Assignment 3 - Rooting Trees with Apache Spark
Sanyam Gupta (sanyamgu)
26/11/2021

Overview
Implementation of an Apache spark program to find roots of connected components in a forest. Program has been created in Python by leveraging the Pyspark library. The performance of the program has been analysed on CCR's Planax partition. The stdout file generated for the longest job has been reported.

Algorithm
The input is edge pair (u v) where v is a parent of u. This, in combination with an inverse RDD, are used to perform a join. The resulting K, V pairs are parsed to perform an update. At each step a node is updated with the value of its parent's parent. This proceeds until all nodes point to root. Since root nodes point to themselves, this is a simple match of total nodes with nodes which have root as their parent.

```python
#!/usr/bin/python
import sys
import time
import pyspark

def makeKey(line):
    temp = line.split(" ")
    return (temp[0], temp[1])

start_time = time.time()
input_file = sys.argv[1]
out_dir = sys.argv[2]

sc = pyspark.SparkContext(appName='local')

transformed = sc.textFile(input_file).map(makeKey)
inverted = transformed.map(lambda ele: (ele[1], ele[0]))
num_elements = transformed.count()
roots = transformed.filter(lambda val: val[0] == val[1])
while (num_elements != inverted.join(roots).count()):
    transformed = inverted.join(transformed).map(lambda pair: (pair[1][0], pair[1][1]))
    inverted = transformed.map(lambda ele: (ele[1], ele[0]))

transformed.saveAsTextFile(out_dir)
print("{:.2f}".format(time.time() - start_time))
sc.stop()
```

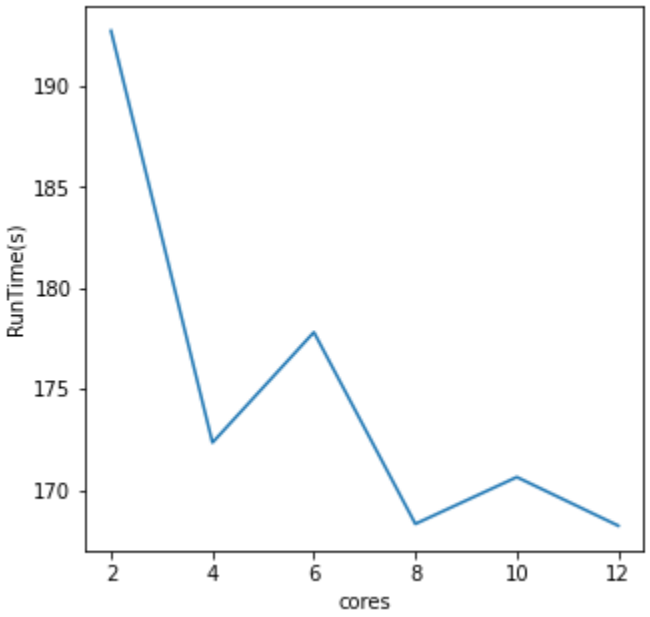
Ascertaining Performance

Performance stats have been gathered from the following Job IDs - 8636766, 8636777, 8636779, 8636784, 8636796, 8637005, 8637047, 8637061, 8637062, 8637063, 8637064, 8637065, 8637071, 8637072, 8637077, 8637080, 8637081.

Average running times

The depicted times have been averaged over 3 runs.

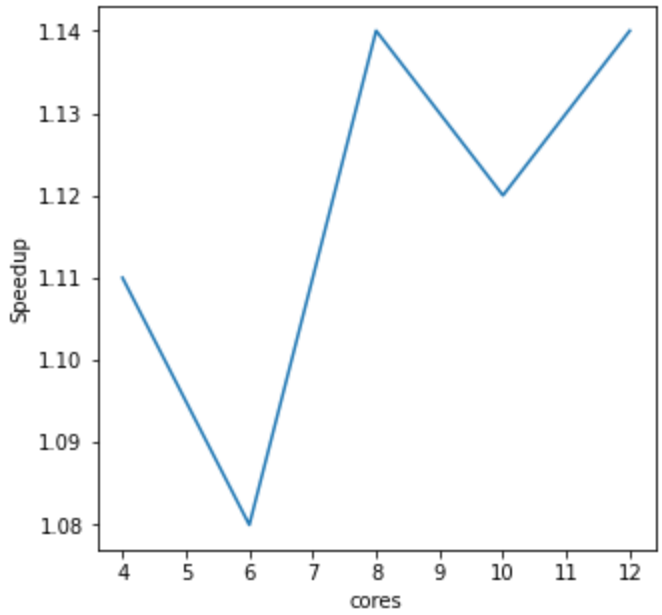
Nodes	1	2	3	4	5	6
Executors	2 (base)	4	6	8	10	12
time (s)	192.75	172.34	177.81	168.31	170.63	168.22



Speedup

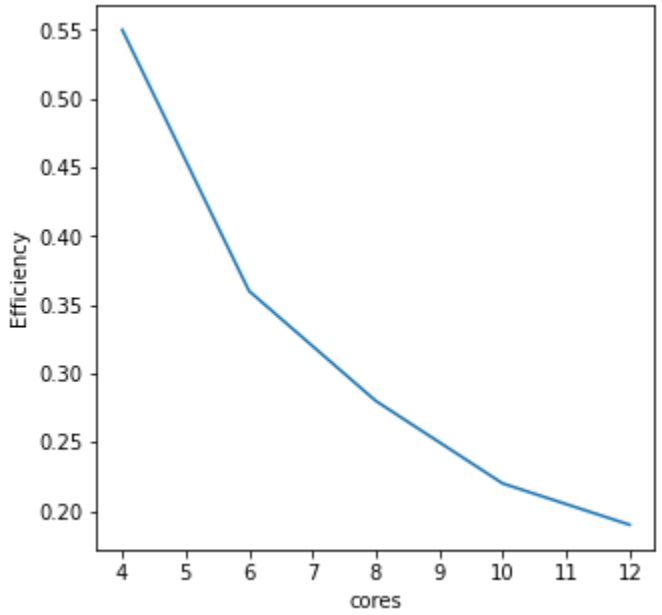
T over 2 processors are taken as a base case.

Nodes	1	2	3	4	5	6
Executors	2 (base)	4	6	8	10	12
time (s)	1	1.11	1.08	1.14	1.12	1.14



Efficiency

Nodes	1	2	3	4	5	6
Executers	2 (base)	4	6	8	10	12
time (s)	1	0.55	0.36	0.28	0.22	0.19



Strong Scaling

The runtime decreases as processors increase. However, efficiency is not preserved. A clear reason is communication overheads. Since the volume of test data is not large, processors are not being used to the fullest. Albeit, data shows the algorithm is slightly strongly scalable.

Further Improvement

The algorithm is not the most effective due to large all to all and shuffle operations. These can be handled to improve performance.