Max-flow Parallelization in OpenMP vs. DSLs

Sophia Zalewski, Scott Routledge

Abstract

In this project, we compare the advantages and disadvantages of Domain Specific Languages (DSLs) for max-flow problems. We:

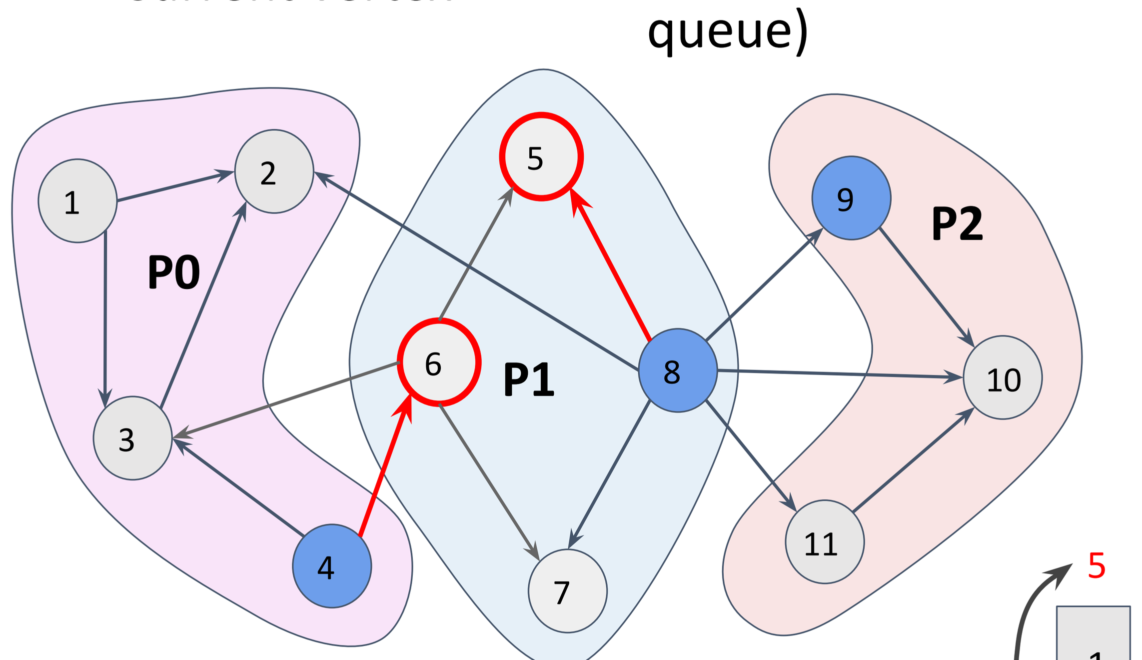
- Design our own compact DSL for graph vertex problems (**GraphLabLite**)
- Implement three different algorithms to solve max flow problems (Ford Fulkerson, Dinic's, Push-Relabel), and compare the difference between the OpenMP-parallelized algorithms and DSL-parallelized algorithm

Work Scheduling

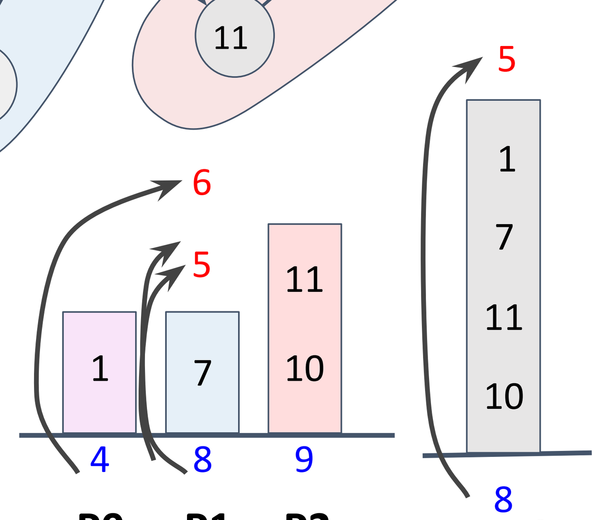
- Vertex Partitioning - vertices in the same neighborhood (ideally) assigned to the same processor, each proc does GAS on its assigned vertices, using locks / critical sections when needed depending on consistency model
- Simultaneous - GAS on every vertex, every iter
- Signaling - current vertex "signals" which neighbor vertices GAS is done on next (if any), user supplies signal call
  - Central Queue, Multi-Queue

