Organisation: Julia (Under NumFocus)

Library: LightGraphs.jl

Title: Parallel Graph Development

Table of Contents

Contributions to LightGraphs.jl

Pull Requests:

1. #844: Greedy coloring

Implemented a heuristic for vertex coloring.

2. #843: Optimise Kruskal's MST

Obtained a **21.6x speed up** on Kruskal's Minimum Spanning Tree algorithm by optimizing the disjoint set data structure. The benchmarks were performed on a graph with 100,000 vertices and 500,000 edges.

3. #839: Improve memory efficency in Prim MST

Improved the memory efficiency of Prim's Minimum Spanning Tree algorithm by avoiding unnecessary allocation of memory.

Patches that are work in progress can be found in: Parallel Branch

I am waiting for the migration of the code to Julia v0.7 as a feature freeze is in progress.

It includes:

1. **Parallel BFS with reduced locking.**

Obtained a **3.4x speed up** on parallel BFS by reducing the number of atomic operations performed on the thread safe queue. The benchmarks were performed on a random regular graph with 10,000 vertices and 10,000,000 edges.

2. **Johnson's all pairs shortest path**

Implemented Johnson's all pairs shortest algorithm. It is performs the same task as Floyd Warshall but is more efficient when the graph is sparse

3. **Parallel Floyd Warshall**

Produced a multi-threaded implementation of Floyd Warshall. The implementation is fairly trivial. During GSoC, I hope to improve the performance by improving the memory management (Refer to the proposal).

4. **Distributed Floyd Warshall**

Produced a simple distributed implementation of Floyd Warshall.

Proposal

Title

Parallel Graph Development

Abstract

LightGraphs is a Julia library that implements several commonly used graph algorithms. The objective of the package is to provide the functionality of robust network and graph analysis libraries such as NetworkX while maintaining efficiency and user-friendliness. Keeping this objective in mind, my proposal is to produce a parallel (multi-threaded and multi-process) implementation of the graph algorithms already available in LightGraphs as well as implement commonly used heuristics and approximation algorithms for certain graph problems.

Description

My proposal consists of producing a parallel implementation of graph algorithms already available in LightGraphs along with the implementation of approximation algorithms and their relevant data structures.

A) Implement approximation algorithms and heuristics for the following NP-hard problems along with their parallel implementations:-

i. Travelling Salesman

This will be solved using Christofede's algorithms, which can be easily implemented if edge matching, minimum spanning tree and Euler tour are available as a subroutine. This has an approximation factor of 1.5. The **Euler tour subroutine** will have to be implemented.

ii. Minimum Vertex cover

This will be solved iteratively choosing both end points of an edge. This has an approximation factor of 2.

iii. Minimum Steiner tree

This is solved by finding the MST of the metric closure of the terminal nodes.

The **metric closure subroutine** will have to be implemented.

iv. Minimum Maximal Matching

Iterate through the edges and choose an edge if none of its end points are in an edge already chosen.

v. Greedy coloring with interchange

In greedy coloring, we iterate through the vertices and assign the lowest color possible to each node and introduce a new color if needed.

Using interchange, we check at every iteration if it is possible to avoid introducing a new color by swapping colors between 2 vertices.

vi. Independent vertex set

Iterate through the vertices and add a vertex to the answer set if none of its neighbour have already been chosen.

vii. Dominating vertex set

The heuristic is the same as independent vertex set.

viii. Local node connectivity

Find a set of node independent paths from source to destination and output a vertex in each path.

Parallel implementations: Before producing a parallel implementation of these algorithms, the subroutines they use will have to be parallelized.

For the algorithms where the nodes are ordered and iterated over in some order, the nodes will be assigned some priority based on the order. The vertices will be partitioned and iterated in parallel.

B) Produce parallel implementations of Floyd Warshall, Bellman Ford and Johnson algorithm.

i) Bellman Ford

It consists of iteratively "relaxing" the edges of the graph until a stable state has been found. If a stable state has not been obtained after |V| iterations then a negative cycle has been detected.

