УНИВЕРЗИТЕТ У НОВОМ САДУ

ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
У НОВОМ САДУ

Огњен Зековић

Анализа графова за дијагностику кварова на мрежи FastTransitNetwork

Семинарски рад

- Мастер академске студије -

Нови Сад, фебруар 2026.

САДЖАЈ

1. Увод

У оквиру пројекта развијен је систем за анализу великих графова који омогућава брзу дијагностику кварова на мрежи. Систем имплементира три кључна алгоритма: BFS (Breadth-First Search) за одређивање достижности чворова, WCC (Weakly Connected Components) за идентификацију повезаних компоненти, и PageRank за одређивање приоритета надзора критичних чворова. Сви алгоритми су имплементирани у програмском језику Rust. За сваки алгоритам направљене су и секвенцијална и паралелна верзија, са циљем да се искористе вишејезгарни процесори и убрза обрада великих графова.

2. Алгоритми

2.1. BFS (Breadth-First Search)

BFS је алгоритам за претрагу графа који обилази чворове по нивоима удаљености од изворног чвора. Користи се за проналажење најкраћег пута (по броју ивица) од извора до свих осталих чворова.

2.1.1. Логика алгоритма

BFS користи ред структуру података за обилазак графа по нивоима:

1. **Иницијализација:**

- Постави удаљеност изворног чвора на 0
- Све остале удаљености постави на -1 (неодређено/недостижно)
- Дода изворни чвор у ред

2. **Итеративна обрада:**

- Извуче први чвор из реда (FIFO принцип)
- За сваког суседа тог чвора:
    - Ако сусед још није посећен (удаљеност = -1):
        - Нова удаљеност = удаљеност тренутног + 1
        - Дода га у ред за каснију обраду

3. **Завршетак:**

- Када се ред испразни, сви достижни чворови су обрађени
- Чворови са удаљеношћу -1 нису достижни из изворног чвора

2.1.2. Разлог за изабран присутп

Јасна је граница између нивоа и чворови на истом нивоу су независни што осигурава лаку паралелизацију без трке за ресурсима.

2.2. WCC (Weakly Connected Components)

WCC алгоритам идентификује слабо повезане компоненте у усмереном графу. Две компоненте су слабо повезане ако постоји пут између њих без обзира на смер ивица (граф се третира као неусмерен).

2.2.1. Логика алгоритма

WCC користи *Union-Find* структуру података:

1. **Иницијализација:**
    - Сваки чвор је у својој одвојеној компоненти
    - Сваки чвор је сам себи родитељ (parent[i] = i)
    - Почетна висина стабла је 0 (rank[i] = 0)
2. **Обрада ивица:**
    - За сваку ивицу (u, v) у графу:
        - Нађе корен (представника) компоненте у којој је root_u = find(u)
        - Нађе корен компоненте у којој је root_v = find(v)
        - Ако су у различитим компонентама (root_u ≠ root_v):
            - Спој их под једну: union(root_u, root_v)
3. **Главна петља:**
    - За сваки чвор i, find(i) враћа ID корена његове компоненте
    - Чворови са истим ID-ом су у истој компоненти
    - Преброје се различити бројеви у листи ID-јева

2.2.2. Разлог за изабран присутп

UnionFind структура садржи листу родитеља/корена (parent) сваког чвора која нам омогућава да лакше дођемо до уније чворова у истој компоненти спајањем једне под другу док не дођемо до исток корена.

Сваком претрагом корена фунцкијом find() директно повезујемо сваки чвор са тим кореном што убрзава сваки следећи позив исте операције. Затим поредимо нивое дубине (rank) сваке компоненте, односно ваког корена, и стављамо увек мањи ранг под већи чиме осигуравамо континуитет поређења нивоа. Интуитивна је паралелизација уније за сваку ивицу сваког чвора.

2.3. PageRank

PageRank је алгоритам који рангира чворове графа на основу њихове "важности". Чвор је важнији ако има више улазних веза, посебно ако те везе долазе од других важних чворова.

