PARALLEL PROCESSING OF GRAPHS

RANJAN SEN
Figure 1.9  Clockwise from top left: a co-occurrence network, ARPA network 1974, London Tube network, and electrical grid.
Graphs and Graph Algorithms

- Euler cycles - Konigsberg bridge problem, Euler 1736
- Many books and research publications
- Drawing N-cube (1974)
- Appel, K. & Haken, W. 4-color theorem (1976)

Semantic Networks (Organize Knowledge)
Vertices: Users, Posts/images
Edges: Social Relationships - Likes (directed - Twitter, undirected - Facebook)

Semantic Networks (Organize Knowledge)
Vertices: Subject, Object
Edges: Predicates
Google Knowledge Graph (570 M, 18 B)

Biological Networks (Protein-Protein Interaction Networks - Interactomes)
Vertices: Protein
Edges: Interactions
Google Knowledge Graph (570 M, 18 B)

Social Networks
Vertices: Users, Posts/images
Edges: Social Relationships - Likes (directed - Twitter, undirected - Facebook)

Graphs and Graph Algorithms

- Euler cycles - Konigsberg bridge problem, Euler 1736
- Many books and research publications
- Drawing N-cube (1974)
- Appel, K. & Haken, W. 4-color theorem (1976)

Properties of graphs that are very large - parallel processing essential for quick response

Complex patterns of parallelism and diversity of graph structures
Graph Algorithms

Identifying leaders

PageRank (Centrality Measures)

Predicting Behavior

Label Propagation (Structured Prediction)

Recommending Products

Finding communities

Counting triangles

Shortest paths

Connected components
Programming with Elixir

Concurrent Oriented Programming (COP)

- lightweight processes & messages
- actor model
- functional programming
- supervisors and workers
  - Elixir - built on Erlang/OTP (2004), ...
    - Nx for numerical applications (2021), Axon - ML library
  - Erlang - Nine 9’s availability switches (1983)
    - YAWS - Erlang web server,
  - OTP - programming patterns

long history of parallel programming
- special way to program
- architecture dependency
- scientific computations
- different from sequential programming

Concurrency subsumes
- parallelism
- Fault-tolerance
- distribution
- scalability
Writing Graph Algorithms in Elixir

Growing community
Common coding language
Vibrant user community
Wide application in web app development

Language features
Actor model
Functional programming
Fault-tolerance
Distributed computing

process & message
262144 max default lightweight processes
Erlang VM (BEAM)

match & functions
pattern matching
first class functions

behaviors & patterns
similar to interface, expresses patterns
adds OTP compliant system development
Comers BFS

BFS(G, s)

1. for each vertex \( u \in G.V - \{s\} \)
2. \( u.\text{color} = \text{WHITE} \)
3. \( u.d = \infty \)
4. \( u.\pi = \text{NIL} \)
5. \( s.\text{color} = \text{GRAY} \)
6. \( s.d = 0 \)
7. \( s.\pi = \text{NIL} \)
8. \( Q = \emptyset \)
9. Enqueue\((Q, s)\)
10. while \( Q \neq \emptyset \)
11. \( u = \text{DEQUEUE}(Q) \)
12. for each \( v \in G.Adj[u] \)
13. \( \text{if } v.\text{color} == \text{WHITE} \)
14. \( v.\text{color} = \text{GRAY} \)
15. \( v.d = u.d + 1 \)
16. \( v.\pi = u \)
17. Enqueue\((Q, v)\)
18. \( u.\text{color} = \text{BLACK} \)

GraphSearch.bfs(adj_l, :s)
c(“graph_search.ex”)
c(“queue.ex”)

adj_l =
%
  r: [:s, :v],
  s: [:r, :w],
  t: [:w, :x, :u],
  u: [:t, :y],
  v: [:r],
  w: [:s, :t, :x],
  x: [:w, :t, :y],
  y: [:x, :u]
libgraph is an Elixir library for graphs

Modules: Graph, Graph.Edge, Graph.Partitioning .... PriorityQueue

Graph defines a graph data structure for directed and undirected graphs. It defines API for creating, manipulating, and querying the structure

‣ The Graph struct (Elixir struct) - has a map of vertex ids to vertices (vertices), a map of vertex ids to their out neighbors (out_edges), a map of vertex ids to their in neighbors (in_edges), a map of vertex ids to vertex labels (vertex_labels)
‣ a map of edge ids ({{vertex_id, vertex_id}}) to a map of edge metadata (edges)
‣ edge metadata is a map of label => weight, and each entry in that map represents a distinct edge (this allows multiple edges in the same direction between same edge pairs)

iex(2)> g = Graph.new(type: :undirected) |> Graph.add_edges([{:a, :b}, {:b, :c}, {:c, :d}, {:b, :d}])
#Graph<type: undirected, vertices: [:a, :b, :c, :d], edges: [:a <-> :b, :b <-> :c, :b <-> :d, :c <-> :d]>

iex(5)> Graph.info(g)
%{num_edges: 4, num_vertices: 4, size_in_bytes: 1688, type: :undirected}

iex(4)> Graph.get_shortest_path(g, :a, :d)
[:a, :b, :d]

iex(3)> cl = Graph.cliques(g)
[[:b, :c, :d], [:b, :a]]
Bron Kerbosch algorithm for maximal cliques using libgraph

BronKerbosch\((R, P, X)\):

\[\begin{align*}
&\text{if } P \text{ and } X \text{ are both empty:} \\
&\quad \text{report } R \text{ as a maximal clique} \\
&\text{for each vertex } v \text{ in } P:\n
&\quad \text{BronKerbosch}(R \cup \{v\}, P \cap N(v), X \cap N(v)) \\
&\quad P := P \setminus \{v\} \\
&\quad X := X \cup \{v\}
\end{align*}\]

