003
5000	10	1	0.000008
10000	10	1	0.000016
50000	10	1	0.000092
100000	10	1	0.000167
500000	10	1	0.000962
1000000	10	1	0.003166
5000000	10	1	0.015701
10000000	10	1	0.030105

Табела 3.1: *Перформансе секвенцијалног алгоритма са променом броја чворова*



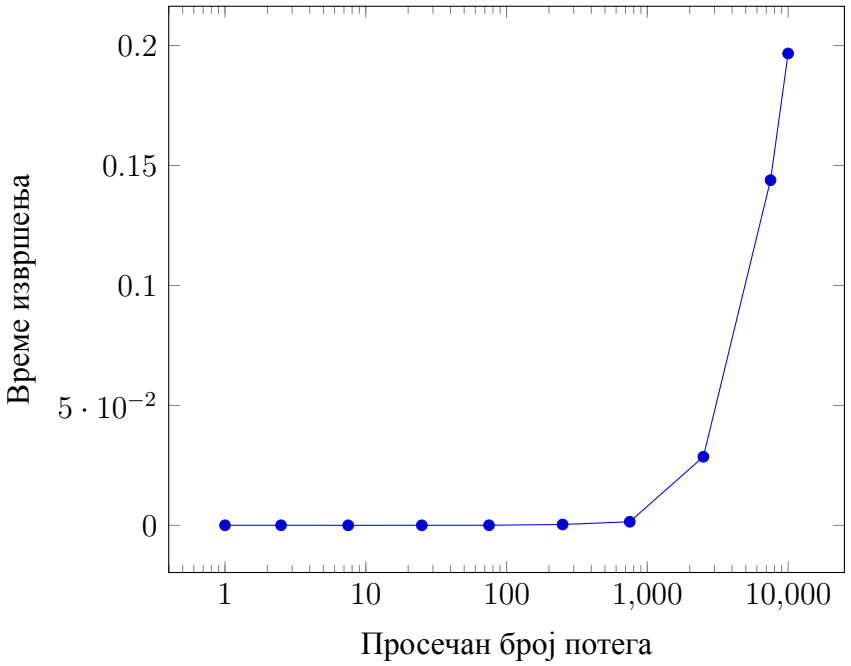
Слика 3.1: *Зависност брзине извршења од броја темена*

Дијаграм који илуструје однос броја потега и времена потребног за проналажење

растојања до досежних чворова из почетног чвора је приказа на слици 3.2. Подаци су приказани и у оквиру табеле 3.2.

Број темена	Максималан број потега	Минималан број потега	Време извршавања
10000	1	1	0.000030
10000	5	1	0.000046
10000	10	5	0.000018
10000	50	10	0.000032
10000	100	50	0.000049
10000	500	100	0.000365
10000	1000	500	0.001497
10000	5000	1000	0.028588
10000	10000	5000	0.143870
10000	10000	10000	0.196691

Табела 3.2: *Перформансе секвенцијалног алгоритма са променом степена повезаности*



Слика 3.2: *Зависност брзине извршења од броја потега*

3.3 Паралелна имплементација *BFS* алгоритма са дистрибуираном меморијом и једнодимензионим партиционисањем чворова

У оквиру ове имплементације, граф је представљен користећи матрицу суседства, односно са два низа, један који представља назубљену матрицу потега и други који представља помераје за одговарајући чвор у матрици потега. Овакво представљање графа је објашњено у секцији 1.2.3. Функција која имплементира паралелни *BFS* са дистрибуираном меморијом је дефинисана у исечку 4. Функција `void bfs_dist(...)` прима:

1. *long long* graph* - низ целобројних вредности који садржи сегменте за суседе одговарајућих темена,
2. *long long* degrees* - низ целобројних вредности који садржи помераје у оквиру низа *graph*,
3. *long long vertex_numb* - целобројна вредност која представља број потега у графу, односно димензију низа *degrees* (*vertex_numb* + 1),
4. *long long start* - целобројна вредност која представља индекс почетног чвора у оквиру низа *degrees* и
5. *long long* distance* - низ целобројних вредности у оквиру којих ће бити бележено растојање од почетног чвора $start$ до чвора v_i представљено у виду броја суседа који деле ова два чвора.

