Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004
DOI: https://doi.org/10.24867/27BE03Aleksic

PARALELNO PRETRAŽIVANJE GRAFA NA ARHITEKTURAMA SA DISTRIBUIRANOM I DELJENOM MEMORIJOM

PARALLEL BREADTH-FIRST SEARCH GRAPH TRAVERSAL ON COMPUTER ARCHITECTURES WITH DISTRIBUTED AND SHARED MEMORY

Stefan Aleksić, *Fakultet tehničkih nauka, Novi Sad*

Oblast – ELEKTROTEHNIKA I RAČUNARSTVO

Kratak sadržaj – *Glavni fokus ovog rada jeste pokušaj unapređenja algoritama pretrage grafova, pogotovo za grafove sa velikim brojem čvorova. Ovaj rad pruža informacije vezane za algoritme paralelne pretrage grafa, na računarskim arhitekturama sa distribuiranom i deljenom memorijom. Svaka od predstavljenih implementacija analizirana je za različite parametre, kako bi se testirale njene performanse. Zaključak samog istraživanja jeste da paralelna pretraga grafa može biti performantna na računarskim arhitekturama sa deljenom memorijom, dok implementacija na arhitekturama sa distribuiranom memorijom jedino ima smisla za grafove koji ne mogu da se smeste u radnoj memoriji jednog procesora.*

Ključne reči: *pretraživanje grafa, distribuirani sistemi, paralelni sistemi*

Abstract – *The main focus of this research paper is an attempt to improve upon graph traversal algorithms, especially in the cases of large graphs. This paper provides information about parallel graph traversal algorithms, as well as their implementation on distributed and parallel computer systems. Every type of implementation is analyzed through different parameters to test its achieved performance. The outcome of this research concluded that parallel graph traversal is performant in the case of computer architectures with shared memory, unfortunatelly only case where it becomes useful for architectures with distributed memory is when graph size becomes too large to fit on one node.*

Keywords: *graph traversal, distributed systems, parallel systems*

1. UVOD

Upotreba apstrakcije uz pomoć grafova za analizu i razumevanje raznih vrsta podataka dobija sve veći značaj. Neki od primera podataka koji mogu da se apstrahuju koristeći grafove podrazumevaju: podatke o interakcijama na društvenim mrežama, podatke bankarskih transakcija, podatke o preporuci raznih reklama korisnicima aplikacija na osnovu njihovih interakcija, komunikacionih podataka poput elektronske pošte i telefonskih mreža, podatke bioloških sistema i različitih oblika relacionih podataka generalno.

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Veljko Petrović, docent.

Kada se govori o veštačkoj inteligenciji, apsolutno je neophodno uvesti neku vrstu grafa i primenjivati raznovrsne algoritme nad njim. Zajednički problemi u matematičkoj oblasti teorije grafova i u oblastima primene uključuju identifikaciju i rangiranje važnih entiteta, otkrivanje anomalija u obrascima ili iznenadnih promena u mrežama, pronalaženje čvrsto povezanih klastera entiteta, itd. Rešenja ovih problema obično uključuju klasične algoritme nad grafovima, kao što su: prebacivanje grafa u strukturu stabla (uklanjanje ciklusa), pronalaženje najkraćih putanja, pronalaženje dvopovezanih komponenti, uparivanja, proračune zasnovane na protoku itd.

Kako bi se zadovoljile potrebe teorijske analize grafova za nove aplikacije koje zahtevaju rad sa strukturama velikih skupova podataka, od suštinskog je značaja ubrzati osnovne probleme grafova koristeći aktuelne paralelne sisteme [1].

U ovom radu će se detaljnije razmatrati problem pronalaženja najkraćeg puta, odnosno najmanjeg broja skokova, između dva proizvoljna čvora u povezanom, neusmerenom grafu i to primenom algoritma za pretragu grafa po širini, poznatog kao *breadth-first traversal*. Biće prikazani primeri već postojećih ideja paralelizacije ovog algoritma, kao i implementacija koje se zasnivaju na pomenutim idejama, u okviru arhitektura sa distribuiranom i deljenom memorijom. Takođe, biće dat i prikaz rezultata dobijenih primenjujući ove implementacije na nasumično generisanim grafovima, kao i nad grafovima koji su specifično pogodni za ovu vrstu obrade.