2.3.1. Логика алгоритма

PageRank се заснива на "Random Surfer" моделу :

1. Корисник почиње на насумичној страници
2. Са вероватноћом α (обично 0.85):
    - Кликне на насумични линк са те странице
3. Са вероватноћом (1-α):
    - "Телепортује се" на потпуно насумичну страницу
4. Понавља до конвергенције (док промена није мања од епсилон)

2.3.2. Разлог за изабран присутп

Ово је алгоритам који је представио *Google* и допуњује модел *Random Surfer* „телепортацијом”, односно скакањем на неки насумични чвор са одређеном вероватноћом како би свака група чворова добила прилику да прође кроз итерацију. Користе се два вектора *rank* и *new_rank* како би се очувала стара вредност нивоа чвора у истој итерацији. Сваки чвор независно дистрибуира свој ранг суседима али се користи атомична операција std::mem::swap() која само мења показиваче ради бржег преношења вредности без утрошка додатне меморије.

3. Репрезентација графа

3.1. Избор структуре података

За репрезентацију графа изабрана је тзв. *adjacency list* структура, односно за сваки чвор чувамо листу његових суседа:

Поље	Тип	Опис
num_nodes	usize	Укупан број чворова у графу
edges	Vec<Vec>	Низ који садржи листу свих суседа чвора

Табела 1 Опис класе Graph

3.2. Меморијска потрошња

За граф са 1,000,000 чворова и 10,000,000 ивица:

- num_nodes: 8 bytes (usize)
- edges: ~10,000,000 × 8 bytes ≈ 76 MB (usize показивачи)
- Укупно: ~76 MB

4. Стратегија верификације

4.1. Unit тестови

BFS тестови:

- Прост граф са 4 чвора
- Неповезани граф (две одвојене компоненте)
- Граф са циклусом

WCC тестови:

- Унија чворова
- Граф са више компоненти
- Потпуно повезан граф
- Потпуно неповезан граф

PageRank тестови:

- Циклични граф (сви чворови требају имати једнак ранг)
- Звездаст граф (сви имају исти ранг осим првог који има 0)
- Добијање најважнијих чворова

4.2. Поређење секвенцијалне и паралелне верзије

Кључна провера тачности је поређење резултата секвенцијалне и паралелне имплементације:

```
#[test]
    fn test_bfs_parallel_vs_sequential() {
        let graph = Graph {
            num_nodes: 5,
            edges: vec![
                vec![1, 2], // 0→1, 0→2
                vec![3],    // 1→3
                vec![3],    // 2→3
                vec![4],    // 3→4
                vec![],     // 4
            ],
        };

        let seq = bfs_sequential(&graph, 0);
        let par = bfs_parallel(&graph, 0, 4);

        for i in 0..graph.num_nodes {
            assert_eq!(seq[i], par[i], "Node {}: seq={}, par={}", i,
seq[i], par[i]);
        }
    }
```

Листинг 1 испитивање тачности паралелне верзије

На свим тест графовима (100 - 1,000,000 чворова), паралелне верзије дају идентичне резултате као секвенцијалне.

4.3. Тестирање на реалним графовима

Генерисани су графови различитих типова и величина:

- Линијски графови: Најгори случај за BFS (дубок, узак)

- Звездасти графови: Најбољи случај за паралелизацију

- Насумични графови: Реалистичан сценарио

- Disconnected графови: Провера WCC-а са више компоненти

5. Паралелизација

5.1. BFS

Како више нити може да покуша да постави дистанцу истог чвора, користимо AtomicI32 уместо Vec<i32> да избегнемо трку за ресурсима.

```
let dist: Vec<AtomicI32> = (0..graph.num_nodes)
      .map(|_| AtomicI32::new(-1))
      .collect();
```

Листинг 2 коришћење AtomicI32

Затим треба да обезбедимо да се свака нит посети само једном што можемо да урадимо поређењем и заменом вредности, такође атомична операција.

```
if dist[neighbor].compare_exchange(
      -1,
      level,
      Ordering::Relaxed,
      Ordering::Relaxed)
```

Листинг 3 провера вредности

Ordering::Relaxed само означава да ће бити атомска операција али да редослед других меморијских операција није битан. Не ослања се на податке које је друга нит променила нити утиче на друге нити, свака само проверава да ли је -1 или не.

```
let next_level: Vec<usize> = current_level
            .par_iter()
            .flat_map_iter(|&node| {
                graph.edges[node].iter().filter_map(|&neighbor| {
                    . . .
                })
            })
            .collect();
```

Листинг 4 паралелна обрада нивоа

Функција *par_iter()* аутоматски дели низ чворова на тренутном нивоу *current_level* између доступних нити. Свака нит обрађује комшије својих чворова независно. А collect() чека да све нити заврше пре него што се настави са следећим нивоом.

