Parallelisation of Graph Algorithms

in Julia

Speaking Order

1. Luke Thompson
2. Darcy Cox
3. George Qiao

Group 5

OVERVIEW

PARALLEL COMPUTING IN JULIA

PARALLELISATION OF GRAPH ALGORITHMS

PARALLEL GRAPH IMPLEMENTATION & BENCHMARKING

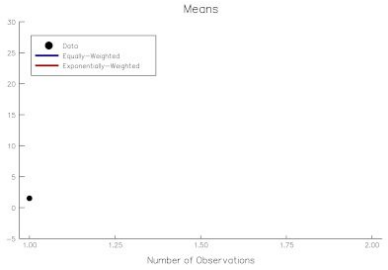
Julia Language

What is Julia?

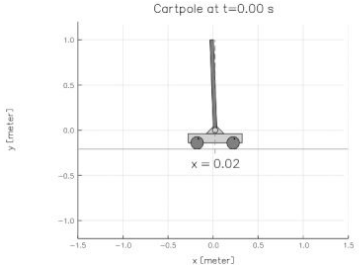
- High Performance
- Compiles to Native Code
- Dynamically Typed
- High Level Expressibility
- Parallel Computing Support

Supported Parallelism

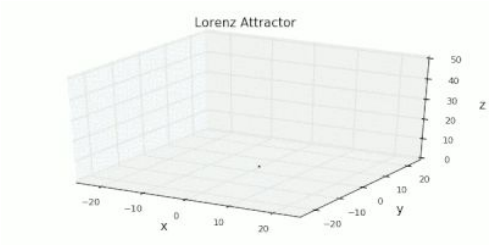
- Julia Tasks (coroutines)
- Multi-Threading
- Multi-Core / Distributed Processing



[1]



[2]



[3]

PARALLEL COMPUTING IN JULIA

Julia Tasks

Task Lifecycle

Key Features

- Suspend and Resume
- Query Task State
- Wait on Completion
- **But**, technically not parallelisation as tasks are multiplexed in single OS thread

```
# verbose method
julia> example() = print("Julia Tasks!")

julia> mytask = Task(example)

julia> istaskscheduled(mytask)
false

julia> schedule(mytask)
Julia Tasks!

julia> istaskdone(mytask)
true

# shorthand method
julia> mytask = @task print("Julia Tasks!")

julia> schedule(mytask)
Julia Tasks!
```

Key Functions

- `schedule(task)` #schedule a task
- `sleep(seconds)` #sleep current task
- `notify(condition, val, all, error)` #wake up tasks waiting on condition passing val
- `yield(task, args)` #immediately schedules task (then calls scheduler)

Julia Tasks

Inter-Task Communication

Channels

- Waitable FIFO queue
- Multiple readers and writers concurrently
- `put!` blocks when channel is full
- `take!` blocks when channel is empty
- Maximum size argument `Channel(`*`size`*`)`

```
c1 = Channel(32) # input data
c2 = Channel(32) # output data

[...] # prepare input data

function foo()
    while true
        data = take!(c1)
        [...]                # process data
        put!(c2, result)     # write out result
    end
end
```

Key Functions

- `put!(`*`channel, value`*`)#adds to queue`
- `take!(`*`channel`*`) #get and remove first item from queue (equivalent to pop() in Java)`
- `fetch(`*`channel`*`) #get first item from queue (equivalent to peek() in Java)`
- `bind(`*`channel, task`*`) #link lifecycle of channel with task`

Multi-Threading

Multi-Threading Functions

Overview

- Shared memory parallelism
- Allows compute bound tasks to execute in parallel
- Execute `for` loop iterations simultaneously
- Uses block allocation of iteration space
- **But**, the number of julia threads must be defined prior to start up using `'export JULIA_NUM_THREADS=4'`

Key Functions

- `@threads #indicate multi-threaded region`
- `nthreads() #number of threads available`
- `threadid() #id if executing thread`

```
julia> using Base.Threads

julia> nthreads()
4

julia> a = zeros(10)
#output omitted

julia> @threads for i = 1:10
            a[i] = threadid()
       end

julia> a
10-element Array{Float64,1}:
 1.0
 1.0
 1.0
 2.0
 2.0
 2.0
 3.0
 3.0
 4.0
 4.0
```

