High-Performance Graph Analytics in Shared Memory

Hans Vandierendonck, Jiawen Sun
Queen's University Belfast

20 July 2018

Image source: Microsoft

WHAT ARE GRAPH ANALYTICS

Graphs represent interactions between people or things

Graph analytics are algorithms that extract information from a graph

Graphs tend to grow large, and often tend to exhibit a power-law degree distribution

➢ “6 degrees of separation”

QUEEN'S UNIVERSITY BELFAST

WHY SHARED MEMORY?

Because of properties of the workload

- Little computation, mostly communication/synchronisation
- Data sets not so large, e.g., Twitter's follower graph fits in memory of a single server [Sharma PVDLB'16]
- Future memory technologies will increase capacity: High-Bandwidth Memory/die-stacking, storage-class memory
- Large-scale shared-memory systems implement a non-uniform memory access (NUMA) model

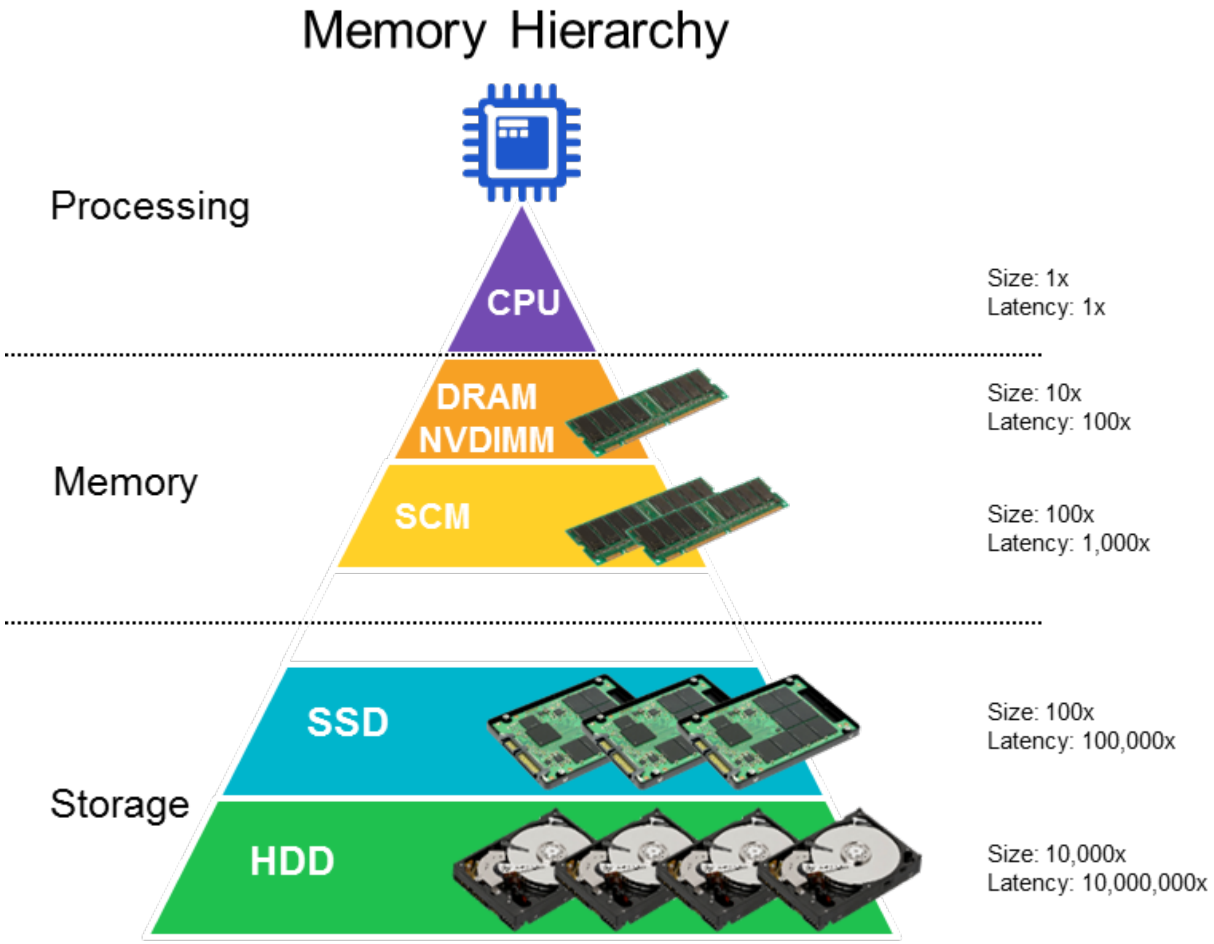


Image source: www.semiengineering.com

WHAT IS NUMA?

