Introduction to Data Science/ Data Intensive Computing (CIS 6930)

PROJECT III Report

Instructor Dr. Sanjay Ranka

Submitted By: Rajat Koujalagi

6138 2944

1. Introduction

In task 1 we implement the page rank algorithm using open MP. We read the file using only one thread and then further calculate the page ranks of the pages using multiple threads.
In task 2 we write a parallel reducer using MPI.

2. Implementation Description

Task 1: Parallel Page rank program in open MP.

Implementation:

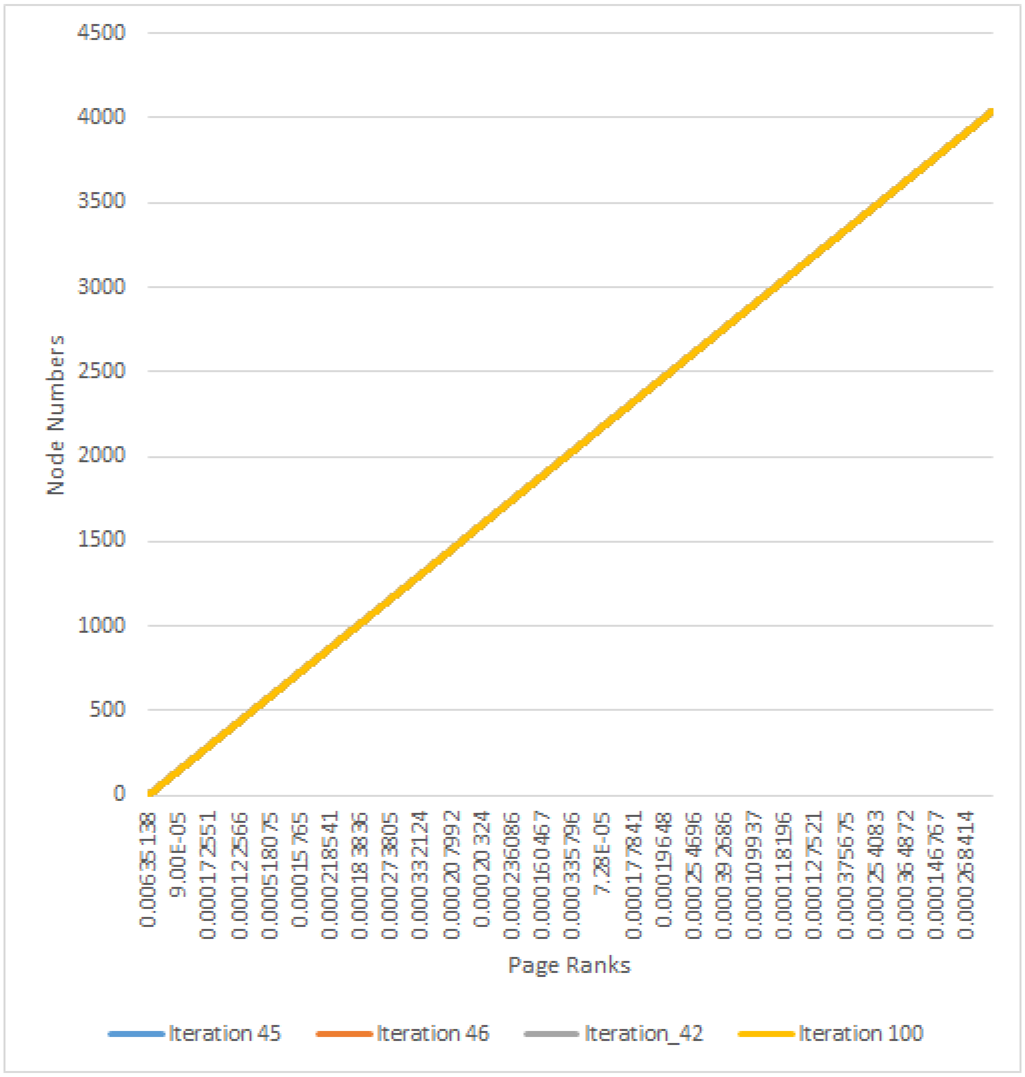
i. File reading is done by the main thread which is Thread_0.
ii. Created the adjacency matrix of size of the number of nodes present in the facebook_combined.txt.
iii. Read the file and assigned the values to the adjacency matrix if an edge exists between two nodes; considering the graph to be an undirected graph.
iv. Assigned the values of the page rank in the page rank vector represented by an array to each node as 1/N; where 'N' is total number of nodes.
v. Then we calculate page rank for each page by applying page rank formula iteratively.
vi. We run it for 'x' iterations in order to find out the point of convergence.

