



Факултет техничких наука

Универзитет у Новом Саду

Рачунарски системи високих перформанси

---

# Оптимизација Pregel vertex-centric модела за обраду графова применом Bulk Synchronous Parallel парадигме

---

*Аутор:*  
Марко Николић

*Индекс:*  
Е2 47/2024

11. јануар 2026.

### **Сажетак**

Овај рад се бави Pregel vertex-centric моделом обраде графова и основама његове оптимизације и паралелизације применом Bulk Synchronous Parallel (BSP) приступа. Pregel vertex-centric модел заснива се на идеји обраде графова при којој сваки чвор извршава сопствене прорачуне и шаље поруке својим суседима током итеративног процеса, све док се не испуни одређени услов или не дође до конвергенције.

У раду су имплементирана четири практична примера примене Pregel vertex-centric модела на PageRank алгоритму. Обухваћена је једна секвенцијална имплементација, као и три паралелне имплементације које користе различите алате и стандарде: OpenMP, OpenMPI и OpenCL. Сви примери су реализовани у програмском језику C++ и показују значајно убрзање паралелних имплементација у односу на секвенцијалну.

## Садржај

<b>1</b>	<b>Увод</b>	<b>1</b>
1.1	Значај обраде великих графова . . . . .	1
1.2	Изазови при обради графова . . . . .	1
<b>2</b>	<b>Теоријске основе</b>	<b>2</b>
2.1	Историјат и примена Pregel модела . . . . .	2
2.2	Историјат и примена Bulk Synchronous Parallel (BSP) парадигме . . . .	4
2.3	Теоријске основе Pregel модела . . . . .	4
2.4	Теоријске основе Bulk Synchronous Parallel (BSP) парадигме . . . . .	6
2.5	Паралелизам Pregel модела . . . . .	8
<b>3</b>	<b>Практична имплементација и резултати</b>	<b>8</b>
3.1	Секвенцијална имплементација . . . . .	8
3.2	Паралелна имплементација (OpenMP) . . . . .	10
3.3	Паралелна имплементација (OpenMPI) . . . . .	13
3.4	Паралелна имплементација (OpenCL) . . . . .	17
3.5	Резултати . . . . .	23
<b>4</b>	<b>Закључак</b>	<b>27</b>

## Списак изворних кодова

1	Секвенцијална имплементација . . . . .	10
2	Паралелна имплементација (OpenMP) . . . . .	12
3	Паралелна имплементација (OpenMPI) . . . . .	17
4	Паралелна имплементација (OpenCL) . . . . .	23

## Списак слика

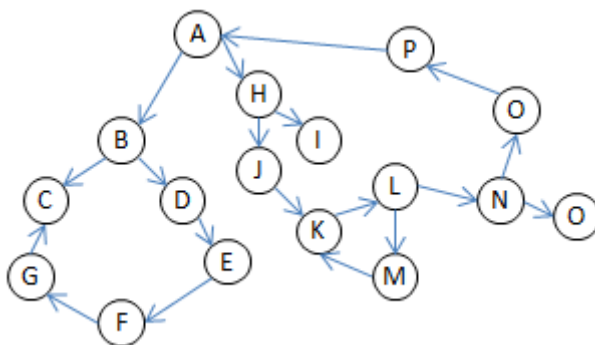
1	Граф . . . . .	1
2	Google лого . . . . .	2
3	Apache Giraph лого . . . . .	3
4	Apache Spark GraphX лого . . . . .	3
5	Apache Hama лого . . . . .	3
6	Vertex-centric алгоритам за проналажење максимума у графу . . . . .	5
7	Bulk Synchronous Parallel (BSP) модел . . . . .	6
8	Типична архитектура дистрибуираног система заснованог на Pregel моделу и Bulk Synchronous Parallel (BSP) парадигми . . . . .	7
9	График времена извршавања са линеарном скалом . . . . .	25
10	График времена извршавања са логаритамском скалом . . . . .	25
11	График убрзања са линеарном скалом . . . . .	26
12	График убрзања са логаритамском скалом . . . . .	26

## 1 Увод

У овом поглављу дат је преглед значаја обраде великих графова, као и кључних изазова који се јављају при њиховој анализи.

### 1.1 Значај обраде великих графова

Са експоненцијалним растом количине података у савременим информационим системима, јавља се све већа потреба за ефикасним методама њихове анализе и обраде. Значајан део ових података поседује сложене међусобне односе који се природно могу моделирати у облику графова (слика 1). Примери таквих система укључују друштвене мреже, системе за препоруку, веб структуре, телекомуникационе мреже и биоинформатичке системе.



Слика 1: Граф

Графови омогућавају интуитивно представљање ентитета у виду чворова и односа између њих у виду грана. Међутим, са повећањем броја чворова и веза, класични алгоритми и секвенцијални приступи постају неефикасни или практично неупотребљиви. Ово је посебно изражено код алгоритама који захтевају итеративну обраду и честу размену информација између повезаних чворова, као што је PageRank алгоритам.

Због тога се развијају специјализовани модели и системи за паралелну и дистрибуирану обраду графова, који омогућавају скалабилну анализу великих скупова података.

### 1.2 Изазови при обради графова

Обрада графова представља изазован проблем због њихове неправилне и нелинеарне структуре. Главни изазови могу се сумирати на следећи начин:

1. Неправилна структура података – За разлику од табеларних података, графови немају униформну структуру. Број суседа по чвору може значајно да варира, што отежава равномерну расподелу посла.

2. Јака међузависност података – Обрада једног чвора често зависи од стања његових суседа, што отежава независну и паралелну обраду.

3. Итеративна природа алгоритама – Многи граф алгоритми, укључујући Page-Rank, захтевају више итерација док не дође до конвергенције, при чему резултат једне итерације утиче на наредну.

4. Меморијска захтевност – Велики графови често не могу бити смештени у меморију једног рачунара, што захтева дистрибуирану обраду.

5. Комуникациони трошкови – При дистрибуираној обради, чворови који су логички повезани могу бити физички смештени на различитим процесима или рачунарима, што повећава трошкове комуникације.

Ови изазови указују на потребу за моделима обраде који су посебно прилагођени структури и природи графовских података. Један од таквих модела је Pregel vertex-centric модел, који у комбинацији са Bulk Synchronous Parallel (BSP) парадигмом омогућава ефикасну и скалабилну обраду великих графова.

## 2 Теоријске основе

У наредном поглављу представљене су теоријске основе и историјат Pregel vertex-centric модела обраде графова, као и Bulk Synchronous Parallel (BSP) парадигме.

### 2.1 Историјат и примена Pregel модела

Pregel модел је представљен 2010. године у раду “Pregel: A System for Large-Scale Graph Processing” [1] у Google-у (слика 2). Развијен је као систем за дистрибуирану обраду великих графова са милијардама чворова и веза.

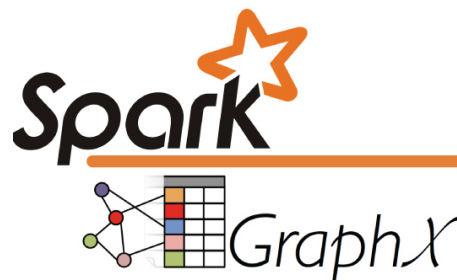


Слика 2: Google лого

Модел користи vertex-centric приступ и омогућава паралелну и итеративну обраду кроз дискретне superstep-ове. Pregel модел је примењен у низу алгоритама унутар Google инфраструктуре, укључујући PageRank, откривање повезаних компоненти и анализу веб графова, а његове идеје су инспирисале системе као што су Apache Giraph [2] (слика 3), Apache Spark GraphX [3] (слика 4), Apache Hama [4] (слика 5).



