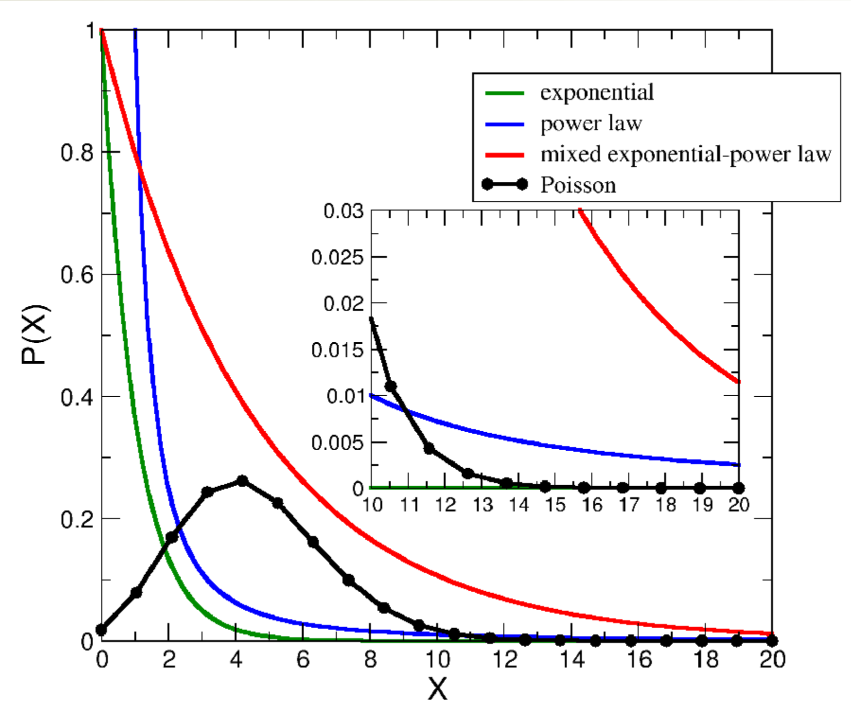
HavoqGT

Brian A. Page
bpage1nd.edu
September 27, 2018

Scale-Free Graphs

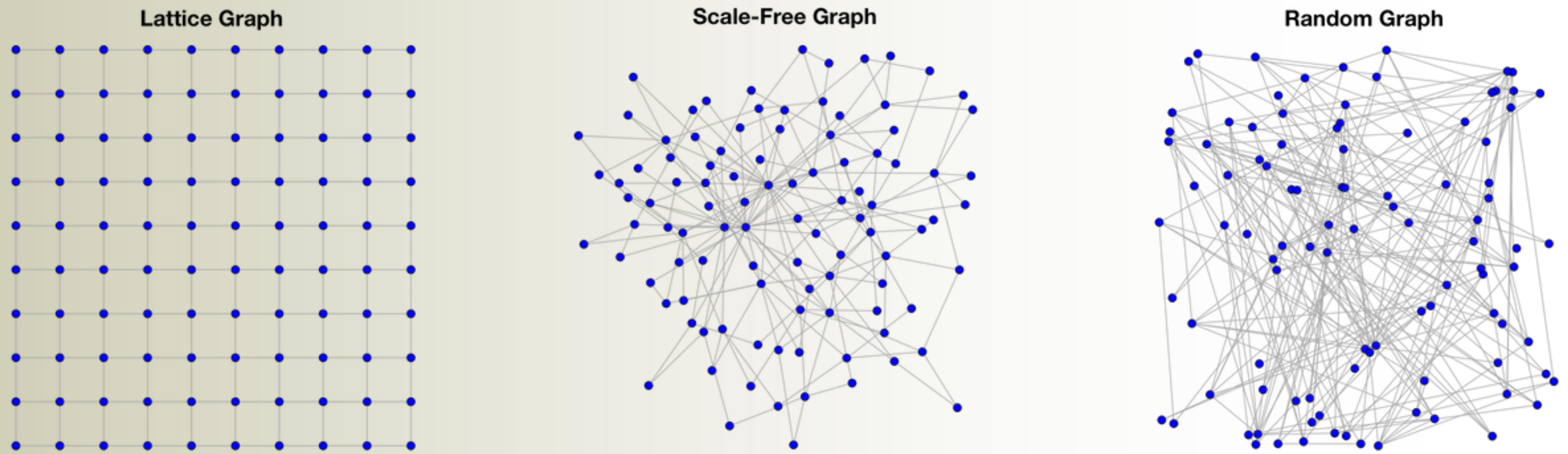
- A ***Scale-Free*** graph is a graph whose vertex degree distribution follows a power law.