Подразумева се да је граф иницијализован у процесу са ранком *MASTER* коришћењем функције 2, док су сви процеси позвали функцију `void bfs_dist(...)`. У оквиру линија 78-83 се врши стандардна иницијализација променљивих *rank* и *size* за генерални комуникатор *MPI_COMM_WORLD*, док се у оквиру променљиве *work_load* рачуна целобројна вредност која представља број индекса који треба да обради сваки од процесора $work\_load = \left\lceil \frac{|V|}{|P|} \right\rceil$.

```
71  void bfs_dist(
72      long long* graph,
73      long long* degrees,
74      long long vertex_numb,
75      long long start,
76      long long* distance
77  ){
78      int rank, size;
79      long long work_load;
80      MPI_Comm_size(MPI_COMM_WORLD, &size);
81      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
82
83      work_load = vertex_numb / size + 1;
84
85      MPI_Bcast(degrees, vertex_numb + 1, MPI_LONG_LONG, MASTER, MPI_COMM_WORLD);
86      if(rank)
87          graph = (long long*) malloc(sizeof(long long) * degrees[vertex_numb]);
88      MPI_Bcast(graph, degrees[vertex_numb], MPI_LONG_LONG, MASTER, MPI_COMM_WORLD);
89
90      long long *d = (long long*) malloc(sizeof(long long) * vertex_numb);
91      for(long long i = 0; i < vertex_numb; i++)
92          d[i] = INFINITY;
93      d[start] = 0;
94
95      long long *F = (long long*) malloc(sizeof(long long) * vertex_numb),
96          *N = (long long*) malloc(sizeof(long long) * size * vertex_numb),
97          *N_recv = (long long*) malloc(sizeof(long long) * vertex_numb),
98          *N_size = (long long*) malloc(sizeof(long long) * size);
99
100     long long level = 0,
101         F_count,
102         F_global_count,
103         my_size;
104
105     while(1) {
106
107         F_count = F_global_count = 0;
108
109         for(long long i = rank * work_load; i < (rank + 1) * work_load; i++)
110             if(i < vertex_numb && d[i] == level)
111                 F[F_count++] = i;
112
113         MPI_Allreduce(&F_count, &F_global_count, 1, MPI_LONG_LONG, MPI_BOR, MPI_COMM_WORLD);
114
115         if(F_global_count == 0)
116             break;
117
118         for(long long i = 0; i < size; i++)
119             N_size[i] = 0;
120
121         for(long long i = 0; i < F_count; i++)
122         {
123             long long current_node = F[i];
124
125             for(long long j = degrees[current_node]; j < degrees[current_node + 1]; j++)
126             {
127                 long long neighbour_node = graph[j],
128                     proc = neighbour_node / work_load,
129                     k = 0;
130
131                 while(k < N_size[proc] && N[proc * vertex_numb + k++] != neighbour_node);
132
133                 if(k == N_size[proc])
134                     N[proc * vertex_numb + N_size[proc]++] = neighbour_node;
135             }
136         }
137
138         for(int i = 0; i < size; i++)
139         {
140             MPI_Sendrecv(&N_size[i], 1, MPI_LONG_LONG, i, level, &my_size, 1, MPI_LONG_LONG, i, level, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
141             MPI_Sendrecv(&N[i * vertex_numb], N_size[i], MPI_LONG_LONG, i, level, N_recv, my_size, MPI_LONG_LONG, i, level, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
142
143             for(long long j = 0; j < my_size; j++)
144                 if(d[N_recv[j]] == INFINITY)
145                     d[N_recv[j]] = level + 1;
146         }
147
148         level++;
149     }
150
151     MPI_Reduce(d, distance, vertex_numb, MPI_LONG_LONG, MPI_MIN, MASTER, MPI_COMM_WORLD);
152
153     free(d);
154     free(N_size);
155     free(N_recv);
156     free(N);
157     free(F);
158 }
```

Изворни код 4: *Паралелна имплементација BFS алгоритма са дистрибуираном меморијом и једнодимензионим партиционисањем чворова*

С обзиром да се граф налази у два низа *degress* и *graph* који су иницијализовани у оквиру процеса са ранком *rank*, неопходно је проследити податке осталим процесима, што се одвија у оквиру линија 85 и 88 респективно. У оквиру линија 86 и 87 се филтрирају не*MASTER* процеси, јер они немају алоциран низ *graph* (величина овог низа није позната унапред, већ се насумично добија).

У оквиру линија 90-93 се врши иницијализација локалног низа за сваки процес, у који се бележе растојања стартног чвора на *INFINITY*, док се растојање до стартног чвора поставља на 0.

У оквиру линија 95-98 се врши иницијализација низова *F*, *N*, *N_recv* и *N_size*. Низови *F* и *NS* имају идентичну улогу као и у алгоритму 2.1, док низови *N_recv* и *N_size* бележе индексе који се размењују са осталим процесима у итерацији и величину сваког од низева који прву шаљу одређеном процесу, а када размене величине, примају низ те дужине, респективно.

У оквиру линија 100-103 се врши иницијализација променљивих *level*, *F_count*, *F_global_count* и *my_size*. Променљива *level* бележи тренутни ниво итерације која се обрађује, променљива *F_count* бележи локалну дужину низа *F* за процес, док променљива *F_global_count* служи да би се у њу касније примила редукована вредност суме *F_count* променљивих од свих процеса. Променљива *my_size* се користи да би се у њу примила вредност која представља дужину низа који се размењује са процесима при разврставању низа локалних темена.

У оквиру линије 105 започиње бесконачна $while$ петља коју извршавају сви процеси све док је задовољен услов да F има елементе бар у једном процесу. Ова логика је имплементирана у оквиру линија 107-116, где сваки процес прво уписује сва темена која се налазе на растојању једнаком тренутној вредности променљиве $level$ у низ F. Након тога се врши $all - reduce$ операција над променљивама *F_count* и вредност уписује у *F_global_count*. Након тога се у линији 115 проверава вредност променљиве *F_global_count* и уколико је 0, биће нула за све процесе и сви ће изаћи на линији 116.

Даље, у линијама 118 и 119 се врши иницијализација елемената низа *N_size* на 0, с обзиром да ће се у том низу налазити дужине скупова N_q из алгоритма 2.1 који ће се размењивати са осталим процесима.

У оквиру петље у линијама 121-136 се врши пролаз кроз низ F сваког од процеса, проналазе се суседна темена (петља у линијама 125-135) и аутоматски разврставају и уписују у одговарајуће сегменте низа N, док се величина сваког сегмента бележи

у оквиру низа *N_size*.

У оквиру петље на линијама 138-146 се врши размена величина низова N_q који се размењују са процесом q на основу алгоритма 2.1, а затим се и врши размена тих низова и уписује у низ *N_recv*. Из овог низа се разматрају темена која представљају локална гранична темена за процес и уколико се није забележило растојање до њих, то се дешава у линијама 143-145. На крају $while$ петље се врши инкрементирање промељиве *level*, чиме се отпочиње следећа итерација.

