Cheat Sheet

Contents

[Homework 3](#_Toc185777996)

[1: Speed up on multicore Processors 3](#_Toc185777997)

[Assumptions 3](#_Toc185777998)

[Task 1: Calculate performance on a dual-core chip 3](#_Toc185777999)

[Task 2: Calculate performance on a quad-core chip 4](#_Toc185778000)

[TA Response 4](#_Toc185778001)

[2: OpenMP Programming 5](#_Toc185778002)

[3: OpenMP Programming 8](#_Toc185778003)

[Main.py 8](#_Toc185778004)

[My Reduce function 10](#_Toc185778005)

[My Reduce-Sequential function 11](#_Toc185778006)

[4: Google File System 12](#_Toc185778007)

[2: Cluster calculations 13](#_Toc185778008)

[5: MapReduce 15](#_Toc185778009)

[OutDegree.java 15](#_Toc185778010)

[WordCount.java 16](#_Toc185778011)

[6: Spark Programming 17](#_Toc185778012)

[PageRank: 17](#_Toc185778013)

[WordCount: 18](#_Toc185778014)

[7: Worse is better 19](#_Toc185778015)

[What I learned 19](#_Toc185778016)

[8: Graph Partitioning 20](#_Toc185778017)

[GraphPartitioner.py: 20](#_Toc185778018)

[9: Graph algorithms with GridGraph 25](#_Toc185778019)

[Kcores.cpp: 25](#_Toc185778020)

[PageRankDelta.cpp: 26](#_Toc185778021)

[10: Graph Mining 28](#_Toc185778022)

[FSM.py 28](#_Toc185778023)

[11: Spark Streaming Top-K 30](#_Toc185778024)

[Top-k.py: 30](#_Toc185778025)

[12: Halide Scheduling 32](#_Toc185778026)

[Dialated-conv.cpp: 32](#_Toc185778027)

# Homework

## 1: Speed up on multicore Processors

### Assumptions

* 90% of a given program can be perfectly parrallelized, the other 10% remains sequential
* We have a fixed power budget
* Total power is proportional to the square of frequency
* Performance is proportional to frequency
* The programs performance on a single-core chip is 1 solutions per seconds

### Task 1: Calculate performance on a dual-core chip

On a single-core system, with the frequency , has the power usage of . Since the total power is fixed, the two dual-chip cores must share this power between them. We can therefore calculate the frequency for each of the cores:

Assuming that on the sequential sections of the program, only one core is working, we can therefore calculate the performance for the throughput for the parallelizable and sequential sections of the program

We can now calculate the total troughput, and therefore performance, of the program on the dual-core chip

### Task 2: Calculate performance on a quad-core chip

As before, we can calculate the frequency for each of the four cores:

Again, we calculate the performance for the throughput for the parallelizable and sequential sections of the program

We now calculate the total throughput

### TA Response

Wrong math on speed averaging. Overall performance does not equal to the sum of the performance times the proportion of each part.

## 2: OpenMP Programming

#include <getopt.h>

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#include <string>

#include <iostream>

#include <sstream>

#include <vector>

#include "./common/CycleTimer.h"

#include "./common/grade.h"

#include "./common/graph.h"

#include "page\_rank.h"

#define USE\_BINARY\_GRAPH 1

#define PageRankDampening 0.3f

#define PageRankConvergence 1e-7d

// used for check correctness

void reference\_serial\_pageRank(Graph g, double \*solution, double damping,

double convergence) {

int numNodes = num\_nodes(g);

double equal\_prob = 1.0 / numNodes;

double \*solution\_new = new double[numNodes];

double \*score\_old = solution;

double \*score\_new = solution\_new;

bool converged = false;

double broadcastScore = 0.0;

double globalDiff = 0.0;

int iter = 0;

for (int i = 0; i < numNodes; ++i) {

solution[i] = equal\_prob;

}

while (!converged && iter < MAXITER) {

iter++;

broadcastScore = 0.0;

globalDiff = 0.0;

for (int i = 0; i < numNodes; ++i) {

score\_new[i] = 0.0;

if (outgoing\_size(g, i) == 0) {

broadcastScore += score\_old[i];

}

const Vertex \*in\_begin = incoming\_begin(g, i);

const Vertex \*in\_end = incoming\_end(g, i);

for (const Vertex \*v = in\_begin; v < in\_end; ++v) {

score\_new[i] += score\_old[\*v] / outgoing\_size(g, \*v);

}

score\_new[i] =

damping \* score\_new[i] + (1.0 - damping) \* equal\_prob;

}

for (int i = 0; i < numNodes; ++i) {

score\_new[i] += damping \* broadcastScore \* equal\_prob;

globalDiff += std::abs(score\_new[i] - score\_old[i]);

}

converged = (globalDiff < convergence);

std::swap(score\_new, score\_old);

}

if (score\_new != solution) {

memcpy(solution, score\_new, sizeof(double) \* numNodes);

}

delete[] solution\_new;

}

int main(int argc, char \*\*argv) {

int num\_threads = -1;

std::string graph\_filename;

if (argc < 3) {

std::cerr << "Usage: <path/to/graph/file> <manual\_set\_thread\_count>\n";

exit(1);

}

int thread\_count = -1;

if (argc == 3) {

thread\_count = atoi(argv[2]);

}

if (thread\_count <= 0) {

std::cerr << "<manual\_set\_thread\_count> must > 0\n";

exit(1);

}

graph\_filename = argv[1];

Graph g;

printf("----------------------------------------------------------\n");

printf("Running with %d threads\n", thread\_count);

printf("----------------------------------------------------------\n");

printf("Loading graph...\n");

if (USE\_BINARY\_GRAPH) {

g = load\_graph\_binary(graph\_filename.c\_str());

} else {

g = load\_graph(argv[1]);

printf("storing binary form of graph!\n");

store\_graph\_binary(graph\_filename.append(".bin").c\_str(), g);

delete g;

exit(1);

}

printf("\n");

printf("Graph stats:\n");

printf(" Filename: %s\n", argv[1]);

printf(" Edges: %d\n", g->num\_edges);

printf(" Nodes: %d\n", g->num\_nodes);

bool pr\_check = true;

double \*sol1;

sol1 = (double \*)malloc(sizeof(double) \* g->num\_nodes);

double \*sol2;

sol2 = (double \*)malloc(sizeof(double) \* g->num\_nodes);

double pagerank\_base;

double pagerank\_time;

double ref\_pagerank\_base;

double ref\_pagerank\_time;

double start;

std::stringstream timing;

std::stringstream ref\_timing;

timing << "Threads Page Rank\n";

ref\_timing << "Serial Reference Page Rank\n";

// Set thread count

omp\_set\_num\_threads(thread\_count);