- ***Hub*** - a vertex where $v_{deg} >> avg_{deg}$
- Fewer hubs of drastically increased degree

The College of Engineering at the University of Notre Dame

Scale-Free Graphs



- Pose significant workload distribution challenges
- Data set size vs node memory issues
- Communication overhead challenges for distributed implementations

HavoqGT: Origins

Lawrence Livermore National Laboratory

Roger Pearce, Maya Gokhale, Nancy Amato

- Asynchronous Visitor Model[1,2]
- Distributed Asynchronous Visitor Model[3]
- Parallel graph traversal for large scale-free graphs
- Code Available at: https://github.com/LLNL/HavoqGT
- Documentation: https://llnl.github.io/havoqgt/index.html

The College of Engineering at the University of Notre Dame

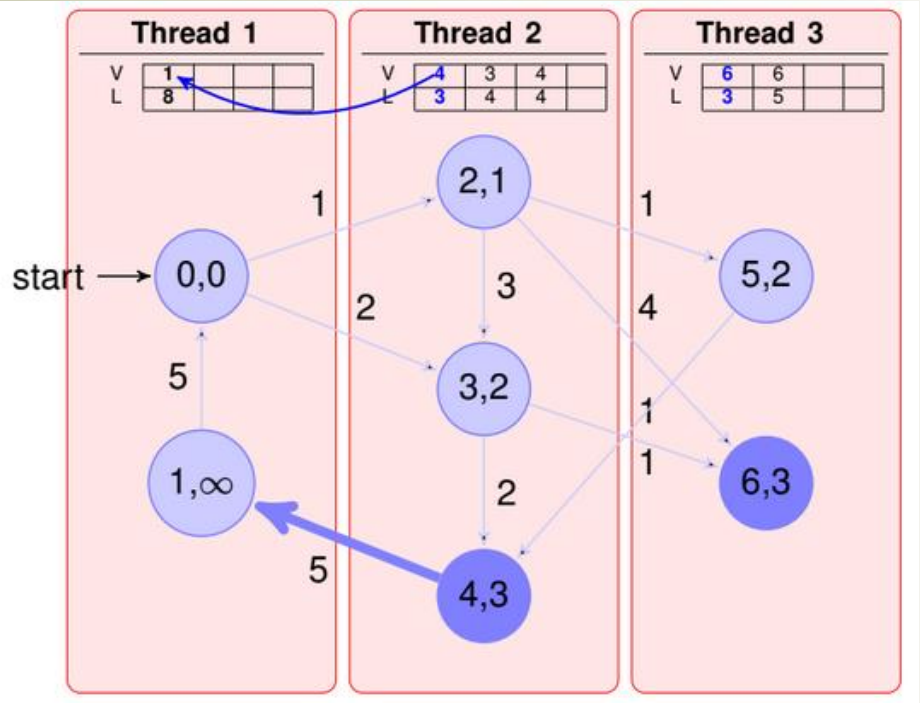
Asynchronous Visitor Model

- Vertices kept in DRAM, associated edge list stored in NVRAM (SSD, etc.)
  - allows larger dataset use without massive IO penalty (39% as measured in [2])

- ***Visitor*** application specific kernel that applied to each vertex
- Uses priority queue(s) to queue visitors for traversal by visitors
- Visitors traverse a graph and return/update based on kernel implementation.