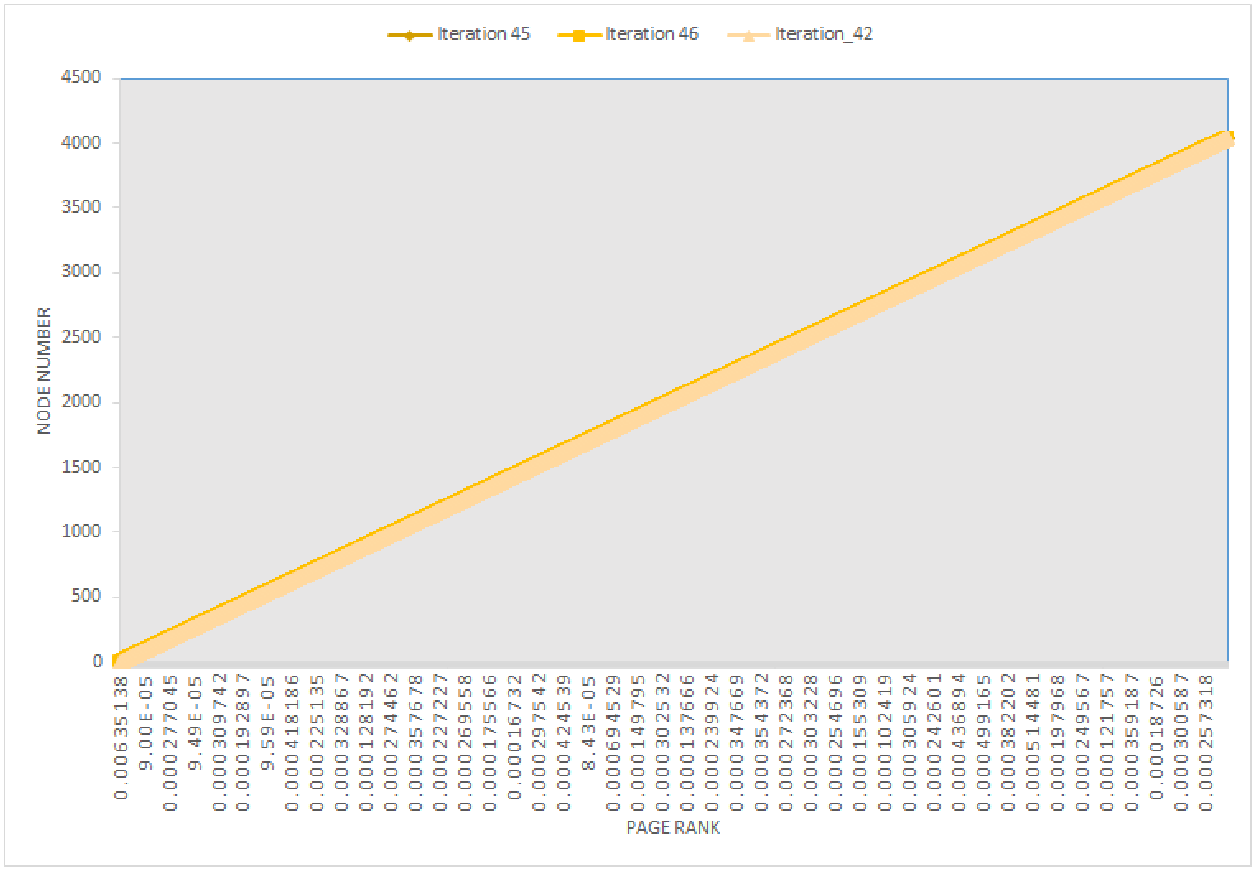
Compiling instructions:

i. export OMP_NUM_THREADS= <x> where x can be set by the user.
ii. g++ EgoFacebook.cpp –fopenmp
iii. ./a.out
The results would be stored in the file "Output_task1.txt".

The code is executed for 100 iterations.

Performance Results: After certain number of iterations the page rank value of each node starts being constant and doesn't change any more; we call this as a point of convergence. After the 42nd iteration the page ranks are constant and they sum to 1 in total. Following graph shows the values of page ranks for iteration 42 till iteration 47. The value after 42nd iteration remains constant for rest of the iterations, hence 42nd iteration is the convergence point.





Iteration 45
Iteration 46
Iteration_42
NODE NUMBER
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0.00635138
9.00E-05
0.000277045
9.49E-05
0.000309742
0.000192897
9.59E-05
0.000418186
0.000225135
0.000328867
0.000128192
0.000274462
0.000357678
0.000227227
0.000269558
0.000175566
0.00016732
0.000297542
0.000424539
8.43E-05
0.000694529
0.000149795
0.000302532
0.000137666
0.000239924
0.000347669
0.000354372
0.000272368
0.000303228
0.000254696
0.000155309
0.000102419
0.000305924
0.000242601
0.000436894
0.000499165
0.000382202
0.000514481
0.000197968
0.000249567
0.000121757
0.000359187
0.00018726
0.000300587
0.000257318
PAGE RANK

Task 2: Parallel reducer using MPI.

Implementation:

i. We start the MPI block by MPI_init.
ii. If the process-id is 0(Master process) then we read the file.
iii. The input is split equally amongst all the processes and if any input remains then it's taken process 0 itself.
iv. Each process sorts its chunk of key value pairs locally. If there exists any duplicate keys then the values are added to and only single copy of key is maintained for each processor.
v. We send the result to process 0.
vi. Process 0 again sorts all the received key value pairs, combines the duplicate keys and writes the key, value to the file.
vii. We exit the MPI block with MPI_finalize.

Compiling instructions:

i. mpicc keyvalue.c
ii. mpirun -np 10 ./a.out 100000_key-value_pairs.csv

Performance results: The parallel reducer works much faster as compared to the serial processing. As the reduction task is scattered amongst various processors the reduction performs faster.

3. Conclusion

During the execution of above tasks I found that parallel processing is much faster and better use of resources available as compared to the serial processing.