GRACL (.grc)

Defne Sonmez	dys2109	System Architect
Eilam Lehrman	esl2160	Language Guru
Hadley Callaway	hcc2134	Manager
Maya Venkatraman	mv2731	System Architect
Pelin Cetin	pc2807	Tester

1. Introduction & Motivation

GRACL (GRAph Concurrency Language) improves the efficiency of common graph algorithms such as Breadth-First-Search (BFS), Depth-First-Search (DFS), Dijkstra, and Traveling Salesman Problem. GRACL uses concurrency (with lightweight multithreading as in Go) and allows programmers to initialize and modify graphs easily with built-in data structures specific to our language. This combination enables the user to implement concurrent graph algorithms that converge more quickly than their traditional counterparts. Improved runtime efficiency has enormous potential to improve modern-day computing in areas such as transportation networks, metadata relation-building database systems, packet switching, neural networks, etc. We were inspired by some of the past projects focused on graphs (such as GRAIL from Spring 2017) to use syntax with elements from Java, Python, and C. We plan to make the following features available to the programmer: graphs, nodes, threads, and locks. We will discuss their implementation at greater length in our “Syntax” section.

2. Language Overview

GRACL is statically scoped, as well as strongly and statically typed to fully leverage the efficiency of the compiler. Like Java, GRACL creates copies of the references and passes them as values to methods.

GRACL supports mutable data types because immutability does not make sense for our application space. In typical graph searches, one must maintain a frontier or data structure of visited nodes. Immutability would force processes to create a new frontier or visited node structure every single time a new node is visited, making mutable objects far preferable.

We will ensure that threads are collected under the hood after they terminate and that orphan or zombie processes are properly reaped. There will be strict evaluation. While there will be no complete garbage collection, we may consider freeing graphs, nodes, lists, and strings for the user at the end of their program execution if time permits.

3. Syntax

a. Primitives

i. bool
ii. char
iii. double
iv. int
v. void
vi. any - a type that allows nodes to store any type of data in their datafields

b. Objects

String	Character arrays with basic functionalities emulating Java Strings, primarily used for printing
List	A standard linked list [] with basic functionalities emulating Java LinkedLists
Node	An object that has a data field and a list of neighboring nodes
Graph	A list of nodes connected by directed or undirected edges representing a graph data type. Data types in the graph do not need to be homogeneous
Thread	A lightweight process sharing memory below the stack with other threads, much like a pthread in C. Initialized with a function that starts the routine the user wishes to have executed
RwLock	A reader/writer lock that allows multiple processes to read protected data simultaneously, but only allows one thread to write at once. No thread may read while another thread is writing
Tuple	A combination of two data types

c. Integer and Double Operators

i. +, -, /, *, %, +=, -=

d. Logical Operators

i. &&, ||, !, <, >, <=, >=, ==, !=

e. Control Flow

/**/	Multiline comment
//	Single line comment
;	Signifies the end of a statement

if(...){} else{};	Conditional statements

for(...){};
for(... in ...){};
while(...){};
	Loops

int myFunc(int x) {
  return x;
};
	Example function

f. Node Properties and Built-In Functions

createNode(String name, any data, Node[] neighbors)	Creates a new node, setting its name, data, and list of neighbor tuples (weight, destinationNode). Returns an error if the passed-in name is not unique
.name	Provides a unique string label by which to access the node
.data	Returns data stored in the node
.neighbors	Returns a list of neighbor nodes
.updateName(String name)	Updates the name field on the node to be the new name passed in
.updateData(any data)	Updates the data field on the node to be the new data passed in
.rwLock() (Note: **possible)	Some concurrent graph algorithms require nodes to be marked as complete so multiple threads don't visit them. We propose creating locks for nodes so that no two threads try to mark a node as complete at the same time **We recognize that it may be difficult to expect the user to protect node accesses with locks. This is something we would appreciate feedback on in our proposal.

g. Graph Properties and Built-In Functions

createGraph(String formattedEdges)	Creates a new graph out of a user's string of edges. Each edge should be formatted start-weight-destination for an undirected edge or start-weight->destination for a directed edge. When creating each node, the start is taken as the node's name and the node's data field is left blank

	Example input: “B-4-C, C-7->D, A-3->C, D-3-B”
.nodes	Returns a list of nodes in the graph
.edges	Returns a string representation of all edges in the graph
.addDirectedEdge(Node A, Node B, int weight)	Adds to node A’s neighbor list a directed edge to node B, representing it as a tuple (int weight, node destination)
.addUndirectedEdge(Node A, Node B, int weight)	Adds an undirected edge between node A and node B. Under the hood, this calls .addDirectedEdge() twice, once to add a directed edge from node A to node B and once to add a directed edge from node B to node A
.updateDirectedEdge(Node A, Node B, int weight)	Updates the directed edge from node A to node B to have a new weight
.updateUndirectedEdge(Node A, Node B, int weight)	Updates the undirected edge between node A and node B. Under the hood, this calls .updateDirectedEdge() twice, once to update the directed edge from node A to node B and once to update the directed edge from node B to node A
.removeDirectedEdge(Node A, Node B)	Removes the directed edge from node A to node B
.removeUndirectedEdge(Node A, Node B)	Removes the directed edge from node A to node B, and then removes the directed edge from node B to node A
.addNode(Node n)	Adds the passed-in node to the graph
.removeNode(Node n)	Removes the passed-in node from the graph and deletes corresponding edges