Multi-Threading

Atomic Operations

Overview

- Thread-safe way to avoid race conditions
- Wraps values and provides methods
- Create using `Atomic{T}(val)`
- Wrapped value **must** be a primitive type

Key Functions

- `atomic_add!(x, val)` `#add val to x`
- `atomic_sub!(x, val)` `#subtract val from x`
- `atomic_and!(x, val)` `#bitwise and of x and val`
- `atomic_or!(x, val)` `#bitwise or of x and val`
- `atomic_cas!(x, old, new)` `#compare and set x`
- `atomic_xchg!(x, new)` `#exchange value in x`

```
julia> using Base.Threads

julia> x = Atomic{Int}(2)
Atomic{Int64}(2)

julia> atomic_add!(x, 10)
2

julia> x
Atomic{Int64}(12)
```

Multi-Core / Distributed Processing

Workers and Local vs Remote

Workers

- All processes for parallel execution (except process 1) are referred to workers
- We define the number of workers on startup using `` `./julia -p n` `` where n is #workers
- Workers can be added / removed on the fly using `addprocs(num)` and `rmprocs(num)`

Paradigm

- Modified version of the message passing paradigm
- One side message passing - programmer explicitly manages one of the two processors

Local vs Remote

- **Remote References** are references which refer to an object located on another processor.
- **Remote Calls** are requests by a processor to call some function on another processor. This returns a remote reference (Future) which can be used to `wait()` on, or `fetch()` the result from.
- Remote calls return immediately and the calling processor proceeds to its next operation. However, if you need to wait on the result, you can use `wait()` to block until completion of the remote call.

Multi-Core / Distributed Processing

Code Availability, Spawning, Sync and Async

Code Availability

- Code must be available on any processor that runs it
- We use `@everywhere` in the include call to make it available to all processors
- This does **not** bring it into scope

```
julia> @everywhere include("Test.jl")
loaded
      From worker 3:    loaded
      From worker 2:    loaded

julia> test = @spawn myid()
Future(2, 1, 5, nothing)

julia> test1 = @spawnat 2 myid()
Future(2, 1, 3, nothing)
```

Spawning

- `@spawn` *`expr`* `# run` *`exp`* `expression on an automatically chosen processor`
- `@spawnat` *`proc expr`* `# run expression on` *`proc`* `processor`

Sync & Async

- `@async` *`task`* `# add task to local machine scheduler`
- `@sync` *`task`* `# blocks until completion of all @async, @spawn, @spawnat, @distributed`

Multi-Core / Distributed Processing

Parallel Maps and Loops

Parallelising Loops

- Loops can be parallelised on any number of processors with the `@distributed` macro
- This can optionally carry out a reduction based on the specified reduction. i.e. `@distributed (`*`reduction`*`)`

```
# count num heads
num = @distributed (+) for i = 1:20000
    Int(rand(Bool))
end
```

Shared Arrays

- Parallelising loops does not work as intended when you need to work with shared arrays since each processor gets its own copy
- To solve this, we use a `SharedArray` which maps into shared memory
- `SharedArray{T,N} # array of type T and size N`

```
using SharedArrays

a = SharedArray{Float64}(10)
@distributed for i = 1:10
    a[i] = i
end
```

PARALLELISATION OF GRAPH ALGORITHMS

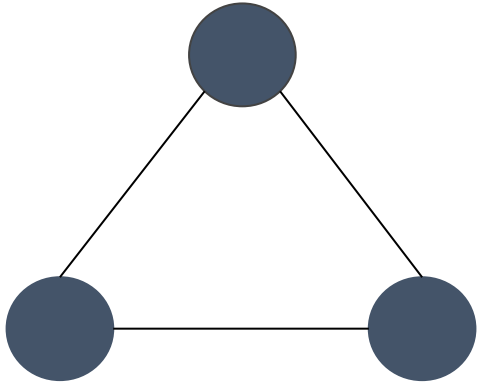
Parallelising Graph Algorithms

Graph applications

- Social networks
- Road networks
- Communication networks

Why parallelise?