// Run implementations

start = CycleTimer::currentSeconds();

pageRank(g, sol1, PageRankDampening, PageRankConvergence);

pagerank\_time = CycleTimer::currentSeconds() - start;

// Run reference implementation

start = CycleTimer::currentSeconds();

reference\_serial\_pageRank(g, sol2, PageRankDampening, PageRankConvergence);

ref\_pagerank\_time = CycleTimer::currentSeconds() - start;

printf("----------------------------------------------------------\n");

std::cout << "Testing Correctness of Page Rank\n";

if (!compareApprox(g, sol2, sol1)) {

pr\_check = false;

}

if (!pr\_check)

std::cout << "Your Page Rank is not Correct" << std::endl;

else

std::cout << "Your Page Rank is Correct" << std::endl;

char buf[1024];

char ref\_buf[1024];

sprintf(buf, "%4d: %.6f s\n", thread\_count, pagerank\_time);

sprintf(ref\_buf, " 1: %.6f s\n", ref\_pagerank\_time);

timing << buf;

ref\_timing << ref\_buf;

printf("----------------------------------------------------------\n");

std::cout << "Serial Reference Summary" << std::endl;

std::cout << ref\_timing.str();

printf("----------------------------------------------------------\n");

std::cout << "Timing Summary" << std::endl;

std::cout << timing.str();

printf("----------------------------------------------------------\n");

delete g;

return 0;

}

## 3: OpenMP Programming

### Main.py

#include "your\_reduce.h"

#include <cassert>

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <ctime>

#include <random>

#include <sys/time.h>

#define MAX\_LEN 268435456

// return the time in the unit of us

static long get\_time\_us() {

struct timeval my\_time;

gettimeofday(&my\_time, NULL);

long runtime\_us = 1000000 \* my\_time.tv\_sec + my\_time.tv\_usec;

return runtime\_us;

}

int main(int argc, char \*argv[]) {

int size, rank, provided;

MPI\_Init\_thread(&argc, &argv, MPI\_THREAD\_MULTIPLE,

&provided); // enable multi-thread support (for Bonus)

assert(provided == MPI\_THREAD\_MULTIPLE);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

int \*a, \*b; // the array used to do the reduction

int \*res; // the array to record the result of YOUR\_Reduce

int \*res2; // the array to record the result of MPI\_Reduce

long count;

long begin\_time, end\_time, use\_time,

use\_time2; // use\_time for YOUR\_Reduce & use\_time2 for MPI\_Reduce

int i;

// initialize

a = (int \*)malloc(MAX\_LEN \* sizeof(int));

b = (int \*)malloc(MAX\_LEN \* sizeof(int));

res = (int \*)malloc(MAX\_LEN \* sizeof(int));

res2 = (int \*)malloc(MAX\_LEN \* sizeof(int));

memset(a, 0, MAX\_LEN \* sizeof(int));

memset(b, 0, MAX\_LEN \* sizeof(int));

memset(res, 0, MAX\_LEN \* sizeof(int));

memset(res2, 0, MAX\_LEN \* sizeof(int));

std::mt19937 rng;

rng.seed(time(NULL)); // seed to generate the array randomly

for (count = 1; count <= MAX\_LEN;

count \*= 16) // length of array : [ 1 16 256 4'096 65'536 1'048'576

// 16'777'216 268'435'456 ]

// do not report results for length 1

{

// the element of array is generated randomly

for (i = 0; i < count; i++) {

b[i] = a[i] = rng() % MAX\_LEN;

}

// MPI\_Reduce and then print the usetime, the result will be put in

// res2[]

MPI\_Barrier(MPI\_COMM\_WORLD);

begin\_time = get\_time\_us();

MPI\_Reduce(a, res2, count, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

MPI\_Barrier(MPI\_COMM\_WORLD);

end\_time = get\_time\_us();

use\_time2 = end\_time - begin\_time;

if (rank == 0)

printf("%ld int use\_time : %ld us [MPI\_Reduce]\n", count,

use\_time2),

fflush(stdout);

// YOUR\_Reduce and then print the usetime, the result should be put in

// res[]

MPI\_Barrier(MPI\_COMM\_WORLD);

begin\_time = get\_time\_us();

YOUR\_Reduce(b, res, count);

MPI\_Barrier(MPI\_COMM\_WORLD);

end\_time = get\_time\_us();

use\_time = end\_time - begin\_time;

if (rank == 0)

printf("%ld int use\_time : %ld us [YOUR\_Reduce]\n", count,

use\_time),

fflush(stdout);

// check the result of MPI\_Reduce and YOUR\_Reduce

if (rank == 0) {

int correctness = 1;

for (i = 0; i < count; i++) {

if (res2[i] != res[i]) {

correctness = 0;

}

}

if (correctness == 0)

printf("WRONG !!!\n"), fflush(stdout);

else

printf("CORRECT !\n"), fflush(stdout);

}

}

MPI\_Finalize();

return 0;

}

### My Reduce function

#include <mpi.h>

#include <cstring>

#include <stdio.h>

void YOUR\_Reduce(const int \*sendbuf, int \*recvbuf, int count) {

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

// Initialize recvbuf with the values from sendbuf

memcpy(recvbuf, sendbuf, count \* sizeof(int));

// Allocate temp\_buffer once

int\* temp\_buffer = new int[count];

// Binary tree reduction

for (int step = 1; step < size; step \*= 2) {

if (rank % (2 \* step) == 0) {

// Root process collects data

if (rank + step < size) {

MPI\_Recv(temp\_buffer, count, MPI\_INT, rank + step, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

// Combine results using a loop

for (int i = 0; i < count; i++) {

recvbuf[i] += temp\_buffer[i];

}

}

} else if (rank % step == 0) {

// Send to parent

MPI\_Send(recvbuf, count, MPI\_INT, rank - step, 0, MPI\_COMM\_WORLD);

break;

}

}

// Clean up

delete[] temp\_buffer;

}

### My Reduce-Sequential function

#include <mpi.h>

#include <cstring>

#include <stdio.h>

#include <omp.h> // Include OpenMP header

void YOUR\_Reduce(const int \*sendbuf, int \*recvbuf, int count) {

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

// Initialize recvbuf with the values from sendbuf

memcpy(recvbuf, sendbuf, count \* sizeof(int));

// Allocate temp\_buffer once

int\* temp\_buffer = new int[count];

// Binary tree reduction

for (int step = 1; step < size; step \*= 2) {

if (rank % (2 \* step) == 0) {

// Root process collects data

if (rank + step < size) {

MPI\_Request request;

// Start non-blocking receive

MPI\_Irecv(temp\_buffer, count, MPI\_INT, rank + step, 0, MPI\_COMM\_WORLD, &request);

// Wait for the receive to complete

MPI\_Wait(&request, MPI\_STATUS\_IGNORE);