Слика 3: Apache Giraph лого



Слика 4: Apache Spark GraphX лого



Слика 5: Apache Hama лого



## 2.2 Историјат и примена Bulk Synchronous Parallel (BSP) парадигме

Bulk Synchronous Parallel (BSP) парадигма је предложена 1990. године од стране Леслија Г. Валијанта као модел за паралелно рачунање. BSP дефинише извршавање паралелних алгоритама у низу фаза које укључују локално рачунање, комуникацију и глобалну синхронизацију.

BSP је широко коришћен као концептуална основа за дистрибуирану и паралелну обраду података, укључујући и Pregel модел.

## 2.3 Теоријске основе Pregel модела

Pregel модел заснован је на vertex-centric парадигми, где сваки чвор у графу самостално обрађује своје податке и комуницира са суседним чворовима кроз поруке. Овај приступ омогућава паралелну и итеративну обраду великих графова, без потребе да се граф обрађује као целина.

Обрада се одвија у изолованим корацима, познатим као superstep-ови, где сваки superstep садржи следеће фазе:

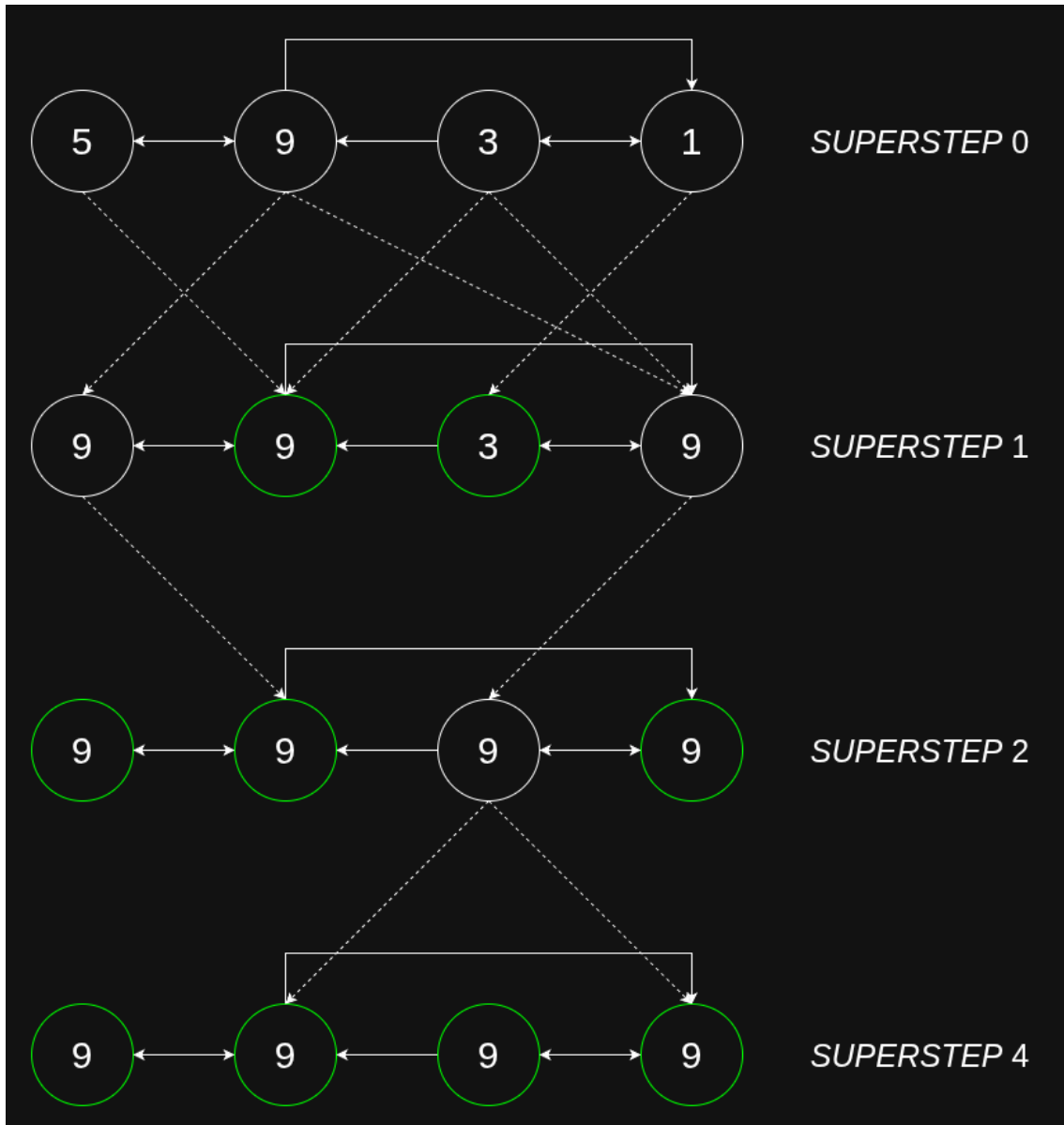
1. Примање порука – Чворови примају поруке које су им други чворови послали у претходном superstep-у.
2. Локално израчунавање – Чвор ажурира своје стање користећи примљене поруке и претходно стање.
3. Слање порука – Чвор шаље резултате обраде својим суседима, који ће их користити у следећем superstep-у.

Чворови се могу привремено деактивирати ако немају посла, али се аутоматски активирају ако им касније стигне нова порука. Алгоритам се завршава када сви чворови буду деактивирани и када више нема порука за слање.

Описани модел обраде података приказан је на слици 6 на примеру проналажења максимума:

1. Чворови графа у свакој итерацији (superstep-у) алгоритма својим суседима шаљу информацију о највећој вредности за коју тај чвор зна.
2. Чвор који прими поруку пореди добијену вредност са највећом вредношћу за коју он зна.
3. Уколико је нова вредност већа, ажурира своје стање и шаље поруку са новом вредношћу суседима.
4. Уколико је тренутна вредност ипак већа и не долази до промене, чвор гласа за прекид алгоритма и привремено се деактивира.

5. Алгоритам се извршава докле год има активних чворова или порука.

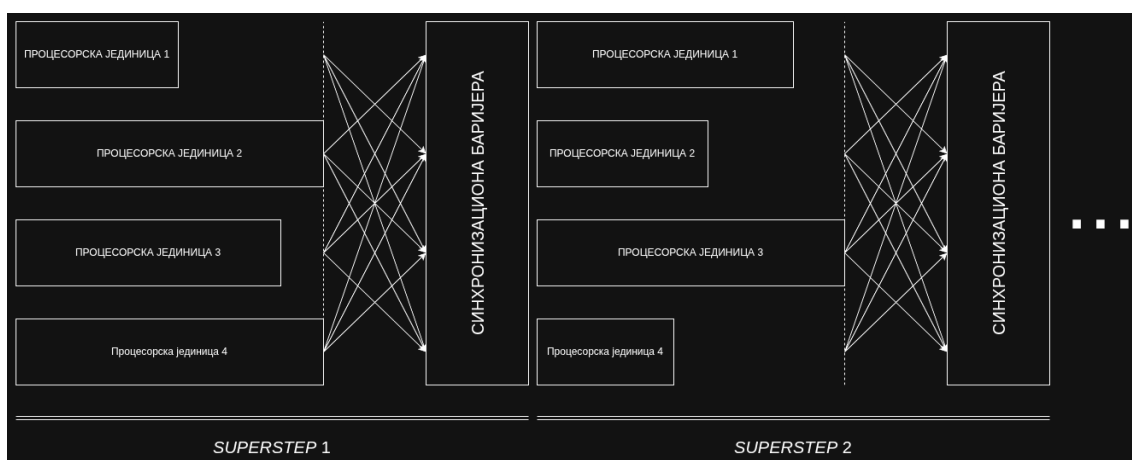


Слика 6: Vertex-centric алгоритам за проналажење максимума у графу

## 2.4 Теоријске основе Bulk Synchronous Parallel (BSP) парадигме

Bulk Synchronous Parallel (BSP) модел организује паралелну обраду података у дискретне кораке, познате као superstep-ови [5]. Сваки superstep се састоји из три фазе:

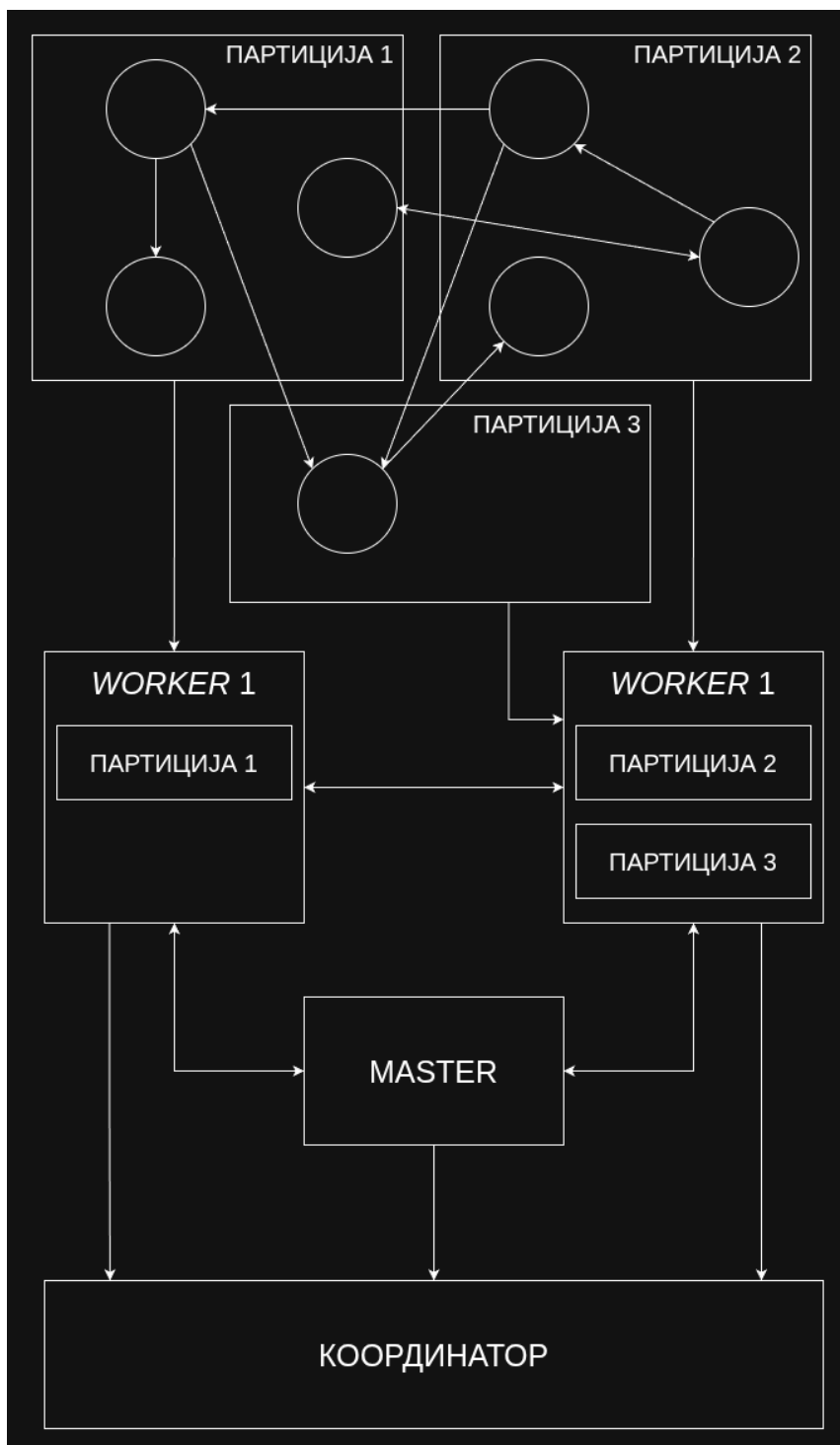
1. Локална обрада – Свака процесорска јединица обрађује свој подскуп података независно од осталих.
2. Комуникација – Јединице размењују поруке како би обезбедиле податке потребне за следећи superstep.
3. Синхронизациона баријера – Сви процесори се синхронизују пре него што започну наредни superstep, што осигурава конзистентност обраде.



Слика 7: Bulk Synchronous Parallel (BSP) модел

BSP омогућава скалабилну и предвидљиву паралелну обраду и представља концептуалну основу за системе као што су Pregel, где superstep механизам одређује редослед извршавања vertex-centric алгоритама на графу.

Pregel модел у комбинацији са BSP парадигмом често се примењује у дистрибуираном окружењу, где се за ефикасну обраду великих графова користе три основне компоненте (слика 7): master, worker и координаторски сервис. Ова архитектура омогућава расподелу података, паралелну обраду и синхронизацију између великог броја процесора. Master контролише ток извршавања superstep-ова, додељује партиције графа worker-има и прати њихово стање током обраде. Worker обрађује свој локални подскуп чворова, врши израчунавања и размењује поруке са другим worker-има како би се подаци ажурирали у наредном superstep-у. Координаторски сервис чувају метаподатке о кластеру, омогућавају конфигурацију и синхронизацију између свих компоненти.



Слика 8: Типична архитектура дистрибуираног система заснованог на Pregel моделу и Bulk Synchronous Parallel (BSP) парадигми

## 2.5 Паралелизам Pregel модела

Vertex-centric приступ, који лежи у основи Pregel модела, посебно је погодан за паралелизацију јер сваки чвор графа самостално обрађује своје податке током superstep-a. Ова независност чворова омогућава да се обрада расподели међу великим бројем процесора, чиме се постиже ефикасна паралелна обрада.

Такође, vertex-centric модел подржава широк спектар граф алгоритама, укључујући PageRank, проналажење најкраћих путева, израчунавање максимума или минимума, детекцију компоненти повезаности и многе друге итеративне алгоритме, што га чини веома флексибилним за различите примене у анализи великих графова.

## 3 Практична имплементација и резултати

У овом поглављу приказана су четири практична примера примене Pregel vertex-centric модела на PageRank алгоритму. Обухваћена је секвенцијална имплементација и три паралелне имплементације које користе различите алате и стандарде: OpenMP, OpenMPI и OpenCL. Примери су реализовани у програмском језику C++ и илуструју значајно убрзање паралелних верзија у односу на секвенцијалну. Такође, Упоредјена су времена извршавања и мере убрзања паралелних верзија.

### 3.1 Секвенцијална имплементација

Секвенцијална имплементација представља базну верзију алгоритма која служи као референтна тачка за мерење убрзања паралелних верзија. Иако се извршава на једној нити, алгоритам је дизајниран по принципима Pregel модела и BSP парадигме.

Граф се репрезентује помоћу три структуре:

1. pageIds - хеш мапа која мапира имена страница у јединствене целобројне идентификаторе.
2. pageNames - вектор који чува имена страница.
3. outEdges - вектор вектора који за сваки чвор чува листу његових излазних суседа.

За размену порука између superstep-ова користе се две структуре:

1. inbox - вектор вектора који за сваки чвор чува примљене поруке.
2. outbox - вектор вектора који за сваки чвор чува поруке за слање.