- Very useful algorithms
- Commonly run with huge amounts of data
- Gain **faster runtimes**



A **graph** is a set of nodes and a set of edges

Parallelising Graph Algorithms

Common graph algorithms

- Dijkstra - shortest path
- Breadth-first search (BFS) - graph traversal
- Depth-first search (DFS) - graph traversal
- Prim’s algorithm - minimum spanning tree
- Many more

Selected for parallelisation

- Prim’s algorithm
- Breadth-first search

Prim's Algorithm - Sequential

Given a weighted graph, find its minimum spanning tree

```
procedure PRIM_MST(V, E, w, r)
begin
    V_T := {r};
    d[r] := 0;
    for all v ∈ (V − V_T) do
        if edge (r, v) exists set d[v] := w(r, v);
        else set d[v] := ∞;
    while V_T ≠ V do
    begin
        find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
        V_T := V_T ∪ {u};
        for all v ∈ (V − V_T) do
            d[v] := min{d[v], w(u, v)};
    endwhile
end PRIM_MST
```

Algorithm Features:

- **Builds MST incrementally**
  Adds one new node each iteration..
- **Distance vector**
  d[v] represents edge with smallest weight of those ending at node v and originating from a node in MST.
- **Greedy approach**
  At each iteration adds the minimal edge to the MST.

Prim's Algorithm - Sequential

Given a weighted graph, find its minimum spanning tree

```
procedure PRIM_MST(V, E, w, r)
begin
    V_T := {r};
    d[r] := 0;
    for all v ∈ (V − V_T) do
        if edge (r, v) exists set d[v] := w(r, v);
        else set d[v] := ∞;
    while V_T ≠ V do
    begin
        find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
        V_T := V_T ∪ {u};
        for all v ∈ (V − V_T) do
            d[v] := min{d[v], w(u, v)};
    endwhile
end PRIM_MST
```

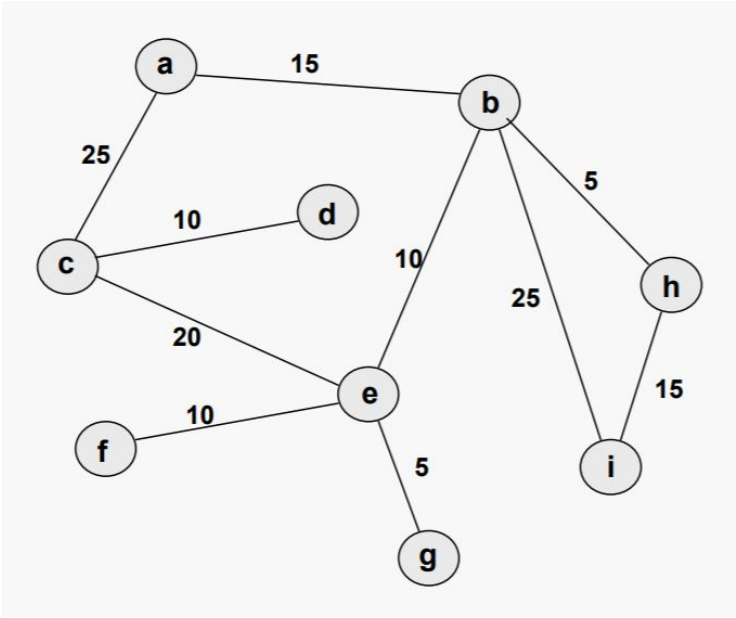
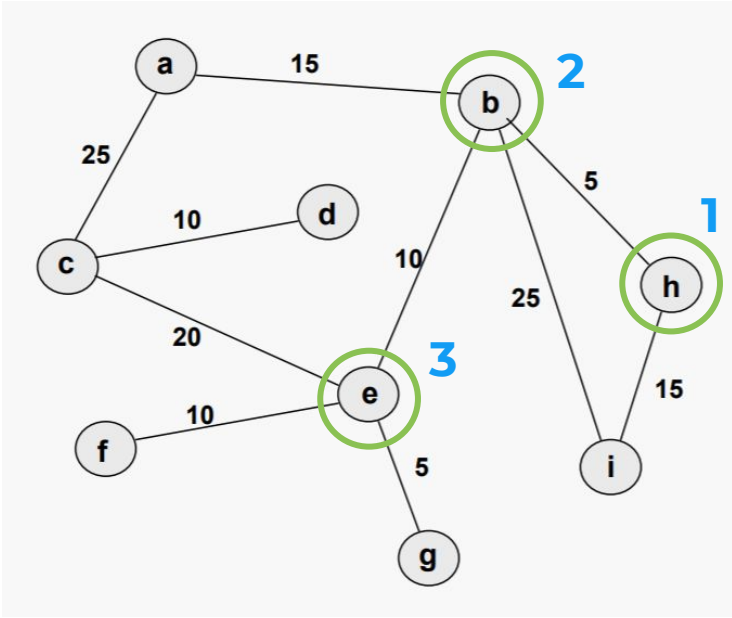