// Combine results in parallel

#pragma omp parallel for

for (int i = 0; i < count; i++) {

recvbuf[i] += temp\_buffer[i];

}

}

} else if (rank % step == 0) {

// Send to parent using non-blocking send

MPI\_Request request;

MPI\_Isend(recvbuf, count, MPI\_INT, rank - step, 0, MPI\_COMM\_WORLD, &request);

break;

}

}

// Clean up

delete[] temp\_buffer;

}

## 4: Google File System

#### 1: GFS Questions

#### 1.1: How does the master node get the locations of each chunks at startup?

At startup, the master node does not store the chunk location information persistently. Instead, it retrieves the location of each chunk by polling all chunkservers. Each chunkserver reports the chunks it holds, and the master node updates its information accordingly. Additionally, whenever a chunkserver joins the cluster, the master node updates the chunk locations. The master keeps this information updated by sending periodic HeartBeat messages, making sure it is updated on chunkplacement and the chunkserver status. This approach ensures that the master always has up-to-date information about chunk locations in the system.

#### 1.2: What is the benefit of this approach comparing with the approach that the master persists this information?

The main benefit of this approach is the reduction in complexity related to maintaining consistency between the master and the chunkservers. Persisting chunk location information would require the system to handle various events such as chunkserver failures, renaming, and rejoining, which can lead to stale or inconsistent information. By polling the chunkservers at startup and using regular HeartBeat messages, the master node avoids the issues of synchronization and ensures accurate, up-to-date chunk location information at all times. This also simplifies the system’s design, making it more robust against common failures in a distributed environment.

### 2: Cluster calculations

We assume a cluster of 1000 servers, each server having 10 disks with 10TB storage capacity and 100MB/s I/O bandwidth per disk. The servers are connected by a 1Gbps (125MBps) ethernet cables, as nothing else is written I assume this bandwidth is per machine and not the total transfer cap in the system.

#### 2.1: What is the minimum time required to recovery a node failure

The total I/O bandwidth of all disks in a node is:

*Total bandwidth = 10disks ×100MB/s = 1000MB/s*

The network bandwidth available per node is:

*Network bandwidth = 1Gbps = 1024Mbps = 128MB/s*

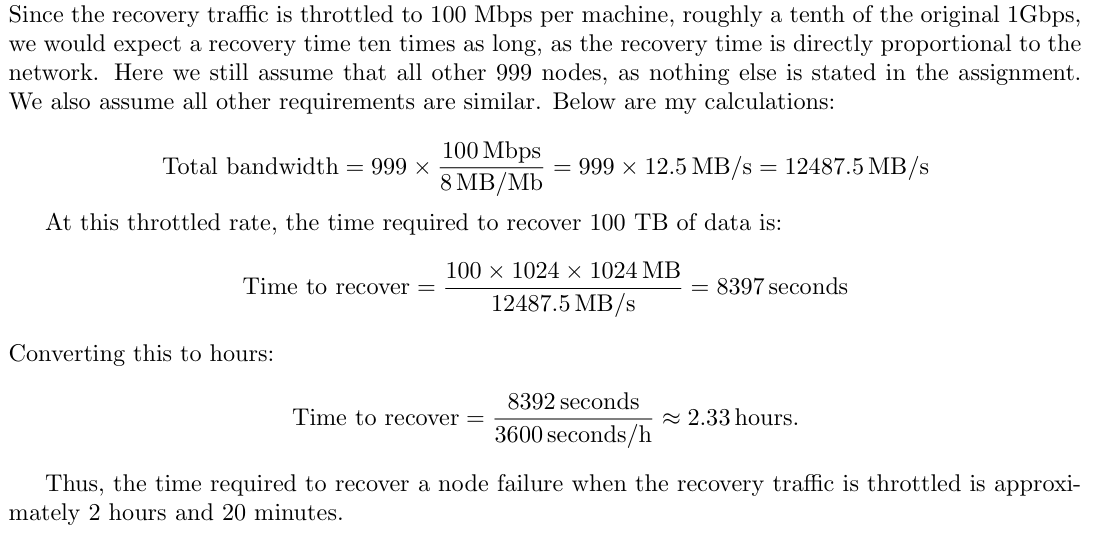
Since the network bandwidth is lower than the total I/O bandwidth of the disks, the recovery time will be limited by the network speed. Furthermore, we assume that since we are calculating the minimum time requred, that all other 999 nodes will use their resources to assist upon a node failure. Assuming that the 999 remaining servers work in perfect parallel to recover the data from the failed node, the total network bandwidth available is

*Total bandwidth = 999 ×128MB/s = 127872MB/s*

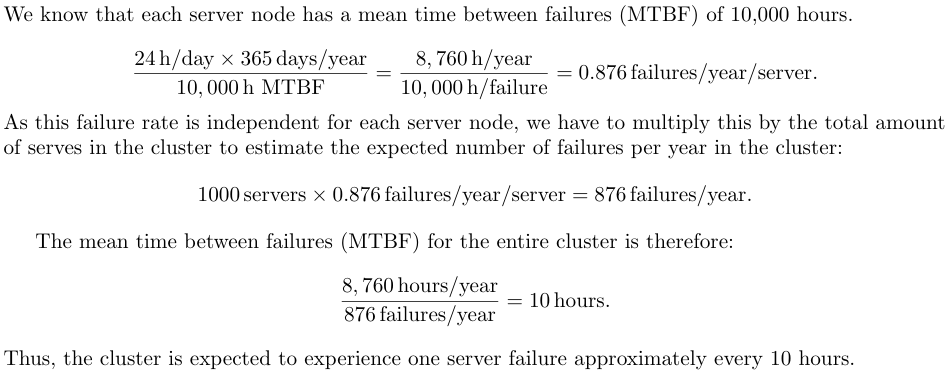
To transfer 100 TB of data at this rate, the time required is:

Thus, the minimum time required to recover a node failure, assuming all other servers participate in the recovery, is approximately 14 minutes.

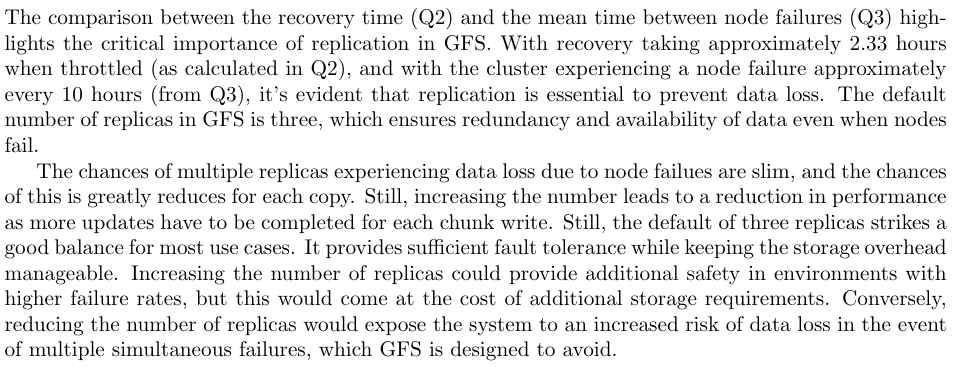
#### 2.2: Time to recover a failure node with throttled recovery bandwidth



#### Q3: How many server failures are likely to occur in a year in this cluster? What is the mean time between node failures in this cluster?