\@doc 

```
implments the bron kerbosch with pivot (max degree vertex in p)
```

def bronKerbosch2(g, r, p, x) do
  if MapSet.size(p) == 0 and MapSet.size(x) == 0 do
    IO.puts("Found clique: #{inspect(r)} \
")
  else
    p=Enum.sort_by(p, &Graph.degree(g,&1), :desc) |> MapSet.new
    for v <- MapSet.to_list(p), x, reduce: {p,x} do
      {p,x} ->
      bronKerbosch2(g, MapSet.union(r, MapSet.new([v])),
      MapSet.intersection(p, MapSet.new(Graph.neighbors(g,v))),
      MapSet.intersection(x, MapSet.new(Graph.neighbors(g,v))))
      p= MapSet.difference(p, MapSet.new([v]))
      x= MapSet.union(x, MapSet.new([v]))
    end
  end
end
```

iex(1)> g = Graph.new(type: :undirected) |
...(1)> Graph.add_edges([{:a, :b}, {:a, :c}, {:a, :e}, {:b,:c}, {:b,:d},
...(1)> {:b,:f}, {:c,:d}, {:c, :f}, {:d, :e}, {:d, :f}])

iex(2)> BronKerbosch2.bronKerbosch2(g, MapSet.new(),
...(2)> MapSet.new([:a, :b, :c, :d, :e, :f]), MapSet.new())
Found clique: #MapSet<[:a, :b, :c]>
Found clique: #MapSet<[:a, :e]>
Found clique: #MapSet<[:b, :c, :d, :f]>
Found clique: #MapSet<[:d, :e]>
Available OTP compliant behaviors (interfaces)

- **Generic server** - pattern for client server
- **Generic supervisor** - pattern for tracing and error reporting
- **Generic event** - event handling pattern
- **Task, Agent, GenStage** - concurrency pattern, producer consumer pattern

- **Supervision Tree**
  - hierarchical process structure
  - started by a supervisor process
defmodule Parser do
  @doc "Parses a string"
  @callback parse(String.t):{:ok, term} | {:error, String.t}
  @doc "Lists all supported file extensions"
  @callback extensions() :: [String.t]
end

defmodule JSONParser do
  @behaviour Parser
  @impl Parser
  def parse(str), do: {:ok, "some json " <> str}
  @impl Parser
  def extensions, do: ["json"]
end

defmodule YAMLParser do
  @behaviour Parser
  @impl Parser
  def parse(str), do: {:ok, "some yaml " >> str}
  @impl Parser
  def extensions, do: ["yaml"]
end
Fault-tolerance in Elixir

Resilient signal processing in telcom - Erlang

nodes - machine

process - supervisors and workers

ports - communication

message communication

defmodule MyApp.Supervisor do
  use Task.Supervisor

  def start_link(init_arg) do
    Supervisor.start_link(__MODULE__, init_arg, name: __MODULE__)
  end

  @impl true
  def init(__init_arg) do
    children = [
      {Stack, [:hello]}
    ]

    Supervisor.init(children, strategy: :one_for_one)
  end
end
Trapping signals

Process.flag(:trap_exit, true)