Image retrieved from: http://courses.cs.vt.edu/~cs3114/Fall10/Notes/T22.WeightedGraphs.pdf

Prim's Algorithm - Sequential

Given a weighted graph, find its minimum spanning tree

```
procedure PRIM_MST(V, E, w, r)
begin
    V_T := {r};
    d[r] := 0;
    for all v ∈ (V − V_T) do
        if edge (r, v) exists set d[v] := w(r, v);
        else set d[v] := ∞;
    while V_T ≠ V do
    begin
        find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
        V_T := V_T ∪ {u};
        for all v ∈ (V − V_T) do
            d[v] := min{d[v], w(u, v)};
    endwhile
end PRIM_MST
```



Prim's Algorithm - Parallel

Given a weighted graph, find its minimum spanning tree

```
procedure PRIM_MST(V, E, w, r)
begin
    V_T := {r};
    d[r] := 0;
    for all v ∈ (V − V_T) do
        if edge (r, v) exists set d[v] := w(r, v);
        else set d[v] := ∞;
    while V_T ≠ V do
    begin
        find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
        V_T := V_T ∪ {u};
        for all v ∈ (V − V_T) do
            d[v] := min{d[v], w(u, v)};
    endwhile
end PRIM_MST
```

Outer loop difficult to parallelise.

Inner steps are parallelisable

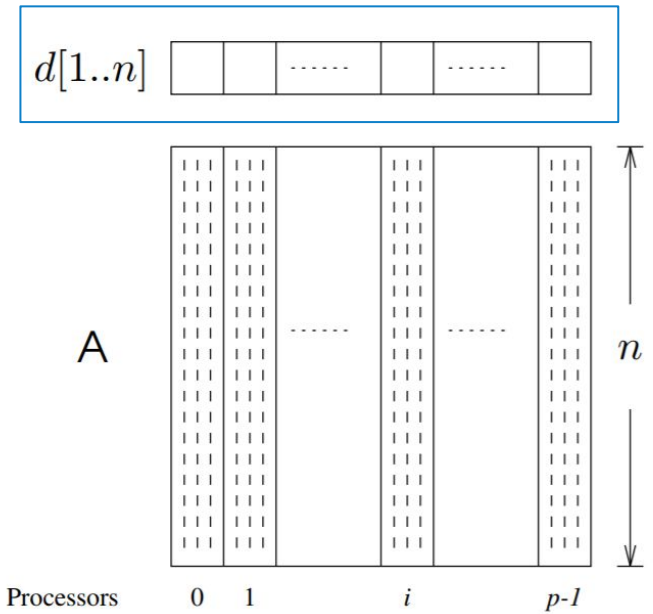
Prim's Algorithm - Parallel

Data partitioning

- Assumes adjacency matrix graph representation
- Split up matrix and distance vector into chunks
- Each processor handles its own distance vector and adjacency matrix partition.

Inner loop steps

- Each processor calculates local minimum edge based on their respective distance vectors.
- Local minima are "reduced" to a global minimum
- The global minimum is added to the MST.
- Update distance vector by looping over the row of the added node, parallelising on columns.



