



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:*

E2 47/2024

11. јануар 2026.

## Сажетак

Овај рад се бави Pregel vertex-centric моделом обраде графова и основана његове оптимизације и паралелизације применом Bulk Synchronous Parallel (BSP) приступа. Pregel vertex-centric модел заснива се на идеји обраде графова при којој сваки чвр 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 Увод</b>	<b>1</b>
1.1 Значај обраде великих графова . . . . .	1
1.2 Изазови при обради графова . . . . .	1
<b>2 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 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 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

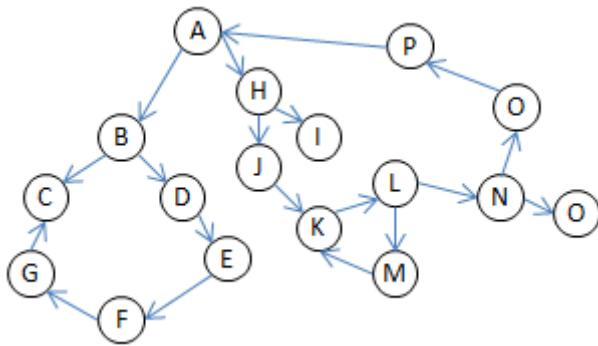
1	Граф . . . . .	1
2	Google лого . . . . .	2
3	Apache Giraph лого . . . . .	3
4	Apache Spark GraphX лого . . . . .	3
5	Apache Hama лого . . . . .	3
6	Vertex-centric алгоритам за проналажење максимума у графу . . . . .	5
7	Bulk Synchronous Parallel (BSP) модел . . . . .	6
8	Типична архитектура дистрибуираног 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 нелинейне структуре. Главни изазови могу се сумирати на следећи начин:

1. Неправилна структура података – За разлику од табеларних података, графови немају унiformну структуру. Број суседа по чврлу може значајно да варира, што отежава равномерну расподелу посла.
2. Јака међувисност података – Обрада једног чврла 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 дистрибуирајућој обради, чворови који су логички повезани могу бити физички смештени на различитим процесима или рачунарима, што повећава трошкове комуникације.

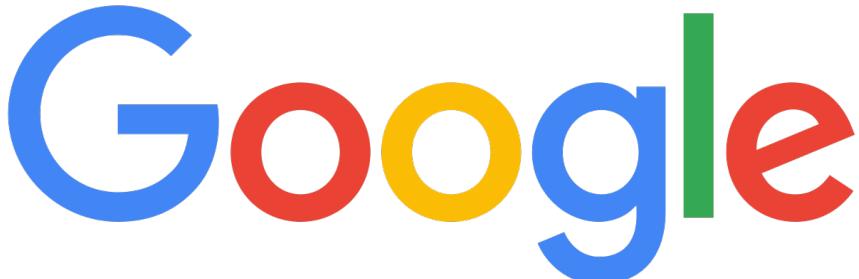
Ови изазови указују на потребу за моделима обраде који су посебно прилагођени структури и природи графовских података. Један од таквих модела је Pregel vertex-centric модел, који у комбинацији са Bulk Synchronous Parallel (BSP) парадигмом омогућава ефикасну и скалабилну обраду великих графова.

## 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.

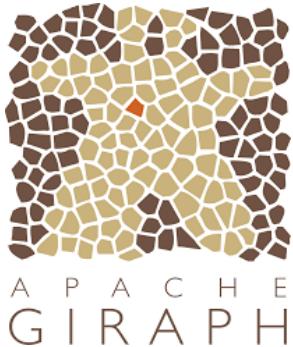
### 2.1 Историјат и примена Pregel модела

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 чврла и веза.