```
iex(9)> spawn(fn ->
...(9)>           Process.flag(:trap_exit, true)
...(9)>           spawn_link(fn -> raise("Something went wrong") end)
...(9)>           receive do
...(9)>             msg -> IO.inspect(msg)
...(9)> end end)
#PID<0.190.0>
iex(10)>
12:36:32.195 [error] Process #PID<0.191.0> raised an exception
** (RuntimeError) Something went wrong
  (stdlib 3.13) erl_eval.erl:678: :erl_eval.do_apply/6
  {:EXIT, #PID<0.191.0>,
  %{RuntimeError{message: "Something went wrong"},
  [{:erl_eval, :do_apply, 6, [file: 'erl_eval.erl', line: 678]}]}}
```
Monitors

Uni-directional link

proc 1

monitor

{:DOWN, #Ref.., :process, PID, :noproc}

proc 2

iex(10)> target_pid = spawn(fn ->
        ...
        Process.sleep(1000)
        ...
      end)
#PID<0.195.0>

iex(11)> Process.monitor(target_pid)
#Reference<0.2879387170.3972005889.9679>

iex(12)> receive do
  ...
  msg -> IO.inspect(msg)
  ...
end
{:DOWN, #Reference<0.2879387170.3972005889.9679>, :process, #PID<0.195.0>, :noproc}

printed by the IO.inspect

returned value (receive)
Supervisors

A generic process that manages life cycle of other processes in a system. A supervisor can start other processes - which are its children. It uses links, monitors and exit traps.

Processes that are not supervisor are called workers.

Supervisor.start_link starts a supervisor - traps exits, starts child processes. It performs corrective actions: restart the worker ... etc.

Can be given by a list as argument or as a callback module.

Supervisors and workers are arranged in a Supervision Tree.
Parallel lines

Parallel Connection

Parallel code and execution
Graph Parallel Pattern

Pregel - The Vertex Program Abstraction

vertex programs interact by sending messages

Iterative Bulk synchronous executions

Partitioning - vertex-cut based followed by merge

GraphLab: split high degree vertices, run them on different machines

Gather Apply Scatter decomposition

2D partitioning

Application graph property and communication cost!
Distributed Programming in Elixir

A node is a system running the Erlang VM with a given name

location transparency

creating a cluster
pick one node as master
this is where we run the commands
it also acts as a worker

check connection: Node.list
run program on the single node
see it runs on other nodes

send many many requests to a
web site and calculate response
times, success/failures from
several workers

blitzy example:
load tester for a web site

iex(11)> Blitzy.Worker.start("hex.pm")
Running on #node-
nonode@nohost

[nonode@nohost-#PID<0.226.0>]
completed in 88.027 msecs
{:ok, 88.027}
Problem: Ring of processes distributed in multiple nodes - elect a unique leader

1. The message alphabet $M$ is exactly the set of UIDs
2. For each $i$, the states in $\text{states}(i)$ consist of the following
   1. $u$, a UID, initially $i$'s UID
   2. $\text{send}$, a UID or null, initially $i$’s UID
   3. $\text{status}$, with values in $\{\text{unknown, leader}\}$, initially unknown
3. The set of start states $\text{start}(i)$ consists of the single state
   1. for each $i$, the message-generation $\text{msgs}(i)$ is as follows
      send the current value of $\text{send}$ to process $i + 1$
4. For each $i$, the transition function $\text{trans}(i)$ is as follows
   1. $\text{send} := \text{null}$
   2. if the incoming message is $v$, a UID, then
      case
      $v > u$: $\text{send} := v$
      $v = u$: $\text{status} := \text{leader}$
      $v < u$: do nothing
   endcase
Computing architecture for parallel processing

Vector machines
Special purpose machines
Embedded computing
Cluster computing
Cloud computing
Multi-core computing

Graph processing libraries

Boost (PBGL), Pregel, Pegasus, GraphLab, PowerGraph, GraphX, Neo4j GDS

KnowledgeDiscovery Toolkit (KDT), GPS, Graph, Grace
Parallelism in graph algorithm is different from that in high performance computing

Data-centric abstraction is key to understanding parallelism in graph based applications

Large graphs: billions of nodes and trillions of edges - Web search engine, graph analytics

Complex patterns of parallelism in diverse dynamic graph structures (irregular structure)
Bridge analogy connecting users to a parallel IT industry (from Ref 2)
Analogy connecting computer users on the right to the IT industry on the left.

- emphasizing software architecture, autotuning and separate support for productivity vs. performance programming
- architecting parallel software with design patterns, not just parallel programming languages
  - graphical models -> DB, ML and Music
- split productive and efficiency layers: a composition and coordination language to glue together the libraries and programming frameworks produced by the efficiency-layer programming
Parallelism for Irregular structures - such as Graphs

- dependency graph - compute/control centric
- operator formulation - data centric
  - action of operators on graph data structures (similar to Dijkstra's "Program = Algorithms + Data Structures")
- amorphous data-parallelism
- structural analysis (Tao analysis):
  - abstraction of algorithms that distills out properties important for parallelization, hiding unnecessary detail.
  - compile time, inspector-executor or optimistic parallelization

Dense matrix computation << High Performance Computing

Irregular data structures (lists, trees, graphs)

- social network analysis: sparse graphs - betweenness centrality, vmaxflow are used to get next network properties
- Machine-Learning (survey propagation) based on message passing a factor graph (a sparse bipartite graph)
- Data-mining: k-means and agglomerative clustering operates on sets and multi-sets
- Simulations - event driven (discrete) over network nodes
- Optimizing compilers << iterative and elimination based data flow analysis on structures like inter-procedural control flow graphs
- Computational science: n-body methods are spatial decomposition trees, finite element methods use 2-D and 3D meshes produces using algorithms like Delaunay mesh generation and refinement

Boost and STAPL
Delaunay Mesh Refinement (DMR) - an irregular algorithm used in finite-element meshing and graphics.

Delaunay triangulation

Operator formulation:
algorithm -> action on data structure

Data-centric -> functional
Active elements: At each point during the execution of a graph algorithm, there are certain nodes or edges in the graph where computation might be performed - these nodes are called active elements
To process an active node, an operator is applied to it resulting an activity

Neighborhood: Performing an activity may require reading or writing other nodes and edges in the graph. The set of nodes and edges that are read or written while performing an activity is its neighborhood

Ordering: The activities may execute in an order (ordered and unordered algorithms)

In DMR the mesh is represented as nodes and edges, a bad triangle is an activity, a cavity is the neighborhood.

DMR can be performed in parallel since bad triangles whose cavities do not overlap can be refined in parallel in most irregular algorithms, dependences between loop iterations (activities) are functions of runtime values.

modeling don't care non-determinism not possible in dependence graphs - two activities can be in either order but not concurrently

Show Elixir demo Add an example and Galois
Algorithms to Programs

- wordlists to keep track of active nodes - do not express the don’t-care nondeterminism
  - Unordered-set Iterator (Galois):
    - foreach (e in Set S) {B(e)} The loop body B(e) is executed for each element e of S the order in which iterations execute is indeterminate
      - there may be dependencies between iterations and elements may be added to S
  - Ordered-set Iterator (Galois):
    - foreach (e in OrderedSet S) {B(e)} The loop body B(e) is executed for each element e of S the order in which iterations execute is determinate (chooses the minimum element)
      - there may be dependencies between iterations and elements may be added to S
    - break statement in the body of a Galois iterator allows early exits out of the iterator

Implementation of the concurrent data-structures in Elixir
- shared memory or distributed?
- Elixir Concurrency Oriented Programming!!

the operator is applied to the data-structure until there are no more active nodes
Amorphous Data-parallelism

Active elements and neighborhoods each is a site where a processor can perform computation subject to neighborhood constraints.

ADP is a generalization of data parallelism - (i) concurrent operations may conflict with each other,
(ii) activities can be created dynamically and (iii) activities may modify the underlying data structures

Baseline: speculative parallel execution: the graph is stored in shared memory and active nodes are processed by some number of threads. Neighborhoods are allowed to overlap if activities do not modify nodes and edges in the intersection of these neighborhoods

Parallelism profiles - available parallelism for the (irregular) algorithm for a given input
Available parallelism - number of active nodes that can be processed in parallel at a step of the algorithm for a given input,
(i) assuming that there is an unbounded number of processors,
(ii) an activity takes one unit time step to execute,
(iii) the system has perfect knowledge of neighborhood and ordering constraints and
(iv) a maximal set of non-conflicting activities is executed in parallel

Boruyka’s unordered MST algorithm
Prim’s ordered MST algorithm (shows more parallelism)
**Implementation:**

Elixir: active nodes are tasks and shared memory in ETS - no need for locks - may be extended to distributed memory

The Erlang Virtual Machine - BEAM scheduler will determine the order of task executions
it will use graph APIs to operate on the ADT to perform operations on graphs as needed
The neighborhood of an activity is determined at run time by the execution of the APIs that touches new nodes or edges in the graph

Overhead of the baseline can be reduced by examining structure:
- **Cautious** implementation of an operator - reads all the elements it is to operation on (used in DMR implementation - Galois has used this for other algorithms)

Scheduling strategy in baseline - autonomous scheduling
coordinated scheduling - only non-conflicting iterations are scheduled (compile-time, just-in-time, run-time) - based on construction of a dependency graph at different points during program compilation and execution

**Runtime coordination**
unordered data-driven algorithms (active node need to execute to know neighbors! new nodes are created)
topology-driven algorithms (no new nodes or edges created - Morph)
Software is the main problem in bridging the gap between users and the parallel IT industry.

The aggressive goal of the parallel revolution is to make it as easy to write programs that are as efficient, portable, and correct and that scale as the number of cores increases as it has been to write programs for sequential computers.

The goal is to find compelling applications that thrust for more computing than is currently available and absorb biennially increasing number of cores for the next decade or two.

Advances could be in, say, worst-case response time, battery life, reliability, or security. Greater end-user value from an increasing number of cores.
Parallel Graph programming with Elixir - an early attempt

1. Shared Memory multi-core (70%)
2. Fault-tolerance (10%)
3. Distributed (20%)
### Running parallel code in Elixir

<table>
<thead>
<tr>
<th>Lightweight processes and messages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Task</strong>: spawn, manage, shutdown a process from another process</td>
</tr>
<tr>
<td><strong>Task.async</strong> - starts a new process using a function (which becomes the body of the new process)</td>
</tr>
<tr>
<td><strong>Task.await</strong> - awaits for the rest returned by the task</td>
</tr>
<tr>
<td><strong>Task.yield</strong> - same as Task.await</td>
</tr>
</tbody>
</table>

#### Link and monitoring of processes:

- **Supervisor for Task**: The Task module implements the `child_spec/1` function for use by a supervisor.
- **Dynamic supervision of Tasks using Task.Supervisor**

### Erlang Table Store (ETS)

- **atomics**
- **Agent based state - client/server behavior**

### Code Snippet