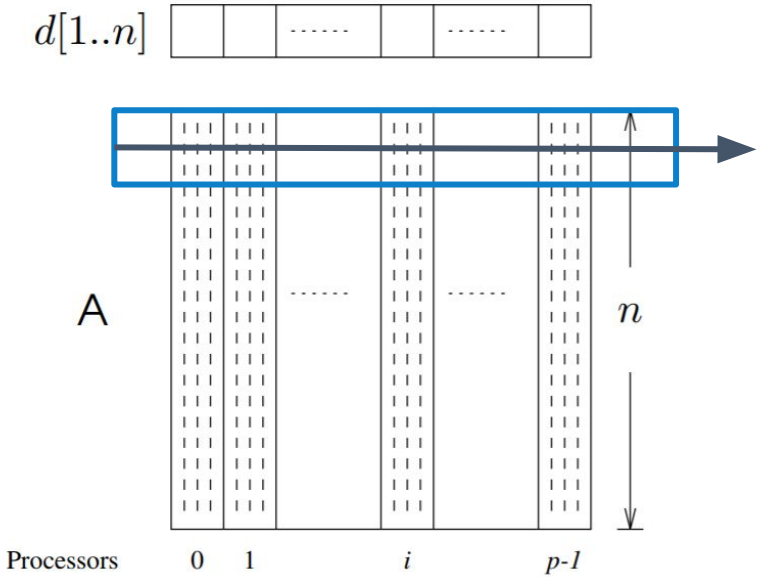
Prim's Algorithm - Parallel

Data partitioning

- Assumes adjacency matrix graph representation
- Split up matrix and distance vector into chunks
- Each processor maintains its own distance vector partition.

Inner loop steps

- Each processor calculates local minimum edge based on their respective distance vectors.
- Local minima are "reduced" to a global minimum
- The global minimum is added to the MST.
- Update distance vector by looping over the row of the added node, parallelising on columns.



Breadth-First Search - Sequential

Given a graph and a start node, visit every node reachable from the start node

High-level Approach

Visit each node **layer by layer:**

- Visit the source node to begin
- As we visit a new node, mark all its neighbours as "to visit".
- Visit all nodes marked "to visit" until there are none remaining

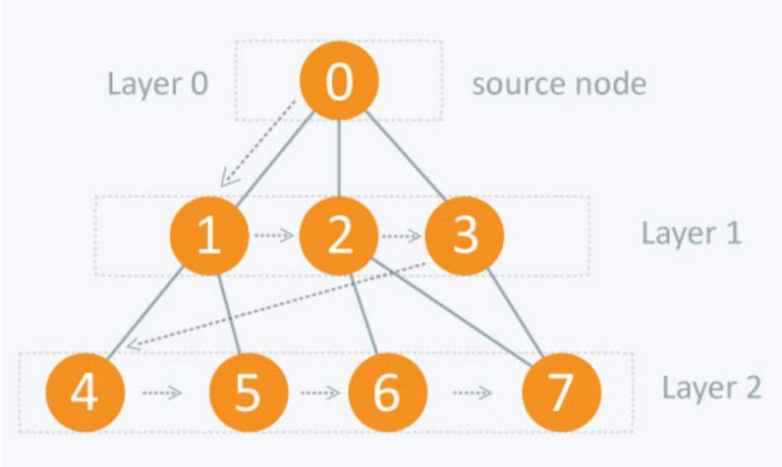


Image retrieved from: https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

Breadth-First Search - Sequential

Given a graph and a start node, visit every node reachable from the start node

Implementation

- Boolean array to track visited / discovered nodes
- Collections for the current level, and the next level.
- Alternatively use a single queue - harder to parallelise

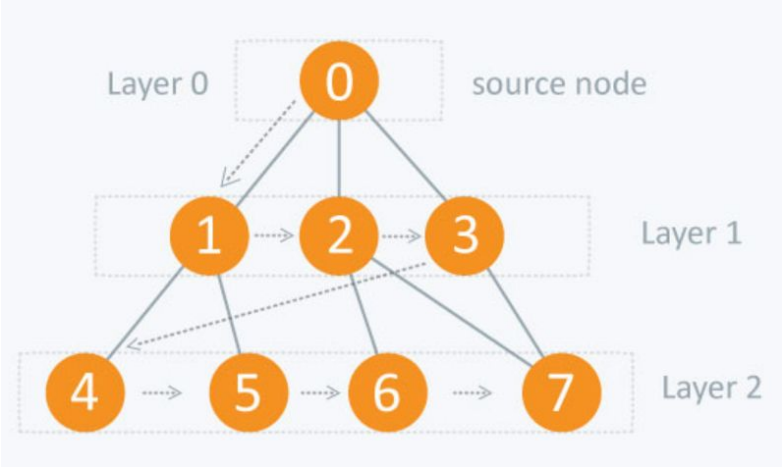
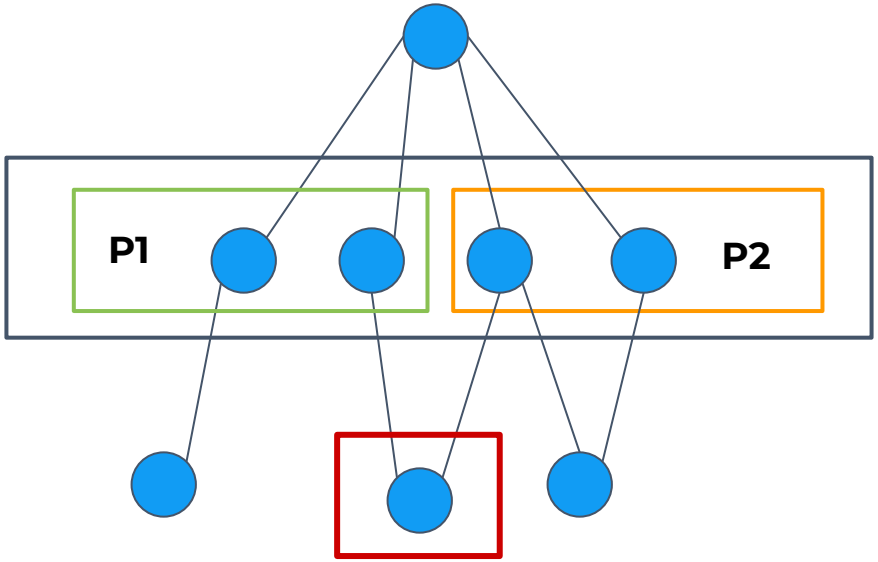


