Playing With PageRank

Comparing different implementations

Group 15
Naisila Puka 21600336
Endi Merkuri 21600337
Fatbardh Feta 21600334
CS 425, 6 December 2018, Bilkent

Outline

- Brief Introduction
- OpenMP
- Parallel Implementation
- Spark
- MapReduce Implementation
- Comparisons

Brief Introduction

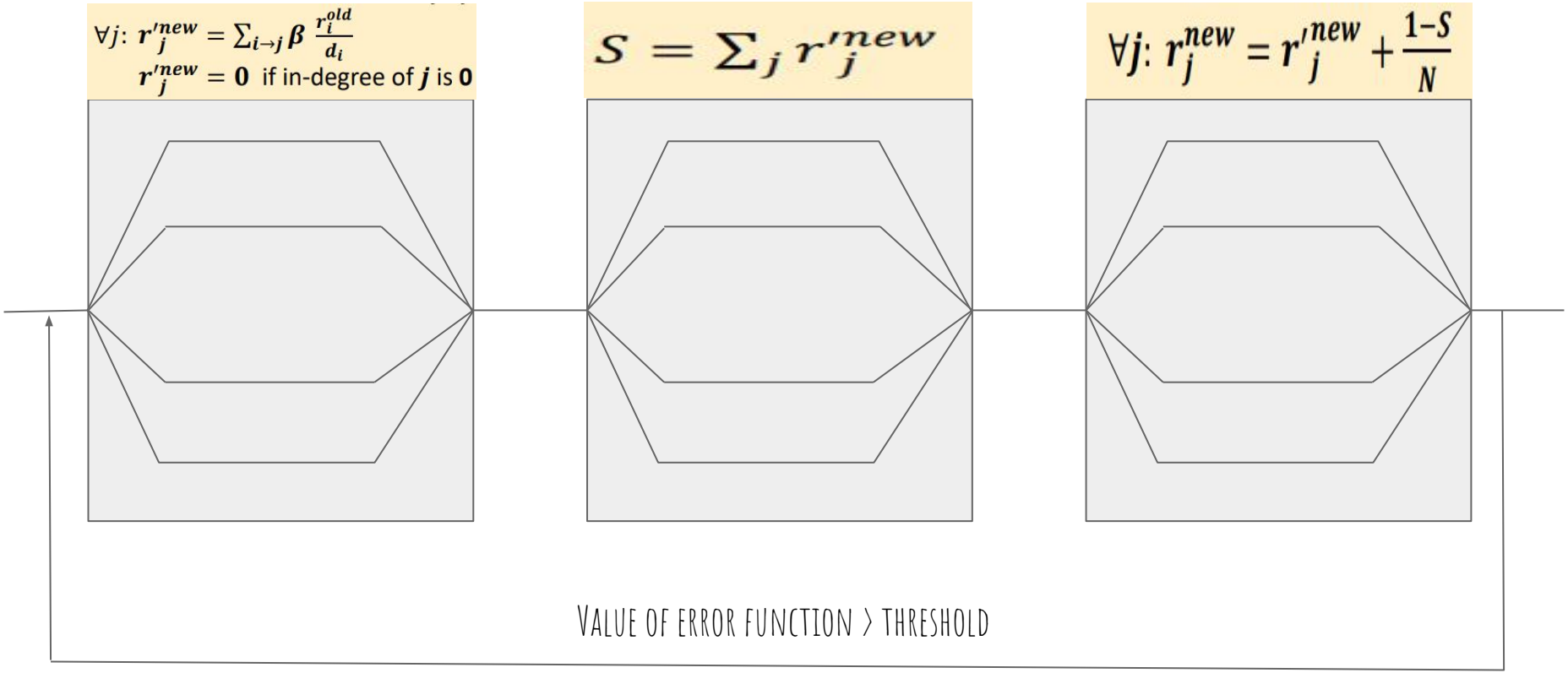
- In this project we implemented the PageRank algorithm in three different ways:
  - Serial (C++)
  - Parallel shared-memory using OpenMP (C++)
  - Distributed version using MapReduce algorithm in PySpark (Python)
- We used two different datasets from Web Graphs of Stanford Dataset Collection and made comparisons in the amount of time taken in the completion of PageRank algorithm upon these datasets

OpenMp introduction:

OpenMP API is a portable, scalable model that gives **shared-memory parallel programmers** a simple and flexible interface for **developing parallel applications.**

- The **Pragma OMP parallel** is used to fork additional threads to carry out the work enclosed in the construct in parallel.
- Among other languages, it also uses C++.