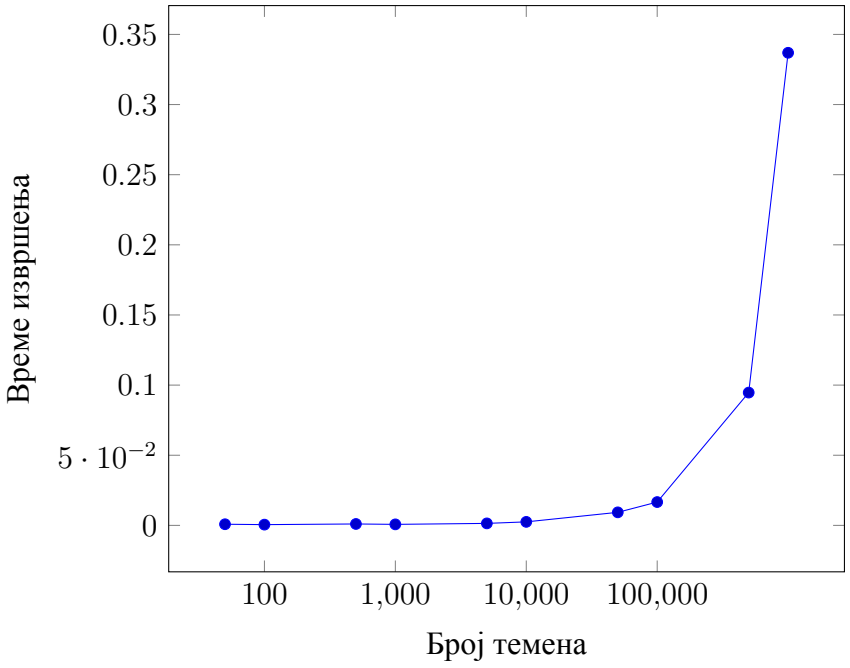
Када више ниједан процес нема гранична темена, то значи да су се од почетног чвора истражила сва темена до којих постоје потези, те се у оквиру линије 151 врши редукција локалног низа растојања d и редуковане вредности уписују у низ *distance*. У оквиру линија 153-157 се врши деалоцирање низова које је алоцирала функција и то у обрнутом редоследу, што су и њене последње нструкције.

3.3.1 Перформансе паралелне имплементације *BFS* алгоритма са једнодимензионим партиционисањем

Дијаграм који илуструје однос броја чворова и времена потребног за проналажење растојања до досежних чворова из почетног чвора је приказа на слици 3.3. Подаци су приказани и у оквиру табеле 3.3.

Број темена	Максималан број потега	Минималан број потега	Време извршавања	Број процеса
50	10	1	0.000818	10
100	10	1	0.000531	10
500	10	1	0.001006	10
1000	10	1	0.000723	10
5000	10	1	0.001419	10
10000	10	1	0.002502	10
50000	10	1	0.009297	10
100000	10	1	0.016624	10
500000	10	1	0.094666	10
1000000	10	1	0.336910	10

Табела 3.3: *Перформансе паралелне имплементације BFS алгоритма са дисртибуираном меморијом и једнодимензионим партиционисањем са променом броја чворова*

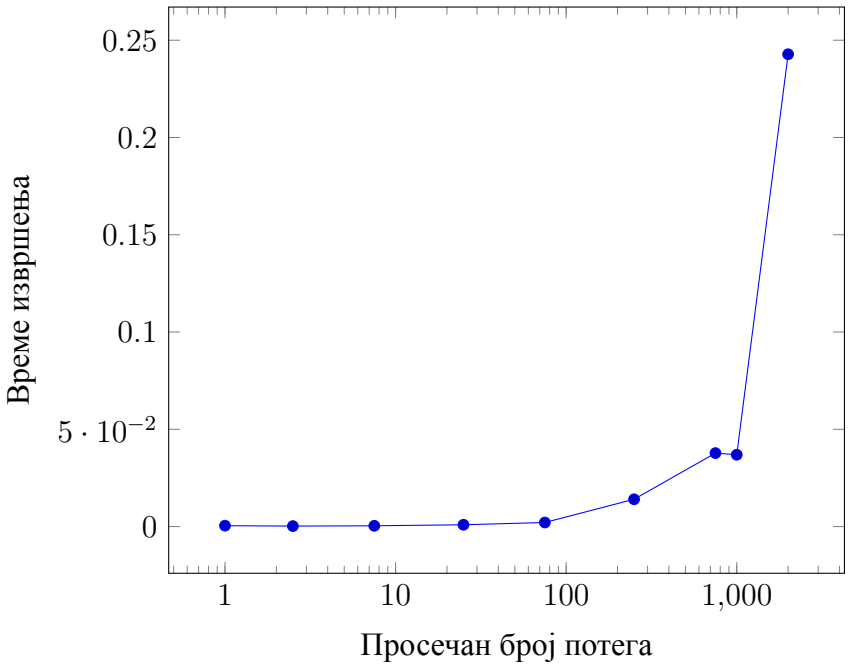


Слика 3.3: *Зависност брзине извршења паралелне имплементације BFS алгоритма са дистрибуираном меморијом и једнодимензионим партиционисањем од броја темена*

Дијаграм који илуструје однос броја потега и времена потребног за проналажење растојања до досежних чворова из почетног чвора је приказа на слици 3.4. Подаци су приказани и у оквиру табеле 3.4.

Број темена	Максималан број потега	Минималан број потега	Време извршавања	Број процеса
1000	1	1	0.000453	10
1000	5	1	0.000244	10
1000	10	5	0.000398	10
1000	50	10	0.000926	10
1000	100	50	0.002091	10
1000	500	100	0.014035	10
1000	1000	500	0.037758	10
1000	1000	1000	0.036923	10
2000	2000	2000	0.242799	10

Табела 3.4: *Перформансе паралелне имплементације BFS алгоритма са дисртибуираном меморијом и једнодимензионим партиционисањем са променом степена повезаности*