.

5.2. WCC

Као и у претходном случају и овде морамо да обезбедимо атомичан приступ ресурсима.

```
struct ConcurrentUnionFind {
    parent: Vec<AtomicUsize>,
}
```

Листинг 5 класа ConcurrentUnionFind

Више нити може истовремено да позива функцију за проналазак коренског чвора али иако не успе није проблем, покушаће поново док не дође до врха.

```
fn find(&self, mut x: usize) -> usize {
        loop {
            let parent = self.parent[x].load(Ordering::Acquire);

            if parent == x {
                return x;
            }

            let grandparent = self.parent[parent].load(Ordering::Acquire);

            let _ = self.parent[x].compare_exchange(
                parent, //expected
                grandparent,
                Ordering::Release,
                Ordering::Relaxed,
            );

            x = grandparent;
        }
    }
```

Листинг 6 провера вредности

Ordering::Acquire осигурава да ће бити видљиво све што је претходно постављено са Ordering::Release. Исто је и са union() операцијом, иако више нити успе да споји два чвора свакако ће доћи до истог корена, односно не ремети се ништа.

Затим остаје само да се паралелизује обрада ивица на тривијалан начин:

```
(0..graph.num_nodes).into_par_iter().for_each(|u| {
        for &v in &graph.edges[u] {
            uf.union(u, v);
        }
    });
```

Листинг 7 главна петља WCC

5.3. PageRank

Поново је неопходно избећи трку за ресурсима али у овом случају не може да се користи тип који одговара, а то је float, јер не постоји атомични float. Из тог разлога се користи unsigned уз неопходну конверзију.

```
let new_rank_atomic: Vec<AtomicU64> =
        (0..n).map(|_| AtomicU64::new(f64_to_bits(0.0))).collect();
```

Листинг 8 иницијализација атомичне float *вредности*

```
fn atomic_add_f64(atomic: &AtomicU64, increment: f64) {
    let mut current = atomic.load(Ordering::Acquire);

    loop {
        let current_f64 = f64_from_bits(current);
        let new_f64 = current_f64 + increment;
        let new = f64_to_bits(new_f64);
        match atomic.compare_exchange(current, new, Ordering::Release, Or-
dering::Acquire) {
            Ok(_) => return,
            Err(actual) => {
                current = actual;
            }
        }
    }
}
```

Листинг 9 атомично додавање f64 *вредности*

Затим свака нит обрађује део чвора:

```
(0..n).into_par_iter().for_each(|u| {
            let out_degree = graph.edges[u].len();

            if out_degree > 0 {
                let contrib = rank[u] / out_degree as f64;
                let weighted_contrib = alpha * contrib;

                for &v in &graph.edges[u] {
                    atomic_add_f64(&new_rank_atomic[v], weighted_contrib);
                }
            }
        });
```

Листинг 10 главна петља PageRank

Нити могу безбедно да додају у низ new_rank_atomic[v] истовремено.

Затим може паралелно да се претвори вредност назад у float тип и дода телепортација, односно да се скочи на следећу насумично одабрану групу чворова.

```
let new_rank: Vec<f64> = new_rank_atomic
            .par_iter()
            .map(|atomic| {
                let val = f64_from_bits(atomic.load(Ordering::Acquire));
                val + teleport
            })
            .collect();
```

Листинг 11 телепортација PageRank

6. Резултати и анализа перформанси

6.1. Експериментална поставка

Хардвер:

- Процесор: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
- Број физичких језгара: 4
- RAM: 16 GB
- ОС: Windows 11

Софтвер:

- Rust verzija: 1.92.0
- Compiler optimizations: --release
- Rayon thread pool за паралелизацију

Назив графа	Број чворова	Број ивица
Medium	100,000	1,000,000
Large	1,000,000	10,000,000

Табела 2 Табела графова

Конфигурације:

- Секвенцијално
- Паралелно са 2, 4, 8 нити
- 3 итерације по конфигурацији (узета просечна вредност)