```elixir
iex(1)> defmodule AsyncStreamTest do
iex(2)>     def expensive_fun(d) do
iex(3)>         Process.sleep(5050)  # more than default timeout of 5000 millisec
iex(4)>         d
iex(5)>     end
iex(6)> end

iex(2)> collection = Enum.to_list(1..10)

iex(3)> max_concurrency = System.schedulers_online() * 2

iex(4)> stream = Task.async_stream(collection, AsyncStreamTest, :expensive_fun, [], timeout: 6000, max_concurrency: max_concurrency) |> Enum.to_list end
```
Shared Memory in Elixir

Erlang Term Store (ETS)
Multiple process can access ETS as global store

Elixir Task module
Task.async_stream

Statically supervised tasks

Elixir Task module
Task.Supervisor submodule

Distributed tasks
Since Elixir provides a Task.Supervisor, it is easy to use one to dynamically start tasks across nodes:

```elixir
# On the remote node
task.Supervisor.start_link(name: MyApp.DistSupervisor)
# On the client
supervisor = {MyApp.DistSupervisor, :remoteLocal}
task.Supervisor.async(supervisor, MyMod, :my_fun, [arg1, arg2, arg3])
```

Shared Memory in Elixir

Atomic operations
Erlang atomics
Multiple process can access atomically operate on atomic array

```elixir
compare_exchange(Ref, Ix, Expected, Desired) -> ok | integer()

Types

Ref = atomics_ref()
Ix = Expected = Desired = integer()
```

Atomically compares the atomic with Expected, and if those are equal, set atomic to Desired. Returns ok if Desired was written. Returns the actual atomic value if not equal to Expected.

Atomic operations - Floating point array
using a wrapper that represents float in integer while preserving precision
Shared Memory PRAM

def jump(p) do
    {rock, agent} = Agent.start_link(fn -> [] end)
    Agent.update(agent, fn_state -> p end)
    # n is the size of the list p
    n = Enum.count(p)
    jump_iter(agent, p, n, [])
end

def jump_iter(agent, p, n, _p_suc) do
    Task.sync_stream(1..n, fn i ->
        if (Enum.at(p, Enum.at(p, i-1)-1) == i) do
            Agent.update(agent, fn p ->
                list_replace_at(p, i-1, Enum.at(p, Enum.at(p, i-1)-1))
            end
        end
    end) |> Enum.to_list
    p_suc = Agent.get(agent, fn s -> s end)
    if (p_suc := p) do
        p = p_suc
        jump_iter(agent, p, n, p_suc) # repeat to get next pointer jumping
    else
        p_suc
    end
end

iex(1)> PrtJump.jump([5, 5, 6, 6, 6, 8, 8, 8, 10, 11, 12, 13, 13])
[8, 8, 8, 8, 8, 8, 8, 13, 13, 13, 13, 13, 13]
Prefix sum (Ja Ja 1992)

Parallel Prefix sum

Input: An array of \( n = 2^k \) elements where \( k \geq 0 \)
Output: The prefix sum

1. for \( i \leftarrow 1 \) to \( n/2 \) pardo
   \( y_i = x_{2i-1} \times x_{2i} \)
2. Recursively compute prefix sum of \( \{y_1, \ldots, y_{n/2}\} \) and store them in \( z_1, \ldots, z_{n/2} \)
3. for all \( i \)'s pardo
   if \( i \) even then \( s_i = z_{i/2} \)
   if \( i = 1 \) then \( s_i = x_1 \)
   if \( i \) odd > 1 then \( s_i = z_{(i-1)/2} \times x_i \)

```
def psfm(list):
    lsz = Enum.count(list)
    alist = atomics.new(lsz, [])
    for i in range(1, lsz, 2):
        atomics.put(alist, i, Enum.at(list, i-1))
        for j in range(1, lsz, 2):
            atomics.put(alist, j, Enum.at(list, j-1))
    # step 1
    #apply(__MODULE__, psfmo, [alist, 0, lsz])  # k is initially 0, increments by 1
    psfmo(alist, 0, lsz)  # k is initially 0, increments by 1
    end

def psfmo(alist, k, lsz) when lsz == 1, do: nil
    define psfmo(alist, k, lsz) do
        kh = math.pow(2, k)
        n = div(lsz, kh)
        s = Task.async_stream(1..div(n, 2),
            fn i ->
                atomics.put(alist, 2*kh * i, Enum.at(alist, 2*kh*i - kh) + Enum.at(alist, 2*kh*i))
            end)
        s |> Enum.to_list
        # step 2
        alists = atomics.new(div(lsz, 2), [])
        # put the y values from alist
        for i in range(1, div(lsz, 2), 2):
            atomics.put(alists, i, Enum.at(alist, i+2))
        if i == 1 then
            Enum.to_list(alists, i)  # used for debugging
        psfmo(alists, 0, div(lsz, 2))
        # step 3
        Task.async_stream(1..lsz, fn i ->
            rem(i, 2) == 0 ->
                atomics.put(alist, i, Enum.at(alist, i+1))
            if i == 1 then
                Enum.to_list(alists, div(i, 2))
            true ->
                atomics.put(alist, i, Enum.at(alist, i-1, 2) + Enum.at(alist, i))
            end
        end)
        Enum.to_list(alists)
    end
```

```
step 3:  1  3  6  10  15  21  28  36
step 2:  1  3  3  10  5  11  7  36
        1  3  3  10  5  11  7  26
        x1  x2  x3  x4
step 1:  1  3  3  7  5  11  7  15
        i  ->  1  2  3  4  5  6  7  8
```

```
]ex(1)> list = [1,2,3,4,5,6,7,8]
[1, 2, 3, 4, 5, 6, 7, 8]
]ex(3)> Enum.reduce(list, [1], fn x, acc -> acc ++ [List.last(acc) + x] end)
[1, 2, 4, 7, 11, 16, 22, 29, 37]
```
Running parallel applications - using Agent

Agent.start_link(fn -> counter = 0 end, name: @name)
Agent.update(@name, fn counter -> counter + amt end)
Agent.get(@name, fn counter -> IO.inspect(counter) end)

MAINTAINS STATE
EASILY DISTRIBUTED

BENCHMARKING - BENCHCEE
MATRIX-VECTOR MULTIPLICATION

```
kapgari.inc> mix run agent_tasks_matrix_benchmark.exs
Operating System: macOS
CPU Information: Intel(R) Core(TM) i5-4260U CPU @ 1.40GHz
Number of Available Cores: 4
Available memory: 8 GB
Elixir 1.12.2
Erlang 24.0.4

Benchmark suite executing with the following configuration:
warmup: 2 s
time: 5 s
memory time: 0 ns
parallel: 1
inputs: none specified
Estimated total run time: 28 s

Benchmarking 1 process...
Benchmarking 2 process...
Benchmarking 4 process...
Benchmarking 8 process...