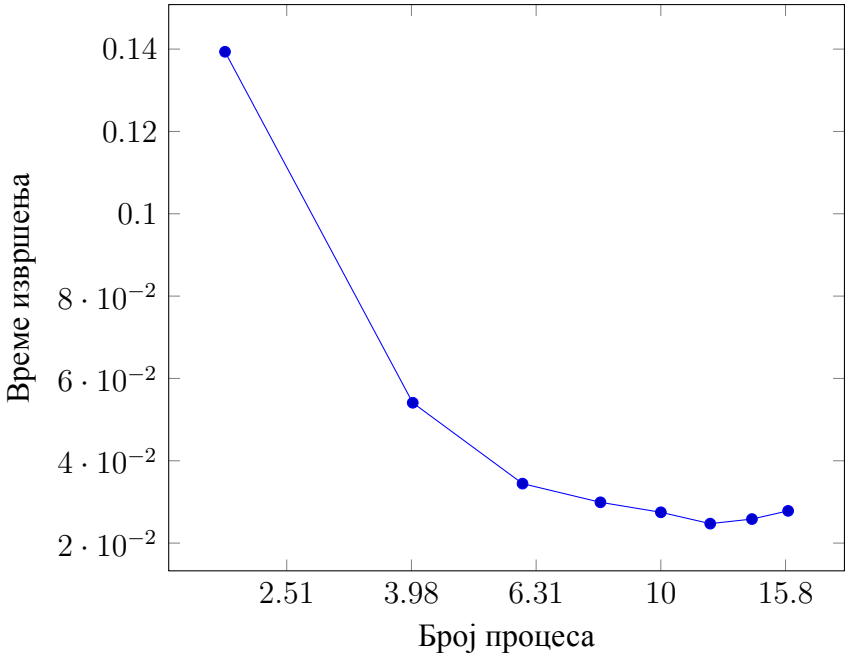
Слика 3.4: *Зависност брзине извршења паралелне имплементације BFS алгоритма са дистрибуираном меморијом и једнодимензионим партиционисањем од броја потега*

Дијаграм који илуструје однос броја процеса и времена потребног за пронала-

жење растојања до досежних чворова из почетног чвора је приказа на слици 3.5. Подаци су приказани и у оквиру табеле 3.5.

Број темена	Максималан број потега	Минималан број потега	Време извршавања	Број процеса
1000	1000	100	0.139349	2
1000	1000	100	0.054088	4
1000	1000	100	0.034449	6
1000	1000	100	0.029910	8
1000	1000	100	0.027482	10
1000	1000	100	0.024727	12
1000	1000	100	0.025838	14
1000	1000	100	0.027829	16

Табела 3.5: *Перформансе паралелне имплементације BFS алгоритма са дисртибуираном меморијом и једнодимензионим партиционисањем са променом броја процеса*



Слика 3.5: *Зависност брзине извршења паралелне имплементације BFS алгоритма са дистрибуираном меморијом и једнодимензионим партиционисањем од броја процеса*

3.4 Паралелна имплементација *BFS* алгоритма са дељеном меморијом и једнодимензионим партиционисањем чворова

Имплементација је одрађена користећи *OpenMP* алат за програмски језик *C* у оквиру кога је имплементирана следећа функција. Граф је и у овој имплементацији представљен користећи матрицу суседства, односно два низа, један који представља назубљену матрицу потега и други који представља помераје за одговарајући чвор у матрици потега. Функција која имплементира паралелни *BFS* са дељеном меморијом је дефинисана у исечку 5. Функција `void bfs_par(...)` прима:

1. *long long* graph* - низ целобројних вредности који садржи сегменте за суседе одговарајућих темена,
2. *long long* degrees* - низ целобројних вредности који садржи помераје у оквиру низа *graph*,
3. *long long vertex_numb* - целобројна вредност која представља број потега у графу, односно димензију низа *degrees* (*vertex_numb* + 1),
4. *long long start* - целобројна вредност која представља индекс почетног чвора у оквиру низа *degrees*,
5. *long long* distance* - низ целобројних вредности у оквиру којих ће бити бележено растојање од почетног чвора $start$ до чвора v_i представљено у виду броја суседа који деле ова два чвора и
6. *long long num_threads* - целобројну вредност која представља жељени број нити са којима желимо да извшимо функцију.

```
71  void bfs_par(
72      long long* graph,
73      long long* degrees,
74      long long vertex_numb,
75      long long start,
76      long long* distance,
77      long long num_threads
78  ){
79      long long F_global_count = 0,
80          work_load = vertex_numb / num_threads + 1;
81
82      long long *N = (long long*) malloc(sizeof(long long) * num_threads * num_threads * vertex_numb),
83          *N_size = (long long*) malloc(sizeof(long long) * num_threads * num_threads);
84
85      #pragma omp parallel num_threads(num_threads)
86      {
87          long long thread_num = omp_get_thread_num();
88
89          #pragma omp for
90          for(long long i = 0; i < vertex_numb; i++)
91              distance[i] = INFINITY;
92
93          #pragma omp single
94              distance[start] = 0;
95
96          long long* F = (long long*) malloc(sizeof(long long) * vertex_numb),
97                  * dups = (long long*) malloc(sizeof(long long) * num_threads * vertex_numb),
98                  F_count,
99                  level = 0;
100
101         for(long long i = 0; i < num_threads * vertex_numb; dups[i++] = 0);
102
103         while(1){
104
105             F_count = 0;
106
107             for(long long i = 0; i < work_load; i++){
108                 long long index = thread_num * work_load + i;
109                 if(index < vertex_numb && distance[index] == level){
110                     F[F_count++] = index;
111                 }
112             }
113
114             #pragma omp single
115                 F_global_count = 0;
116
117             #pragma omp critical
118                 F_global_count |= F_count;
119
120             #pragma omp barrier
121
122             if(F_global_count == 0)
123                 break;
124
125             #pragma omp for
126             for(long long i = 0; i < num_threads * num_threads; i++)
127                 N_size[i] = 0;
128
129             for(long long i = 0; i < F_count; i++)
130             {
131                 long long node = F[i];
132                 for(long long j = degrees[node]; j < degrees[node + 1]; j++){
133                     long long neighbour = graph[j],
134                         thread_dest = neighbour / work_load,
135                         offset = thread_dest * num_threads + thread_num;
136                     if(dups[thread_dest * num_threads + neighbour] == 0){
137                         N[offset * vertex_numb + N_size[offset]++] = neighbour;
138                         dups[thread_dest * num_threads + neighbour] = 1;
139                     }
140                 }
141             }
142
143             #pragma omp barrier
144
145             for(long long i = 0; i < num_threads; i++)
146             {
147                 long long offset = thread_num * num_threads + i;
148                 for(long long j = 0; j < N_size[offset]; j++)
149                     if(distance[N[offset * vertex_numb + j]] == INFINITY)
150                         distance[N[offset * vertex_numb + j]] = level + 1;
151             }
152
153             level++;
154         }
155
156         free(dups);
157         free(F);
158     }
159
160     free(N_size);
161     free(N);
162 }
```