#### Q4: What is the implication of the number of replicas used in GFS based on the results from Q2 and Q3?



## 5: MapReduce

### OutDegree.java

import java.io.IOException;

import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.GenericOptionsParser;

public class OutDegree {

public static class OutDegreeMapper

extends Mapper<Object, Text, Text, IntWritable> {

private final static IntWritable one = new IntWritable(1);

private Text node = new Text();

public void map(Object key, Text value, Context context

) throws IOException, InterruptedException {

StringTokenizer itr = new StringTokenizer(value.toString());

itr.nextToken(); // Skip the first token ("a")

if (itr.hasMoreTokens()) {

String sourceNode = itr.nextToken(); // The source node (e.g., "a")

node.set(sourceNode);

// Emit the source node with a count of 1 for each outgoing edge

context.write(node, one);

}

}

}

public static class OutDegreeReducer

extends Reducer<Text, IntWritable, Text, IntWritable> {

private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values,

Context context

) throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get(); // Sum up all counts for each node

}

result.set(sum);

context.write(key, result); // Emit the node and its total out-degree

}

}

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();

if (otherArgs.length < 2) {

System.err.println("Usage: outdegree <in> <out>");

System.exit(2);

}

Job job = new Job(conf, "outdegree");

job.setJarByClass(OutDegree.class);

job.setMapperClass(OutDegreeMapper.class);

job.setCombinerClass(OutDegreeReducer.class);

job.setReducerClass(OutDegreeReducer.class);

job.setNumReduceTasks(1);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

for (int i = 0; i < otherArgs.length - 1; ++i) {

FileInputFormat.addInputPath(job, new Path(otherArgs[i]));

}

FileOutputFormat.setOutputPath(job, new Path(otherArgs[otherArgs.length - 1]));

System.exit(job.waitForCompletion(true) ? 0 : 1); }}

### WordCount.java

import java.io.IOException;

import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {

private final static IntWritable one = new IntWritable(1);

private Text word = new Text();

public void map(Object key, Text value, Context context) throws IOException, InterruptedException {

StringTokenizer itr = new StringTokenizer(value.toString());

while (itr.hasMoreTokens()) {

word.set(itr.nextToken());

context.write(word, one);

}

}

}

public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values, Context context)

throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get();

}

result.set(sum);

context.write(key, result);

}

}

public static int run(String input, String output) throws Exception {

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "word count");

job.setJarByClass(WordCount.class);

job.setMapperClass(TokenizerMapper.class);

job.setCombinerClass(IntSumReducer.class);

job.setReducerClass(IntSumReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job, new Path(input));

FileOutputFormat.setOutputPath(job, new Path(output));

return job.waitForCompletion(true) ? 0 : -1;

}

public static void main(String[] args) throws Exception {

if (args.length != 2) {

System.err.println("Usage: WordCount <in> <frontiers> <out>");

System.exit(2);

}

if (run(args[0], args[1]) < 0) {

System.exit(1);

}

}

}

## 6: Spark Programming

### PageRank:

import sys

from pyspark import SparkConf, SparkContext

import time

if \_\_name\_\_ == '\_\_main\_\_':

conf = SparkConf()

sc = SparkContext(conf=conf)

sc.setLogLevel("ERROR") # Added to avoid Warnings cluttering the terminal

file\_path = sys.argv[1]

lines = sc.textFile(file\_path)

# Parameters

damping = 0.8

num\_iterations = 50

top\_i\_nodes = 5

dynamic = sys.argv[2]

first = time.time()

# Parse the lines into (source, destination) pairs and remove duplicates

edges = lines.map(lambda line: tuple(map(int, line.split()))).distinct()

# We estimate the amount of nodes

if(dynamic):

# This method takes aprox. 0.5s longer but is dynamic

nodes = edges.flatMap(lambda edge: edge).distinct()

n = nodes.count()

else:

# This method is faster but not dynamic

n = 100 if "small" in file\_path else 1000 if "full" in file\_path else None

# Create an adjacency list as (node, [neighbors])

adj\_list = edges.groupByKey().mapValues(list).cache()

# Initialize each node's PageRank value

page\_ranks = adj\_list.mapValues(lambda \_: 1.0 / n)

for i in range(num\_iterations):

# Broadcast the adjacency list for efficient access

adjacency\_broadcast = sc.broadcast(adj\_list.collectAsMap())

# Compute contributions for each node's neighbors

contributions = page\_ranks.flatMap(lambda node\_rank: [

(neighbor, node\_rank[1] / len(adjacency\_broadcast.value.get(node\_rank[0], [])))

for neighbor in adjacency\_broadcast.value.get(node\_rank[0], [])

])

# Aggregate contributions and calculate new PageRank values

page\_ranks = contributions.reduceByKey(lambda a, b: a + b).mapValues(

lambda rank: (1 - damping) / n + damping \* rank

)

# Get the top 5 nodes with the highest PageRank scores

highest = page\_ranks.takeOrdered(top\_i\_nodes, key=lambda x: -x[1])

# Print the top 5 nodes

print("Top 5 nodes with highest PageRank scores:")

for node, score in highest:

print(f"Node {node}: {score}")

last = time.time()

print("Total program time: %.2f seconds" % (last - first))

sc.stop()

### WordCount:

import re

import sys

from pyspark import SparkConf, SparkContext

import time

if \_\_name\_\_ == '\_\_main\_\_':

conf = SparkConf()

sc = SparkContext(conf=conf)

sc.setLogLevel("ERROR") # Added to avoid Warnings cluttering the terminal

lines = sc.textFile(sys.argv[1])

top\_i\_words = 10

first = time.time()

# We split the lines into words

words = lines.flatMap(lambda line: re.split(r'[^\w]+', line))

# We now count every word

word\_counts = words.countByValue()

top\_words = sorted(word\_counts.items(), key=lambda x: -x[1])[:top\_i\_words]

# Print the top i words

print(f"Top {top\_i\_words} words:")

for word, count in top\_words:

print(f"({repr(word)}, {count})")