<table>
<thead>
<tr>
<th>Name</th>
<th>ips</th>
<th>average</th>
<th>deviation</th>
<th>median</th>
<th>99th %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 process</td>
<td>92.81</td>
<td>10.77 ms</td>
<td>±20.66%</td>
<td>10.35 ms</td>
<td>21.05 ms</td>
</tr>
<tr>
<td>4 process</td>
<td>67.26</td>
<td>14.87 ms</td>
<td>±13.12%</td>
<td>14.31 ms</td>
<td>21.80 ms</td>
</tr>
<tr>
<td>2 process</td>
<td>63.34</td>
<td>15.79 ms</td>
<td>±11.86%</td>
<td>15.60 ms</td>
<td>20.72 ms</td>
</tr>
<tr>
<td>8 process</td>
<td>36.72</td>
<td>27.23 ms</td>
<td>±14.43%</td>
<td>26.79 ms</td>
<td>42.38 ms</td>
</tr>
</tbody>
</table>

Comparison:
<table>
<thead>
<tr>
<th>Name</th>
<th>ips</th>
<th>average</th>
<th>deviation</th>
<th>median</th>
<th>99th %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 process</td>
<td>92.81</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 process</td>
<td>67.26</td>
<td>1.38x slower</td>
<td>+4.09 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 process</td>
<td>63.34</td>
<td>1.47x slower</td>
<td>+5.01 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 process</td>
<td>36.72</td>
<td>2.53x slower</td>
<td>+16.46 ms</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
defmodule Matrices do
  defstruct [matrix_A: [], matrix_B: [], matrix_C: []]
end
defmodule AgentTasksMatrix do
  @moduledoc ""
  Parallel program for matrix vector multiplication
  matrix is A and vector is B
  n: matrix size n x n
  p: number of processor
  r: n/p is a integer
  Agent stores matrix A and vector B, as well as the product C in a list of arrays.
  ...
  use Agent
  @name __MODULE__
  import MatrixModule
  ###
  # External API
  def start_link, do: Agent.start_link(fn -> struct(Matrices) end, name: @name)
  def print_matrix(matrix) do
    if matrix == "matrix_A", do: Agent.get(@name, & &1.matrix_A), else:
      if matrix == "matrix_B", do: Agent.get(@name, & &1.matrix_B), else:
        if matrix == "matrix_C", do: Agent.get(@name, & &1.matrix_C), else:
          Agent.get(__MODULE__, & &1)
        end
      end
    end
    @doc ""
    set_matrix(matrix1, data_file1, matrix2, data_file2)
    will read data to be fed to the matrix_A, matrix_B
    ...
    def set_matrix(matrix1, data_file1, matrix2, data_file2) do
      cond do
        matrix1 != [] ->
          Agent.update(@name, fn x -> %Matrices{ x | matrix_A: MatrixModule.read_matrix(data_file1)} end)
        true ->
      end
      cond do
        matrix2 != [] ->
          Agent.update(@name, fn x -> %Matrices{ x | matrix_b: MatrixModule.read_matrix(data_file2)} end)
        true ->
      end
    end
  end
  # Internals
  defp partition_matrix(i, n,p,matrix) do
    r = div(n,p)
    if matrix == "matrix_A", do: Agent.get(@name, & &1.matrix_A)
    |> Enum.slice((i-1)*r, r)
    end
  end
  defp mult_partition(id,n,p) do
    part = partition_matrix(id,n,p, "matrix_A")
    #"Matrix_B:
    mat_B = Agent.get(@name, & &1.matrix_B)
    v = MatrixModule.mult_matrix(part, mat_B)
    end
  end
  defp write_product(c) do
    Agent.update(@name, fn x -> %Matrices{ x | matrix_C: c} end)
  end
  def run_tasks(p) do
    n = n_rows(Agent.get(@name, & &1.matrix_A))
    temp = 1..n
    |> Stream.map(fn id -> Task.async(fn -> mult_partition(id, n,p) end) end)
    |> Enum.map(&Task.await/1)
    |> Enum.concat()
    write_product(inspect(temp))
    nil
  end
  # Internals
  defp partition_matrix(i, n,p,matrix) do
    r = div(n,p)
    if matrix == "matrix_A", do: Agent.get(@name, & &1.matrix_A)
    |> Enum.slice((i-1)*r, r)
    end
  end
  defp mult_partition(id,n,p) do
    part = partition_matrix(id,n,p, "matrix_A")
    #"Matrix_B:
    mat_B = Agent.get(@name, & &1.matrix_B)
    v = MatrixModule.mult_matrix(part, mat_B)
    end
  end
  defp write_product(c) do
    Agent.update(@name, fn x -> %Matrices{ x | matrix_C: c} end)
  end
end
MPI notes

MPI - MPI_Init: every MPI process makes a call to this function which allows the system to do any set up work

When initialized, every active process becomes a member of a communicator MPI_COMM_WORLD. A communicator is an opaque object that provides the environment for message passing among processes. Other functions such as MPI_BARRIER (sync all messages), MPI_BCAST and many more are available.

MPI_Send and MPI_Receive

MPI process will usually use the above two functions to get the partition it is supposed to work on

MPI - MPI_Init: every MPI process makes a call to this function which allows the system to do any set up work

When initialized, every active process becomes a member of a communicator MPI_COMM_WORLD. A communicator is an opaque object that provides the environment for message passing among processes. Other functions such as MPI_BARRIER (sync all messages), MPI_BCAST and many more are available.

MPI_Send and MPI_Receive

MPI process will usually use the above two functions to get the partition it is supposed to work on
Shortest paths - Floyd’s Algorithm (adapted from MPI version in Quinn’s book)

1. ONE PROCESS WILL READ THE FILE AND DISTRIBUTE THE ROWS MEANT FOR EACH PROCESS. IT READS THE ROWS IT IS TO PROCESS TOO.
2. ONCE THE COMPUTATION IS OVER THE SAME PROCESS WILL PRINT OUT THE FINAL DISTANCE MATRIX (AFTER THE SHORTEST PATHS HAVE BEEN FOUND)
3. PROCESS 0 DOES THE IO AND DISTRIBUTES THE DATA TO ITSELF AND OTHER PROCESSES.
4. EACH PROCESS HAS PART OF THE ROWS
5. ALL PROCESSES CARRY OUT BROADCAST OF THESE ROWS TO THE OTHER PROCESSES
6. EACH PROCESS THEN CALCULATES THE DISTANCE MATRIX FOR ITS ROWS
7. EACH PROCESS SENDS THE ROWS BACK TO PROCESS 0
8. PROCESS 0 ASSEMBLES THE ROWS RECEIVED FROM ITS OWN LOCAL DATA AND THOSE RECEIVED