Изворни код 5: *Паралелна имплементација BFS алгоритма са дистрибуираном меморијом и једнодимензионим партиционисањем чворова*

У оквиру линија 79-83 се врши иницирање дељених променљивих *F_global_count*, *work_load*, низа *N* и низа *N_size*. С обзиром да овде нећемо имати комуникацију између нити, а коришћење синхронизације је јако скупоцено за перформансе, нећемо штедети на меморији и низ *N* ће бити алоциран на величину од $num\_threads^2 \cdot vertex\_numb$, где ће свака нит имати свој сегмент у оквиру ког може уписивати темена која припадају осталим нитима, из којег ће остале нити касније моћи да преузму локална темена која су им додељена. Идеја је да постоји тродимензиони низ $N[receiving\_thread][sending\_thread][vertexes]$.

Паралелни регион се креира у оквиру сегмента у линијама 85-158 са жељеним бројем нити. У оквиру линије 87 се рачуна колико нити је креирано у паралелном региону и та врдност смешта у локалну променљиву $thread\_numb$. У оквиру линија 89 и 93 се врши расподела посла тако да све нити врше иницијализацију елемената дељеног вектора *distance* на вредност *INFINITY*, а једна нит касније индекс вредност елемента $distance[start]$ коригује вредношћу 0.

Након овога се у линијама 96-99 врши иницијализација осталих локалних променљивих. Променљива *F* ће бити локални низ где ће свака нит бележити своја локална темена која су гранична за тренутну итерацију. Променљива *F_count* ће бележити локалну вредност дужине низа *F*. Локална променљива *level* ће бележити ниво, односно дубину графа до које се дошло у тренутној итерацији, док ће локални низ $dups$ бележити које индексе је тренутна нит уписала у низ N за одговарајућу нит. у линији 101 се елементи овог вектора иницијализују на 0, с обзиром да још увек није покренуто уношење вредности, а ово је потребно одрадити само једном, с обзиром да не желимо да разматрамо темена која су већ послата одговарајучој нити на обраду.

У оквиру линије 103 се отпочиње бесконачна *while* петља коју извршавају све нити засебно. Инструкција на линији 105 иницијализује променљибу којом се бележи величину низа *F* на 0. У оквиру линија 107-112 се за одговарајући сегмент који је додељен локалној нити врши тражење оних темена који за своје растојање од почетног чвора имају растојање једнако дубини графа тренутне итерације. Логика линија 114-123 јесте да одради редукцију величина свих локалних низова F, упишу у дељиву промељиву и уколико је вредност 0, то значи да смо истражили сва темена до којих имамо потеге и да треба да се изађе из бесконачне $while$ петље.

У линијама 125-127 се врши инцијализација свих елемената низа *N_size* на 0, а онда свака од нити извршава петљу у оквиру линија 129-141. Логика ове петље јесте да за сваки од локалних темена из низа *F* нађе суседе и уколико се они не налазе у низу дупликата *dups*, упише их у њега, као и на одговарајућу позицију и оквиру

тродимензионог низа *N*, из ког ће заинтересоване нити касније очитати вредности. С обзиром да је неопходно и опсег до ког су вредности ових елемената валидни, врши се и инкрементирање одговарајућег елементе у оквиру низа *N_size*. Сам померај у оквиру ова два низа се добија на следећи начин. Када целобројно поделимо вредност индекса за суседно теме са вредношћу количине посла који добија свака нит, добићемо идентификатор нити којем је додељено то суседно теме. Да би се добио померај од ког треба да уписује тренутна нит, идентификатор нити којој припада суседни чвор треба помножити са укупним бројем нити и на ту вредност додати идентификатор нити која врши упис. Овај померај ће се користити у оквиру низа *N_size*, а да бисмо добили померај у оквиру низа *N*, тај померај је неопходно помножити са укупним бројем чворова у графу и на њега додати број претхдно уписаних темена за тај сегмент, што је вредност из низа *N_size* са претходно израчунатим померајем.