2. PRETHODNA ISTRAŽIVANJA

Najpoznatiji algoritmi za pretraživanje grafova su algoritam **obilaska grafa po širini (engl. *breadth-first search*)** i algoritam **obilaska grafa po dubini (engl. d*epth-first search*)**. Ideja ovih algoritama je da pronađu najkraći put od izvornog čvora do proizvoljnog ciljnog čvora u grafu. Od interesa za ovaj rad jeste algoritam *BFS*. Pseudokod višeizvornog *BFS* algoritma za obilazak neusmerenog grafa je prikazan u okviru [Pseudokod 1].

Kako bi se podelio posao obrade na entitete u paralelnom okruženju, neophodno je obaviti neku vrstu particionisanja grafa kao celine. Ideja na koji način je moguće izvršiti ovakvo particionisanje je predstavljena u okviru rada [2]. Predloženi algoritam je *BFS* algoritam sinhronizovan po nivoima koji napreduje nivo po nivo, počevši od izvornog čvora, gde je nivo temena definisan kao njegov graf udaljenosti od izvora. **Razmatrani su samo usmereni grafovi**.

```
Алгоритам   Вишеизворни обилазак графа по ширини
   Улаз: Граф G, скуп изворних темена S
   Излаз: Листа путања за свако теме P
procedure multisourceBreadthFirstSearch(G, S)
    create queue Q
    create list P where P[v_i] ← ∅
    for all sources s in S do
        enqueue(Q, s)
        P[s] ← s
    while Q is not empty do
        w ← dequeue(Q)
        mark w
        for all edges e in adjacentEdges(G, w) do
            x ← adjacentVertex(w, e)
            if x is not marked then
                P[x] ← append(x, P[w])
                enqueue(Q, x)
```

Pseudokod 1 Višeizvorni obilazak neusmerenog grafa po širini

Jednodimenziono particionisanje grafa podrazumeva raspodelu temena grafa, tako da svaki čvor i svi potezi koji proizilaze iz tog čvora pripadaju samo jednom procesoru. Skup temena koji pripadaju jednom procesoru q će dalje biti nazivan skup *lokalnih temena*. U nastavku je prikazana ilustracija jednodimenzionog particionisanja temena grafa na P particija korišćenjem matrice susedstva A, koja je podeljena tako da su lokalna temena za neki procesor q kontinualna.

$$\begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_P \end{bmatrix}$$

Indeksi podmatrica susedstva A_i predstavljaju indikatore procesora kojima su dodeljene. Potezi koji proizilaze od temena v_j formiraju listu potega, koja je predstavljena listom temena u okviru j-og reda matrice susedstva A. U okviru slike [Pseudokod 2] je dat pseudokod algoritma za distribuirano pretraživanje grafa sa jednodimenzionim particionisanjem, počevši od čvora v_s. U okviru algoritma, svako teme v_i biva označeno svojim nivoom, u okviru niza $L_{v_s}(v_i)$, što označava rastojanje čvora v_s od čvora v_i. Niz L_{v_s} je distribuiran u skladu sa distribucijom temena, tako da procesor P_i poseduje rastojanja početnog temena v_s do svojih lokalnih temena $\{ v_{(i,0)}, v_{(i,1)}, \dots, v_{(i,m)} \}$ gde je $m = \frac{|V|}{|P|}$.

```
Алгоритам   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 ∈ V
procedure bfsDistributedSearch(G(V, E), s)
    Initialize L_s(v) ← { inf,  v ≠ s
                        { 0,    v = s
    for level ← 0 to ∞ do
        F ← {v|L_s(v) = level}, скуп локалних темена за процес са нивоом level
        if F is empty за све процесе then
            Терминирај спољашњу петљу свим процесима
        N ← {суседна темена из скупа F (нису нужно локална темена за тренутни процес)}
        for all processes q do
            N_q ← {темена из N која припадају процесу q}
            Send N_q to process q
            Receive N̂_q from process q
        N̂ ← ∪_q N̂_q   (N̂_q је могуће да има дупликате)
        for v ∈ N̂ and L_s(v) = ∞ do
            L_s(v) ← level + 1
    return L_s
```