Processor's Current vertex

Signalled vertex (added to GAS work queue)



- Sample vertex scheduling (top)
- Multi-queue (center)
- Central Queue (right)



GraphLabLite (Our DSL)

- Iterative vertex-centric operations, runs Gather (accumulate data from neighbors), Apply (update vertex data) Scatter (distribute updated data to neighbors) until converge
- Inspired by CMU's GraphLab implementation

Graph Representation

data stored at each edge and vertex + read-only global data, locks for each vertex and edge, proc info

```
struct tVertex{
  • void *data
  • lock v_lock
  • Int proc_id
  • vec<int>
    border_edges_in,
    border_edges_out

struct tEdge{
  • Int u,v
  • void *data
  • lock e_lock

class tGraph{
  • Int num nodes
  • vec<tEdge> edges
  • vec<int> in_edges
  • vec<int> out_edges
  • void *global_data
```

Which neighboring edges belong to a different processor (for enforcing consistency)

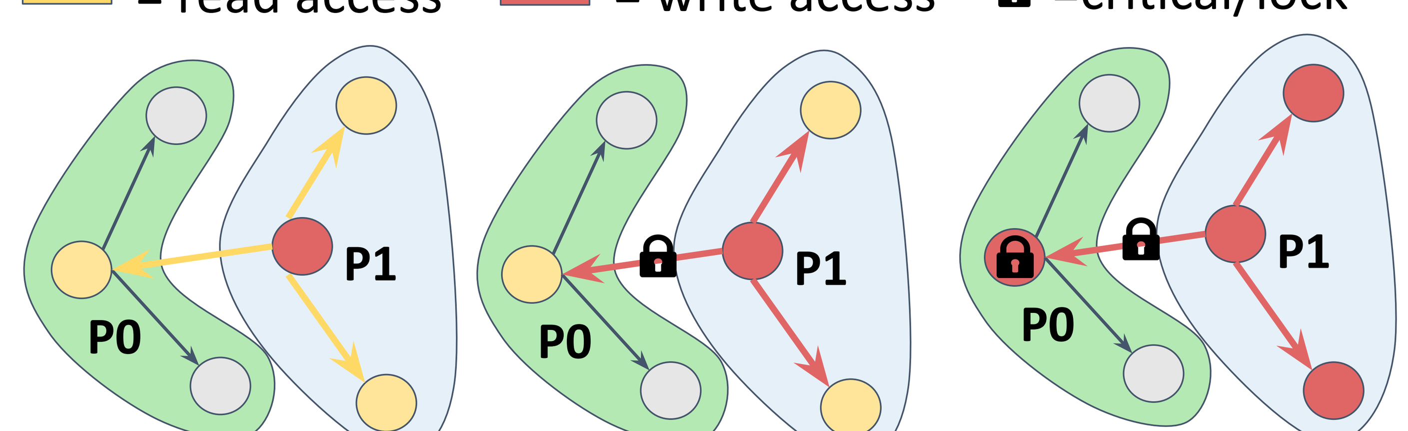
Which processor this vertex is assigned to

```
tGraph::gather_context INGOING;
tGraph::scatter_context OUTGOING;
tGraph::consist = FULL;
tGraph::schedule = SIMULTANEOUS;
```

User picks edges to do scatter/gather on, specifies schedule and consistency model

Consistency Models

= read access = write access =critical/lock

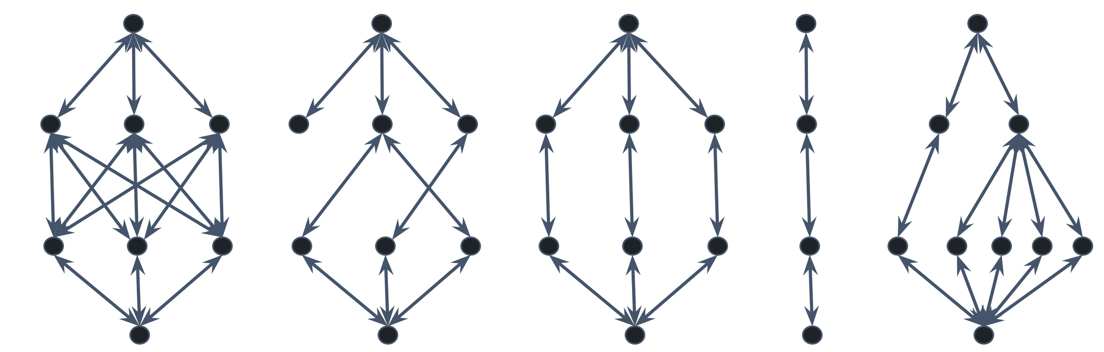


Vertex Cons. Edge Cons. Full Consistency

- **Vertex** - no locks or critical sections during GAS
- **Edge** - Lock this vertex's "border edges"
- **Full** - Lock border edges + neighbor vertices who are assigned to a different processor or can be accessed by another processor (has non-empty border edges)