Функција rankPages (изворни код 1) имплементира итеративно рачунање PageRank вредности. Иницијалне вредности свих чворова су постављене на  $1/n$ . Алго-

питајте се у superstep итерацијама где се у сваком superstep-у извршавају следеће операције:

1. Агрегација порука - Сваки чвор сабира све поруке из свог inbox-а.
2. Рачунање нове вредности - Нова PageRank вредност се рачуна по формули  $PR(v) = (1 - d)/n + d * \Sigma(msg)$ , где је  $d$  damping фактор (0.85),  $n$  број чворова, а  $\Sigma(msg)$  сума примљених порука.
3. Слање порука - Сваки чвор дели своју тренутну PageRank вредност једнако између свих својих суседа и ставља поруке у њихов outbox. Висећи чворови (без излазних грана) не шаљу поруке, већ се њихова маса акумулира.
4. Редистрибуција масе висећих чворова - Акумулирана вредност од висећих чворова се равномерно дистрибуира свим чворовима.
5. Синхронизација - Outbox постаје inbox за следећу итерацију, inbox се брише. Ово симулира глобалну баријеру BSP модела.

Алгоритам се зауставља када се достигне максималан број superstep-ова или када ниједан чвор не пошаље поруку (конвергенција).

---

```
1 vector<double> rankPages (
2     const unordered_map<string, int>& pageIds,
3     const vector<string>& pageNames,
4     const vector<vector<int>>& outEdges,
5     int maxSupersteps) {
6     int n = pageIds.size();
7
8     vector<double> pageRanks(n, 1.0 / n);
9     vector<double> nextPageRanks(n, 0.0);
10    vector<vector<double>> inbox(n);
11    vector<vector<double>> outbox(n);
12
13    double danglingMass, sum, share, danglingShare;
14    bool messagesSent = true;
15
16    for (int step = 0; step < maxSupersteps && messagesSent; ++step) {
17        danglingMass = 0.0;
18        messagesSent = false;
19
20        for (int v = 0; v < n; ++v) {
21            sum = 0.0;
22            for (double msg : inbox[v]) {
23                sum += msg;
24            }
25            ...
```

---

```
1  ...
2      nextPageRanks[v] = (1.0 - DAMPING) / n + DAMPING * sum;
3
4      if (outEdges[v].empty()) {
5          danglingMass += pageRanks[v];
6      } else {
7          share = pageRanks[v] / outEdges[v].size();
8          for (int u : outEdges[v]) {
9              outbox[u].push_back(share);
10             messagesSent = true;
11         }
12     }
13 }
14
15 danglingShare = DAMPING * danglingMass / n;
16
17 for (int v = 0; v < n; ++v) {
18     nextPageRanks[v] += danglingShare;
19 }
20
21 inbox.swap(outbox);
22 for (auto& box : outbox) {
23     box.clear();
24 }
25
26 pageRanks.swap(nextPageRanks);
27 fill(nextPageRanks.begin(), nextPageRanks.end(), 0.0);
28 }
29
30 return pageRanks;
31 }
```

---

Изворни код 1: Секвенцијална имплементација

## 3.2 Паралелна имплементација (OpenMP)

Паралелна имплементација користи OpenMP за паралелизацију на систему са дељеном меморијом. Структура алгоритма остаје иста као у секвенцијалној верзији, superstep-ови се и даље извршавају итеративно, али се рачунање унутар сваког superstep-а паралелизује. Кључна разлика у односу на секвенцијалну верзију је што се обрада чворова унутар superstep-а извршава паралелно на више нити.

Граф се репрезентује истим структурама као у секвенцијалној верзији:

1. pageIds - хеш мапа која мапира имена страница у јединствене целобројне идентификаторе.

2. pageNames - вектор који чува имена страница.
3. outEdges - вектор вектора који за сваки чвор чува листу његових излазних су-седа.

За размену порука између superstep-ова користе се исте две структуре:

1. inbox - вектор вектора који за сваки чвор чува примљене поруке.
2. outbox - вектор вектора који за сваки чвор чува поруке за слање.

Функција rankPages (изворни код 2) имплементира итеративно рачунање PageRank вредности са паралелизацијом. Иницијалне вредности свих чворова су постављене на  $1/n$ . Алгоритам ради у superstep итерацијама где се у сваком superstep-у извршавају следеће операције:

1. Агрегација порука - Сваки чвор сабира све поруке из свог inbox-а. Петља кроз све чворове је паралелизована, свака нит обрађује свој подскуп чворова независно.
2. Рачунање нове вредности - Нова PageRank вредност се рачуна по формули  $PR(v) = (1 - d)/n + d * \Sigma(msg)$ , где је  $d$  damping фактор (0.85),  $n$  број чворова, а  $\Sigma(msg)$  сума примљених порука. Ово се такође извршава паралелно.
3. Слање порука - Сваки чвор дели своју тренутну PageRank вредност свим својим суседима и ставља поруке у њихов outbox. Висећи чворови (без излазних грана) не шаљу поруке, већ се њихова маса акумулира. Петља је паралелизована, али писање у outbox захтева синхронизацију јер више нити може истовремено писати у исти outbox.
4. Редистрибуција масе висећих чворова - Акумулирана вредност од висећих чворова се равномерно дистрибуира свим чворовима. Петља је паралелизована.
5. Синхронизација - Outbox постаје inbox за следећу итерацију, inbox се брише. Ово симулира глобалну баријеру BSP модела.

Алгоритам се зауставља када се достигне максималан број superstep-ова или када ниједан чвор не пошаље поруку (конвергенција).

---

```
1 vector<double> rankPages(  
2     unordered_map<string, int>& pageIds,  
3     vector<string>& pageNames,  
4     vector<vector<int>>& outEdges,  
5     int maxSupersteps) {  
6     int n = pageIds.size();  
7  
8     vector<double> pageRanks(n, 1.0 / n);  
9     vector<double> nextPageRanks(n, 0.0);  
10    ...  
11
```

---



```
1  ...
2  vector<double> inbox(n, 0.0);
3  vector<double> outbox(n, 0.0);
4
5  double danglingMass;
6  bool messagesSent = true;
7
8  int numThreads = omp_get_max_threads();
9
10 for (int step = 0; step < maxSupersteps && messagesSent; ++step) {
11     danglingMass = 0.0;
12     messagesSent = false;
13
14     fill(outbox.begin(), outbox.end(), 0.0);
15
16     #pragma omp parallel for reduction(|:messagesSent) reduction(+:danglingMass)
17     for (int v = 0; v < n; ++v) {
18         double sum = inbox[v];
19         nextPageRanks[v] = (1.0 - DAMPING) / n + DAMPING * sum;
20
21         if (outEdges[v].empty()) {
22             danglingMass += pageRanks[v];
23         } else {
24             double share = pageRanks[v] / outEdges[v].size();
25             for (int u : outEdges[v]) {
26                 #pragma omp atomic
27                 outbox[u] += share;
28                 messagesSent = true;
29             }
30         }
31     }
32
33     double danglingShare = DAMPING * danglingMass / n;
34
35     #pragma omp parallel for
36     for (int v = 0; v < n; ++v) {
37         nextPageRanks[v] += danglingShare;
38     }
39
40     swap(inbox, outbox);
41     pageRanks.swap(nextPageRanks);
42     fill(nextPageRanks.begin(), nextPageRanks.end(), 0.0);
43 }
44
45 return pageRanks;
46 }
```

---

Изворни код 2: Паралелна имплементација (OpenMP)

### 3.3 Паралелна имплементација (OpenMPI)

Паралелна имплементација користи OpenMPI за паралелизацију на систему са дистрибуираном меморијом. Ова имплементација концептуално највише личи на оригиналне Pregel и BSP моделе, где је граф партиционисан између више процеса, а сваки процес обрађује свој део чворова. Superstep-ови се и даље извршавају итеративно, али се рачунање унутар сваког superstep-а дистрибуира између процеса.