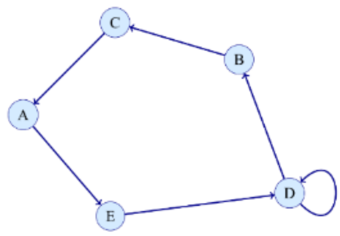
Pseudokod 2: Paralelni BFS algoritam sa jednodimenzionim particionisanjem

Algoritam funkcioniše na sledeći način. Na svakom nivou (dubine grafa), svaki od procesora ima skup F koji predstavlja skup lokalnih temena koji se nalaze na rastojanju *level* od početnog čvora. Lista potega koja se dobija od svakog od temena iz skupa F biva unifikovana u skup potega N koji predstavlja skup susednih temena. Pojedina temena u okviru skupa N neće biti iz skupa lokalnih temena koji je dodeljen procesoru. Za ova temena, vrši se razmena sa procesorom q, kome se temena koja njemu pripadaju šalju, a temena koja pripadaju trenutnom procesoru primaju od istog procesora q. Svaki od procesora vrši ovu razmenu temena i formira konačan skup $\widehat{N}$, skup temena do kojih se stiglo u trenutnom nivou, a koji su iz skupa lokalnih temena za trenutni procesor.

3. TEORIJSKE OSNOVE

Graf (engl. *graph*), posmatrano iz ugla diskretne matematike, odnosno preciznije **teorije grafova (engl. *graph theory*)**, predstavlja strukturu koja se sastoji od skupa objekata koji među sobom mogu imati specifičnu povezanost, odnosno relaciju. Sami objekti koji se povezuju se nazivaju **temenima (engl. *vertex*)**, dok su veze između tih objekata **potezi, odnosno ivice (engl. *edge*)** [3]. Posmatrano iz ugla računarstva, graf predstavlja apstraktnu strukturu podataka koja služi za samo modelovanje grafova iz matematičke oblasti teorije grafova na računaru [4].

Putanja (engl. *path*) predstavlja sekvencu ili niz potega, takvih da se odredišno teme i − og potega poklapa sa polaznim čvorom i + 1-og potega [4]. Kod neusmerenih grafova će zbog njihove prirode važiti da ukoliko postoji put od čvora A do čvora B, postojaće i put od čvora B do čvora A, dok kod usmerenih grafova ovo neće biti pravilo. Put u grafu otvara novu taksonomiju u svetu grafova, na **povezane (engl. *connected*)** i **nepovezane (engl. *disconnected*)** grafove. Povezan graf predstavlja strukturu grafa koja je povezana u smislu *topološkog prostora*, odnosno govori da postoji put od proizvoljnog izvornog temena do proizvoljnog odredišnog temena. Nasuprot ovome, ukoliko se za graf ne može tvrditi da postoji putanja između dva proizvoljna temena, takav graf je nepovezan [4]. **Najkraći put (engl. *shortest path*)** između dva čvora u grafu, odnosno putanju sa najmanjom kardinalnošću od svih mogućih putanja između dva zadata čvora u grafu [5].



Slika 1: *Primer usmerenog grafa*

Kada bi se graf sa slike [Slika 1] predstavio matrično, dobila bi se slabo posednuta matrica, te je najpogodniji način predstavljanja grafova koji nemaju veliki broj potega, korišćenjem nazubljenih matrica. U nastavku sledi postupak kako graf sa slike [Slika 1] prevesti u nazubljenu matricu susedstva.