Generating Tests (Method)

- Wrote a flow-graph generating python script, generates 5 different graph types (user specifies num nodes and layers + type)

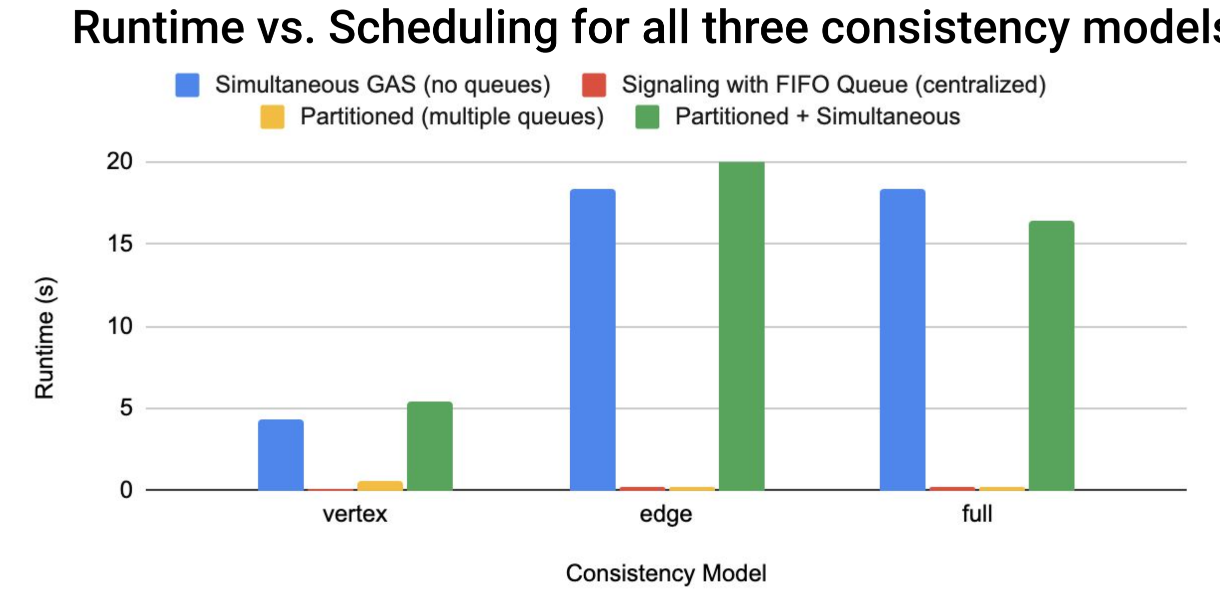


(1) Dense Layer Graph, (2) Sparse Layer Graph (3) Exclusive Path Graph, (4) Line Graph, (5) Unbalanced Graph