Само главни процес (rank 0) учитава комплетан граф након чега га дели на партиције и дистрибуира их осталим процесима помоћу MPI\_Send операција. Сваки процес добија свој опсег чворова, што обезбеђује равномерну дистрибуцију. Након дистрибуције, сваки процес чува само своје локалне чворове са њиховим излазним и улазним гранама.

Граф се партиционише тако да сваки MPI процес добија свој подскуп чворова:

1. localPageIds - локална мапа која садржи само чворове додељене овом процесу.
2. localPageNames - вектор који чува имена локалних страница.
3. localOutEdges - вектор вектора који за сваки локални чвор чува листу његових излазних суседа (који могу бити на другим процесима).

За размену порука између superstep-ова користе се структуре:

1. inbox - вектор вектора који за сваки локални чвор чува примљене поруке.
2. outbox - вектор вектора који за сваки локални чвор чува поруке за слање.

Функција rankPages (изворни код 3) имплементира итеративно рачунање PageRank вредности где сваки процес обрађује свој део графа. Иницијалне вредности свих чворова су постављене на  $1/n$ . Алгоритам ради у superstep итерацијама где се у сваком superstep-у извршавају следеће операције:

1. Агрегација порука - Сваки процес сабира поруке за своје локалне чворове из inbox-а.
2. Рачунање нове вредности - Нова PageRank вредност се рачуна по формули  $PR(v) = (1 - d)/n + d * \Sigma(msg)$ , где је  $d$  damping фактор (0.85),  $n$  укупан број чворова у графу, а  $\Sigma(msg)$  сума примљених порука. Сваки процес рачуна вредности само за своје локалне чворове.
3. Слање порука - Сваки процес обрађује своје локалне чворове и генерише поруке за њихове суседе. Висећи чворови (без излазних грана) не шаљу поруке, већ се њихова маса акумулира локално. Поруке намењене чворовима на другим процесима се групишу за слање преко MPI-ја.
4. Редистрибуција масе висећих чворова - Сваки процес израчунава локални допринос dangling масе. Користи се MPI\_Allreduce операција да би се сабрале све локалне масе и резултат дистрибуирао свим процесима. Затим се акумулирана вред-

ност равномерно дистрибуира свим локалним чворовима.

5. Синхронизација - Процеси размењују поруке коришћењем MPI\_Alltoall или сличних колективних операција што представља синхронизациону баријеру. Outbox постаје inbox за следећу итерацију.

Алгоритам се зауставља када се достигне максималан број superstep-ова или када ниједан процес не пошаље поруку.

---

```
1 vector<double> rankPages(  
2     unordered_map<string, int>& pageIds,  
3     vector<string>& pageNames,  
4     vector<vector<int>>& outEdges,  
5     vector<vector<int>>& inEdges,  
6     int maxSupersteps) {  
7     int rank, size;  
8     MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
9     MPI_Comm_size(MPI_COMM_WORLD, &size);  
10  
11     int n = 0;  
12     if (rank == 0) {  
13         n = pageIds.size();  
14     }  
15     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);  
16  
17     int verticesPerProcess = (n + size - 1) / size;  
18     int verticesStart = rank * verticesPerProcess;  
19     int verticesEnd = min(verticesStart + verticesPerProcess, n);  
20     int localN = max(0, verticesEnd - verticesStart);  
21  
22     vector<vector<int>> localOutEdges(localN);  
23     vector<vector<int>> localInEdges(localN);  
24  
25     // Distribute graph partitions  
26     if (rank == 0) {  
27         for (int process = 1; process < size; ++process) {  
28             int processStart = process * verticesPerProcess;  
29             int processEnd = min(processStart + verticesPerProcess, n);  
30             int processVertexCount = max(0, processEnd - processStart);  
31  
32             MPI_Send(  
33                 &processVertexCount,  
34                 1, MPI_INT, process, 0, MPI_COMM_WORLD  
35             );  
36     ...
```

---

---

```
1  ...
2      for (int u = processStart; u < processEnd; ++u) {
3          int edgeCount = outEdges[u].size();
4          MPI_Send(
5              &edgeCount,
6              1, MPI_INT, process, 0, MPI_COMM_WORLD
7          );
8          MPI_Send(
9              outEdges[u].data(),
10             edgeCount, MPI_INT, process, 0, MPI_COMM_WORLD
11          );
12
13             edgeCount = inEdges[u].size();
14             MPI_Send(
15                 &edgeCount,
16                 1, MPI_INT, process, 0, MPI_COMM_WORLD
17             );
18             MPI_Send(
19                 inEdges[u].data(),
20                 edgeCount, MPI_INT, process, 0, MPI_COMM_WORLD
21             );
22         }
23     }
24
25     for (int i = 0; i < localN; ++i) {
26         localOutEdges[i] = outEdges[verticiesStart + i];
27         localInEdges[i] = inEdges[verticiesStart + i];
28     }
29 } else {
30     MPI_Recv(
31         &localN,
32         1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE
33     );
34     localOutEdges.resize(localN);
35     localInEdges.resize(localN);
36
37     for (int i = 0; i < localN; ++i) {
38         int edgeCount;
39         MPI_Recv(
40             &edgeCount,
41             1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE
42         );
43         localOutEdges[i].resize(edgeCount);
44         MPI_Recv(
45             localOutEdges[i].data(), edgeCount,
46             MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE
47         );
48     }
49     ...
```

---

---

```
1  ...
2      MPI_Recv(
3          &edgeCount,
4          1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE
5      );
6      localInEdges[i].resize(edgeCount);
7      MPI_Recv(
8          localInEdges[i].data(), edgeCount,
9          MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE
10     );
11 }
12 }
13
14 // PageRank algorithm
15 vector<double> localPageRanks(localN, 1.0 / n);
16 vector<double> nextLocalPageRanks(localN, 0.0);
17 vector<double> messages(n, 0.0);
18
19 bool messagesSent = true;
20
21 for (int step = 0; step < maxSupersteps && messagesSent; ++step) {
22     messagesSent = false;
23     fill(nextLocalPageRanks.begin(), nextLocalPageRanks.end(), 0.0);
24     fill(messages.begin(), messages.end(), 0.0);
25
26     double localDangling = 0.0;
27
28     for (int i = 0; i < localN; ++i) {
29         if (localOutEdges[i].empty()) {
30             localDangling += localPageRanks[i];
31         } else {
32             double share = localPageRanks[i] / localOutEdges[i].size();
33             for (int u : localOutEdges[i]) {
34                 messages[u] += share;
35             }
36             messagesSent = true;
37         }
38     }
39
40     MPI_Allreduce(
41         MPI_IN_PLACE, messages.data(),
42         n, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD
43     );
44
45     double danglingMass = 0.0;
46     MPI_Allreduce(
47         &localDangling, &danglingMass,
48         1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD
49     );
50     ...
```

---

```
1  ...
2      double danglingShare = DAMPING * danglingMass / n;
3
4      for (int i = 0; i < localN; ++i) {
5          int v = verticesStart + i;
6          nextLocalPageRanks[i] = (1.0 - DAMPING)/n +
7              DAMPING * messages[v] + danglingShare;
8      }
9
10     localPageRanks.swap(nextLocalPageRanks);
11
12     int anyMessage = messagesSent ? 1 : 0;
13     MPI_Allreduce(
14         MPI_IN_PLACE, &anyMessage,
15         1, MPI_INT, MPI_LOR, MPI_COMM_WORLD
16     );
17     messagesSent = anyMessage;
18 }
19
20 vector<double> pageRanks(n, 0.0);
21 vector<int> counts(size), displacements(size);
22 for (int i = 0; i < size; ++i) {
23     int start = i * verticesPerProcess;
24     int end = min(start + verticesPerProcess, n);
25     counts[i] = end - start;
26     displacements[i] = start;
27 }
28
29 MPI_Gatherv(
30     localPageRanks.data(), localN, MPI_DOUBLE, pageRanks.data(),
31     counts.data(), displacements.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD
32 );
33
34 return pageRanks;
35 }
```

---

Изворни код 3: Паралелна имплементација (OpenMPI)

### 3.4 Паралелна имплементација (OpenCL)

Паралелна имплементација користи OpenCL за убрзање извршавања на различитим процесорским јединицама (CPU, GPU, акцелератори). За разлику од OpenMP имплементације која користи само CPU нити, OpenCL омогућава извршавање на специјализованим хардверским јединицама. Superstep-ови се и даље извршавају итеративно, али се рачунање унутар сваког superstep-а извршава на акцелераторској јединици.