iex(14)> mat_get = fn m, i, j ->
...(14)>   Enum.at(Enum.at(m,i),j)
...(14)> end

iex(20)> :timer.tc(fn -> for k <- 0..n-1, i <- 0..n-1, j <- 0..n-1, reduce: data do
...(20)>     acc -> List.replace_at(acc,i, List.replace_at(Enum.at(acc,i), j, min(mat_get.(acc, i,j),mat_get.(acc, i,k) + mat_get.(acc, k, j)))))
...(20)>   end
...(20)>   end)
{12853, 
[[0.0, 2.0, 5.0, 3.0, 6.0, 9.0],
 [999.0, 0.0, 6.0, 1.0, 4.0, 7.0],
 [999.0, 15.0, 0.0, 4.0, 7.0, 10.0],
 [999.0, 11.0, 5.0, 0.0, 3.0, 6.0],
 [999.0, 8.0, 2.0, 5.0, 0.0, 3.0],
 [999.0, 5.0, 6.0, 2.0, 4.0, 0.0]]}
iex(13)> :timer.tc(fn -> ComputSp.main_start("hello", 6, 4);Agent.get(:process_p, fn {:main, data} -> data end) end)
{93,
 [0.0, 2.0, 5.0, 3.0, 6.0, 9.0],
 [999.0, 0.0, 6.0, 1.0, 4.0, 7.0],
 [999.0, 15.0, 0.0, 4.0, 7.0, 10.0],
 [999.0, 11.0, 5.0, 0.0, 3.0, 6.0],
 [999.0, 8.0, 2.0, 5.0, 0.0, 3.0],
 [999.0, 5.0, 6.0, 2.0, 4.0, 0.0]}

Ligra Shared Memory Parallel Programming Framework

A single multicore server can support more than a terabyte of memory -> graphs of tens or even hundreds billions of edges.
Shared-memory multicores are generally more efficient on a per core, per dollar and per joule basis than dict memory systems

Two simple routines - mapping over edges and for mapping over vertices can be applied to any subset of the vertices:
BFS, graph radii estimation, graph connectivity, betweenness centrality, PageRank and single-source shorts paths.
in CilkPlus, good speedup on a 40-core machine

Graph traversals easy to write!

**Pseudocode for Breadth First Search in Ligra framework:**

```
1: Parents = {-1, ..., -1}                     // initialized to all -1's
2: 3: procedure UPDATE(s, d)
4:      return (CAS(&Parents[d], -1, s))
5: 6: procedure COND(i)
7:      return (Parents[i] == -1)
8: 9: procedure BFS(G, r)                    // r is the root
10: Parents[r] = r                           
11: Frontier = {r}                           // vertexSet initialized to contain only r
12: while (SIZE(Frontier) ≠ 0) do
13:    Frontier = EDGEMAP(G, Frontier, UPDATE, COND)
```

data/set1.data

```
size of graph is: 15
root for search?: 1
calling edgeMapSparse
u: #MapSet<1>
calling edgeMapSparse
u: #MapSet<2, 3, 4>
calling edgeMapSparse
u: #MapSet<5, 6, 7, 8, 9>
calling edgeMapSparse
u: #MapSet<10, 11, 12, 13, 14, 15>
Time: 9528 mill-secs
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```
Ligra implementation in Elixir

- The graph is given by \( (g, \text{parents}) \) - \( g \) is a ETS table, parents is an atomics ref array, vertices are labeled 1 to \(|V|\) initial values in parents are all -1.
- The init function should read the graph and create the parents array, init returns \( \text{grph} = (g, \text{parents}) \)

```elixir
def edgeMap(grph, u, f, fargs, c, carbs)
  if graph_size(grph) > 20 do
    # convert u to atomics array
    n = graph_size(grph)
    u = set_to_array(u, n)
    IO.puts("calling edgeMapDense")
    edgeMapDense(grph, u, f, fargs, c, carbs)
  else
    IO.puts("calling edgeMapSparse")
    edgeMapSparse(grph, u, f, fargs, c, carbs)
  end
end
```

- For sparse vertexSubSet u is a MapSet, the initial type of u is a MapSet, we convert it to an atomics if it is the case of dense.

```elixir
# user supplied functions
def c(parents, u) when u == 0 do # cond
  atomics.get(parents, u) == -1
end
def c(_, u) when u == 0 do # cond
  false
end
def f(_, s, d) do # UPDATE
  IO.puts("values of s and d: " || Enum.join([s, d], ", "))
  res = atomics.compare_exchange(parents, d, -1, s)
  if res == not do true else false end
end
```

Algorithm 1

1: procedure EDGEMAP
2: if (\( U \) + sum of out-degrees of \( U \) >= threshold) then
3: return EDGEMAPDENSE(\( G, U, F, C \))
4: else return EDGEMAPSPARSE(\( G, U, F, C \))

Algorithm 2

1: procedure EDGEMAPSPARSE(\( G, U, F, C \))
2: Out = \{\}
3: parfor each \( u \in U \) do
4:    parfor \( ngh \in N^+(u) \) do
5:      if \( C(u, ngh) == 1 \) and \( F(u, ngh) == 1 \) then
6:        Add ngh to Out
7:    end
8: end
9: Remove duplicates from Out
10: return Out

For line 3:

```elixir
def set_to_array(u, n) do
  a = atomics.new(n, 1)
  for i <- 1..n do
    atomics.put(a, i, 0)
  end
  a
end
```

```elixir
def mapset_to_list(u) do
  MapSet.to_list(u)
end
```

```elixir
def neighbors(grph, g, u, ngh) do
  ngh = ngh_u(g, List.first(u_list))
  Enum.filter(fn x -> x == ngh end)
  Task.async_stream(u_list, fn u when u == ngh ->
    Enum.filter(fn x -> x == ngh end)
  end)
end
```
Framework in Elixir

graph is given by \( (g, \text{parents}) \) where \( g \) is a ETS table and \( \text{parents} \) is a atomics ref array, vertices are labeled 1 to \(|V|\), initial values in \( \text{parents} \) are all -1. The \text{init} function should read the graph and create the parents array, \text{init} returns \( \text{grph} = (g, \text{parents}) \).

Algorithm 3 \text{EDGE\_MAP\_DENSE}.

1. procedure \text{EDGE\_MAP\_DENSE}(G, U, F, C)
2. \hspace{1em} Out = {}  
3. \hspace{2em} parfor \( i \in \{0, \ldots, |V| - 1\} \) do
4. \hspace{3em} if \( C(i) == 1 \) then
5. \hspace{4em} \hspace{1em} for \( \text{ngth} \in N^{-}(i) \) do
6. \hspace{5em} \hspace{1em} \hspace{1em} if \( (\text{ngth} \in U \text{ and } F(\text{ngth}, i) == 1) \) then
7. \hspace{6em} \hspace{2em} Add \( i \) to \( \text{Out} \)
8. \hspace{3em} if \( (C(i) == 0) \) then break
9. \hspace{2em} return \( \text{Out} \)

Algorithm 4 \text{VERTEX\_MAP}.

1. procedure \text{VERTEX\_MAP}(U, F)
2. \hspace{1em} Out = {}  
3. \hspace{2em} parfor \( u \in U \) do
4. \hspace{3em} if \( F(u) == 1 \) then Add \( u \) to \( \text{Out} \)
5. \hspace{2em} return \( \text{Out} \)

Algorithm 5 \text{EDGE\_MAP\_DENSE-WRITE}.

1. procedure \text{EDGE\_MAP\_DENSE-WRITE}(G, U, F, C)
2. \hspace{1em} Out = {}  
3. \hspace{2em} parfor \( i \in \{0, \ldots, |V| - 1\} \) do
4. \hspace{3em} if \( i \in U \) then
5. \hspace{4em} \hspace{1em} parfor \( \text{ngth} \in N^{+}(i) \) do
6. \hspace{5em} \hspace{2em} if \( (F(\text{ngth}, i) == 1 \text{ and } C(\text{ngth}) == 1) \) then
7. \hspace{6em} \hspace{3em} Add \( \text{ngth} \) to \( \text{Out} \)
8. \hspace{3em} return \( \text{Out} \)