The College of Engineering at the University of Notre Dame

Asynchronous Visitor Model



- Independent threads or processes schedule work on other’s work queues via a visitor queue
- Priority based on vertex ID

- Load imbalance heavily reliant on graph structure

The College of Engineering at the University of Notre Dame

HavoqGT: Objectives

- Design highly scalable graph traversal framework for very large scale-free data sets
- Minimize memory and network latency
- Improve storage and workload imbalance caused by high degree ***hubs***
- Develop techniques that tolerate data latancies

The College of Engineering at the University of Notre Dame

HavoqGT: Graph Representation

- Input graph from file
- Older implementations used distributed vertex sets and associated edge-lists
- Latest: adjacency list generated from compressed sparse matrix (?)

HavoqGT: Execution

1. Read in graph
2. Perform delgate partitioning and load balancing
3. Create and queue initial visitors
4. Perform traversal as per algorithm design
5. End traversal when all visitor queues are empty (no other vertices needs to be traversed)

HavoqGT: Visitor

- User defines vertex-centric behavior to be executed on traversed vertices
- Have the ability to pass visitor state to other vertices
- Only operate on the subset of adjacent edges local to the partition

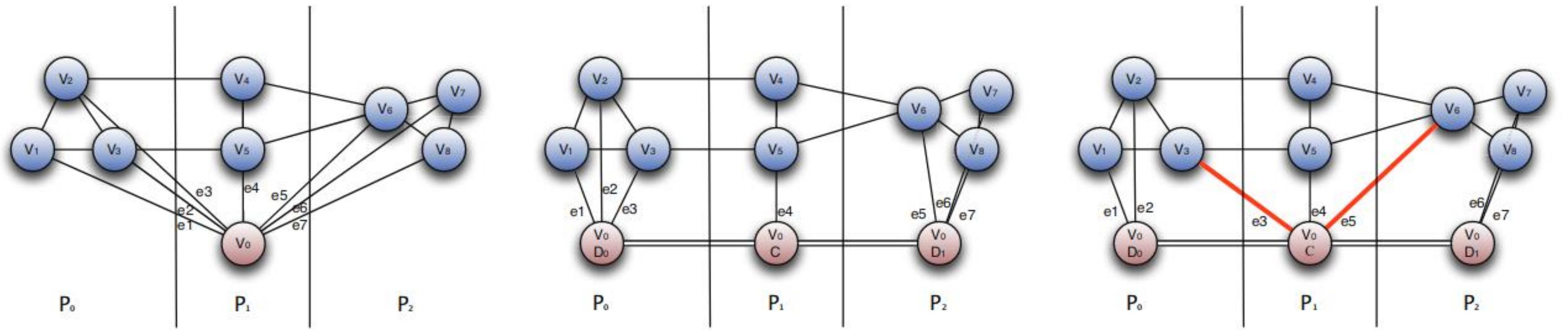
Algorithm 1 BFS & SSSP Visitor

```
visitor state: vertex ← vertex to be visited
visitor state: length ← path length
visitor state: parent ← path parent

delegate behavior: pre_visit_parent

procedure PRE_VISIT(vertex_data)
    if length < vertex_data.length then
        vertex_data.length ← length
        vertex_data.parent ← parent
        return true
    end if
    return false
end procedure

procedure VISIT(graph, visitor_queue)
    if length == graph[vertex].length then
        for all vi ∈ out_edges(g, vertex) do
                                        ▷ Creates and queues new visitors
            new_len ← length + edge_weight(g, vertex, vi)
                                        ▷ edge_weight equals 1 for BFS
            new_vis ← bfs_visitor(vi, new_len, vertex)
            visitor_queue.push(new_vis)
        end for
        return bcast_delegates
    else
        return terminate_visit
    end if
end procedure

procedure OPERATOR < ()(visitor_a, visitor_b)
                                ▷ Less than comparison, sorts by length
    return visitor_a.length < visitor_b.length
end procedure
```