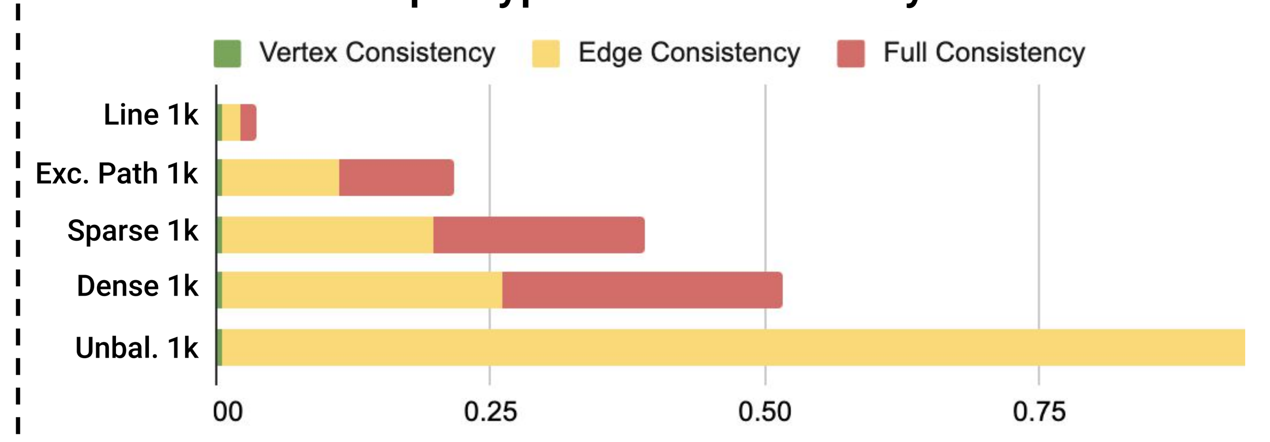
Push Relabel in GraphLabLite

- Push flow to vertices of lower "height" or relabel / increase height until graph converges

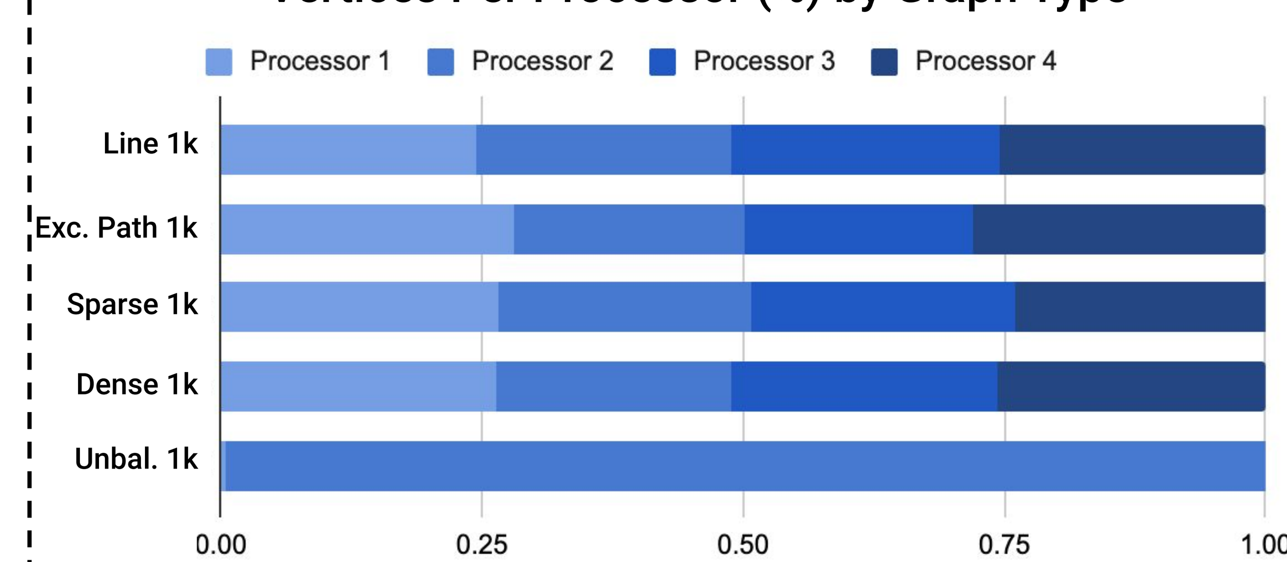
Runtime vs. Scheduling for all three consistency models



Vertices Requiring Locks/Critical Sections (%) by Graph Type and Consistency Model