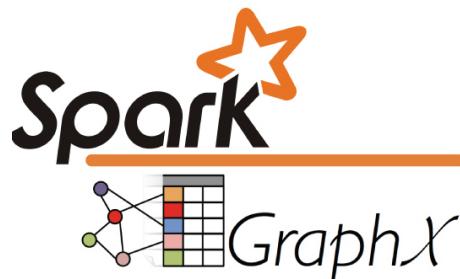


Слика 2: Google лого

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:

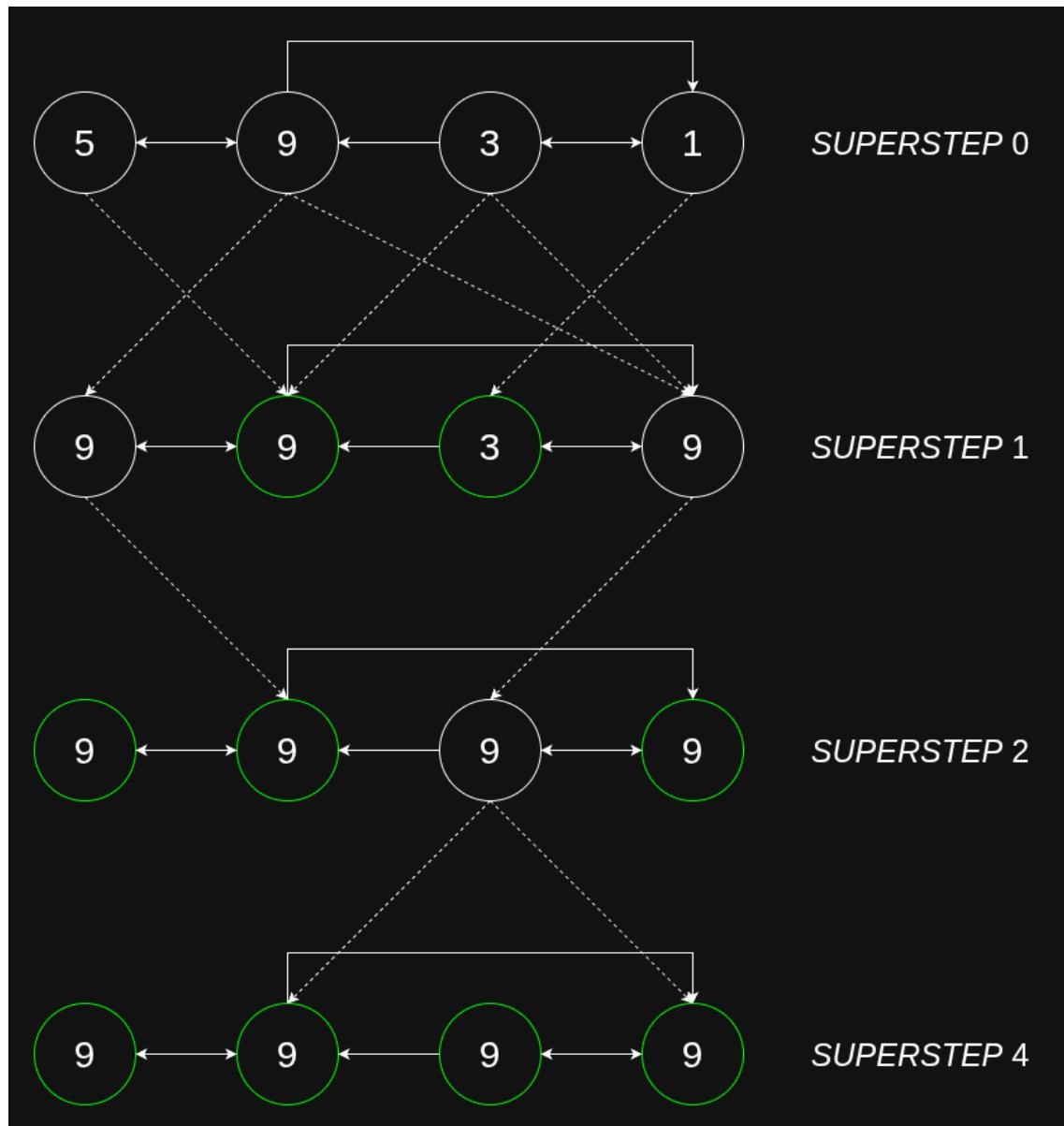
1. Примање порука – Чворови примају поруке које су им други чворови послали у претходном superstep-у.
2. Локално израчунавање – Чвор ажурира своје стање користећи примљене поруке и претходно стање.
3. Слање порука – Чвор шаље резултате обраде својим суседима, који ће их користити у следећем superstep-у.