HavoqGT: Delegates

- ***Delegates*** are *hubs* or very high degree vertices
- They are main cause of load imbalance in scale-free graphs
- Maintain a copy of the state for the vertex and a portion of the adjacency list of the vertex.
- When visited, a visitor operates <u>only</u> on the subset of edges managed by the local delegate

HavoqGT: Partitioning



(a) 1D Partitioning

(b) Distributed Delegates Partitioning. P_1 contains the *controller*.

(c) Distributed Delegates Partitioning after balancing.

- Delegates are distributed amongst partitions such to insure balanced work distribution
- One Delegate chosen as ***controller***
- Edges are moved to any partition to further correct imbalance

The College of Engineering at the University of Notre Dame

Algorithm Design

- Non-delegate visitor behavior is traditional traversal method specific
- Delegate behavior must be chosen for each visitor

pre_visit_parent	Visitor is sent to parent delegate and executes *pre_visit*. If *pre_visit* returns *true*, visitor continues to visit parents until the *controller* is reached.	$O(h_{tree})$	BFS, SSSP
lazy_merge_parent	Lazily merges visitors using an asynchronous reduction tree. Merges visitors locally, and sends to parent in reduction tree when local visitor queue is idle. When *controller* is reached, normal visitation proceeds. Requires that visitors provide a *merge* function.	$O(h_{tree})$	k-core
post_merge	Visitors are merged into parent reduction tree after traversal completes. Requires that visitors provide a *merge* function.	$O(h_{tree})$	PageRank

- Controller commands are selected for a visitor's return procedure

Behavior	Description	Complexity
bcast_delegates	Controller broadcasts the current visitor to all delegates.	$O(h_{tree})$
terminate_visit	Controller terminates the current visitor without sending to delegates.	$\Theta(1)$

The College of Engineering at the University of Notre Dame

Algorithm Design Cont.

- Controller and delegate operations are coordinated through asynchronous broadcast and reduction via MPI

- A controller can broadcast commands to all delegates or choose to not propagate visitors to other delegates by calling *terminate_visit()*

Example: BFS

```
template<typename Graph, typename LevelData, typename ParentData>
class bfs_visitor {
public:
  typedef typename Graph::vertex_locator                 vertex_locator;
  bfs_visitor(): m_level(std::numeric_limits<uint64_t>::max())  { }
  bfs_visitor(vertex_locator _vertex, uint64_t _level, vertex_locator _parent)
    : vertex(_vertex)
    , m_parent(_parent)
    , m_level(_level) { }

  bfs_visitor(vertex_locator _vertex)
    : vertex(_vertex)
    , m_parent(_vertex)
    , m_level(0) { }


  bool pre_visit() const {
    bool do_visit  = (*level_data())[vertex] > level();
    if(do_visit) {
      (*level_data())[vertex] = level();
    }
    return do_visit;
  }

  template<typename VisitorQueueHandle>
  bool visit(Graph& g, VisitorQueueHandle vis_queue) const {
    if(level() <= (*level_data())[vertex]) {
      (*level_data())[vertex] = level();
      (*parent_data())[vertex] = parent();

      typedef typename Graph::edge_iterator eitr_type;
      for(eitr_type eitr = g.edges_begin(vertex); eitr != g.edges_end(vertex); ++eitr) {
        vertex_locator neighbor = eitr.target();
        //std::cout << "Visiting neighbor: " << g.locator_to_label(neighbor) << std::endl;
        bfs_visitor new_visitor(neighbor, level() + 1,
            vertex);
        vis_queue->queue_visitor(new_visitor);
      }
      return true;
    }
    return false;
  }
```

```
  uint64_t level() const {  return m_level; }
  vertex_locator parent() const  { return m_parent; }

  friend inline bool operator>(const bfs_visitor& v1, const bfs_visitor& v2) {
    //return v1.level() > v2.level();
    if(v1.level() > v2.level())
    {
      return true;
    } else if(v1.level() < v2.level())
    {
      return false;
    }
    return !(v1.vertex < v2.vertex);
  }

  // friend inline bool operator<(const bfs_visitor& v1, const bfs_visitor& v2) {
  //   return v1.level() < v2.level();
  // }

  static void set_level_data(LevelData* _data) { level_data() = _data; }

  static LevelData*& level_data() {
    static LevelData* data;
    return data;
  }

  static void set_parent_data(ParentData* _data) { parent_data() = _data; }
  static ParentData*& parent_data() {
    static ParentData* data;
    return data;
  }
  vertex_locator   vertex;
  //uint64_t         m_parent : 40;
  vertex_locator  m_parent;
  uint64_t         m_level : 8;
} __attribute__ ((packed));
```


