Parallelisation of Graphing Algorithms in Julia

Group 4

1. Alexander Maehl
2. Sam Broadhead
3. William Ning

Agenda

- Introduction to Julia
- Graphing algorithms
- Parallelization of Graphing algorithms
- Our implementation/benchmarking

What is Julia

Julia is

- Open source
- High-level
- High-performance
- Dynamic programming language

Designed for numerical computing

“Looks like python, feels like lisp, runs like C”

Syntax

Python

```python
def sum(a):
    s = 0.0
    for x in a:
        s += x
    return s
```

Julia

```julia
function sum(a)
    s = 0.0
    for x in a
        s += x
    end
    return s
end
```

- 1-based indexing 😡
- Macro support
- Homoiconicity
- Structs

Generality

Julia has metaprogramming support similar to Lisp

```
macro name (expr, expr)
    ...modify evaluation...
End

@name (expr, expr)
```

Compilation in Julia

Julia “runs like c” mostly due to it’s compilation

Julia uses a JIT (just-in-time) compiler based on LLVM to generate native machine code

optimize unnecessary static branches out at runtime

Julia Tasks

Used to execute of multiple functions co-operativly

Declare a channel

```
c1 = Channel(32)
```

```
c2 = Channel(32)
```

!take and !fetch data from and !put data into channels

```
data = take!(c1)
```

```
put!(c2, result)
```

close () channels when done with them

```
close(c1);
```

```
close(c2);
```

Native Threads

Currently Experimental (only supports for loops)

@distributed

```
@distributed [reducer] for var = range
            body
        End
```

@Threads

```
Threads.@threads for var = range
            body
        End
```

(does not support optional reduction parameter)

Processes

Future

- Remotecall the function
- Fetch() the result

```
./julia -p 2
```

Create workers

RemoteChannel

- Create remote channel
- !put() !take() to/from remote channel

```
RemoteChannel(pid::Integer=myid())
```

Create remote channel

Parallel computing constructs in Julia

Julia Tasks

- Useful for concurrency
- “Appear” as multiple threads
- In Julia all executed on one system thread
- Not hugely useful for parallel speedup
- Uses Tasks (Coroutines) with Channels to communicate

Native Threads

- Somewhat minimal support in Julia (still experimental)
- @thread and @distributed
- Fork-join approach
- OpenMP style loop parallelisation

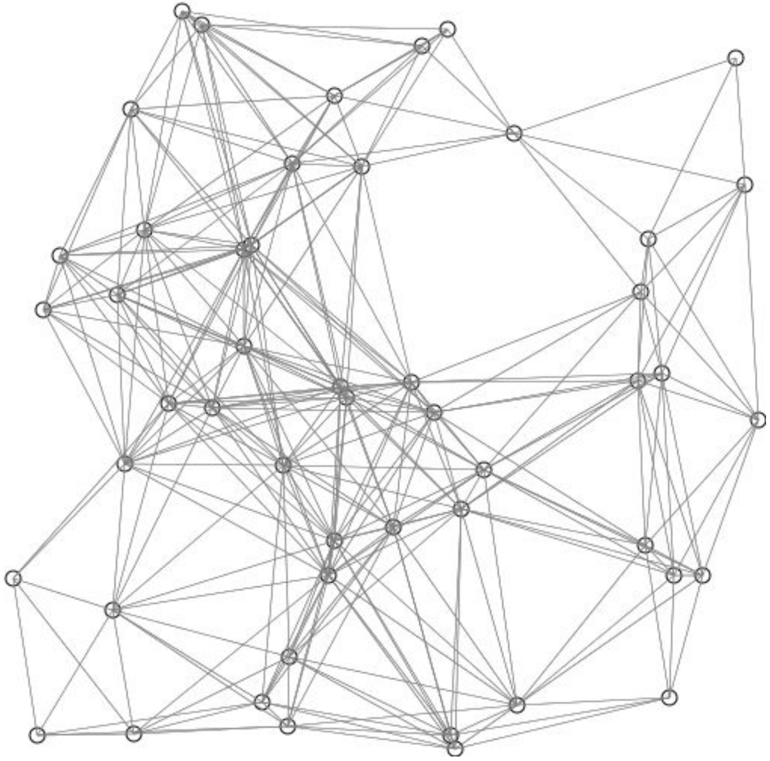
Processes

- Expensive and heavyweight
- Message passing
- Coarse grain granularity

Graph algorithms

Algorithms that solve a variety of problems that exist within graph theory. Categories include:

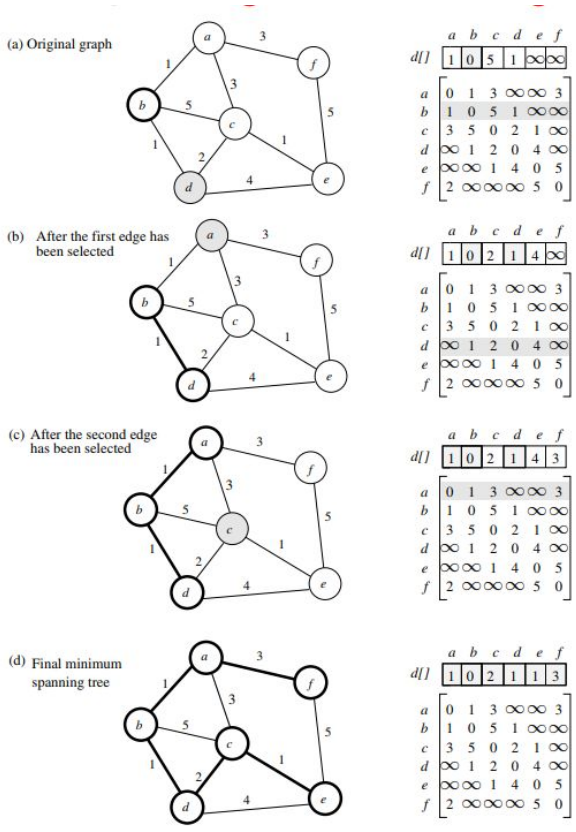
- Minimal Spanning Tree
- Graph Traversal/Searching
- Shortest Path
- Maximal Independent Subset
- Transitive Closure



https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:DijkstraDemo.gif

Minimum Spanning Tree

- A subset of the edges
  - In a graph that is connected, edge-weighted and undirected.
  - No cycles
  - Minimal possible total edge weight
- Inherently Sequential
  - Most MST algorithms “grow” the spanning tree
  - Greedy algorithms
- Somewhat parallelizable
  - The selection process of candidate nodes can be parallelized
  - Distance matrix can be partitioned between processors
- Overhead considerations
  - Alternative methods of speedup may be more optimal, (Binary heap adjacency list etc.)



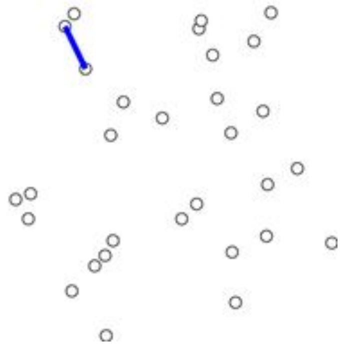
Prims Algorithm

A. Grama, A. Gupta, G. Karypis, and V. Kumar, “Graph Algorithms.”

Prim's Algorithm

A greedy algorithm for finding a minimum spanning tree in an undirected graph.

- Choose a vertex and build tree around it
- Building of tree is mostly sequential
- We can parallelise the process of selecting the shortest edge.
- Time complexity depends on data structure used

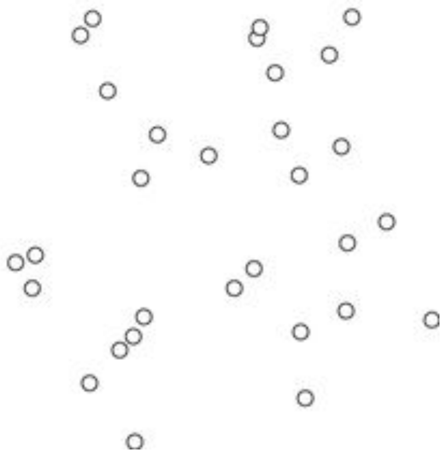


https://en.wikipedia.org/wiki/Prim%27s_algorithm#/media/File:PrimAlgDemo.gif

Kruskal's Algorithm

Another greedy algorithm to find a minimum spanning tree.