Relax edge e = (a, b) with source s: dist(b) = min(dist(b), dist(a) + weight(e)

In the parallel implementation, the edges will be relaxed in parallel. Some thought will have to be put into load balancing the edges to be relaxed in each iteration.

ii) Floyd Warshall

Each node is chosen as a pivot exactly once. The shortest path from all pairs of nodes will be relaxed through the pivot element

Relaxing pair u, v through pivot p: dist(u, v) = min(dist(u, v), dist(u, p)+dist(p, v))

Since the pivot data is used frequently, efficient memory management will dramatically improve performance.

iii) Johnson

It uses Bellman Ford once and Dijkstra |V| times for most of its operations. Hence, parallelizing Bellman Ford should be sufficient for this problem.

C) Implement the Disjoint Set Data Structure and make it available as a utility of LightGraphs [3].

This data structure can efficiently

- Check if two vertices are in the same set (used to represent connected component).
- Merge two sets.

I have already implemented this in my pull request for improving the efficiency of Kruskal's algorithm.

D) Use the data structure to implement Karger's global min cut algorithm [8].

The implementation is similar to Kruskal's MST. Instead of sorting the edges, a random permutation is chosen.

E) Implement parallel Disjoint Set data structure and use it to parallelize Kruskal's Minimum Spanning Tree.

Checking if two nodes are in the same connected component is easier than merging two connected components. In the parallel implementation of Disjoint Set data structure, helper threads will perform the |E| checks and the main thread will perform the |V| merges. Hence, a significant speed up will be obtained for dense graphs. However, the speed-up will not be noticeable unless a parallel implementation of sorting edges is available.

I wish to discuss how to carry out the communication between the main thread and worker threads during the community bonding period.

F) Implement a parallel priority queue and use to produce a parallel implementation of Dijkstra and Prim [6].

Both Prim and Dijkstra use a priority queue to keep track of the cheapest node across a cut. The availability parallel priority queue would make it easy to implement both of them in parallel.

Preferably, the queue should support Decrease_Key operation (along with Insert, Delete and Get_Min) to improve efficiency as each node will have to be stored at most once in the heap.

I can implement a parallel binary heap supporting the above operations.

In [6] a priority queue that can support Insert, Delete and Get_Min more efficiently was discovered. I will implement that as well and compare the performance.

G) Produce a parallel implementation of flow algorithms, similar to the flo library [7].

i) Edmund-Karp

Run every BFS in parallel. I was thinking of an implementation where frontiers are expanded from the source as well as the destination until the two frontiers come in contact with each other.

ii) Dinic

Run every BFS in parallel (Similar to Edmund-Karp). I could attempt to run the flow search in parallel but the interleaving of flows may ruin the guarantee of the upper bound of iterations required

iii) Push-relabel

Each node is assigned a height. Each node iteratively attempts to push flow from it to a neighbor with lower height or increase your own height until none of them have excess flow.

Time-line

Community Bonding Period [23rd April to 14th May]

Before coding begins, I need to familiarize myself with the practices expected to be followed in the code. More importantly, I need to familiarize myself with the inner workings of Julia's parallel functionality and how to perform optimization techniques for parallel computing such as tiling. In particular, I would like to discuss the implementation of *parallel disjoint set data structure* and *parallel priority queue*.

My end semester exams begin on 1st May and end on 5th May. Due to that, I would not be free between 28th April and 5th May.

Week 1-2 **[May 14 - 28 May]**

- Approximations Algorithms and Heuristics

By then end of the second week, all 8 algorithms along with their test benches should be implemented and documented.

Week 3-5 **[May 28 - June 18]**

- Parallelize shortest path:

1. Floyd Warshall [4]
2. Bellman-Ford [5]
3. Johnson

- Disjoint Set data structure (Parallel and Sequential) [3]
- Karger's min cut algorithm
- Parallelize Kruskal's MST [1]

First, I will implement parallel versions of commonly used shortest path algorithms, except Dijkstra. Parallel Dijkstra will be implemented towards the end of the project as it requires the implementation of a parallel priority queue.

Then, I will implement the Disjoint Set data structure and make it available as a functionality. I will use it to implement Karger's algorithm for global min cut.

I will implement a parallel Disjoint Set Data Structure and use it to produce a parallel implementation of Kruskal's MST. This needs to be discussed in the community bonding period.

I have allocated a buffer period in case anything in the first 5 weeks gets delayed.

I am optimistic that at least the following will be ready for phase 1 evaluation (15th June)

- Parallel shortest path algorithms.
- All approximation algorithms and heuristics.
- Sequential Disjoint Set Data Structure.
- Karger's algorithm.