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

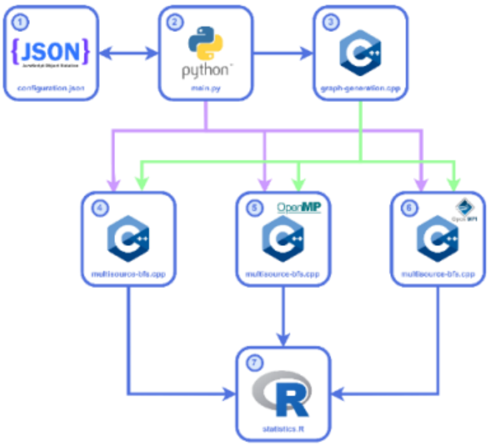
Sada, ukoliko bi se umesto 1 na odgovarajućoj poziciji u matrici iskoristili identifikator samog čvora ka kome postoji poteg, dobija se matrica koja izgleda ovako.

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

U okviru implementacije će se upravo koristiti ovakav način reprezentacije, kako bi se posao obrade mogao pravilno rasporediti na entitete za obradu.

4. IMPLEMENTACIJA

Implementacija je u radu realizovana kako za sekvencijalnu obradu koja će biti referentna, tako i za obradu u okviru distribuiranih arhitektura, kao i arhitektura sa deljenom memorijom. Ideja paralelne, odnosno distribuirane obrade je data u okviru [Pseudokod 2]. Implementacija samih algoritama je odrađena u okviru programskog jezika ***C++***, dok su za paralelno, odnosno distribuirano programiranje iskorišćeni alati ***OpenMP*** i ***OpenMPI***, respektivno.
Kako bi se mogla testirati obrada različitih implementacija, kreiran je odgovarajući **tok podataka (engl. *pipeline*)** koji generiše rezultate, vrši obradu i na kraju analizira dobijene rezultate. Ovaj tok podataka je prikazan na slici [Slika 2].



Slika 2: *Arhitektura toka podataka*

Sam tok podataka teče od (1) ***JSON*** konfiguracionog fajla u kome su postavljani svi neophodni parametri za pokretanje ***Python*** skripte (2). Skripta pokreće **generisanje nasumičnih, neusmerenih, povezanih grafova** (3), a nakon toga svaka od implementacija (4, 5, 6) vrši njihovu obradu. Na kraju, uz pomoć (7) ***R*** skripti se vrši analiza performansi svake od implementacija.
Sama definicija metode koja generiše nasumičan, neusmeren, povezan graf je prikazana na slici [Slika 3].

Što se tiče paralelne implementacije BFS algoritma za arhitekture sa deljenom memorijom, ona je prikazana u okviru slike [Slika 4]. U okviru implementacije se mogu odvojiti regioni od interesa. U okviru linija 46 – 60 se izvršava inicijalizacija svih promenljivih koje se koriste pri obradi i ovaj segment je sekvencijalni deo, odnosno deo koji izvršava glavna master nit. Nakon toga sledi paralelni region u okviru linija 61 – 125, gde se vrši inicijalizacija privatnih promenljivih, a kasnije se otpočinje i sama obradu u okviru *while* petlje (linije 83 – 124). Petlja se izvršava sve dok u globalu postoje čvorovi koji nisu obrađeni, a u okviru nje se raspoređuju čvorovi do kojih se trenutno došlo ostalim nitima (linije 85 – 95), upisuju obrađeni primljeni čvorovi u parcijalne putanje (linije 99 – 111) i na kraju vrši obrada uslova da li se petlja završava (linije 113 – 123).

```cpp
void GenerationGraphHandler::generate_graph(map<string, Object>& parameters)
{
    srand(parameters[SEED]);

    long node_number = parameters[NODE_NUMBER],
        max_degree = parameters[MAX_DEGREE],
        min_degree = parameters[MIN_DEGREE],
        available_nodes,
        random_choice,
        neighbour;

    graph.resize(node_number);

    vector<long> degrees(node_number);
    vector<long> possibilities(node_number);

    for(long i = 0; i < node_number; i++){
        degrees[i] = 0;
        possibilities[i] = i;
        graph[i] = {};
    }

    available_nodes = node_number;
    for(long i = 0; i < node_number; i++)
    {
        long degree = rand() % (max_degree - min_degree) + min_degree;
        for(long j = degrees[i]; j < degree; j++)
        {
            do {
                random_choice = rand() % available_nodes;
                neighbour = possibilities[random_choice];
            } while(neighbour == i);

            graph[i].push_back(neighbour);
            degrees[i]++;

            graph[neighbour].push_back(i);
            degrees[neighbour]++;

            if(degrees[neighbour] == max_degree)
                possibilities[random_choice] = possibilities[--available_nodes];
        }
    }
}
```