- Similar to Prim's but selects an edge instead of a vertex
- Better for sparse graphs
- Next selected edge not necessarily connected to previous edges.
- Also inherently a sequential algorithm

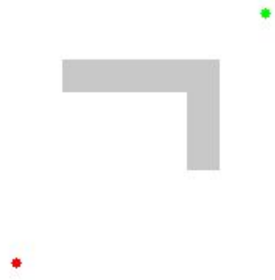


https://en.wikipedia.org/wiki/Kruskal%27s_algorithm#/media/File:KruskalDemo.gif

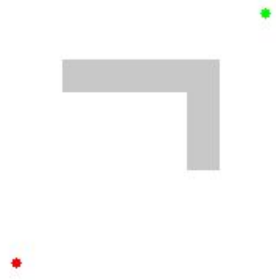
Traversing / Path finding

Search throughout a graph, starting from a source vertex. Includes finding shortest paths.

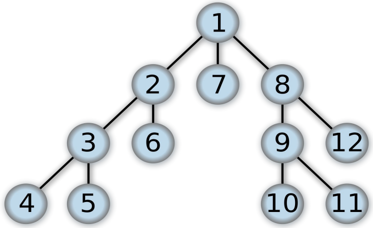
- Each node visited exactly once
- Examples include:
  - A*
  - Dijkstra's
  - BFS
  - DFS



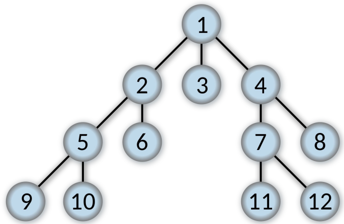
Finding a shortest path with A*



Finding a shortest path with Dijkstra's



Order nodes are visited in DFS



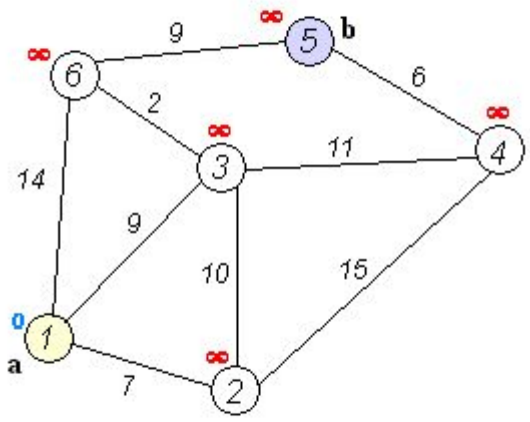
Order nodes are visited in BFS

https://upload.wikimedia.org/wikipedia/commons/thumb/3/33/Breadth-first-tree.svg/300px-Breadth-first-tree.svg.png
https://upload.wikimedia.org/wikipedia/commons/thumb/1/1f/Depth-first-tree.svg/300px-Depth-first-tree.svg.png
https://upload.wikimedia.org/wikipedia/commons/2/23/Dijkstras_progress_animation.gif
https://upload.wikimedia.org/wikipedia/commons/5/5d/Astar_progress_animation.gif

Dijkstra's Algorithm

One of the most well known graph algorithms.

- Published in 1959
- Finds the shortest path to nodes in a graph from a given source node, producing a shortest path tree.
- Parallelizable in a way similar to Prims and Kruskal:
  - After a node is visited the adjacency matrix can be partitioned amongst the available processors n.
  - Each processor seeks to find a local minimum next node
  - Reduction is then done to find global minimum for next iteration
- Time complexity of O(E log V) with help of a binary heap

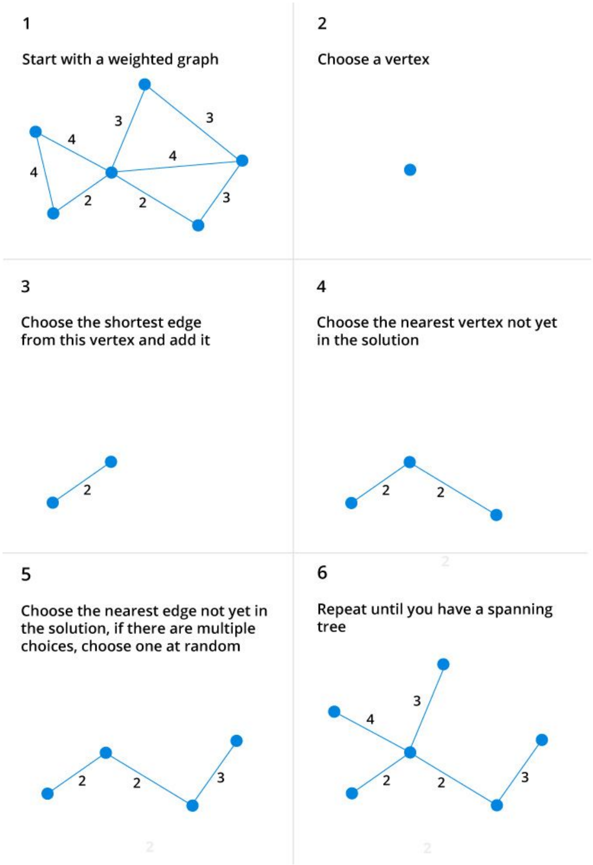


https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

Prims in parallel

While the process of building the tree is sequential, we can parallelize the process of choosing the closest node.