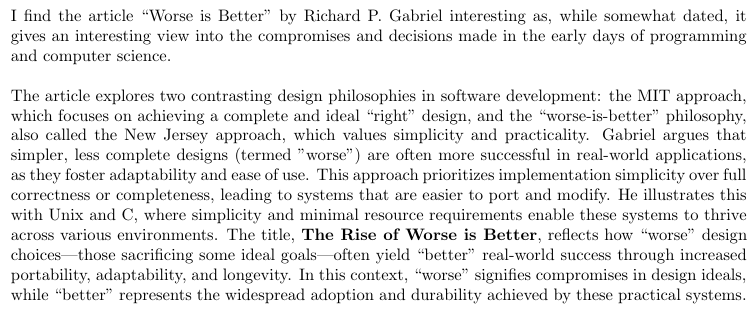
last = time.time()

print("Total program time: %.2f seconds" % (last - first))

sc.stop()

## 7: Worse is better

### What I learned



## 8: Graph Partitioning

### GraphPartitioner.py:

import sys

import struct

import random

from collections import defaultdict

import matplotlib.pyplot as plt

import networkx as nx

class GraphPartitioner:

def \_\_init\_\_(self, file\_path, num\_partitions, show\_details):

self.file\_path = file\_path

self.num\_partitions = num\_partitions

self.show\_details = show\_details

self.edges = self.read\_graph()

self.directed = self.isDirected()

def isDirected(self):

# Some of the graphs are undirected

return not any(file\_name in file\_path for file\_name in ["small-5", "synthesized-1b"])

# A (Slightly modified) replication of the C code already existing

# Reads the data and saves all edges in the graph

def read\_graph(self):

edges = []

with open(self.file\_path, "rb") as file:

data = file.read(8) # Read first 8 bytes

while data:

src, dst = struct.unpack("ii", data) # (4 byte src, 4 byte dst)

edges.append((src, dst))

data = file.read(8) # Read the next 8 bytes

print("Done reading edges\n")

return edges

# Show graphs in subplots for each partition

def show\_graph(self, partitions, title="Graph Partitions"):

height = 4

width = len(partitions) \* height

fig, axes = plt.subplots(1, len(partitions), figsize=(width, height))

if len(partitions) == 1:

axes = [axes] # Ensure axes is always iterable for a single partition

for i, (partition\_id, partition) in enumerate(sorted(partitions.items())):

ax = axes[i]

G = nx.DiGraph() if self.directed else nx.Graph()

G.add\_edges\_from(partition['edges'])

# Differentiate master vertices by color, or color all nodes lightgreen if none exist

master\_vertices = partition.get('master\_vertices', set())

# If there is no master\_vertices value, then we are on the Full graph and all are master vertices

if not master\_vertices:

node\_colors = ["lightgreen" for \_ in G.nodes()]

# If only some are master vertices, color them lightgreen and the rest skyblue

else:

node\_colors = ["lightgreen" if node in master\_vertices else "skyblue" for node in G.nodes()]

pos = nx.spring\_layout(G) # Use spring layout for better visualization

nx.draw(G, pos, with\_labels=True, node\_color=node\_colors, font\_weight="bold",

node\_size=500, edge\_color="gray", ax=ax, arrows=self.directed)

ax.set\_title(f"Partition {partition\_id}")

fig.suptitle(title, fontsize=16)

plt.tight\_layout(rect=[0, 0, 1, 0.95])

plt.show()

# Balanced p-way edge-cut partitioning

def edge\_cut\_partition(self):

partitions = defaultdict(

lambda: {

'master\_vertices': set(),

'total\_vertices': set(),

'replicated\_edges': 0,

'edges': []

}

)

vertex\_to\_partition = {}

for src, dst in self.edges:

# Hash vertices to assign them to partitions

src\_partition = vertex\_to\_partition.get(src, hash(src) % self.num\_partitions)

dst\_partition = vertex\_to\_partition.get(dst, hash(dst) % self.num\_partitions)

# Assign the vertex to the partition

vertex\_to\_partition[src] = src\_partition

vertex\_to\_partition[dst] = dst\_partition

# Assign edges and count replicated edges

if src\_partition != dst\_partition:

partitions[src\_partition]['replicated\_edges'] += 1

partitions[dst\_partition]['replicated\_edges'] += 1

partitions[src\_partition]['edges'].append((src, dst))

partitions[dst\_partition]['edges'].append((src, dst))

# Update master and total vertices

partitions[src\_partition]['master\_vertices'].add(src)

partitions[dst\_partition]['master\_vertices'].add(dst)

partitions[src\_partition]['total\_vertices'].update([src, dst])

partitions[dst\_partition]['total\_vertices'].update([src, dst])

# Print partition statistics

for i, partition in sorted(partitions.items()):

print(f"Partition {i}")

print(len(partition['master\_vertices']))

print(len(partition['total\_vertices']))

print(partition['replicated\_edges'])

print(len(partition['edges']))

# Show the partitions side by side

if self.show\_details:

self.show\_graph(partitions, title="Edge-Cut Partitioning")

# Balanced p-way vertex-cut partitioning

def vertex\_cut\_partition(self):

partitions = defaultdict(

lambda: {

'master\_vertices': set(),

'total\_vertices': set(),

'edges': []

}

)

vertex\_partitions = defaultdict(set) # Store all partitions a vertex is assigned to

for src, dst in self.edges:

# Hash the edge to assign it to a partition

edge\_partition = hash((src, dst)) % self.num\_partitions

partitions[edge\_partition]['edges'].append((src, dst))

# Update vertex partitions

vertex\_partitions[src].add(edge\_partition)

vertex\_partitions[dst].add(edge\_partition)

# Calculate master and total vertices for each partition

for vertex, assigned\_partitions in vertex\_partitions.items():

# Choose one partition as master

master\_partition = random.choice(list(assigned\_partitions))

for partition\_id in assigned\_partitions:

partitions[partition\_id]['total\_vertices'].add(vertex)

if partition\_id == master\_partition:

partitions[partition\_id]['master\_vertices'].add(vertex)

# Print partition statistics

for i, partition in sorted(partitions.items()):

print(f"Partition {i}")

print(f"{len(partition['master\_vertices'])}")

print(f"{len(partition['total\_vertices'])}")

print(f"{len(partition['edges'])}")

# Show the partitions side by side

if self.show\_details:

self.show\_graph(partitions, title="Vertex-Cut Partitioning")

# Greedy heuristic vertex-cut partitioning

def greedy\_heuristic\_partition(self):

partitions = defaultdict(

lambda: {

'master\_vertices': set(),

'total\_vertices': set(),

'edges': []

}

)

vertex\_degrees = defaultdict(int)