Slika 3: Generisanje grafa

```cpp
44  void ParallelGraphHandler::multisource_bfs(map<string, Object>& parameters)
45  {
46      int thread_number = parameters[THREAD_NUMBER];
47
48      long node_number = parameters[NODE_NUMBER],
49          work_load = node_number / thread_number;
50
51      vector<long> sources = parameters[SOURCES].get_long_array();
52      vector<vector<vector<two_longs>>> next(thread_number);
53
54      paths.resize(node_number);
55
56      fill(paths.begin(), paths.end(), two_longs{-1, -1});
57
58      long level = 0,
59          frontiers_global_count = 0;
60
61      #pragma omp parallel num_threads(thread_number) firstprivate(level)
62      {
63          int tid = omp_get_thread_num();
64          long frontiers_count;
65          vector<long> frontiers{};
66
67          for(long source : sources)
68          {
69              if(tid == source / work_load)
70              {
71                  paths[source] = { 0, -1 };
72                  frontiers.push_back(source);
73              }
74          }
75
76          #pragma omp master
77          frontiers_global_count = sources.size();
78
79          next[tid].resize(thread_number);
80
81          #pragma omp barrier
82
83          while(frontiers_global_count > 0)
84          {
85              for(long current_node : frontiers)
86              {
87                  for(long next_node : graph[current_node])
88                  {
89                      long thread = next_node / work_load;
90                      if(paths[next_node].first == -1)
91                          next[thread][tid].push_back({ current_node, next_node });
92                  }
93              }
94
95              #pragma omp barrier
96
97              frontiers.clear();
98
99              for(long i = 0; i < thread_number; i++)
100             {
101                 for(two_longs next_node : next[tid][i])
102                 {
103                     if(paths[next_node.second].first == -1)
104                     {
105                         paths[next_node.second] = { level + 1, next_node.first };
106                         frontiers.push_back(next_node.second);
107                     }
108                 }
109
110                 next[tid][i].clear();
111             }
112
113             frontiers_count = frontiers.size();
114
115             #pragma omp single
116             frontiers_global_count = 0;
117
118             #pragma omp reduction (+:frontiers_global_count)
119             frontiers_global_count += frontiers_count;
120
121             level++;
122
123             #pragma omp barrier
124         }
125     }
126 }
```

Slika 4: *Paralelni BFS za arhitekture sa deljenom memorijom*

Implementacija algoritma paralelne obrade grafa pomoću *BFS* za arhitekture sa distribuiranom memorijom prikazana na slici [Slika 5]. Ova implementacija je dosta složenija, u odnosu na implementaciju na arhitekturama sa deljenom memorijom, odnosno sekvencijalnu, ali se mogu podeliti slični regioni u kodu. Na linijama 127 – 155 se vrši inicijalizacija promenljivih koje se koriste pri obradi, ovo podrazumeva i sam tip podataka koji se razmenjuje (linije 153 – 155).

Nakon ovoga, na linijama 157 – 227 se nalazi sama *while* petlja, koja se izvršava ponovo sve dok postoji u globalu barem jedan čvor do kog se stiglo do trenutne dubine obrade, što predstavlja i sam brojač ove petlje. Na linijama 159 – 170 se vrši prikupljanje čvorova do kojih se stiglo u trenutnoj iteraciji i uslov izlaska na liniji 167. Nakon toga, u okviru linija 170 – 188 se vrši preraspodela čvorova suseda u odgovarajuće redove za slanje procesima koji su odgovorni za njihovu obradu.

Na linijama 193, svaki proces komunicira sa ostalim procesima u kanalu, prima i šalje particionisane redove čvorova, a zatim na odgovarajuće mesto u listi parcijalnih putanja, upisuje prethodnika od kog se do trenutnog čvora stiglo u putanji, kao i nivo na kom se on nalazi.