h. Thread Properties and Built-In Functions

createThread(any func, any param1, any param2, ...)	Creates a thread that begins by executing the function func applied to the given parameters. This returns a thread object which GRACL detaches upon completion behind the scenes
joinThreads(Thread[] threads)	Blocks until all threads specified in the list complete

i. RwLock Properties and Built-In Functions

createRwLock()	Creates an rwLock
.racquire()	Before entering a critical section of code, a thread has to wait to acquire the rwLock for reading. Once the lock is acquired it may execute the critical section
.wacquire()	Before entering a critical section of code, a thread has to wait to acquire the rwLock for writing. Once the lock is acquired it may execute the critical section
.rrelease()	After executing a critical section, a thread must release the rwLock for reading
.wrelease()	After executing a critical section, a thread must release the rwLock for writing

j. Other Functions

print(“Hello world”);	Print function that prints strings

4. Basic Operations & Examples
    a. Making and modifying a graph:

```
Graph g = createGraph("B-4-C, C-7->D, A-3->C, D-3-B");
A.updateData(True);
B.updateData(5);
C.updateData("Eilam");
D.updateData("GRACL");
g.removeData(D);
Node E = createNode("E", 7, [(6,A)]);
g.addNode(E);
```

    b. A simple concurrency example:

```
// Assuming this gets allocated on the heap, as in Java
String buf = "Hello";

RwLock lock = createRwLock();

void startRoutine() {
    lock.wacquire();
    print(buf);
    buf = "World";
    lock.wrelease();
}

for (int i = 0; i < 2; i++) {
    createThread(startRoutine);
}

/*
Sample output:

"Hello"
"World"

*/
```

c. Concurrent DFS:

```
Node[] path = [];
Node[] visited = [];
RwLock visitedLock = createRwLock();
RwLock pathLock = createRwLock();

bool goalTest(Node goal, Node current){
    return goal.name == current.name;
    // Searching for a specific node object using its unique name
}
```

```
void normalDFS(Graph graph, Node current, Node goal, Node[] visited, Node[] myPath){
    pathLock.racquire();
    // Another thread found the goal already and this thread should terminate
    if (path != []) {
        // Release the read lock on path
        pathLock.rrelease();
        return;
    } else {
        // Release the read lock on path
        pathLock.rrelease();
        myPath.add(current);
        // If goal found
        if (goalTest(goal, current)) {
            // Modify shared memory for path to goal
            pathLock.wacquire();
            path = myPath;
            // Release the write lock on path
            pathLock.wrelease();
            return;
        } else {
            // Add current to shared memory list of visited nodes
            visitedLock.wacquire();
            visited.add(current);
            // Release the write lock on visited
            visitedLock.wrelease();
            Node[] neighbors = current.neighbors;
            for ((weight, neighbor) in neighbors) {
                visitedLock.racquire();
                // If visited doesn't contain neighbor, call normal DFS on neighbor
                if (!visited.contains(neighbor)) {
                    // Release the read lock on visited
                    visitedLock.rrelease();
                    normalDFS(graph, neighbor, goal, visited, myPath);
                } else {
                    // Release the read lock on visited
                    visitedLock.rrelease();
                }
            }
            return;
        }
    }
}
```

```
Node[] multithreadDFS(Graph graph, Node start, Node goal) {
    if (goalTest(goal, start)) {
        return [];
    } else {
        Node[] neighbors = start.neighbors;
        visited.add(start);
        Thread[] threads = [];
        // Create a thread for each top-level child of start to perform search in parallel
        for ((weight, neighbor) in neighbors) {
            threads.add(createThread(normalDFS, graph, neighbor, goal, visited, [start]));
        }
        joinThreads(threads);
        return path;
    }
}
```

5. References
    a. GRAIL: A Graph-Construction Language Proposal (Spring 2017):
        i. cs.columbia.edu/~sedwards/classes/2017/4115-spring/proposals/GRAIL.pdf
    b. Grape (.grp) (Fall 2018):
        i. cs.columbia.edu/~sedwards/classes/2018/4115-fall/proposals/Grape.pdf
    c. Pass by Reference vs. Pass by Value in Java:
        i. tutorialspoint.com/Pass-by-reference-vs-Pass-by-Value-in-java
    d. Goroutines:
        i. gobyexample.com/goroutines
    e. Java Implementation of DFS:
        i. geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
    f. Concurrent DFS:
        i. github.com/ZdravkoHvarlingov/Concurrent-DFS
        ii. link.springer.com/chapter/10.1007/978-3-642-54862-8_14
    g. Parallel BFS:
        i. en.wikipedia.org/wiki/Parallel_breadth-first_search