```
    def edgeMapDense(graph, u, fargs, c, cargs) do
      g = elem(graph, 0)
      n = Framework.graph_size(graph)
      Task.async_stream(1..n, fn i ->
        if apply(c, cargs ++ [i]) do
          parfor ngh <- Framework.gh_u_map(g, i) do
            if(keyisatom(t=apply(f, fargs+t[j], j))) do i end
        end
      end)
      Enum.filter(fn _, x) -> x == nil and x == nil end |> MapSet.new() end
```

```
    def vertexMap(u, fargs) do
      Task.async_stream(u, fn i ->
        if apply(f, fargs++ [i]) do i end
      end)
      Enum.filter(fn _, x) -> x == nil and x == nil end |> MapSet.new()
```

Betweenness Centrality

For a graph $G = (V, E)$ and some $s, t \in V$, let $\sigma_{st}$ be the number of shortest paths from $s$ to $t$ in $G$.

For vertices $s, t, v \in V$, define $\sigma_{st}(v)$ to be the number of shortest paths from $s$ to $t$ that pass through $v$.

Define $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$ to be the pair-dependency of $s, t$ on $v$.

The betweenness centrality of a vertex $v$, denoted by $C_B(v)$, is equal to $\sum_{s \neq v \neq t \in V} \delta_{st}(v)$. There are $O(|V|^2)$ pair-dependency terms associated with each vertex, hence this method requires $O(|V|^3)$ operations.

How to have a parallel version?

:atomic.compare_exchange

EdgeMap

VertexMap
Dependency

\[ \sigma_{rv} = \sum_{u \in P_r(v)} \sigma_{ru} \]

Recursive evaluation using BFS & backward traversal

\[ \sigma_{rv} = \sum_{u \in P_r(v)} \sigma_{ru} \]
\[ \delta_{r*}(v) = \sum_{w: v \in P_r(w)} \frac{\sigma_{rw}}{\sigma_{rv}} \times (1 + \delta_{r*}(w)) \]

Dependencies for root 1

Dependencies for root 2
def bc(g, r, gs, visit, pathsUpdate, numPaths, depUpdate, dependencies, cond, visited, levels) do
    frontier = MapSet.new([r])
    IO.puts("Frontier before: #{inspect(frontier)}")
    # call bc recursively
    {frontier,numPaths} = bc_rec(g, gs, frontier, pathsUpdate, numPaths, cond, visited, levels, frontier)
    IO.puts("Frontier after: #{inspect(frontier)}")
    IO.puts("numPaths:")
    for i <- 1..gs, do: IO.inspect("#{inspect(:atomics.get(numPaths,i))}")
    # initialize 'visited' to 0
    for i <- 1..gs, do: :atomics.put(visited, i, 0)
    #curLevel = curLevel - 1
    [{_,temp}] = :ets.lookup(levels, :currentLevel)
    :ets.insert(levels, {:currentLevel, temp - 1})
    # transpose graph
    g = transpose(g,gs)
    cur_level = temp - 1
    IO.puts("cur_level:  #{cur_level}")
    # need to pack numPaths and dependencies arrays in a tuple to be arg of f/3 in edgeMap
    numPaths_dependencies = {numPaths, dependencies}
    # second while implementation
    bc_rec_s(g, gs, frontier, visit, depUpdate, numPaths_dependencies, cond, visited, levels, cur_level)
end

the whiles are implemented using recursion

bc_rec and bc_rec_s
def bc_rec(_g, gs, frontier, _pathsUpdate, numPaths, _cond, _visited, _levels, frontier_acc)
    when frontier == @empty, do: {frontier_acc, numPaths}
    def bc_rec(g, gs, frontier, pathsUpdate, numPaths, cond, visited, levels, frontier_acc) do
        # a hack of using an :atomics of size gs is used to conform to the edgeMap interface
        # note that all data are stored in global ref type structure (except frontier)
        # changes in frontier is accumulated in the frontier_acc
        # using common framework interface edgeMap
        # edgeMap(G,U,F,C) as in paper replaced by edgeMap(g, u, f, fargs, c, cargs)
        fargs = [numPaths]
        cargs = [visited]
        #### apply EdgeMap
        frontier = edgeMap({g,:atomics.new(gs,[])}, frontier, &pathsUpdate/3, fargs, &cond/2, cargs)
        [_, cur_level] = :ets.lookup(levels, :currentLevel)
        [_, temp] = :ets.lookup(levels, :levs)
        temp = Map.put(temp, cur_level, frontier)
        :ets.insert(levels, {:levs, temp})
        vargs = [visited]
        #### apply VertexMap
        frontier = vertexMap(frontier, &visit/2, vargs)
        cur_level = cur_level - 1
        IO.puts("cur_level: #{cur_level}")
        bc_rec_s(g, gs, frontier, visit, depUpdate, numPaths_dependencies, cond, visited, levels, cur_level)
    end
    def bc_rec_s(g, gs, _frontier, visit, depUpdate, numPaths_dependencies, cond, visited, levels, cur_level) when cur_level >= 0 do
        [_, temp] = :ets.lookup(levels, :levs)
        frontier = Map.get(temp, cur_level)
        dargs = [numPaths_dependencies]  # the tuple is put into list to be used in edgeMap func f/3 args
        cargs = [visited]
        # a hack of using an :atomics of size gs is used to conform to the edgeMap interface
        edgeMap({g,:atomics.new(gs,[])}, frontier, &depUpdate/3, dargs, &cond/2, cargs)
        cur_level = cur_level - 1
        IO.puts("cur_level: #{cur_level}")
        bc_rec_s(g, gs, frontier, visit, depUpdate, numPaths_dependencies, cond, visited, levels, cur_level)
    end
    def bc_rec_s(_g, _gs, _frontier, _visit, _depUpdate, numPaths_dependencies, _cond, _visited, _levels, cur_level) when cur_level < 0 do