6.2. Резултати мерења

Алгоритам	Граф	Секвенцијално	2 нити	4 нити	8 нити
BFS	Small	7.45	7.60	4.98	4.98
BFS	Large	174.66	135.37	77.99	58.51
WCC	Small	15.04	11.44	6.42	5.95
WCC	Large	296.65	125.69	82.11	60.57
PageRank	Small	363.39	302.32	212.23	175.55
PageRank	Large	7381.31	4427.69	2691.55	2179.36

Табела 3 Просечна времена извршавања (ms)

Алгоритам	Граф	2 нити	4 нити	8 нити
BFS	Small	0.98×	1.50×	1.50×
BFS	Large	1.29×	2.24×	2.98×
WCC	Small	1.31×	2.34×	2.53×
WCC	Large	2.36×	3.61×	4.90
PageRank	Small	1.20×	1.71×	2.07×
PageRank	Large	1.67×	2.74×	3.39×

Табела 4 Убрзање

7. Дискусија

7.1. Анализа убрзања

7.1.1. WCC

Најбоље перформансе показала је паралелизација WCC алгоритма, постиже убрзање од 4.90 пута на 8 нити за велики граф. За боље убрзање је заслужно:

1. Свака ивица може да се обради независно и оперцаије унију за различите парове не ометају једна другу
2. До конфликта долази само када две нити покушавају да споје исте компоненте, што је баш ретко, и свакако лако атомично решиво

7.1.2. PageRank

PageRank постиже 3.39× убрзање на великом графу са 8 нити. Атомичне операције atomic_add_f64(&new_rank[v], contrib); су скупе. Имамо и проблем популарних чворова, јер ако неки чвор има 1000 улазних ивица, то доводи до велике шансе да више нити покушава да ажурира његов ранг. Такође, може доћи до неравномерне расподеле посла где нека нит добије 10 чворова који су скупљи за обраду, због чега га остале чекају.

7.1.3. BFS

Ради по принципу синхронизације нити по нивоима, чиме се захтева баријера. После сваког нивоа све нити морају да чекају најспорију.

7.2. Идентификована уска грла

7.2.1. Атомичне операције

Атомичне оперцаије коштају сваки пут када приступимо ресурсу, дакле множи се проблем са бројем приступа у петљи. Једно од решења би наравно могло да буде додељивање бафера нитима, односно неког локалног низа за ажурирање, како бисмо на крају све нити синхронизовали. Међутим ово троши додатно меморије, грубо израчунато: 8 нити × 1M чворова × 8 bytes = 64MB додатно.

7.2.2. Подела посла међу нитима

Посао се дели у блокове исте дужине што је описано у 6.1.2. Проблем је што нема динамичке поделе посла.

8. Закључак

У оквиру пројекта развијен је систем за паралелну анализу великих графова у програмском језику Rust, са имплементацијама три кључна алгоритма: BFS за одређивање достижности чворова, WCC за идентификацију повезаних компоненти, и PageRank за рангирање важности чворова. Сваки алгоритам је имплементиран у секвенцијалној и паралелној верзији, са циљем убрзања обраде коришћењем вишејезгарних процесора.

Резултати тестова на графовима до 1,000,000 чворова показују значајна убрзања: WCC постиже најбоље перформансе са убрзањем од 4.90× на 8 нити, Page Rank постиже 3.39× убрзање, док BFS постиже 2.98× убрзање на великим графовима. *Adjacency list* репрезентација графа показала се као оптималан избор за графове, обезбеђујући меморијску ефикасност и брзу итерацију преко суседа чворова.

Анализа је открила да паралелизација није увек корисна, као на графовима испод 100,000 чворова, јер превише времена одлази на креирање нити и синхронизација превазилази добит од паралелне обраде. Главна уска грла идентификована су као атомичне операције у PageRank-у (које успоравају обраду популарних чворова), меморијска ограничења за веома велике графове, и баријера синхронизације по нивоима у BFS алгоритму. Упркос томе, постигнута убрзања и ефикасност (61-118% на 2-4 нити за WCC) демонстрирају да је Rust са Rayon библиотеком одличан избор за развој високоперформантних паралелних система.

Систем је практично применљив за анализу мрежа средњих величина (до неколико милиона чворова) и представља солидну основу за даљи развој, укључујући имплементацију алгоритма уз спољну меморију (external memory) за графове који не стају у RAM, и проширење на архитектуре дистрибуираних рачунара за граф-анализу на нивоу милијарди ивица.