# Calculate degrees for all vertices

for src, dst in self.edges:

vertex\_degrees[src] += 1

vertex\_degrees[dst] += 1

# Assign vertices to partitions using greedy heuristic

for src, dst in self.edges:

src\_partition = vertex\_degrees[src] % self.num\_partitions

dst\_partition = vertex\_degrees[dst] % self.num\_partitions

partitions[src\_partition]['edges'].append((src, dst))

partitions[dst\_partition]['edges'].append((src, dst))

partitions[src\_partition]['total\_vertices'].add(src)

partitions[dst\_partition]['total\_vertices'].add(dst)

master\_partition = random.choice([src\_partition, dst\_partition])

if master\_partition == src\_partition:

partitions[src\_partition]['master\_vertices'].add(src)

else:

partitions[dst\_partition]['master\_vertices'].add(dst)

# Ensure every partition gets at least one vertex

for i in range(self.num\_partitions):

if not partitions[i]['total\_vertices']:

random\_vertex = random.choice(list(vertex\_degrees.keys()))

partitions[i]['total\_vertices'].add(random\_vertex)

partitions[i]['master\_vertices'].add(random\_vertex)

# Print partition statistics

for i, partition in sorted(partitions.items()):

print(f"Partition {i}")

print(len(partition['master\_vertices']))

print(len(partition['total\_vertices']))

print(len(partition['edges']))

# Show the partitions side by side

if self.show\_details:

self.show\_graph(partitions, title="Greedy Vertex-Cut Partitioning")

# Helper function to get the neighbors of a vertex

def get\_neighbors(self, vertex):

neighbors = set()

for src, dst in self.edges:

if src == vertex:

neighbors.add(dst)

elif dst == vertex:

neighbors.add(src)

return neighbors

# Helper function to get partition for a vertex

def get\_partition\_for\_vertex(self, vertex, partitions):

for partition\_id, partition in partitions.items():

if vertex in partition['total\_vertices']:

return partition\_id

return None

# Balanced p-way Hybrid-Cut based on PowerLyra

def hybrid\_cut\_partition(self, theta=10):

partitions = defaultdict(

lambda: {

'master\_vertices': set(),

'total\_vertices': set(),

'edges': []

}

)

vertex\_degree = defaultdict(int) # Store vertex degrees

vertex\_partitions = defaultdict(set) # Tracks the partitions each vertex is assigned to

# Calculate degrees for all vertices

for src, dst in self.edges:

vertex\_degree[src] += 1

vertex\_degree[dst] += 1

for src, dst in self.edges:

if vertex\_degree[src] > theta or vertex\_degree[dst] > theta:

# High-degree vertices: use edge-cut strategy

src\_partition = hash(src) % self.num\_partitions

dst\_partition = hash(dst) % self.num\_partitions

partitions[src\_partition]['edges'].append((src, dst))

partitions[dst\_partition]['edges'].append((src, dst))

partitions[src\_partition]['master\_vertices'].add(src)

partitions[dst\_partition]['master\_vertices'].add(dst)

partitions[src\_partition]['total\_vertices'].update([src, dst])

partitions[dst\_partition]['total\_vertices'].update([src, dst])

else:

# Low-degree vertices: use vertex-cut (greedy heuristic)

min\_partition = None

min\_replication = float('inf')

for partition\_id in range(self.num\_partitions):

replication\_cost = (

int(src not in partitions[partition\_id]['master\_vertices']) +

int(dst not in partitions[partition\_id]['master\_vertices'])

)

if replication\_cost < min\_replication:

min\_replication = replication\_cost

min\_partition = partition\_id

partitions[min\_partition]['edges'].append((src, dst))

partitions[min\_partition]['master\_vertices'].update([src, dst])

partitions[min\_partition]['total\_vertices'].update([src, dst])

vertex\_partitions[src].add(min\_partition)

vertex\_partitions[dst].add(min\_partition)

# Print partition statistics

for i, partition in sorted(partitions.items()):

print(f"Partition {i}")

print(f"{len(partition['master\_vertices'])}")

print(f"{len(partition['total\_vertices'])}")

print(f"{len(partition['edges'])}")

if self.show\_details:

self.show\_graph(partitions, title=f"Hybrid-Cut Partitioning (Theta={theta})")

if \_\_name\_\_ == "\_\_main\_\_":

file\_path = sys.argv[1] if len(sys.argv) > 1 else "small-5.graph"

num\_partitions = int(sys.argv[2]) if len(sys.argv) > 2 else 3

show\_details = (sys.argv[3]!="False") if len(sys.argv) > 3 else False

partitioner = GraphPartitioner(file\_path, num\_partitions, show\_details)

# Display the graph if applicable

if show\_details:

partitioner.show\_graph({0: {'edges': partitioner.edges}}, title="Full Graph")

# Partition the graph

print("\nEdge-Cut Partitioning:")

partitioner.edge\_cut\_partition()

print("\nVertex-Cut Partitioning:")

partitioner.vertex\_cut\_partition()

print("\nHybrid-Cut Partitioning:")

partitioner.hybrid\_cut\_partition(theta=2)

print("\nGreedy-Cut Partitioning:")

partitioner.greedy\_heuristic\_partition()

## 9: Graph algorithms with GridGraph

### Kcores.cpp:

#include "core/graph.hpp"

int main(int argc, char \*\*argv) {

if (argc < 3) {

fprintf(stderr, "usage: kcores [path] [k] [memory budget in GB]\n");

exit(-1);

}

std::string path = argv[1];

int k = atoi(argv[2]);

long memory\_bytes = (argc >= 4) ? atol(argv[3]) \* 1024l \* 1024l \* 1024l : 8l \* 1024l \* 1024l \* 1024l;

Graph graph(path);

graph.set\_memory\_bytes(memory\_bytes);

Bitmap \*active\_in = graph.alloc\_bitmap();

Bitmap \*active\_out = graph.alloc\_bitmap();

BigVector<int> degree(graph.path + "/degree", graph.vertices);

BigVector<int> core(graph.path + "/core", graph.vertices);

long vertex\_data\_bytes = (long)graph.vertices \* (sizeof(int) + sizeof(int));

graph.set\_vertex\_data\_bytes(vertex\_data\_bytes);

// Initialize degree and active vertices

active\_out->fill();

degree.fill(0);

graph.stream\_edges<VertexId>(

[&](Edge &e) {

write\_add(&degree[e.source], 1);

return 0;

},

nullptr, 0, 0);

// Initialize core and set initial active vertices

int active\_vertices = graph.stream\_vertices<VertexId>(

[&](VertexId i) {

core[i] = (degree[i] >= k) ? 1 : 0;

return core[i];

});

printf("Initialization complete: %d active vertices\n", active\_vertices);