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.

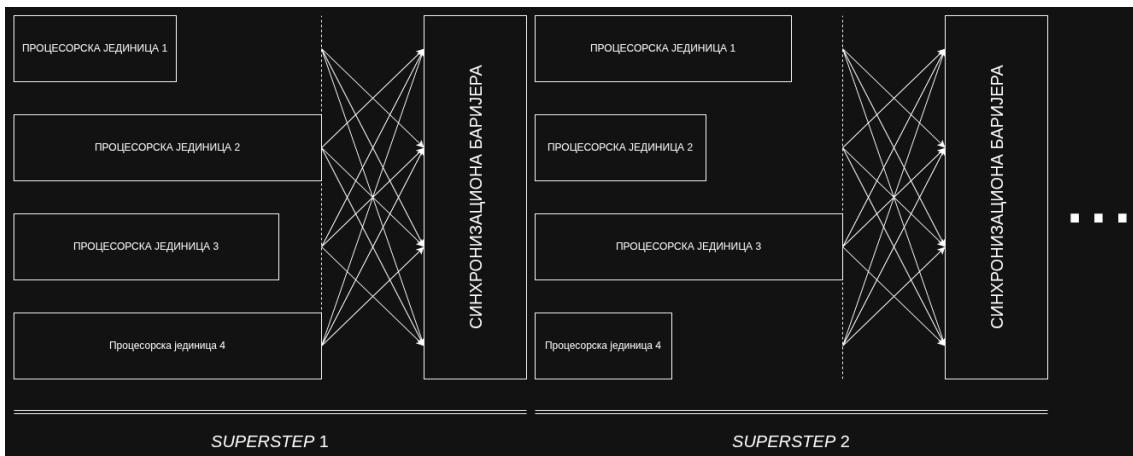


Слика 6: Vertex-centric алгоритам за проналажење максимума у графу

## 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:

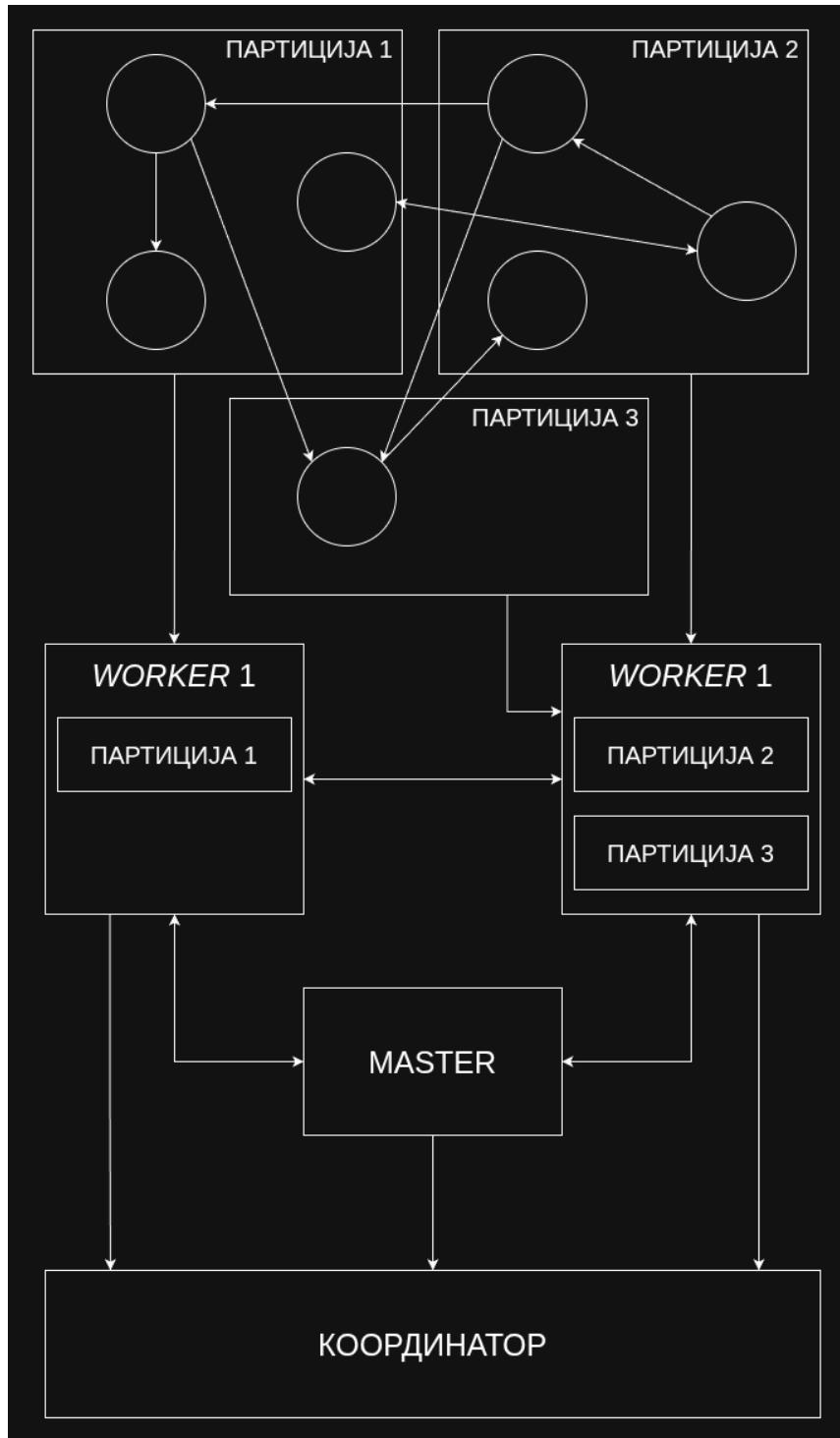
1. Локална обрада – Свака процесорска јединица обрађује свој подскуп података независно од осталих.
2. Комуникација – Јединице размењују поруке како би обезбедиле податке потребне за следећи superstep.
3. Синхронизациона баријера – Сви процесори се синхронизују пре него што започну наредни superstep,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

ритам ради у superstep итерацијама где се у сваком superstep-у извршавају следеће операције:

1. Агрегација порука - Сваки чвор сабира све поруке из свог inbox-a.
2. Рачунање нове вредности - Нова PageRank вредност се рачуна по формулама  $PR(v) = (1 - d)/n + d * \sum_{msg}$ , где је  $d$  damping фактор (0.85),  $n$  број чворова, а  $\sum_{msg}$  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        ...
26    }
27
28    ...
29
30    ...
31
32    ...
33
34    ...
35
36    ...
37
38    ...
39
40    ...
41
42    ...
43
44    ...
45
46    ...
47
48    ...
49
50    ...
51
52    ...
53
54    ...
55
56    ...
57
58    ...
59
59
60    ...
61
62    ...
63
64    ...
65
66    ...
67
67
68    ...
69
69
70    ...
71
72    ...
73
74    ...
75
75
76    ...
77
77
78    ...
79
79
80    ...
81
81
82    ...
83
83
84    ...
85
85
86    ...
87
87
88    ...
89
89
90    ...
91
91
92    ...
93
93
94    ...
95
95
96    ...
97
97
98    ...
99
99
100   ...
100
101
102
103
104
105
106
107
108
109
109
110
111
112
113
114
115
116
117
118
119
119
120
121
122
123
124
125
125
126
127
127
128
129
129
130
131
132
132
133
133
134
134
135
135
136
136
137
137
138
138
139
139
140
140
141
141
142
142
143
143
144
144
145
145
146
146
147
147
148
148
149
149
150
150
151
151
152
152
153
153
154
154
155
155
156
156
157
157
158
158
159
159
160
160
161
161
162
162
163
163
164
164
165
165
166
166
167
167
168
168
169
169
170
170
171
171
172
172
173
173
174
174
175
175
176
176
177
177
178
178
179
179
180
180
181
181
182
182
183
183
184
184
185
185
186
186
187
187
188
188
189
189
190
190
191
191
192
192
193
193
194
194
195
195
196
196
197
197
198
198
199
199
200
200
201
201
202
202
203
203
204
204
205
205
206
206
207
207
208
208
209
209
210
210
211
211
212
212
213
213
214
214
215
215
216
216
217
217
218
218
219
219
220
220
221
221
222
222
223
223
224
224
225
225
226
226
227
227
228
228
229
229
230
230
231
231
232
232
233
233
234
234
235
235
236
236
237
237
238
238
239
239
240
240
241
241
242
242
243
243
244
244
245
245
246
246
247
247
248
248
249
249
250
250
251
251
252
252
253
253
254
254
255
255
256
256
257
257
258
258
259
259
260
260
261
261
262
262
263
263
264
264
265
265
266
266
267
267
268
268
269
269
270
270
271
271
272
272
273
273
274
274
275
275
276
276
277
277
278
278
279
279
280
280
281
281
282
282
283
283
284
284
285
285
286
286
287
287
288
288
289
289
290
290
291
291
292
292
293
293
294
294
295
295
296
296
297
297
298
298
299
299
300
300
301
301
302
302
303
303
304
304
305
305
306
306
307
307
308
308
309
309
310
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1
```

```
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 суседа.

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 чвррова 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 чврлове је паралелизована, свака нит обрађује свој подскуп чвррова независно.
2. Рачунање нове вредности - Нова PageRank вредност се рачуна по формулама  $PR(v) = (1 - d)/n + d * \sum(\text{msg})$ , где је  $d$  damping фактор ( $0.85$ ),  $n$  број чвррова, а  $\sum(\text{msg})$  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 чврлови (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 чвррова - Акумулирана вредност од висећих чвррова се равномерно дистрибуира свим чврловима. 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-a.
2. Рачунање нове вредности - Нова PageRank вредност се рачуна по формулама  $PR(v) = (1 - d)/n + d * \sum(\text{msg})$ , где је  $d$  damping фактор (0.85),  $n$  укупан број чворова у графу, а  $\sum(\text{msg})$  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-ja.
4. Редистрибуција масе висећих чворова - Сваки процес израчунава локални до-принос 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. Out-box 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 verticiesStart = rank * verticesPerProcess;
19    int verticiesEnd = min(verticiesStart + verticesPerProcess, n);
20    int localN = max(0, verticiesEnd - verticiesStart);
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    ...
37 }
```

---

```
1 ...
2     ...
3         for (int u = processStart; u < processEnd; ++u) {
4             int edgeCount = outEdges[u].size();
5             MPI_Send(
6                 &edgeCount,
7                 1, MPI_INT, process, 0, MPI_COMM_WORLD
8             );
9             MPI_Send(
10                outEdges[u].data(),
11                edgeCount, MPI_INT, process, 0, MPI_COMM_WORLD
12            );
13
14             edgeCount = inEdges[u].size();
15             MPI_Send(
16                 &edgeCount,
17                 1, MPI_INT, process, 0, MPI_COMM_WORLD
18             );
19             MPI_Send(
20                inEdges[u].data(),
21                edgeCount, MPI_INT, process, 0, MPI_COMM_WORLD
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     ...
}
```

---

```
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 = verticiesStart + 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, акселератори). 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 акселераторској 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 Page-Rank вредности на акцелераторској јединици. Иницијалне вредности свих чвирова 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) обраћује један чвор. Сваки чвор сабира поруке из свог inbox-а и рачуна нову PageRank вредност по формули  $PR(v) = (1 - d)/n + d * \sum(\text{msg})$ , где је  $d$  damping фактор (0.85),  $n$  број чвирова, а  $\sum(\text{msg})$  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 чвирова - Покреће се посебан kernel који сабира dangling масу од свих висећих чвирова користећи атомске операције. Затим се покреће још један kernel који равномерно дистрибуира ову масу свим чвровима.
4. Синхронизација - Подаци се размењују између inbox и outbox бафера 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
43    ...
44}
```

```
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     ...
```

---

```
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 имплементираног 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.