Image retrieved from: https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

Breadth-First Search - Parallel

Approach

- Like prim's, can't parallelise the "outer loop"
- Parallelise on each level:
  - Split level into p equal-sized chunks
  - Each processor visits each node in its own chunk
  - Possible contention adding to next level collection and checking visited array.



Both P1 and P2:

- If neighbour is not discovered
- Set to discovered
- Add to next level

PARALLEL GRAPH IMPLEMENTATION & BENCHMARKING

Benchmarking in Julia

BenchmarkTools

- @benchmark
- @btime
- @belapsed

@benchmark

- Specify parameters
- Returns a Trial object

```
julia> @benchmark rand() samples=1000
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     4.908 ns (0.00% GC)
  median time:      5.664 ns (0.00% GC)
  mean time:        6.545 ns (0.00% GC)
  maximum time:     156.695 ns (0.00% GC)
  --------------
  samples:          1000
  evals/sample:     1000
```

Benchmarking in Julia

Statistics

- minimum(Array arr)
- median(Array arr)
- mean(Array arr)
- maximum(Array arr)

```
julia> benchmark = @benchmark rand() samples=1000
…

julia> dump(benchmark)
BenchmarkTools.Trial
  params: BenchmarkTools.Parameters
      seconds: Float64 5.0
      samples: Int64 1000
      evals: Int64 1000
      overhead: Float64 0.0
      gctrial: Bool true
      gcsample: Bool false
      time_tolerance: Float64 0.05
      memory_tolerance: Float64 0.01
  times: Array{Float64}((1000,)) [4.909 … 30.206]
  gctimes: Array{Float64}((1000,)) [0.0 …  0.0]
  memory: Int64 0
  allocs: Int64 0

julia> println(median(benchmark))
TrialEstimate(6.041 ns)
```

Benchmarking in Julia

Setup

- 1000 samples
- 120 second timeout
- A set of randomly generated graph of various sizes

CPU

- Intel i5-4570 @ 3.20GHz
- Can be pushed to 3.60GHz
- 4 physical cores and threads

Benchmarking in Julia

Nested Loop

```
cheapest, index

for each node already visited
  for every other node
    if edge is smaller than cheapest
      update cheapest and index
    end if
  end for
end for
```

Sequential

- 1000 node graph
- 625 seconds

Parallel

- 1000 node graph
- 25 seconds

Benchmarking in Julia

Parallel Distance Vector Implementation

cheapestNode(distanceVector, mst)

```
addprocs(n)

...

cheapest = @distributed min for i = 1:length(distanceVector)
    if node not in mst
        ...
        calculate cheapest node
        ...
    end
    (minCost, nodeIndex)
end

return cheapest[2]
```

Benchmarking in Julia

Parallel Distance Vector Implementation

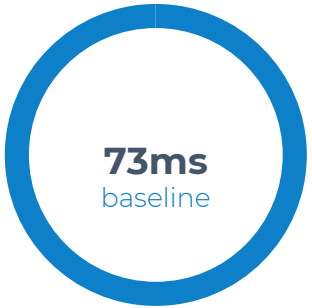
updateVector(...)

