Lab 3: Multithreading

Henry Lambson

Advanced Application Programing

11/8/2022

Rust, WASM, and Threads

To be able to run this project, some setup may be required. Firstly, the basic Rust package must be installed in order to run the Rust version of the code. This can be done here: <https://www.rust-lang.org/tools/install> . For threading, I used wasm-mt which can just be included in the cargo.toml file under dependencies, however it can only be used on the nightly version of Rust, so that must be included in the cargo.toml file under toolchain and must be installed through rustup and set it as the current version for the code to run. This can be done with the command: “rustup default nightly”. I used wasm-pack in order to transpile the Rust into wasm, which can be installed through npm, but is not needed in order to run the project. In order for the threading library to work in browser, I needed to change the command to “wasm-pack build –release –target no modules”. The –no-modules target makes it so the JavaScript must be included in a script tag in the index.html in order to run. The index.js and bootstrap.js are no longer called, but my app wouldn’t run if I deleted them so I just left them in.

Once the setup is done, the project can be ran by going into the graph-algorithms folder and running the command “cargo run” for the main.rs to run. For this version of the library, I elected to only include multithreading in the portion of the code which gets transpiled, so there is no multithreading in the rust version of the functions (lib.rs has all of the threading, main.rs does not). To run the JavaScript, go into the www folder, and run the command “npm run start” to launch a local server on port 8080.

I realized after the GitHub issues I emailed about that I needed to be clearer in this section.

1. Install rust through the link provided.
2. Install wasm-pack at <https://rustwasm.github.io/wasm-pack/installer/>
3. Install rust nightly with “rustup default nightly”
4. Cd into graph-algorithms
5. Wasm-pack build –release –target no-modules
6. Copy the pkg folder created into the www folder
7. Cd into www
8. npm install
9. npm run start

API

create\_cycle\_graph(v:u32) -> JsValue Function to create a graph object that is a cycle with v vertices. This gets returned as a JsValue so that it can be transpiled into WASM since it is a custom object. An example of a cycle with 3 vertices would be a triangle, with 4 vertices a square, 5 vertices a pentagon, etc.

create\_cycle\_graph\_thread(v:u32) -> Result<JsValue,JsValue>

Function to create a graph object that is a cycle with v vertices via spinning up a thread. This gets returned as a Result<JsValue,JsValue> because it is asynchronous and gets treated as a promise by JavaScript.

create\_random\_graph(v:u32, n:u32) -> JsValue

Function to create a graph with v vertices with each vertex having up to n edges, chosen randomly for each vertex. Each edge will have a weight from 1 to 100, also chosen randomly. Returned as a JsValue to transpile into WASM.

create\_random\_graph\_thread(v:u32, n:u32) -> Result<JsValue,JsValue>

Function to create a random graph with v vertices with each vertex having up to n edges, chosen randomly for each vertex via spinning up a thread. This gets returned as a Result<JsValue,JsValue> because it is asynchronous and gets treated as a promise by JavaScript.

breadth\_first\_search(graph:JsValue, root:u32, target:u32) -> Vec<u32>

Function to run Breadth First Search on a graph starting at root and ending at target. Returns a vector of the vertices visited. Will return an empty vector if no path can be found.

breadth\_first\_search\_thread(graph:JsValue, root:u32, target:u32) -> Result<JsValue,JsValue>

Function to run Breadth First Search on a graph starting at root and ending at target via spinning up a thread. This gets returned as a Result<JsValue,JsValue> because it is asynchronous and gets treated as a promise by JavaScript.

depth\_first\_search(graph:JsValue, root:u32, target:u32) -> Vec<u32>

Function to run Depth First Search on a graph starting at root and ending at target. Returns a vector of the vertices visited. Will return an empty vector if no path can be found.

depth\_first\_search\_thread(graph:JsValue, root:u32, target:u32) -> Result<JsValue,JsValue>

Function to run Depth First Search on a graph starting at root and ending at target via spinning up a thread. This gets returned as a Result<JsValue,JsValue> because it is asynchronous and gets treated as a promise by JavaScript.

kruskal(graph:JsValue, v:u32) -> Vec<u32>

Function to run Kruskal’s algorithm for finding the minimum spanning tree of a graph. Takes in a graph to run on and the total number of vertices of that graph v. Returns a vector of the order of the nodes in the minimum spanning tree.

kruskal\_thread(graph:JsValue, v:u32) -> Result<JsValue,JsValue>

Function to run Kruskal’s algorithm for finding the minimum spanning tree of a graph via spinning up a thread. Takes in a graph to run on and the total number of vertices of that graph v. This gets returned as a Result<JsValue,JsValue> because it is asynchronous and gets treated as a promise by JavaScript.

Design

The threaded versions of each function are designed to be able to be used independently of one another so that they can be run in any order in parallel. This way, multiple graphs can all have multiple searches run on them simultaneously. They are implemented with simplicity on the JavaScript side of things in mind, so all the user must do is call the functions without worry of other overhead. The main assumption made in this implementation is that the user would want to run multiple searches in parallel and not having each search be parallelized to increase performance. The pro of this is that the whole library can be run in parallel on as many graphs as the user wishes. The con of this is that the individual performance of each function has not been optimized by parallelizing them.

Testing

To test my new threaded library, I decided to compare the time taken to run BFS, DFS, and Kruskal on the same graph in parallel to the time taken to run them sequentially. I chose to do this on a cycle graph for simplicity and consistency. Since I will be running BFS and DFS on a cycle, it they just have to follow the path forward and backward until they reach the target, so I chose a root of 1 and the target furthest away from the root, which would be the median of the graph (i.e., 1 and 500 for a size 1000 graph). I initially found that with small graphs, running sequentially would be significantly faster because it avoids the additional overhead that utilizing the threads bring.

Graph of size 1000

|  |  |
| --- | --- |
| Multithread time (ms) | Single thread time (ms) |
| 76.59999990463257 | 12 |
| 66.5 | 11.099999904632568 |
| 73.7000002861023 | 11.099999904632568 |
| 68.69999980926514 | 11.799999713897705 |
| 67.7000002861023 | 11.099999904632568 |

Once I began scaling up the size of the graph however, the multithreaded versions of the functions began outperforming running them sequentially.

Graph of size 30,000

|  |  |
| --- | --- |
| Multithread time (ms) | Single thread time (ms) |
| 4411.099999904633 | 5716.800000190735 |
| 4359.299999713898 | 5700.700000286102 |
| 4422.299999713898 | 5738 |
| 4904.900000095367 | 5759.700000286102 |
| 4331.5 | 5800.39999961853 |

Since the multithreaded version is supposed to increase performance, I decided to continue testing on a large graph rather than a small one. I ran my test 100 times on a cycle ­­­­­­­­graph with­ 30,000 vertices. The full data for all 100 runs is in the excel file.

Y-axis is in milliseconds

Looking at this graph of the means with 95% confidence intervals, there is a statistical difference in the performance of the multithreaded functions versus running them sequentially. This does heavily vary on the size of the graph that is chosen, as a smaller graph would cause the means to grow closer, and then to spread out again as the single threading begins to outpace the multithreading on small graphs. However, if we kept scaling the graph to be larger and larger, the gap between the two would only keep growing:

Graph of size 100,000

|  |  |
| --- | --- |
| Multithread timing (ms) | Single thread timing (ms) |
| 45585.800000190735 | 62291.90000009537 |
| 44975.7999997139 | 62018.09999990463 |