- The distance array is partitioned amongst the available threads in a distributed for loop
- Each processor finds minimum distance in their array partition
- A reduction then occurs to find the globally closest node as the next node for the tree
- Updating the distance array after a node is also be made parallel



"Prims Algorithm," *Prim's Algorithm*. [Online]. Available: https://www.programiz.com/dsa/prim-algorithm. [Accessed: 13-May-2019].

Dijkstra's all-sources in parallel

Source Partitioning

- Use p processors, distribute the vertices between the processors.

- Each processor sequentially executes a single source Dijkstra algorithm on its allocated vertices

- Can only use as many processors as vertices in the graph

- Lower overhead

Source Parallel

- Used if we have more processors than vertices

- P processors split into n nodes

- n/p processors working on each node

- Greater exploitation of parallelism

Initial Benchmarks

Run Time

- Is a parallel solution faster than a sequential one?
- What types of graphs are better suited for a parallel approach?

Efficiency

- How effectively does our solution use additional resources?

Dataset

- Randomly generated graphs, both sparse and dense

Machine 1 (Toshiba Portégé A600):

- Intel Core 2 Duo SU9300 / 1.2 GHz
- 2 Cores
- 2GB RAM
- Lubuntu 18.10 64bit

Machine 2 (Acer Aspire s7)

- Intel Core i7-5500U / 2.4Ghz up to 3Ghz
- 2 Cores with HyperThreading (4 Threads)
- 8GB RAM
- Ubuntu 18.04 64bit

Dijkstra all-sources

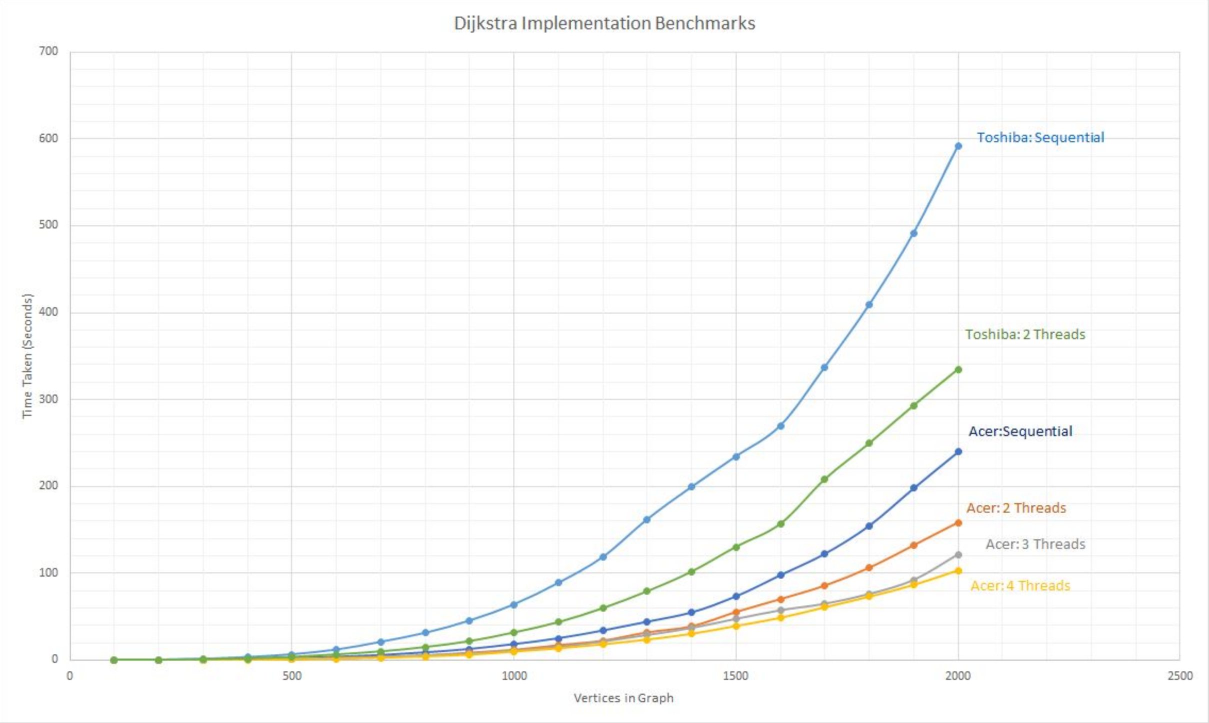
Source Partitioned all-sources Dijkstra implementation