Parallel Implementation with OpenMP



Parallel Implementation with OpenMP

- Computer Setup:

2 Cores, 4 Hardware Threads

Parallel Algorithm Pseudocode

Preprocessing

While the error function is greater than the threshold:

Parallel: For each node

Update previous rank and set current rank to 0

Parallel: For each node

Add its pagerank contribution to its outgoing links

Parallel: For each node

Multiply its current rank with beta

Calculate the sum of all the ranks

Parallel Algorithm Pseudocode (continued)

Find the leakage due to dead-ends and calculate teleportation probability

Parallel: For each node

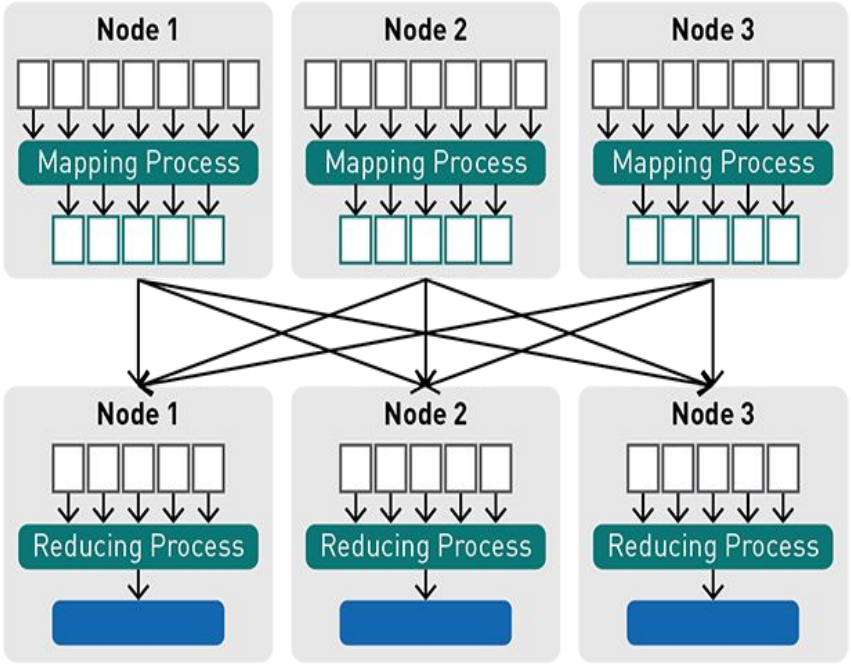
Add the teleportation probability to the nodes current rank

Calculate the error using L1-norm

Spark Python API (PySpark)

Overview:

1. Spark is a general-purpose **distributed data processing** engine.
2. RDD (resilient distributed dataset) is a **fault-tolerant** collection of elements that can be operated on in parallel.



Important PySpark functions:

- **rdd** : creates a **resilient distributed dataset** from a given Data Structure.
- **flatMap** : is a transformation operation that is applied to all the elements of the RDD and returns a new RDD which can have a different size from its ancestor(map is an other similar function to flatMap).
- **reduceByKey** : is an RDD function that merges the values of each key using an associative reduce function.
- **join** : joins two given data frames on a column based on the function that is passed.

Distributed version using MapReduce in PySpark

$\beta = 0.85$

1 -> Map Reduce Algorithm for Data Processing

2 -> Another Map Reduce Algorithm for PageRank iterations

3 -> Reinsert PageRank Leakage

4 -> Check for Convergence

Distributed version using MapReduce in PySpark

- Computer Setup:

Master Node, 2 Cores

Worker 1, 2 Cores

Worker 2, 2 Cores

Worker 3, 2 Cores

First Map-Reduce

Data in the form:
`URL neighborURL`

```
URL1 OUTLINK1_1
URL1 OUTLINK1_2
URL1 OUTLINK1_3
URL2 OUTLINK2_1
URL3 OUTLINK3_1
URL3 OUTLINK3_2
URL3 OUTLINK3_3
URL3 OUTLINK3_4
.
.
.
```

Data in the form:
`URL [list of neighbor URLs, list size]`

```
URL1 ([OUTLINK1_1, OUTLINK1_2, OUTLINK1_3], 3)
URL2 ([OUTLINK2_1], 1)
URL3 ([OUTLINK3_1, OUTLINK3_2, OUTLINK3_3, OUTLINK3_4], 4)
.
.
.
```

First Map-Reduce

Map Function

```
for each URL1 OUTLINK1_1 line of the document
    generate <key = URL1, value = OUTLINK1_1>
```

Reduce Function

```
for each <URL, OUTLINK_LIST>
    OUT_DEGREE <- length(OUTLINK_LIST)
    generate <key = URL, value = (OUTLINK_LIST, OUT_DEGREE)>
```

First Map-Reduce: Complexity Evaluation

Replication rate: 1

Average Reducer Size: $\bar{d}$

Communication Cost: 2E

# of Reducers: N

$\bar{d}$ → average number of outgoing links in the graph

E → number of edges

Second Map-Reduce (The Real Thing)

➢ Recall:

PageRank Algorithm from course slides

Contributions to out_links will be calculated using Map-Reduce!