Граф се учитава на хост систему (CPU) и затим конвертује у CSR (Compressed Sparse Row) формат који је оптималан за паралелни приступ. CSR формат се састоји од два низа: edges који садржи све суседе редом, и offsets који за сваки чвор чува почетну позицију његових суседа у edges низу. Овај формат омогућава брз приступ суседима било ког чвора без потребе за динамичком алокацијом меморије.

Граф се репрезентује помоћу:

1. pageIds - хеш мапа која мапира имена страница у јединствене целобројне идентификаторе.
2. pageNames - вектор који чува имена страница.
3. edges - CSR низ свих излазних суседа.
4. offsets - CSR низ почетних позиција за сваки чвор.

За размену порука између superstep-ова користе се структуре:

1. inbox - низ који за сваки чвор чува збир примљених порука.
2. outbox - низ који за сваки чвор чува збир порука за слање.

Функција rankPages (изворни код 4) имплементира итеративно рачунање PageRank вредности на акцелераторској јединици. Иницијалне вредности свих чворова су постављене на  $1/n$ . Сви подаци се копирају у меморију акцелератора. Алгоритам ради у superstep итерацијама где се у сваком superstep-у извршавају следеће операције:

1. Агрегација порука - Покреће се OpenCL kernel где сваки work-item (нит) обрађује један чвор. Сваки чвор сабира поруке из свог inbox-а и рачуна нову PageRank вредност по формули  $PR(v) = (1 - d)/n + d * \Sigma(msg)$ , где је  $d$  damping фактор (0.85),  $n$  број чворова, а  $\Sigma(msg)$  сума примљених порука.

2. Слање порука - У истом kernel-у, сваки чвор дели своју тренутну PageRank вредност својим суседима и додаје поруке у њихов outbox. Висећи чворови (без излазних грана) не шаљу поруке, већ се њихова маса акумулира. Пошто више work-item-а може писати у исти outbox, користе се атомске операције за избегавање трке над подацима.

3. Редистрибуција масе висећих чворова - Покреће се посебан kernel који сабира dangling масу од свих висећих чворова користећи атомске операције. Затим се покреће још један kernel који равномерно дистрибуира ову масу свим чворовима.

4. Синхронизација - Подаци се размењују између inbox и outbox баффера у меморији акцелератора. Ово симулира глобалну баријеру BSP модела. OpenCL обезбеђује синхронизацију на нивоу work-group-а и глобалну синхронизацију између позива kernel-а.

Алгоритам се зауставља када се достигне максималан број superstep-ова. Након завршетка, резултати се копирају назад у хост меморију.

---

```
1 vector<double> rankPages(
2     unordered_map<string, int>& pageIds,
3     vector<string>& pageNames,
4     vector<int>& edges, vector<int>& offsets,
5     int maxSupersteps) {
6     int n = pageIds.size();
7     int m = edges.size();
8
9     cl_int err;
10    cl_platform_id platform;
11    cl_device_id device;
12    cl_context context;
13    cl_command_queue queue;
14    cl_program program;
15
16    err = clGetPlatformIDs(1, &platform, NULL);
17    checkError(err, "clGetPlatformIDs");
18
19    err = clGetDeviceIDs(
20        platform, CL_DEVICE_TYPE_GPU,
21        1, &device, NULL
22    );
23    if (err != CL_SUCCESS) {
24        err = clGetDeviceIDs(
25            platform, CL_DEVICE_TYPE_CPU,
26            1, &device, NULL
27        );
28        checkError(err, "clGetDeviceIDs");
29    }
30
31    context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
32    checkError(err, "clCreateContext");
33
34    queue = clCreateCommandQueue(context, device, 0, &err);
35    checkError(err, "clCreateCommandQueue");
36
37    program = clCreateProgramWithSource(
38        context, 1, &kernelSource,
39        NULL, &err
40    );
41    checkError(err, "clCreateProgramWithSource");
42    ...
```

---



---

```
1  ...
2  err = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
3  if (err != CL_SUCCESS) {
4      size_t log_size;
5      clGetProgramBuildInfo(
6          program, device, CL_PROGRAM_BUILD_LOG,
7          0, NULL, &log_size
8      );
9
10     char* log = new char[log_size];
11     clGetProgramBuildInfo(
12         program, device, CL_PROGRAM_BUILD_LOG,
13         log_size, log, NULL
14     );
15     cerr << "Build error:\n" << log << endl;
16     delete[] log;
17
18     exit(1);
19 }
20
21 cl_kernel pageRankKernel = clCreateKernel(
22     program, "pageRankKernel", &err
23 );
24 checkError(err, "clCreateKernel pageRankKernel");
25
26 cl_kernel danglingMassKernel = clCreateKernel(
27     program, "danglingMassKernel", &err
28 );
29 checkError(err, "clCreateKernel danglingMassKernel");
30
31 cl_kernel addDanglingMassKernel = clCreateKernel(
32     program, "addDanglingMassKernel", &err
33 );
34 checkError(err, "clCreateKernel addDanglingMassKernel");
35
36 vector<double> h_pageRanks(n, 1.0 / n);
37 vector<double> h_inbox(n, 0.0);
38
39 cl_mem d_pageRanks = clCreateBuffer(
40     context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
41     n * sizeof(double), h_pageRanks.data(), &err
42 );
43 checkError(err, "clCreateBuffer pageRanks");
44
45 cl_mem d_nextPageRanks = clCreateBuffer(
46     context, CL_MEM_READ_WRITE,
47     n * sizeof(double), NULL, &err
48 );
49 checkError(err, "clCreateBuffer nextPageRanks");
50 ...
```

---

---

```
1  ...
2      cl_mem d_inbox = clCreateBuffer(
3          context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
4          n * sizeof(double), h_inbox.data(), &err
5      );
6      checkError(err, "clCreateBuffer inbox");
7
8      cl_mem d_outbox = clCreateBuffer(
9          context, CL_MEM_READ_WRITE,
10         n * sizeof(double), NULL, &err
11     );
12     checkError(err, "clCreateBuffer outbox");
13
14     cl_mem d_edges = clCreateBuffer(
15         context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
16         m * sizeof(int), edges.data(), &err
17     );
18     checkError(err, "clCreateBuffer edges");
19
20     cl_mem d_offsets = clCreateBuffer(
21         context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
22         (n + 1) * sizeof(int), offsets.data(), &err
23     );
24     checkError(err, "clCreateBuffer offsets");
25
26     cl_mem d_danglingMass = clCreateBuffer(
27         context, CL_MEM_READ_WRITE,
28         sizeof(double), NULL, &err
29     );
30     checkError(err, "clCreateBuffer danglingMass");
31
32     size_t globalWorkSize = ((n + 255) / 256) * 256;
33     size_t localWorkSize = 256;
34
35     for (int step = 0; step < maxSupersteps; ++step) {
36         double zero = 0.0;
37         err = clEnqueueFillBuffer(
38             queue, d_outbox, &zero, sizeof(double),
39             0, n * sizeof(double), 0, NULL, NULL
40         );
41         checkError(err, "clEnqueueFillBuffer outbox");
42
43         err = clEnqueueFillBuffer(
44             queue, d_danglingMass, &zero, sizeof(double),
45             0, sizeof(double), 0, NULL, NULL
46         );
47         checkError(err, "clEnqueueFillBuffer danglingMass");
48
49         clSetKernelArg(pageRankKernel, 0, sizeof(cl_mem), &d_inbox);
50     }
51     ...
```