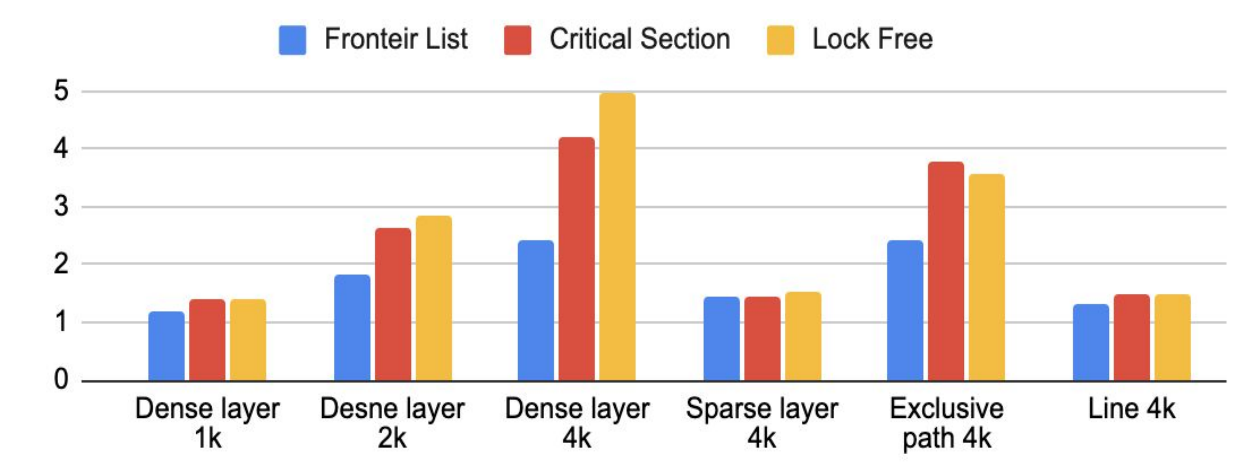
Vertices Per Processor (%) by Graph Type



Ford Fulkerson in OpenMP

- Uses BFS to repeatedly push flow from source to sink (parallelize the BFS)
- Lock-Free BFS fastest on all graph types
- Fastest speedup on dense layer graph with 4k nodes, 120 levels (5x speedup)

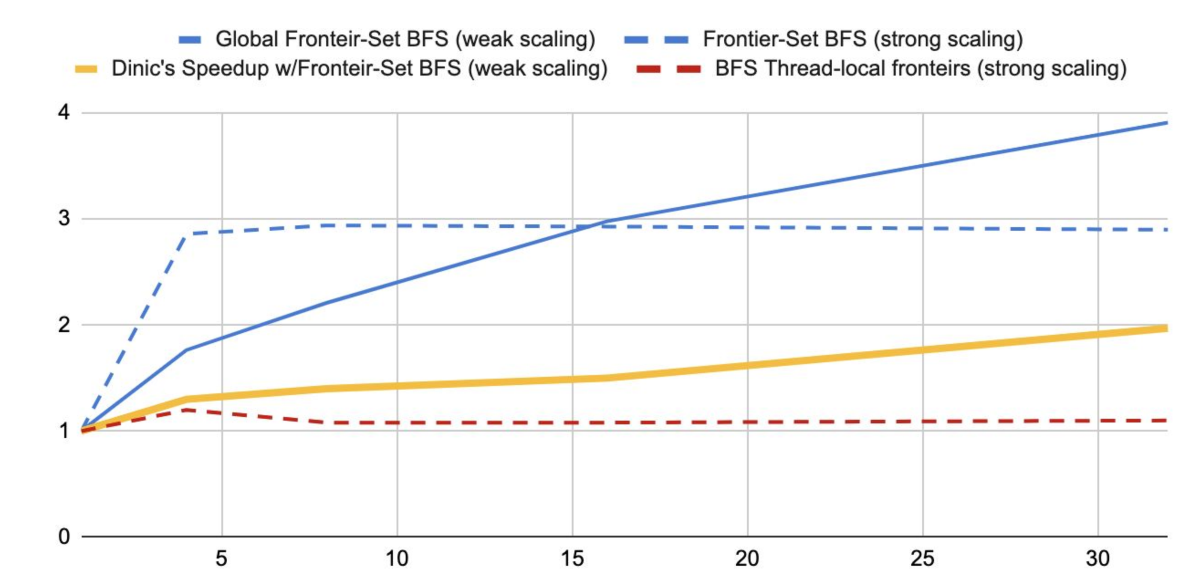
Ford Fulkerson Speedup vs. BFS Implementation



Dinic's in OpenMP

- BFS to make a "level graph", then DFS to push "blocking flows"
- Dinic's BFS (adjacency list) reaches around 4x speedup (weak scaling) on 32 threads => 2x Dinic's speedup

Dinic's BFS Speedup vs. Implementation, Scaling, #Threads



Comparison

- Overhead to learning DSL, but easier to program
- Easy to try different schedules w/o worrying about implementation
- Actual GraphLab implementation would run even faster than ours

Impl.	dense-5k runtime (s)	dense 40k-runtime (s)
FF Par	16.08770	—
Dinics Par	0.06737	0.141852
Push Relabel	0.80958	7.80220