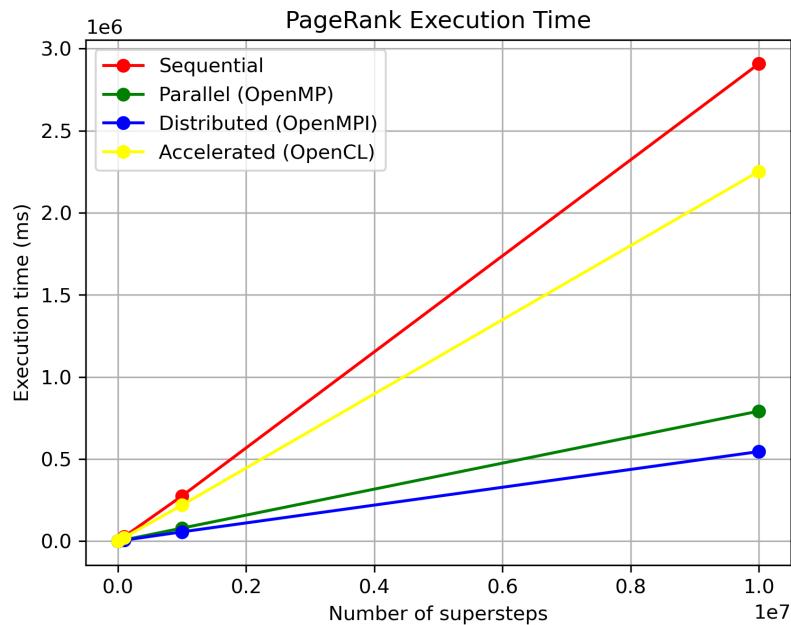
За сваку имплементацију измерена су укупна времена извршавања, а на основу секвенцијалне верзије израчуната су убрзања паралелних решења. Добијени резултати омогућавају директно поређење ефикасности различитих приступа паралелизацији у оквиру истог алгоритма и радног оптерећења.

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.

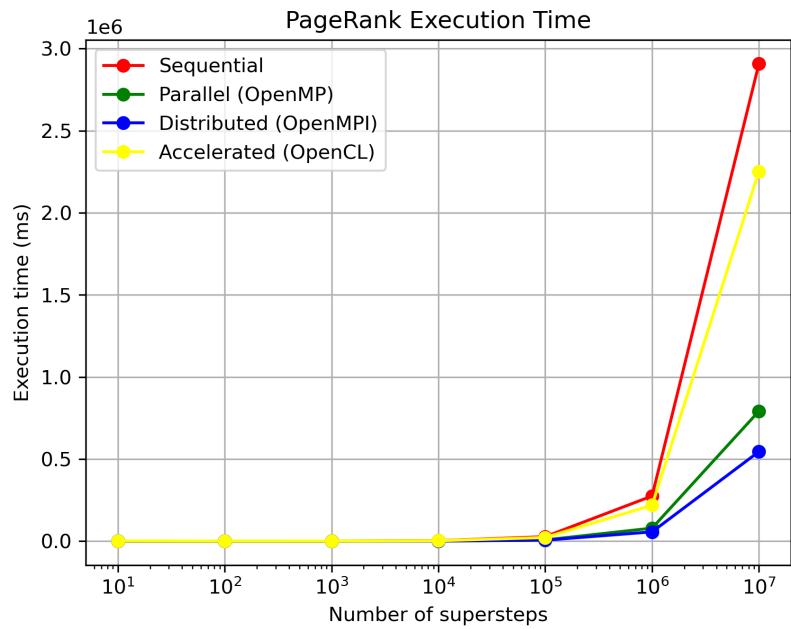
Анализа добијених резултата показује да су све паралелне имплементације оствариле боље перформансе у односу на секвенцијалну верзију алгоритма, што потврђује оправданост примене паралелизације у оквиру Pregel vertex-centric модела. Међу паралелним решењима, имплементације засноване на OpenMP и OpenMPI 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.