---

```
1  ...
2
3      clSetKernelArg(pageRankKernel, 1, sizeof(cl_mem), &d_pageRanks);
4      clSetKernelArg(pageRankKernel, 2, sizeof(cl_mem), &d_offsets);
5      clSetKernelArg(pageRankKernel, 3, sizeof(cl_mem), &d_edges);
6      clSetKernelArg(pageRankKernel, 4, sizeof(cl_mem), &d_nextPageRanks);
7      clSetKernelArg(pageRankKernel, 5, sizeof(cl_mem), &d_outbox);
8      clSetKernelArg(pageRankKernel, 6, sizeof(int), &n);
9      clSetKernelArg(pageRankKernel, 7, sizeof(double), &DAMPING);
10
11     err = clEnqueueNDRangeKernel(
12         queue, pageRankKernel, 1, NULL, &globalWorkSize,
13         &localWorkSize, 0, NULL, NULL
14     );
15     checkError(err, "clEnqueueNDRangeKernel pageRankKernel");
16
17     clSetKernelArg(danglingMassKernel, 0, sizeof(cl_mem), &d_pageRanks);
18     clSetKernelArg(danglingMassKernel, 1, sizeof(cl_mem), &d_offsets);
19     clSetKernelArg(danglingMassKernel, 2, sizeof(cl_mem), &d_danglingMass);
20     clSetKernelArg(danglingMassKernel, 3, sizeof(int), &n);
21
22     err = clEnqueueNDRangeKernel(
23         queue, danglingMassKernel, 1, NULL, &globalWorkSize,
24         &localWorkSize, 0, NULL, NULL
25     );
26     checkError(err, "clEnqueueNDRangeKernel danglingMassKernel");
27
28     double danglingMass;
29     err = clEnqueueReadBuffer(
30         queue, d_danglingMass, CL_TRUE, 0,
31         sizeof(double), &danglingMass, 0, NULL, NULL
32     );
33     checkError(err, "clEnqueueReadBuffer danglingMass");
34
35     double danglingShare = DAMPING * danglingMass / n;
36
37     clSetKernelArg(
38         addDanglingMassKernel, 0,
39         sizeof(cl_mem), &d_nextPageRanks
40     );
41     clSetKernelArg(
42         addDanglingMassKernel, 1,
43         sizeof(double), &danglingShare
44     );
45     clSetKernelArg(
46         addDanglingMassKernel,
47         2, sizeof(int), &n
48     );
49     ...
```

---

```
1  ...
2      err = clEnqueueNDRangeKernel(
3          queue, addDanglingMassKernel, 1, NULL,
4          &globalWorkSize, &localWorkSize, 0, NULL, NULL
5      );
6      checkError(err, "clEnqueueNDRangeKernel addDanglingMassKernel");
7
8      err = clEnqueueCopyBuffer(
9          queue, d_outbox, d_inbox, 0, 0,
10         n * sizeof(double), 0, NULL, NULL
11     );
12     checkError(err, "clEnqueueCopyBuffer");
13
14     swap(d_pageRanks, d_nextPageRanks);
15 }
16
17 err = clEnqueueReadBuffer(
18     queue, d_pageRanks, CL_TRUE, 0,
19     n * sizeof(double), h_pageRanks.data(), 0, NULL, NULL
20 );
21 checkError(err, "clEnqueueReadBuffer pageRanks");
22
23 clReleaseMemObject(d_pageRanks);
24 clReleaseMemObject(d_nextPageRanks);
25 clReleaseMemObject(d_inbox);
26 clReleaseMemObject(d_outbox);
27 clReleaseMemObject(d_edges);
28 clReleaseMemObject(d_offsets);
29 clReleaseMemObject(d_danglingMass);
30 clReleaseKernel(pageRankKernel);
31 clReleaseKernel(danglingMassKernel);
32 clReleaseKernel(addDanglingMassKernel);
33 clReleaseProgram(program);
34 clReleaseCommandQueue(queue);
35 clReleaseContext(context);
36
37 return h_pageRanks;
38 }
```

---

Изворни код 4: Паралелна имплементација (OpenCL)

### 3.5 Резултати

У овом одељку приказани су резултати извршавања PageRank алгоритма имплементираних по Pregel vertex-centric моделу. Сви експерименти су извршени над истим графом који се састоји од 10000 чворова, како би поређење између различитих

имплементација било конзистентно.

Алгоритам је покретан са различитим бројем итерација, и то редом за 10, 100, 1000, 10000, 100000, 1000000 и 10000000 итерација. На овај начин омогућено је посматрање понашања алгоритма и утицаја паралелизације како при малом, тако и при веома великом броју superstep-ова.

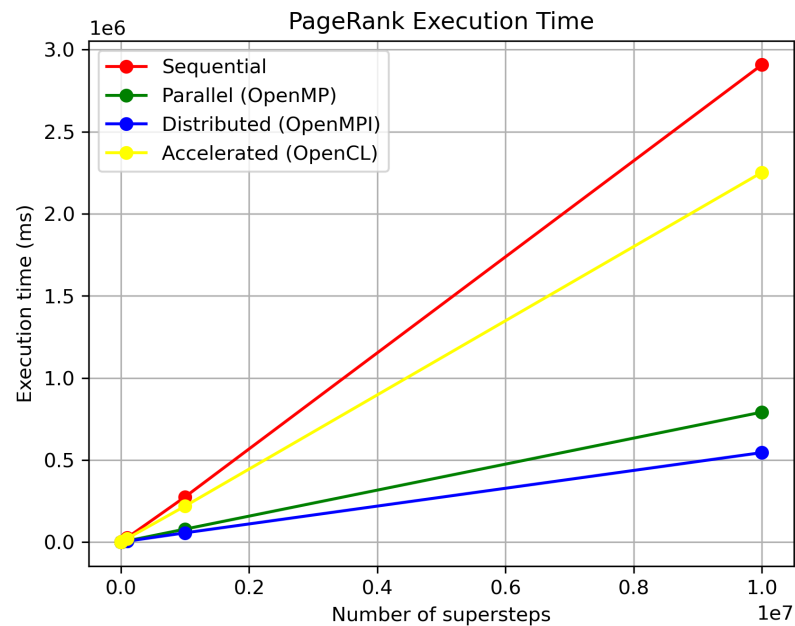
За сваку имплементацију измерена су укупна времена извршавања, а на основу секвенцијалне верзије израчуната су убрзања паралелних решења. Добијени резултати омогућавају директно поређење ефикасности различитих приступа паралелизацији у оквиру истог алгоритма и радног оптерећења.

Графички приказ резултата налази се на сликама 9, 10, 11 и 12. Времена извршавања приказана су на два графика: један са линеарном x-скалом и један са логаритамском x-скалом. На овај начин је лакше уочити понашање алгоритма како при малом, тако и при великом броју superstep-ова. Убрзања паралелних имплементација такође су приказана на два графика са истим скалама, што омогућава поређење ефекта паралелизације у различитим условима извршавања.