// K-core decomposition iterations

int iteration = 0;

while (active\_vertices > 0) {

iteration++;

printf("Iteration %d: %d active vertices\n", iteration, active\_vertices);

std::swap(active\_in, active\_out);

active\_out->clear();

graph.hint(degree, core);

active\_vertices = graph.stream\_edges<VertexId>(

[&](Edge &e) {

if (core[e.source] == 1 && core[e.target] == 0) {

write\_add(&degree[e.target], -1);

if (degree[e.target] < k) {

core[e.target] = 0;

active\_out->set\_bit(e.target);

return 1;

}

}

return 0;

},

active\_in);

}

// Count k-core vertices

int kcore\_vertices = graph.stream\_vertices<VertexId>(

[&](VertexId i) {

return core[i] == 1;

});

printf("K-core (%d-core) decomposition complete: %d vertices remain\n", k, kcore\_vertices);

return 0;

}

### PageRankDelta.cpp:

#include "core/graph.hpp"

int main(int argc, char \*\* argv) {

if (argc < 3) {

fprintf(stderr, "usage: pagerank\_delta [path] [iterations] [memory budget in GB]\n");

exit(-1);

}

std::string path = argv[1];

int iterations = atoi(argv[2]);

long memory\_bytes = (argc >= 4) ? atol(argv[3]) \* 1024l \* 1024l \* 1024l : 8l \* 1024l \* 1024l \* 1024l;

Graph graph(path);

graph.set\_memory\_bytes(memory\_bytes);

BigVector<VertexId> degree(graph.path + "/degree", graph.vertices);

BigVector<float> pagerank(graph.path + "/pagerank", graph.vertices);

BigVector<float> delta(graph.path + "/delta", graph.vertices);

BigVector<float> new\_delta(graph.path + "/new\_delta", graph.vertices);

long vertex\_data\_bytes = (long)graph.vertices \* (sizeof(VertexId) + sizeof(float) \* 3);

graph.set\_vertex\_data\_bytes(vertex\_data\_bytes);

double begin\_time = get\_time();

// Initialize degrees

degree.fill(0);

graph.stream\_edges<VertexId>(

[&](Edge & e) {

write\_add(&degree[e.source], 1);

return 0;

}, nullptr, 0, 0

);

// Initialize Pagerank and Delta

graph.hint(pagerank, delta, new\_delta);

graph.stream\_vertices<VertexId>(

[&](VertexId i) {

pagerank[i] = 0.15f; // Initial PageRank value

delta[i] = 1.0f / degree[i]; // Initial delta

new\_delta[i] = 0;

return 0;

}, nullptr, 0,

[&](std::pair<VertexId, VertexId> vid\_range) {

pagerank.load(vid\_range.first, vid\_range.second);

delta.load(vid\_range.first, vid\_range.second);

new\_delta.load(vid\_range.first, vid\_range.second);

},

[&](std::pair<VertexId, VertexId> vid\_range) {

pagerank.save();

delta.save();

new\_delta.save();

}

);

// PageRank Delta Iterations

for (int iter = 0; iter < iterations; iter++) {

graph.hint(delta, new\_delta);

graph.stream\_edges<VertexId>(

[&](Edge & e) {

write\_add(&new\_delta[e.target], 0.85f \* delta[e.source]);

return 0;

}, nullptr, 0, 1,

[&](std::pair<VertexId, VertexId> source\_vid\_range) {

delta.lock(source\_vid\_range.first, source\_vid\_range.second);

},

[&](std::pair<VertexId, VertexId> source\_vid\_range) {

delta.unlock(source\_vid\_range.first, source\_vid\_range.second);

}

);

graph.hint(pagerank, delta, new\_delta);

graph.stream\_vertices<float>(

[&](VertexId i) {

pagerank[i] += new\_delta[i];

delta[i] = new\_delta[i] / degree[i];

new\_delta[i] = 0; // Reset for next iteration

return 0;

}, nullptr, 0,

[&](std::pair<VertexId, VertexId> vid\_range) {

pagerank.load(vid\_range.first, vid\_range.second);

delta.load(vid\_range.first, vid\_range.second);

new\_delta.load(vid\_range.first, vid\_range.second);

},

[&](std::pair<VertexId, VertexId> vid\_range) {

pagerank.save();

delta.save();

new\_delta.save();

}

);

}

double end\_time = get\_time();

printf("%d iterations of pagerank delta took %.2f seconds\n", iterations, end\_time - begin\_time);

return 0;

}

## 10: Graph Mining

### FSM.py

import pandas as pd

import numpy as np

import json

from collections import defaultdict

from itertools import combinations

from networkx.algorithms import isomorphism

from math import comb

from multiprocessing import Pool

def build\_graph(edges, vertices):

print("Building graph...")

graph = defaultdict(list)

# Ensure unique indices in the vertices DataFrame

vertices = vertices.drop\_duplicates(subset='id').set\_index('id')

# Add all edges to the graph

for \_, row in edges.iterrows():

source, target = row['source\_id'], row['target\_id']

graph[source].append((target, row['amt'], row['strategy\_name'], row['buscode']))

print(f"Total nodes: {len(graph)}, Total edges: {sum(len(v) for v in graph.values())}\n")

return graph

def hash\_edge(source, target, amt, strategy\_name, buscode):

# I think we can go without some of these values, but thought they might be nice to have anyways :-)

return f"{min(source, target)}-{max(source, target)}-{amt}-{strategy\_name}-{buscode}"

def mine\_frequent\_subgraphs(graph, pattern\_size, support\_threshold, output\_file):

print("Mining frequent subgraphs...")

subgraph\_counts = defaultdict(int)

subgraph\_patterns = defaultdict(list)

# Iterate over node combinations

for nodes in combinations(graph.keys(), pattern\_size):

edges = []

for u, v in combinations(nodes, 2):

if v in [neighbor[0] for neighbor in graph[u]]:

edge\_details = next(

(neighbor for neighbor in graph[u] if neighbor[0] == v), None

)

if edge\_details:

edges.append(

hash\_edge(u, v, \*edge\_details[1:])

)

if len(edges) == pattern\_size:

subgraph\_key = "\_".join(sorted(edges))

subgraph\_counts[subgraph\_key] += 1

subgraph\_patterns[subgraph\_key].append(edges)

# Filter frequent subgraphs

frequent\_subgraphs = {

k: v for k, v in subgraph\_counts.items() if v >= support\_threshold

}

save\_results(frequent\_subgraphs, subgraph\_patterns, output\_file)

print("Frequent subgraph mining completed.")

def save\_results(frequent\_subgraphs, subgraph\_patterns, output\_file):

result = []

for subgraph, frequency in frequent\_subgraphs.items():

edges = subgraph\_patterns[subgraph]

result.append({

"frequency": frequency,

"edges": [{"source": edge.split("-")[0], "target": edge.split("-")[1], "details": edge} for edge in edges]

})

with open(output\_file, 'w') as f:

json.dump(result, f, indent=4)

print(f"Results saved to {output\_file}")

if \_\_name\_\_ == "\_\_main\_\_":

print("Reading data...")