Konačno, po izlazu iz same petlje, vrši se prikupljanje svih parcijalnih putanja u jednu i njeno emitovanje **(engl. *broadcast*)** svim procesima u kanalu (linije 231 – 239). Ovim se ujedno i okončava distribuirana obrada grafa.

Slika 5: *Paralelni BFS za arhitekture sa distribuiranom memorijom*

5. REZULTATI

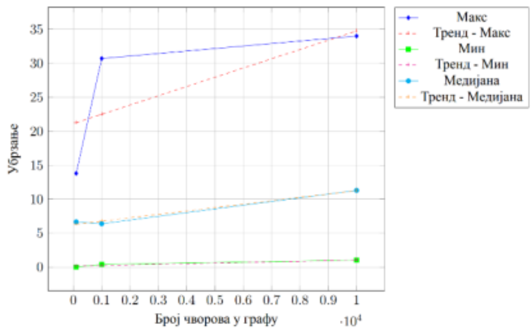
U okviru ove sekcije su prikazani neki od rezultata dobijeni pri analizi prethodno opisanih implementacija algoritma paralelne obrade grafa u odnosu na sekvencijalnu implementaciju pokrenutu nad istim grafom.

Podaci prikazani u okviru [Tabela 1], odnosno ilustrovani na grafikonu [Grafikon 1] predstavljaju zavisnost ubrzanja paralelne obrade na arhitekturama sa deljenom memorijom od ukupnog broja čvorova u grafu. Sam grafikon se sastoji iz tri funkcije: maksimalno dobijeno ubrzanje, medijana ubrzanja i minimalno dobijeno ubrzanje, dok je za svaku od njih odrađena i linearna regresija kako bi se uočio trend.

Broj čvorova	Maksimalno ubrzanje	Minimalno ubrzanje	Medijana ubrzanja
100	13.8	0.0186	6.67
1000	30.7	0.422	6.39
10000	34.0	1.03	11.3

Tabela 1: Zavisnost paralelne implementacije od broja čvorova u grafu

Konkretno je od najvećeg značaja bitno gledati medijanu, jer daje najobjektivniju procenu merenja, s obzirom na osetljivost funkcija minimuma i maksimuma, koje su tu kako bi prikazale same ekstreme koji se dešavaju pri obradi. Konkretno za parametar broj čvorova u grafu, pokazalo se da ubrzanje implementacije raste, to jest da je moguće dobiti **prosečno ubrzanje od 1130%**, dok maksimalno moguće ubrzanje, ponovo dostignuto za najveći broj čvorova, 10000, iznosi 3400%. Naravno, postoje slučajevi gde se ovo nije pokazalo toliko izvanredno, pa je bitno napomenuti da može doći i do **usporenja, odnosno ubrzanja od 1.86%**.



Grafikon 1: *Zavisnost ubrzanja paralelne obrade od broja čvorova u grafu*

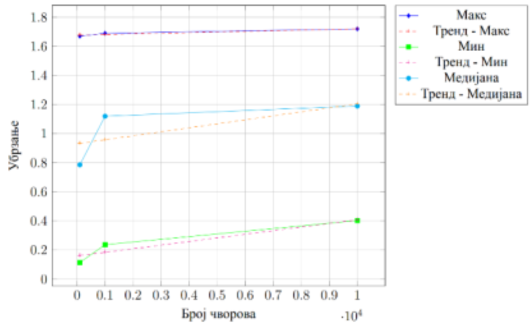
Što se distribuirane implementacije tiče, ona se pokazala u mnogo lošijem svetlu. Međutim, iako je dosta složenija i po performansama dosta lošija od sekvencijalne, jedina je kojom se mogu obraditi grafovi koji ne mogu da se smeste u radnoj memoriji jednog procesora. U okviru tabele [Tabela 2] su prikazana dobijena merenja ubrzanja distribuirane obrade u odnosu na sekvencijalnu implementaciju pokrenutu nad istim grafom, gde je parametar koji se ispituje bio broj grafova u čvoru, odnosno količina posla obrade. Ilustracija ovih podatak je prikazana grafikonom [Grafikon 2].