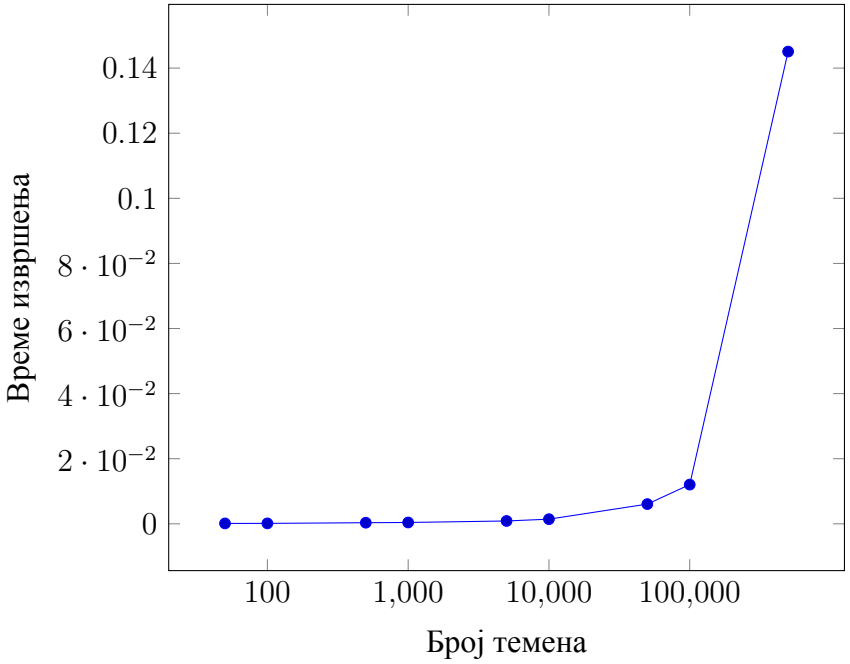
На линији 143 је постављена експлицитна баријера, с обзиром да је неопходно сачекати да све нити разврстају предстојећа гранична темена у низу *N*. На крају, у линијама 145-152, и *for* петљи коју извршава свака нит, врши се рачунање помераја у низу *N*, читају темена која су послата од нити са идентификатором i и уколико у дељеном вектору *distance* одговарајуће теме има вреднист *INFINITY*, обавиће се корекција на вредност $level + 1$. Овде није потребно вршити никакву синхронизацију, нити водити рачуна о услову трке, с обзиром да свака нит ради са индексима локалних темена и ни у једном тренутку не преписује вредности других нити. У последњим линијама кода се врши деалоцирање меморије за низове.

3.4.1 Перформансе имплементације алгоритма

Дијаграм који илуструје однос броја чворова и времена потребног за проналажење растојања до досежних чворова из почетног чвора је приказа на слици 3.6. Подаци су приказани и у оквиру табеле 3.6.

Број темена	Максималан број потега	Минималан број потега	Време извршавања	Број нити
50	10	1	0.000123	10
100	10	1	0.000135	10
500	10	1	0.000344	10
1000	10	1	0.000415	10
5000	10	1	0.000879	10
10000	10	1	0.001432	10
50000	10	1	0.006063	10
100000	10	1	0.012052	10
500000	10	1	0.145075	10

Табела 3.6: *Перформансе паралелне имплементације BFS алгоритма са дељеном меморијом и једнодимензионим партиционисањем са променом броја чворова*

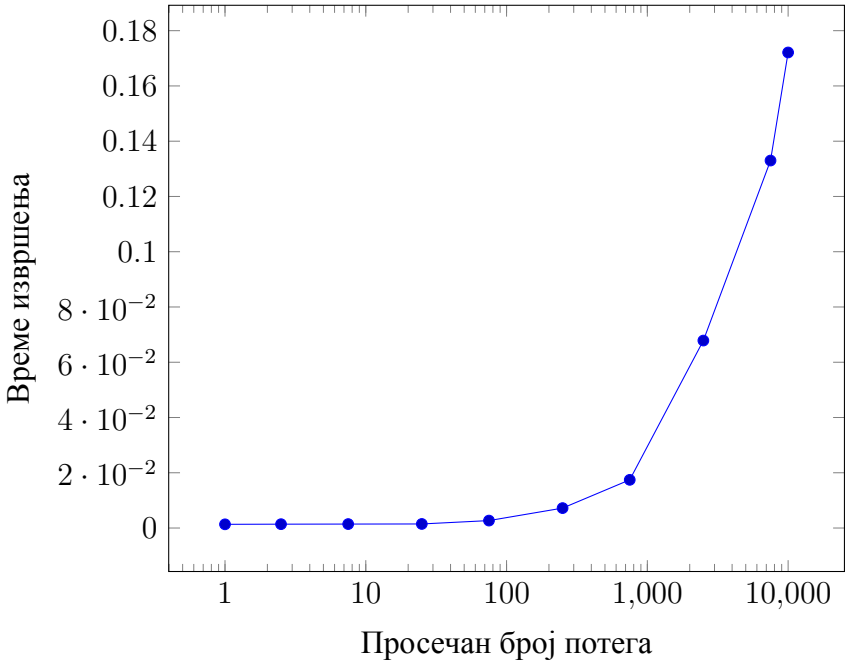


Слика 3.6: *Зависност брзине извршења паралелне имплементације BFS алгоритма са дељеном меморијом и једнодимензионим партиционисањем од броја темена*

Дијаграм који илуструје однос броја потега и времена потребног за проналажење растојања до досежних чворова из почетног чвора је приказа на слици 3.7. Подаци су приказани и у оквиру табеле 3.7.

Број темена	Максималан број потега	Минималан број потега	Време извршавања	Број нити
10000	1	1	0.001341	10
10000	5	1	0.001393	10
10000	10	5	0.001431	10
10000	50	10	0.001470	10
10000	100	50	0.002691	10
10000	500	100	0.007210	10
10000	1000	500	0.017438	10
10000	5000	1000	0.067848	10
10000	10000	5000	0.132995	10
10000	10000	10000	0.172114	10

Табела 3.7: *Перформансе паралелне имплементације BFS алгоритма са дељеном меморијом и једнодимензионим партиционисањем са променом степена повезаности*