- Starting vertices are distributed between available threads. Used @threads macro

- Minimal inter-process communication

- Decent speedup across the board, (relatively efficient)

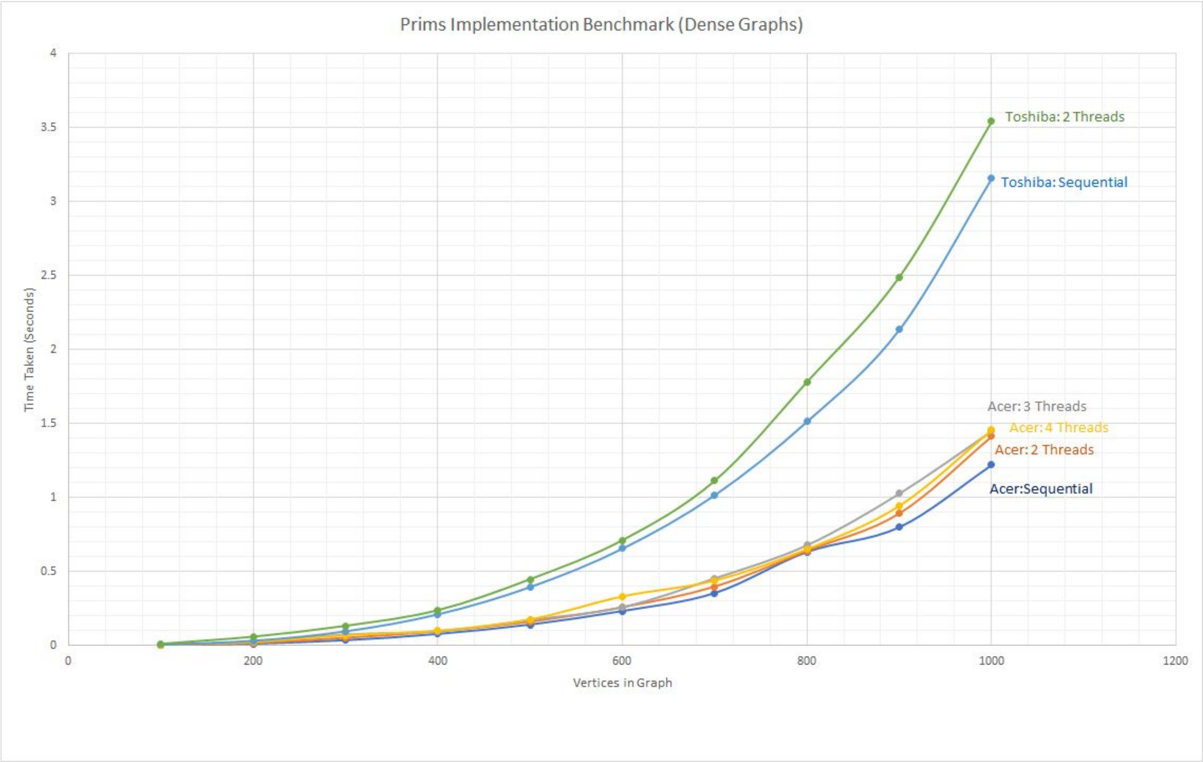
- Next step: source parallel implementation



Prims

Parallelisation becomes much more viable when the problem each thread has to solve is bigger

- Sparse graphs tended to suffer very heavily from overhead
- With more neighbouring nodes to process, the proportion of useful work to overhead increased



Next Steps

- More extensive benchmarking suite
  - SNAP Dataset
  - Benchmark.jl

- Algorithm improvements
  - Further Speedups
  - Data structure improvements
  - More complex algorithms (Maximal Independent set, Luby's)

Any questions?