Broj čvorova	Maksimalno ubrzanje	Minimalno ubrzanje	Medijana ubrzanja
100	1.67	0.114	0.786
1000	1.69	0.237	1.12
10000	1.72	0.403	1.19

Tabela 2: Zavisnost ubrzanja distribuirane obrade od broja čvorova u grafu

U okviru tabele [Tabela 2], odnosno grafikona [Grafikon 2] se ponovo može uočiti optimističan rast ubrzanja sa porastom količine posla, odnosno broja čvorova u grafu. Međutim, taj rast je u odnosu na implementaciju na arhitekturama sa deljenom memorijom dosta manji. **Maksi-**

malna dobijena medijana ubrzanja za distribuiranu implementaciju iznosi 119%. Dobra strana ovoga jeste to što performanse rastu i svakako će u jednom trenutku ubrzanje postati matematički beskonačno, jer neće postojati procesor koji bi sekvencijalno mogao da obradi toliko velike grafove, pa će distribuirano rešenje ujedno postati i jedino za ovakve poslove. No, pri izradi ovog rada, gde nije bilo uslova testiranja ovoga na većem broju računara, rezultati su obeshrabrujući, ali sa poznatim shvatanjem zašto je to tako.



Grafikon 2: *Zavisnost ubrzanja distribuirane obrade od broja čvorova u grafu*

6. ZAKLJUČAK

U ovom radu se detaljnije razmotrio problem pronalaženja najkraćeg puta, odnosno najmanjeg broja skokova, između dva čvora u grafu i to primenom algoritma za pretragu grafa po širini, poznatog kao ***breadth-first traversal***. Implementirani su algoritmi paralelne verzije ovog algoritma, koristeći sisteme sa distribuiranom i deljenom memorijom. Takođe, dat je i prikaz rezultata dobijenih primenjujući pomenute imeplementacije na nasumično generisanim grafovima.

Na osnovu analize implementacija, može se zaključiti da je realizacija paralelnog pretraživanja grafa moguća, ali ni pod raznom jednostavna i nikako apsolutna. Ubrzanja do kojih se došlo su tu isključivo jer se radi o veštački sintezovanim grafovima, kao i činjenica da ta ubrzanja postoje samo za slučajeve visokog stepena povezanosti, odnosno paralelizacije u okviru okruženja koje radi sa deljivom memorijom.

Nažalost, za arhitekture sa distribuiranom memorijom se ne može reći ni to, s obzirom da je najbrže zabeleženo ubrzanje, koje je retko dostignuto, jako malo. Međutim, kao što je već napomenuto, ukoliko se radi o veoma velikim grafovima, koji jednostavno ne mogu stati u radnu memoriji jednog računara, distribuirana obrada je jedino moguće rešenje njihovog procesiranja, s toga je ipak ne treba zanemarivati.

7. LITERATURA

[1] D. A. Bader, Massive Graph Analytics, CRC Press, 2022.

[2] A. Yoo, E. Chow, K. Henderson, W. McLendon, U. Catalyurek i B. Hendrickson, A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L, ACM/IEEE Conference on Supercomputing, 2005.

[3] W. T. Tutte, Cambridge mathematical library: Graph theory, Cambridge: Cambridge University Press, 2001.

[4] K. Mehlhorn, Data structures and algorithms 2, Berlin: Springer, 2011.

[5] M. Neerajkumar, An efficient algorithm for shortest path tree in dynamic graph, LAP Lambert Academic Publishing, 2014.

Kratka biografija

Stefan Aleksić rođen je u Zaječaru 02. marta 2000. godine. Elektronski fakultet u Nišu, studijski program Elektrotehnika i računarstvo upisao je 2018. godine. Nakon završenih osnovnih studija, 2022. godine, upisao je master akademske studije na Fakultetu tehničkih nauka u Novom Sadu, na studijskom programu Računarstvo i automatika.

kontakt: stefan.sa.aleksic@gmail.com