header1 = ['id', 'name', 'timestamp', 'black']

header2 = ['source\_id', 'target\_id', 'timestamp', 'amt', 'strategy\_name', 'trade\_no', 'buscode', 'other']

account = pd.read\_csv('data/account', names=header1, sep=',')

card = pd.read\_csv('data/card', names=header1, sep=',')

account\_to\_account = pd.read\_csv('data/account\_to\_account', names=header2, sep=',', usecols=range(len(header2)))

account\_to\_card = pd.read\_csv('data/account\_to\_card', names=header2, sep=',', usecols=range(len(header2)))

vertices = pd.concat([account, card])

edges = pd.concat([account\_to\_account, account\_to\_card])

edges['amt'] = edges['amt'].round()

graph = build\_graph(edges, vertices)

# Start mining frequent subgraphs

mine\_frequent\_subgraphs(

graph,

pattern\_size=3,

support\_threshold=10000,

output\_file=f"results/bdci\_data.json"

)

## 11: Spark Streaming Top-K

### Top-k.py:

from \_\_future\_\_ import print\_function

import sys

from pyspark import SparkContext

from pyspark.streaming import StreamingContext

from pyspark.streaming.dstream import DStream

# --- Code Setup ---

# Initialize SparkContext and StreamingContext

sc = SparkContext(appName="Py\_HDFSWordCount")

ssc = StreamingContext(sc, 60)

# Create a DStream that listens to the HDFS directory

hdfs\_directory = "hdfs://intro00:8020/user/2024403421/stream"

lines = ssc.textFileStream(hdfs\_directory)

# We use these global variables to keep track of our top-k algorithm

word\_counts = {}

file\_no = 1

last\_file = 5

k = 100

# --- My two functions to perform the Top-K ---

# Function to update the word counts with each new RDD

def update\_count(new\_counts, last\_counts):

# On first batch we initialize an empty dictionary

if last\_counts is None:

last\_counts = {}

# On all continuing files, update the word counts

for word, count in new\_counts:

if word in last\_counts:

last\_counts[word] += count

else:

last\_counts[word] = count

return last\_counts

# Function to process each RDD and compute the top-k frequent words

def process\_rdd(time, rdd):

global word\_counts, file\_no, last\_file, k

if rdd.isEmpty():

return

# Compute word counts for the current RDD

counts = rdd.flatMap(lambda line: line.split(" ")) \

.map(lambda word: (word, 1)) \

.reduceByKey(lambda a, b: a + b)

# Update the global word counts using the updateCounts function

updated\_counts = counts.collect() # Collect current counts to update

word\_counts = update\_count(updated\_counts, word\_counts)

# Sort by count and take top k

top\_k = sorted(word\_counts.items(), key=lambda x: x[1], reverse=True)[:k]

# We structure the output

output = f"----- File Number {file\_no} -----\n"

output += f" --- Top-{k} words so far ---\n"

for word, count in top\_k:

output += f"{word}: {count}\n"

# We write the output to a file

output\_filename = f"output\_file\_{file\_no}.txt"

with open(output\_filename, "w") as f:

f.write(output)

print(output)

file\_no += 1

# Stop the program after processing the last file

if file\_no > last\_file:

print("\nMaximum number of files processed. Stopping the streaming.")

ssc.stop(stopSparkContext=True, stopGraceFully=True)

# --- Use My functions ---

# Process each RDD in the DStream and compute top-k

lines.foreachRDD(process\_rdd)

# Start streaming and wait for termination

ssc.start()

ssc.awaitTermination()

## 12: Halide Scheduling

### Dialated-conv.cpp:

#include "Halide.h"

#include "common.h"

#include <stdio.h>

using namespace Halide;

using namespace Halide::Tools;

int main(int argc, char \*\*argv) {

const int N = 5, CI = 128, CO = 128, W = 100, H = 80, KW = 3, KH = 3;

const int dilation = 15;

ImageParam input(type\_of<float>(), 4);

ImageParam filter(type\_of<float>(), 4);

// Define variables and reduction domain

Var x("x"), y("y"), c("c"), n("n");

Var xo("xo"), yo("yo"), xi("xi"), yi("yi");

Var co("co"), ci("ci");

Func dilated\_conv("dilated\_conv");

RDom r(0, CI, 0, KW, 0, KH);

// Algorithm definition

dilated\_conv(c, x, y, n) = 0.0f;

dilated\_conv(c, x, y, n) += filter(c, r.y, r.z, r.x) \*

input(r.x, x + r.y \* (dilation + 1), y + r.z \* (dilation + 1), n);

// \*\*Scheduling\*\*

// 1. Split the x and y dimensions for tiling

dilated\_conv.compute\_root()

.tile(x, y, xo, yo, xi, yi, 8, 8) // 8x8 tile size (tunable)

.fuse(xo, yo, co)

.parallel(co) // Parallelize outer loop over tiles

.vectorize(xi, 8); // Vectorize inner x dimension

// 2. Optimize the reduction

dilated\_conv.update()

.reorder(r.x, r.y, r.z, c, xi, yi, n)

.unroll(r.z) // Unroll kernel height loop

.unroll(r.y) // Unroll kernel width loop

.vectorize(xi, 8) // Vectorize inner computation

.parallel(n); // Parallelize across batch dimension

// Buffer initialization

Buffer<float, 4> in(CI, W + (KW - 1) \* (dilation + 1), H + (KH - 1) \* (dilation + 1), N);

Buffer<float, 4> fil(CO, KW, KH, CI);

Buffer<float, 4> output\_halide(CO, W, H, N);

// Initialize input and filter with random data

random\_data<float, 4>(in);

random\_data<float, 4>(fil);

input.set(in);

filter.set(fil);

// JIT compile and run

dilated\_conv.realize(output\_halide);

double t\_halide = benchmark(10, 10, [&]() { dilated\_conv.realize(output\_halide); });

Buffer<float, 4> output\_ref(CO, W, H, N);

double t\_onednn = dnnl\_dilated\_conv\_wrapper(in.data(), fil.data(), output\_ref.data(),

{N, CI, CO, W, H, KW, KH, dilation, dilation});

// Check correctness

if (check\_equal<float, 4>(output\_ref, output\_halide)) {

printf("Halide results - OK\n");

} else {

printf("Halide results - FAIL\n");

return 1;

}

float gflops = 2.0f \* (N \* CO \* H \* W) \* (CI \* KH \* KW) / 1e9f;

printf("Halide: %fms, %f GFLOP/s\n", t\_halide \* 1e3, (gflops / t\_halide));

printf("oneDNN: %fms, %f GFLOP/s\n\n", t\_onednn \* 1e3, (gflops / t\_onednn));

printf("Success!\n");

return 0;

}