The College of Engineering at the University of Notre Dame

Example: BFS

1) Create graph type holder

```
typedef hmpi::delegate_partitioned_graph<segment_manager_t> graph_type;
```

2) Read in graph and create/populate proper graph object

```
havoqgt::distributed_db ddb(havoqgt::db_open(), graph_input.c_str());

graph_type *graph = ddb.get_segment_manager()->
  find<graph_type>("graph_obj").first;
```

3) Perform partitioning and distribute graph

```
graph_type::vertex_data<uint8_t,
    std::allocator<uint8_t> > bfs_level_data(*graph);
graph_type::vertex_data<graph_type::vertex_locator,
    std::allocator<graph_type::vertex_locator> >  bfs_parent_data(*graph);
MPI_Barrier(MPI_COMM_WORLD);
if (mpi_rank == 0) {
  std::cout << "BFS data allocated.  Starting BFS from vertex "
            << source_vertex << std::endl;
}
```

4) Call Breadth First Search function

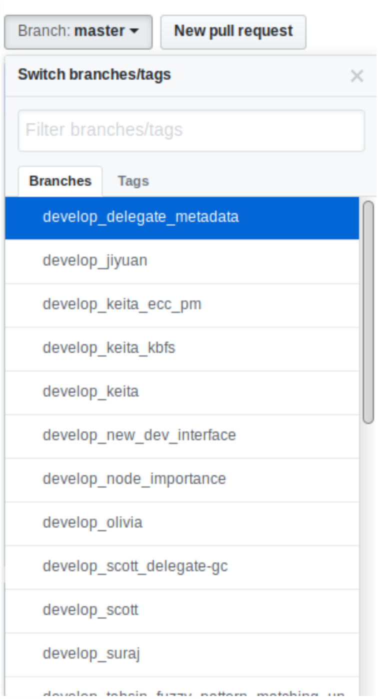
```
hmpi::breadth_first_search(graph, bfs_level_data, bfs_parent_data, source);
```


The College of Engineering
at the University of Notre Dame

Development Progress

- Framework only a couple of years old

- LLNL staff and interns are in active development
- Precise documentation for usage and rational are quite limited



The College of Engineering at the University of Notre Dame

References

[1] Roger Pearce, Maya Gokhale, and Nancy M. Amato. 2010. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '10). IEEE Computer Society, Washington, DC, USA, 1-11. DOI: https://doi.org/10.1109/SC.2010.34

[2] Roger Pearce, Maya Gokhale, and Nancy M. Amato. 2013. Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory. 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. Boston, MA, USA. 1. DOI: https://doi.org/10.1109/IPDPS.2013.72

[3] Roger Pearce, Maya Gokhale, and Nancy M. Amato. 2014. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '14). IEEE Press, Piscataway, NJ, USA, 549-559. DOI: https://doi.org/10.1109/SC.2014.50