        elem(numPaths_dependencies, 1)
    end
    end

def visit(visited, i) do
  :atomics.put(visited, i, 1)
end

def cas(x, d, oldV, newV) do
  if :atomics.compare_exchange(x, d, oldV, newV) == :ok do
    {x, true} else {x, false} end
end

def cas_r(xr, d, oldV, newV) do    # newV given as a real number with factor f
  {f, n} = newV
  n = if is_integer(n) do n/1.0 else n end
  i_newV = Float.round(n, f) * :math.pow(10, f) |> trunc
  if :atomics.compare_exchange(xr, d, oldV, i_newV) == :ok do
    {xr, true} else {xr, false} end
end

def pathsUpdate(numPaths, s, d) do    # must conform to the edgeMap interface function f
  oldV = :atomics.get(numPaths, d)
  newV = oldV + :atomics.get(numPaths, s)
  {acc_numPaths, c} = cas(numPaths, d, oldV, newV)
  rep_pathsUpdate(s, d, numPaths, oldV, c, acc_numPaths)
end

def rep_pathsUpdate(_s, _d, numPaths, oldV, true, acc_numPaths), do:
  (pathsSz = :atomics.info(numPaths) |> Map.get(:size);
   for i<- 1..pathsSz, do: IO.puts(:atomics.get(acc_numPaths,i)); oldV == 0)
def rep_pathsUpdate(s, d, numPaths, oldV, _oldV, _c, acc_numPaths) do
  pathsSz = :atomics.info(numPaths) |> Map.get(:size)
  for i<- 1..pathsSz, do: IO.puts(:atomics.get(acc_numPaths,i))
  rep_pathsUpdate(s, d, numPaths, oldV, c, acc_numPaths)
end

def cond(visited, i) do
  :atomics.get(visited, i) == 0
end

def depUpdate(numPaths_dependencies, s, d) do
  {numPaths, dependencies} = numPaths_dependencies
  # a f value of 2 has been used here
  oldV = :atomics.get(dependencies, d)
  newV = if :atomics.get(numPaths, s) != 0 do
    oldV + round(:atomics.get(numPaths, d)/:atomics.get(numPaths, s) * 
    (1 + :atomics.get(dependencies, s)/100))
  else
    oldV/100
  end
  IO.puts("value of newV: #{newV}")
  newV = {2, newV}
  {acc_dependencies, c} = cas_r(dependencies, d, oldV, newV)
  rec_depUpdate(s, d, numPaths, dependencies, oldV, c, acc_dependencies)
end

def rec_depUpdate(_s, _d, _numPaths, dependencies, oldV, true, acc_dependencies), do:
  (depsSz = :atomics.info(dependencies) |> Map.get(:size);
   # printing scaled down actual (real) values
   for i<- 1.. depsSz, do: IO.puts(:atomics.get(acc_dependencies,i)/100); oldV == 0)
def rec_depUpdate(s, d, numPaths, dependencies, _oldV, _c, acc_dependencies) do
  depsSz = :atomics.info(dependencies) |> Map.get(:size)
  oldV = :atomics.get(dependencies, d)
  newV = if :atomics.get(numPaths, s) != 0 do
    oldV/100 + round(:atomics.get(numPaths, d)/:atomics.get(numPaths, s) * 
    (1 + :atomics.get(dependencies, s)/100))
  else
    oldV/100
  end
  IO.puts("value of newV: #{newV}")
  newV = {2, newV}
  {acc_dependencies, c} = cas_r(dependencies, d, oldV, newV)
  rec_depUpdate(s, d, numPaths, dependencies, oldV, c, acc_dependencies)
end

lines 9 - 11
lines 24
lines 13 - 18
lines 20 - 25
iex(4)> BcTest.runBC("data/set1_bc_un.data")
Reading input graph from file: data/set1_bc_un.data
size of graph is: 15
in_neighbors: [[2, 3, 4], [1, 5, 6], [1, 5, 6, 7], [1, 7, 9, 8], [2, 3, 10], [2, 3, 11], [3, 4, 12, 13], [4, 10, 15], [4, 13, 14], [5, 8], [6], \a, \a\t, \t, \b]
out_neighbors: [[2, 3, 4], [1, 5, 6], [1, 5, 6, 7], [1, 7, 9, 8], [2, 3, 10], [2, 3, 11], [3, 4, 12, 13], [4, 10, 15], [4, 13, 14], [5, 8], [6], \a, \a\t, \t, \b]
##numPaths
1 -> 1
2 -> 1
3 -> 2
4 -> 1
5 -> 4
6 -> 3
7 -> 1
8 -> 1
9 -> 1
10 -> 1
11 -> 3
12 -> 1
13 -> 2
14 -> 1
15 -> 1
##dependencies
1 -> 3.33
2 -> 0.91
3 -> 1.84
4 -> 14.0
5 -> 0.0
6 -> 1.0
7 -> 2.92
8 -> 2.25
9 -> 1.5
10 -> 0.25
11 -> 0.0
12 -> 0.0
13 -> 0.0
14 -> 0.0
15 -> 0.0
[:ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok]
Algorithm 3.1 Finding nodes reachable from X given Z via active trails (d-separation)

Inference in trees using GenServer

- Problem: Given a Bayesian network whose DAG is a tree, determine the probabilities of the values of each node conditional on specified values of the nodes in some subset.

- Inputs: Bayesian network \((G,P)\) whose DAG is a tree, where \(G = (V,E)\) and a set of values \(a\) of a subset \(A\) of \(V\).

- Outputs: The Bayesian network \((G,P)\) updated according to the values in \(a\). The lambda and pi values and messages and \(P(x|a)\) for each \(X\) in \(V\) are considered part of the network.
Distributed application with Graph Database

Supply Chain Management with graphs - some analytics


A scalable architecture with Elixir, Bolt connector and Neo4j
FUTURE WORK

Irregular Graph programming

Distribution

Performance

Reliability

Web applications
REFERENCES

3. Elixir https://elixir-lang.org/
5. The Little Elixir & OTP Guidbook, Benjamin Tao Wei Has, Manning, 2017
8. Parallel Programming in C with MPI and OpenMP, Michael J. Quinn, McGraw Hill, 2004
9. An Introduction to Parallel Algorithms, Joseph JaJa, Addison-Wesley, 1992
11. Introducing Nx - Jose Valim, Lambda Days 2021 https://www.youtube.com/watch?v=fPKMmJpAGWc
12. Tao of Parallelism in Algorithms, Keshav Pingali et al. PLII’11, June 4-8. San Jose, California, USA