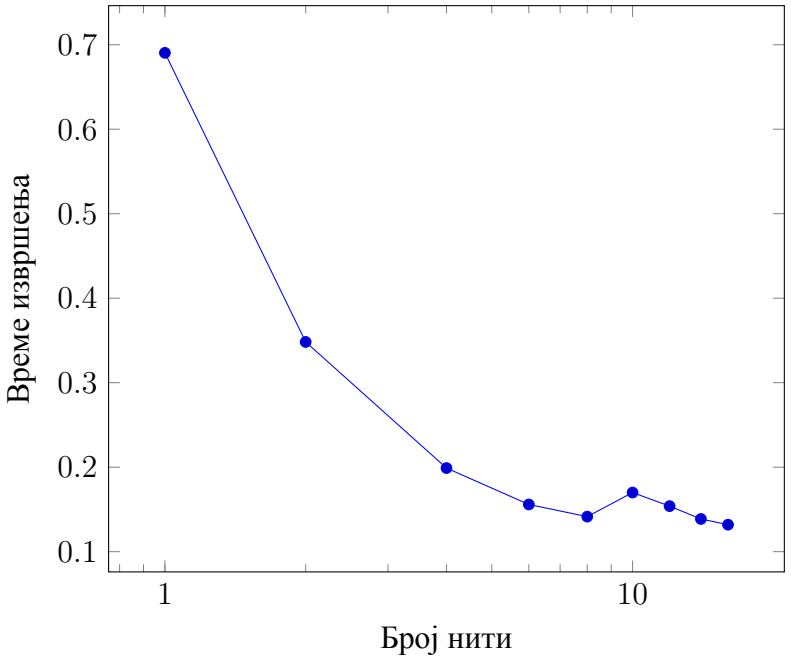
Слика 3.7: *Зависност брзине извршења паралелне имплементације BFS алгоритма са дељеном меморијом и једнодимензионим партиционисањем од броја потега*

Дијаграм који илуструје однос броја процеса и времена потребног за проналажење растојања до досежних чворова из почетног чвора је приказа на слици 3.8.

Подаци су приказани и у оквиру табеле 3.8.

Број темена	Максималан број потега	Минималан број потега	Време извршавања	Број нити
10000	10000	10000	0.690409	1
10000	10000	10000	0.348251	2
10000	10000	10000	0.198964	4
10000	10000	10000	0.155874	6
10000	10000	10000	0.141484	8
10000	10000	10000	0.169981	10
10000	10000	10000	0.153905	12
10000	10000	10000	0.138699	14
10000	10000	10000	0.131912	16

Табела 3.8: *Перформансе паралелне имплементације BFS алгоритма са дељеном меморијом и једнодимензионим партиционисањем са променом броја нити*

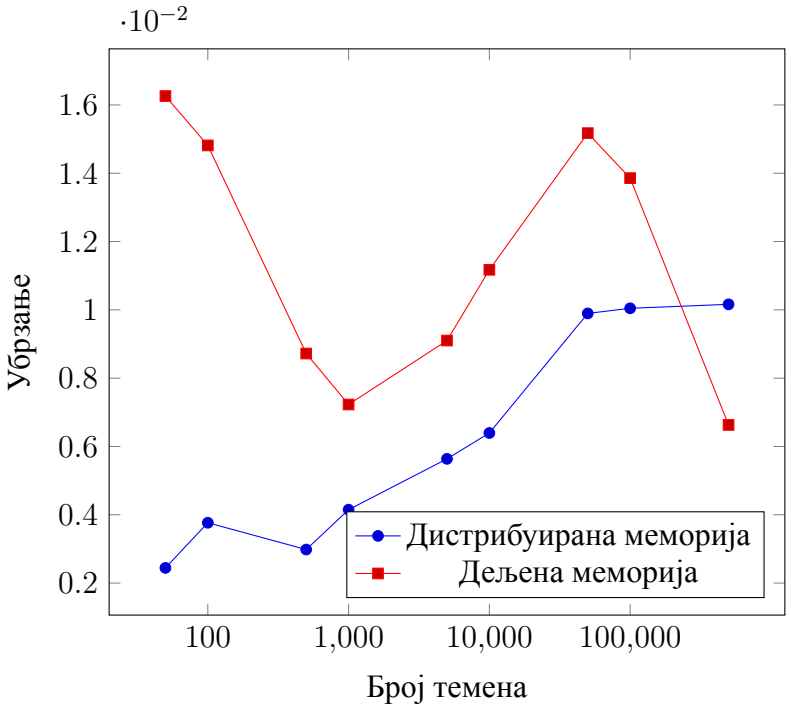


Слика 3.8: *Зависност брзине извршења паралелне имплементације BFS алгоритма са дељеном меморијом и једнодимензионим партиционисањем од броја нити*

3.5 Упоређивање резултата

На крају овог поглавља, односно у оквиру ове секције биће упоредо истакнут однос секвенцијалне и паралелних имплементација, односно убрзања која се постижу у зависности од броја темена, степена повезаности графова као и броја ентитета који паралелно обављају посао.

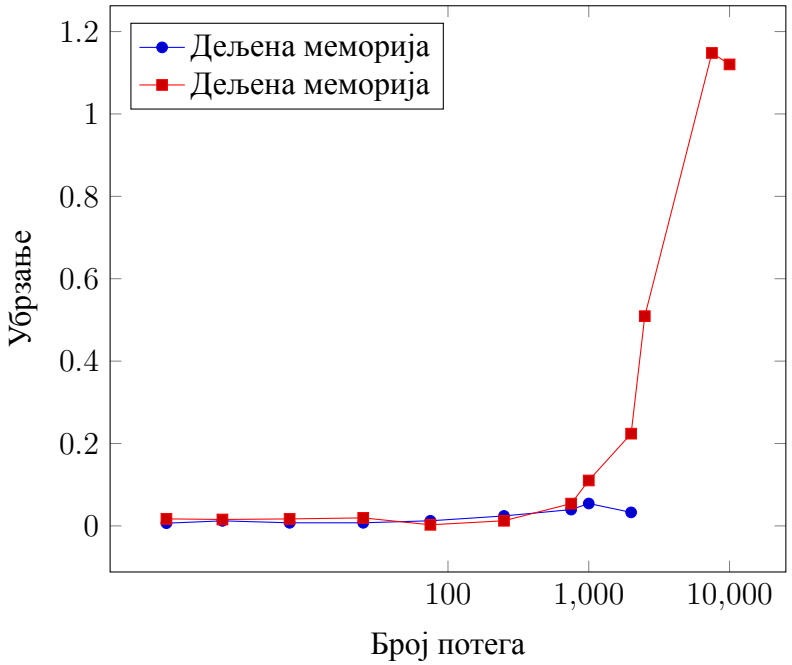
На дијаграму 3.9 представљен је однос брзине извршења секвенцијалне имплементације и паралелних имплементација у окружењу са дистрибуираном и дељеном меморијом, у односу на број темена графа. Ентитети који паралелно обављају посао, као и број потега је овде константан. Са дијаграма се нажалост може уочити да обе имплементације немају позитивне резултате.