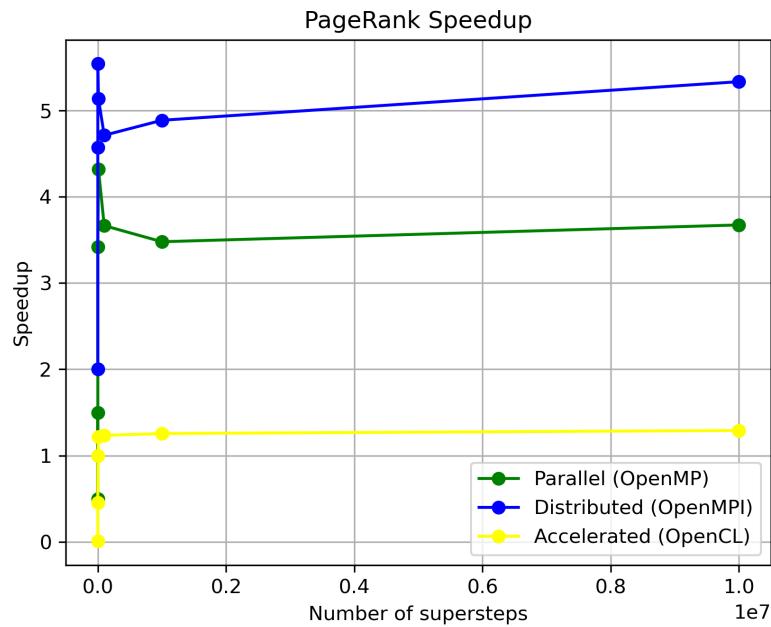
На графику убрзања може се уочити да је убрзање приближно константно за све паралелне имплементације у случајевима када је број superstep-ова доволно велики. Ово указује на то да се трошкови иницијализације и синхронизације амортизују са порастом броја итерација, па доминантан утицај на укупно време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



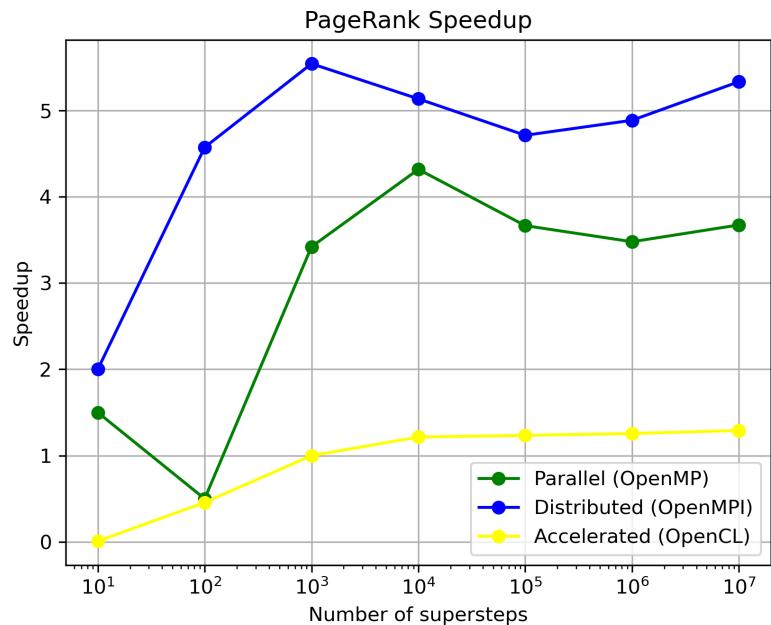
Слика 9: График времена извршавања са линеарном скалом



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 издавају 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/preprint-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.