Week 6 **[June 18 to June 25]**

- Parallel implementation of centrality algorithms

1. Multi-threaded betweeness centrality
2. Multi-threaded closeness centrality
3. Multi-threaded radiality
4. Multi-threaded stress
5. Parallelize Pagerank [2]

Multi-threaded implementation of the centrality algorithms are lacking (Multi-process is already implemented for most of them). I will implement them, along with a parallel implementation of Page Rank.

Week 7-9 **[June 25 to July 16]**

- Parallel implementation of the heuristics and approximation algorithms in week 1-2.

I am optimistic that at least the following will be ready for phase 2 evaluation (13th July)

- Parallel centrality algorithms.
- Parallel heuristics and approximation algorithms.

Week 10-12 **[July 16 - August 6]**

- Parallel flow algorithms

1. Edmund-Karp
2. Dinic
3. Push-relabel

- Implement parallel priority queue. [6]
- Parallel Dijkstra's Shortest Path
- Parallel Prim's Minimum Spanning Tree.

LightGraphsFlow has implemented several flow algorithms. I will produce parallel implementation of them.

Then, I will implement a parallel priority queue and use it to produce a parallel implementation of Dijkstra and Prim. This needs to be discussed during the community bonding period.

All existing bugs will be addressed and documentation will be completed.

I am optimistic that at least the following will be ready for final evaluation

- Parallel flow algorithms

Any extra time remaining will be spent improving the efficiency of the algorithms already implemented in LightGraphs.

About Me

I, Soham Tamba am a final year Computer Science and Engineering student at National Institute of Technology Goa, India. I have spent most of my bachelor's degree studying the optimization of algorithms. Hence, I believe LightGraphs is a perfect fit for me. I have taken part in several coding competitions and have extensively studied the design and analysis of algorithms.

Proficiency in Data Structures and Algorithms

Relevant MOOC Completed:

1. **Advanced Algorithms** (Spring 2016) by Ankur Moitra, MIT.
2. **Design and Analysis of Algorithms** (Spring 2015) by MIT OCW.
3. **Probabilistic Systems Analysis and Applied Probability** (Fall 2013) by MIT OCW.
4. **Introduction to Algorithms** (Fall 2011) by MIT OCW.
5. **Mathematics for Computer Science** (Fall 2010) by MIT OCW.

Relevant Undergraduate Courses:

1. Parallel Algorithms
2. Graph Theory
3. Design and Analysis of Algorithms
4. Computer Networks
5. Theory of Computation
6. Probability and Statistics
7. Data Structures
8. Discrete Mathematics

Coding competitions:

I have taken part in several coding competitions, where the objective was to design an algorithm and implement it to solve ad-hoc problems with near optimal runtime and reasonable memory constraints.

Results and corresponding results in reverse chronological order:

1. **Rank 1** in **Codetron**, 2018. Organized by **Goa Engineering College** during their Techfest, Spectrum.
2. **Rank 1** in **Programmatics**, 2018. Organized by **National Institute of Technology Goa** during their Techfest, Saavyas.
3. **Rank 88** in **ACM ICPC** Asia Amritapuri Site Regional Contest, 2017.
4. **Rank 13** in **Inter-NIT Code-a-thon**, 2017. Organized by The **Indian Society for Technical Education**, NIT Bhopal chapter.
5. **Rank 1** in **Code Heat**, 2016. Organized by **Manipal Institute of Technology**, IECSE.
6. **Rank 202** in the **Morgan Stanley** Code-a-thon, 2016.
7. **Rank 342** in **Google Code APAC Code Jam** Round B, 2016.

Hackerrank profile: Soham_Tamba

Country of Residence: India

Time Zone: GMT + 5:30 (Indian Standard Time)

Logistics

My end semester exams conclude in the first week of May. I may be attending a few interviews for higher study applications that would in sum take no more than a week. Other than that, I have no commitments during the summer and can devote my time entirely to Google Summer of Code.

Contact

Email: sohamtamba@gmail.com

Github: SohamTamba

Mobile: (+91)-9049336787

Skype ID: sohamtamba_1

Slack ID: @sohamtamba

References

[1] Parallel Kruskal

[2] Parallel Pagerank

[3] Parallel Disjoint Set

[4] Parallel Floyd Warshall

[5] Parallel Bellman Ford

[6] Parallel Priority Queue

[7] Flo library

[8] Karger's min cut

[9] NetworkX