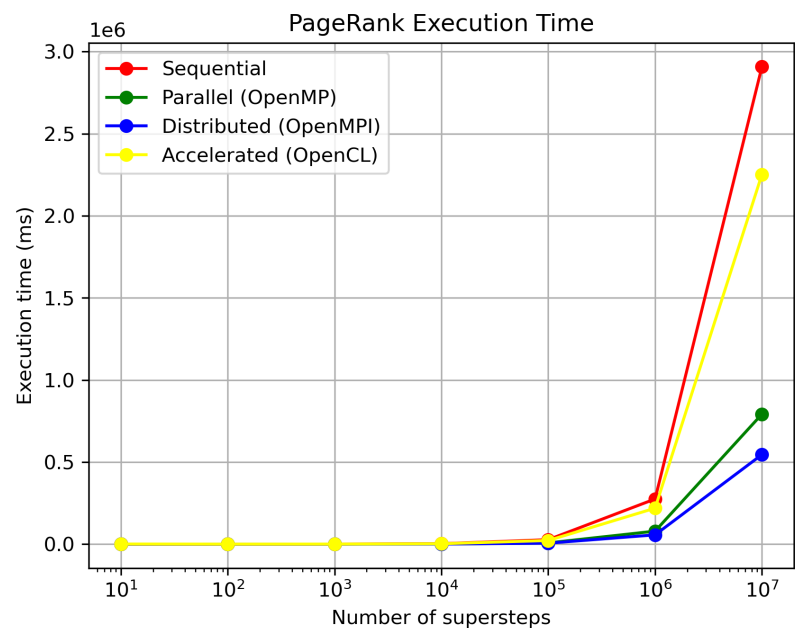
Анализа добијених резултата показује да су све паралелне имплементације оствариле боље перформансе у односу на секвенцијалну верзију алгоритма, што потврђује оправданост примене паралелизације у оквиру Pregel vertex-centric модела. Међу паралелним решењима, имплементације засноване на OpenMP и OpenMPI показале су знатно краћа времена извршавања у поређењу са OpenCL верзијом. Посебно се истиче OpenMPI имплементација, која је остварила највеће убрзање, нарочито при већем броју итерација, где доминира трошак израчунавања над трошковима комуникације и синхронизације.

На графику времена извршавања јасно се уочавају четири приближно праве линије, што указује на линеарну зависност времена извршавања од броја superstep-ова. Овакво понашање је у складу са очекивањима, јер свака итерација алгоритма подразумева константан скуп операција над чворовима графа.

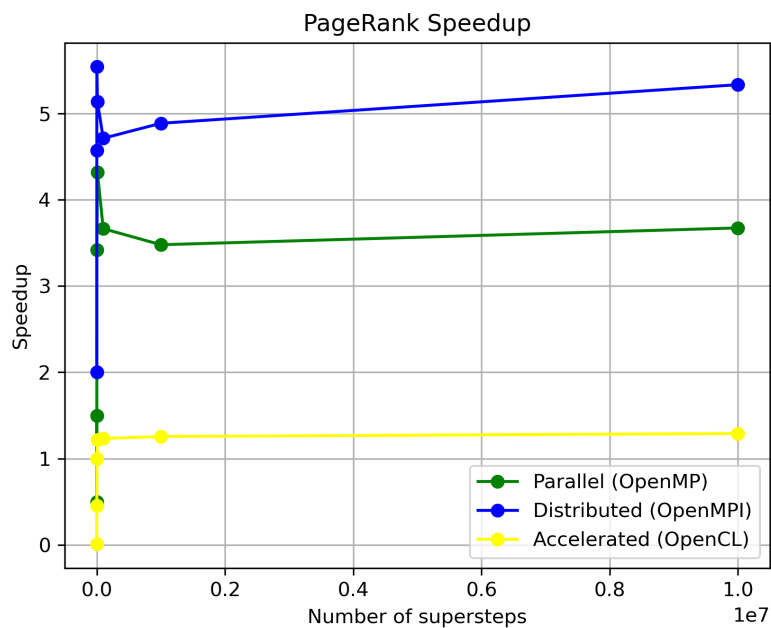
На графику убрзања може се уочити да је убрзање приближно константно за све паралелне имплементације у случајевима када је број superstep-ова довољно велики. Ово указује на то да се трошкови иницијализације и синхронизације амортизују са порастом броја итерација, па доминантан утицај на укупно време извршавања има сама паралелна обрада.



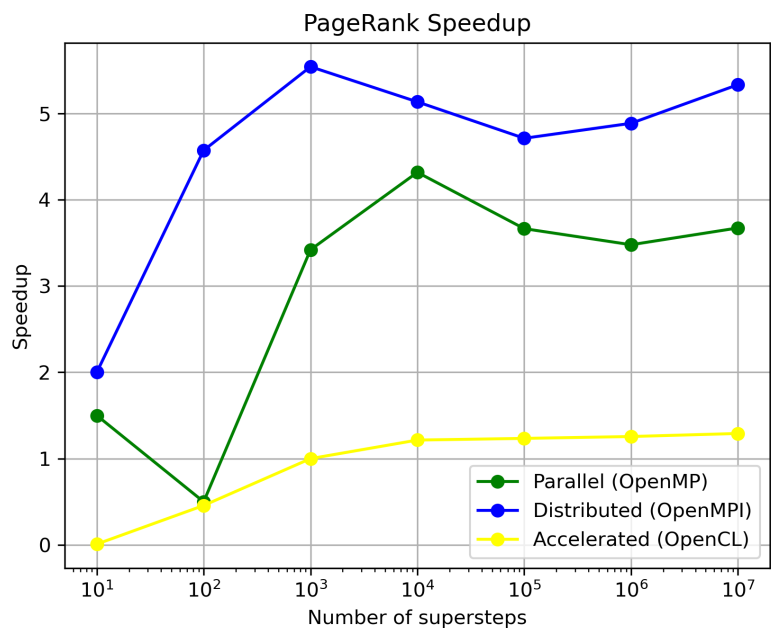
Слика 9: График времена извршавања са линеарном скалом



Слика 10: График времена извршавања са логаритамском скалом



Слика 11: График убрзања са линеарном скалом



Слика 12: График убрзања са логаритамском скалом

## 4 Закључак

У овом раду приказана је примена Pregel vertex-centric модела и Bulk Synchronous Parallel (BSP) парадигме за паралелну обраду графова, са посебним освртом на PageRank алгоритам. Реализоване су једна секвенцијална и три паралелне имплементације (OpenMP, OpenMPI и OpenCL) у програмском језику C++.

Експериментални резултати показали су да све паралелне верзије остварују значајно убрзање у односу на секвенцијалну, при чему имплементације засноване на OpenMP и OpenMPI издвајају по већој ефикасности. Времена извршавања расту линеарно са бројем superstep-ова, што је очекивано због природе BSP парадигме и vertex-centric приступа. Убрзање паралелних имплементација је приближно константно када број superstep-ова постане велики, што указује на амортизацију трошкова синхронизације и комуникације.

Резултати потврђују да су Pregel vertex-centric модел и Bulk Synchronous Parallel (BSP) парадигма погодни за паралелизацију графовских алгоритама.



## Библиографија

- [1] Aart J.C Bik James C. Dehnert Ilan Horn Naty Leiser Grzegorz Czajkowski Grzegorz Malewicz, Matthew H. Austern. Pregel: a system for large-scale graph processing. <https://research.google/pubs/pregel-a-system-for-large-scale-graph-processing/>.
- [2] Apache Giraph. <https://giraph.apache.org/>.
- [3] Apache Spark GraphX. <https://spark.apache.org/graphx/>.
- [4] Apache Hama. <https://hama.apache.org/>.
- [5] Dionysios Logothetis Claudio Martella, Roman Shaposhnik. Practical Graph Analytics with Apache Giraph. O'Reilly Media, Inc.