Слика 3.9: *Убрзање паралелних имплементација у односу на број темена графа*

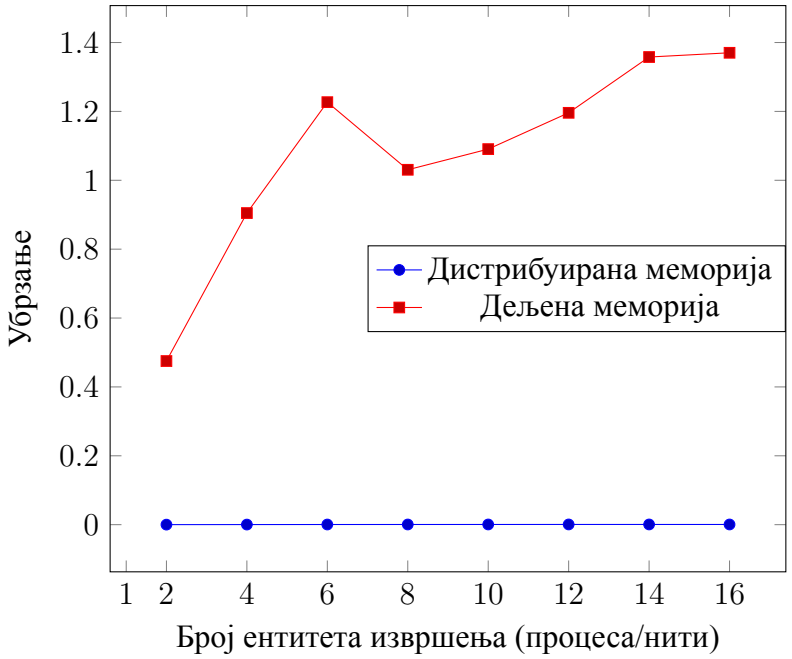
На дијаграму 3.10 представљен је однос брзине извршења секвенцијалне имплементације и паралелних имплементација у окружењу са дистрибуираном и дељеном меморијом, у односу на број потега графа. Ентитети који паралелно обављају посао, као и број темена графа су константни. Са дијаграма се може уочити да за број потега већи од 10^3, паралелна имплементација са дељеном меморијом достиже убрзање

до 1.14. Нажалост, имплементација у окружењу дистрибуиране меморије има негативне резултате.



Слика 3.10: *Убрзање паралелних имплементација у односу на степен повезаности графа*

Коначно, на дијаграму 3.11 представљен је однос брзине извршења секвенцијалне имплементације и паралелних имплементација у окружењу са дистрибуираном и дељеном меморијом, у односу на број ентитета који изврашавају посао у паралели. За окружења са дистрибуираном меморијом, то су процеси, док у окружењима са дељивом меморијом су то нити. Број потега, као и број темена графа су константни. Са дијаграма се може уочити да за број нити већи од 6, паралелна имплементација са дељеном меморијом достиже до чак 1.37. Нажалост, имплементација у окружењу дистрибуиране меморије ни овде нема позитивне резултате.



Слика 3.11: *Убрзање паралелних имплементација у односу на број паралелних ентитета извршења*

Глава 4

Закључак

У овом раду се детаљније размотрио проблем проналажења најкраћег пута, односно најмањег броја скокова, између два чвора у графу и то применом алгоритма за претрагу графа по ширини, познатог као *breadth-first traversal*. Имплементирани су алгоритми паралелне верзије овог алгоритма, користећи системе са дистрибуираном и дељеном меморијом. Такође, дат је и приказ резултата добијених примењујући поменуте имеплементације на насумично генерисаним графовима.

У првом одељку су дате теоријске основе графова. Обрађена је основна терминологија, идеје основних операција над њима, описане су идеје претраживања истих итд.. Такође, у оквиру овог одељка су описани и разни начини репрезентације графа, поједини од којих су коришћени ради олакшане обраде у паралелном окружењу.

У оквиру другог одељка су изнети описи појединих приступа у паралелизацији претраживања графа. Поред описа, ово поглавље садржи и идеје имплементирања сваког од појединих приступа, заједно са алгоритмима и описом истих.

Треће поглавље садржи имплементацију алгоритама за паралелно претраживање графа у дистрибуираним и системима са дељеном меморијом. Такође, на крају описа сваке од њих стоји приказ резултата добијених кроз њихову примену на насумично генерисаним графовима.

На основу анализе имплементација, може се закључити да је реализација паралелног претраживања графа могућа, али ни под разном једноставна и никако апсолутна. Убрзања до којих се дошло су ту искључиво јер се ради о вештачки синтезованим графовима, као и чињеница да та убрзања постоје само за случајеве високог

степена повезаности, односно паралелизације у оквиру окружења које ради са дељивом меморијом.

У оквиру секције која описује идеје које стоје иза паралелизације обраде графа по ширини су описани и методи који нису имплементирани у овом раду, а могуће је да би дали добре резултате. Њихова имплементација може бити корак ближе генералном решењу овог проблема, тако да се читаоци овог рада охрабрују да исту покушају.

Литература

[1] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3):563--581, 1977.

[1]