Set: $r_j^{old} = \frac{1}{N}$

repeat until convergence: $\sum_j \left| r_j^{new} - r_j^{old} \right| > \varepsilon$

- $\forall j$: $r'^{new}_j = \sum_{i \to j} \beta \frac{r_i^{old}}{d_i}$
  $r'^{new}_j = 0$ if in-degree of j is 0
- **Now re-insert the leaked PageRank:**
  $\forall j$: $r_j^{new} = r'^{new}_j + \frac{1-S}{N}$ **where:** $S = \sum_j r'^{new}_j$
- $r^{old} = r^{new}$

By a simple Spark join we have:

`URL, (OUTLINK_LIST, OUT_DEGREE, CURRENT_RANK)`

Compute new ranks from what we have above:

`URL, NEW_RANK`

Second Map-Reduce (The Real Thing)

$$\forall j: r'^{new}_j = \sum_{i \to j} \beta \frac{r^{old}_i}{d_i}$$
$$r'^{new}_j = 0 \text{ if in-degree of } j \text{ is } 0$$

Map Function

```
for each <URL, (OUTLINK_LIST, OUT_DEGREE, CURRENT_RANK)>
    generate <key = URL, value = 0>
    for i = 1 to OUT_DEGREE
        generate <key = OUTLINK_LIST[i], value = 1/CURRENT_RANK>
```

Reduce Function

```
for each <URL, CONTRIBUTIONS_LIST>
    NEW_RANK <- sum(CONTRIBUTIONS_LIST)
    generate <key = URL, value = NEW_RANK>
```

Second Map-Reduce: Complexity Analysis

Replication rate: $\overline{d}$

Average Reducer Size: $\overline{d}$

Communication Cost: 2E

# of Reducers: N

Finishing The Iteration

- Calculate sum of `NEW_RANK` vector for leaked PageRank due to possible dead ends

```
S <- sum(NEW_RANK[])
```

- Reinsert the leaked PageRank in all ranks

```
for each NEW_RANK in NEW_RANK[]

    NEW_RANK <- NEW_RANK + (1 - S)/N
```

- Calculate L1 Norm of Rank Vectors and Check for convergence

RESULTS

- As expected, we observed that the parallel implementation takes shorter amount of time than the serial one.
- We saw that the distributed implementation using Map-Reduce is actually really efficient compared to Parallel implementation.
- Data preprocessing is done faster in Spark.
- The distributed implementation is not feasible with the calculation of PageRank leakage in case the graph has dead-ends.

Web-NotreDame: 325729 Nodes, 1497134 Edges, With Deadends

Convergence tolerance: 0.0001, 33 iterations in total

	Setup	Data Processing	PageRank	Total
Serial	-	0.69	3.41	4.1
Shared-Memory	-	0.74	4.04	4.78
Distributed	12	1.78	655	669

Web-Google: 875713 Nodes, 5105039 Edges, With Deadends

Convergence tolerance: 0.0001, 35 iterations in total

	Setup	Data Processing	PageRank	Total
Serial	-	2.76	17.6	20.4
Shared-Memory	-	2.98	14.2	17.2
Distributed	18 s	2.16	2700	2720

Web-NotreDame: 325729 Nodes, 1497134 Edges, Without Deadends

Convergence tolerance: 0.0001, 33 iterations in total

	Setup	Data Processing	PageRank	Total
Serial	-	0.83	4.64	5.47
Shared-Memory	-	0.91	4.39	5.3
Distributed	11.8	1.78	3.14	16.72

Web-Google: 875713 Nodes, 5105039 Edges, Without Deadends

Convergence tolerance: 0.0001, 35 iterations in total

	Setup	Data Processing	PageRank	Total
Serial	-	3.19	19.4	22.6
Shared-Memory	-	3.06	14	17.1
Distributed	12.1	2.16	3.39	17.65

Patent Citation Network: 3774768 Nodes, 16518948 Edges, Without Deadends

Convergence tolerance: 0.0001, 10 iterations in total

	Setup	Data Processing	PageRank	Total
Serial	-	9.77	29.3	39.1
Shared-Memory	-	10.3	22.7	33
Distributed	11.9	1.78	0.79	14.47

Thank You For Listening!

Any questions?