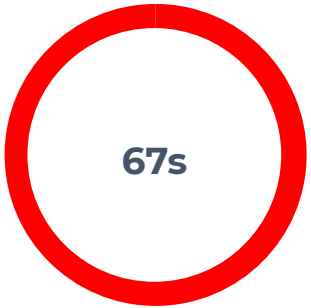
```
addprocs(n)

...

@sync @distributed for i = 1:length(distanceVector)
    if new node edge cost cheaper
        update distance
    end
end
```

Preliminary Results





Sequential

Distance Vector with set of traversed nodes

Parallel

Parallelised version using traversed nodes

Benchmarking in Julia

Parallel Distance Vector (with set of nodes) Implementation

cheapestNode(distanceVector, remaingingNodes)

```
addprocs(n)

...

cheapest = @distributed min for i in collect(remaingingNodes)
    (distanceVector[i], i)
end

return cheapest[2]
```

Preliminary Results

Sequential v2

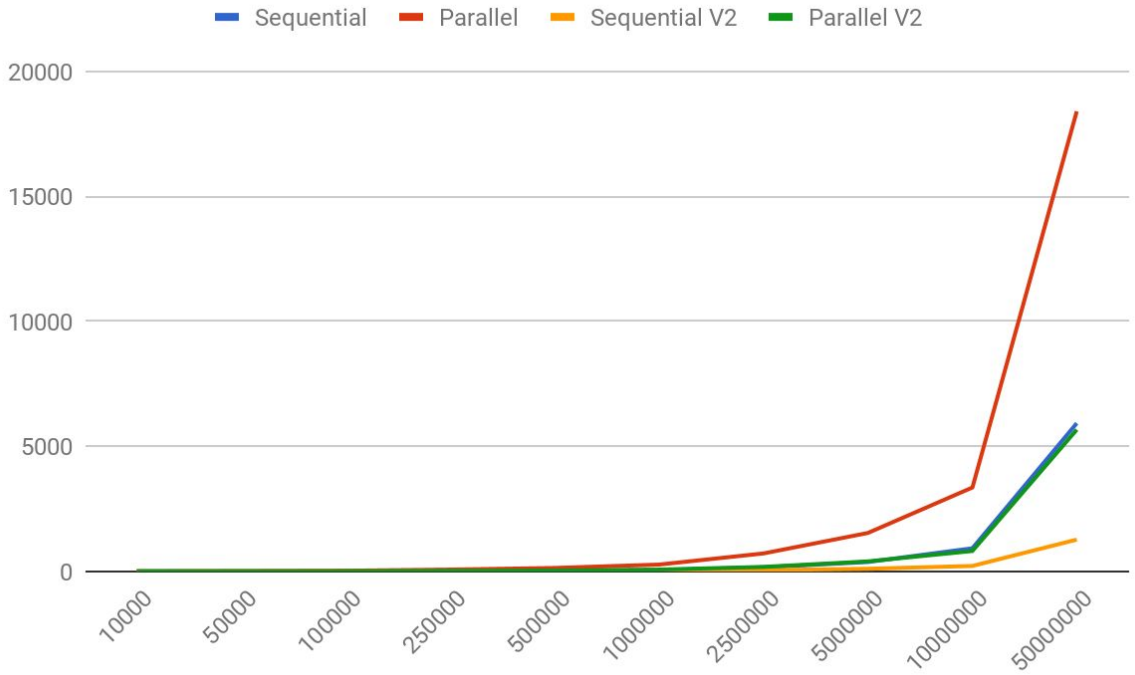
Distance Vector with set of remaining nodes

Parallel v2

Parallelised version using remaining nodes

Preliminary Results

Cheapest Node Function



Future Work

- Further investigate and optimise parallelisation methods
- Implement BFS algorithms
- Investigate more complex graph algorithms
- Compare performance against other languages

THANK YOU

Questions?

REFERENCES

[1] Julia. (2016). *Machine learning* [Screenshot]. Retrieved from https://julialang.org/
[2] Julia. (2016). *Data science* [Screenshot]. Retrieved from https://julialang.org/
[3] Julia. (2016). *Scientific Computing* [Screenshot]. Retrieved from https://julialang.org/