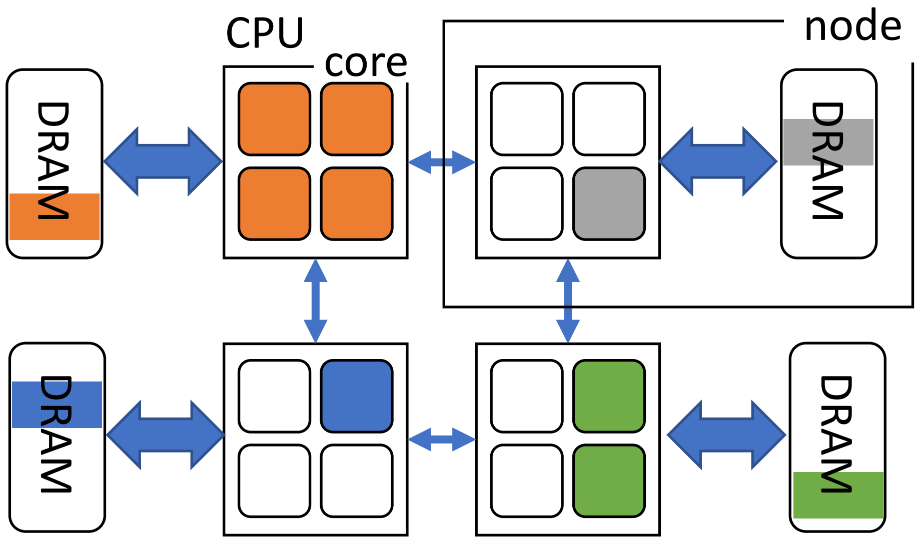
Each CPU socket is connected to local DRAM memory

Inter-node links provide access to “remote” DRAM memory

Local links have higher bandwidth and lower latency than inter-node links

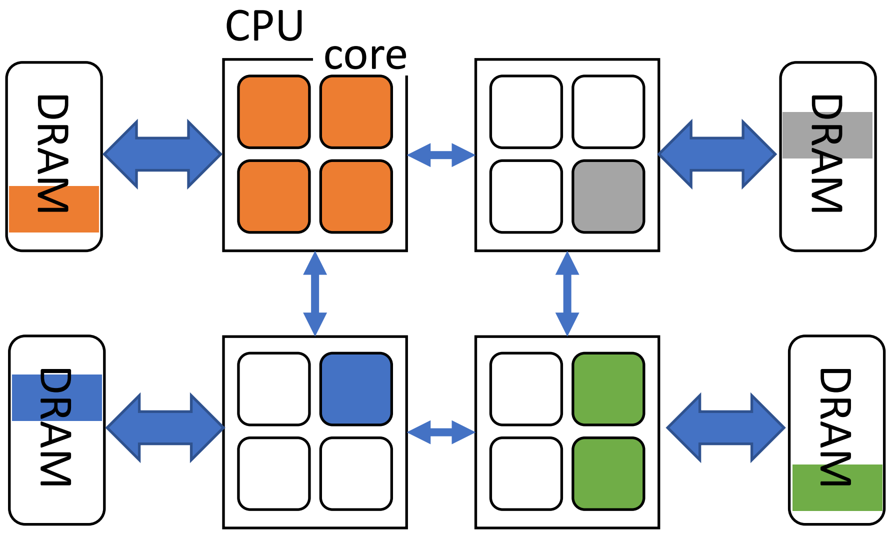
Difference is more pronounced for stores than for loads

In a program optimised for NUMA, CPU cores primarily access local DRAM



GOAL

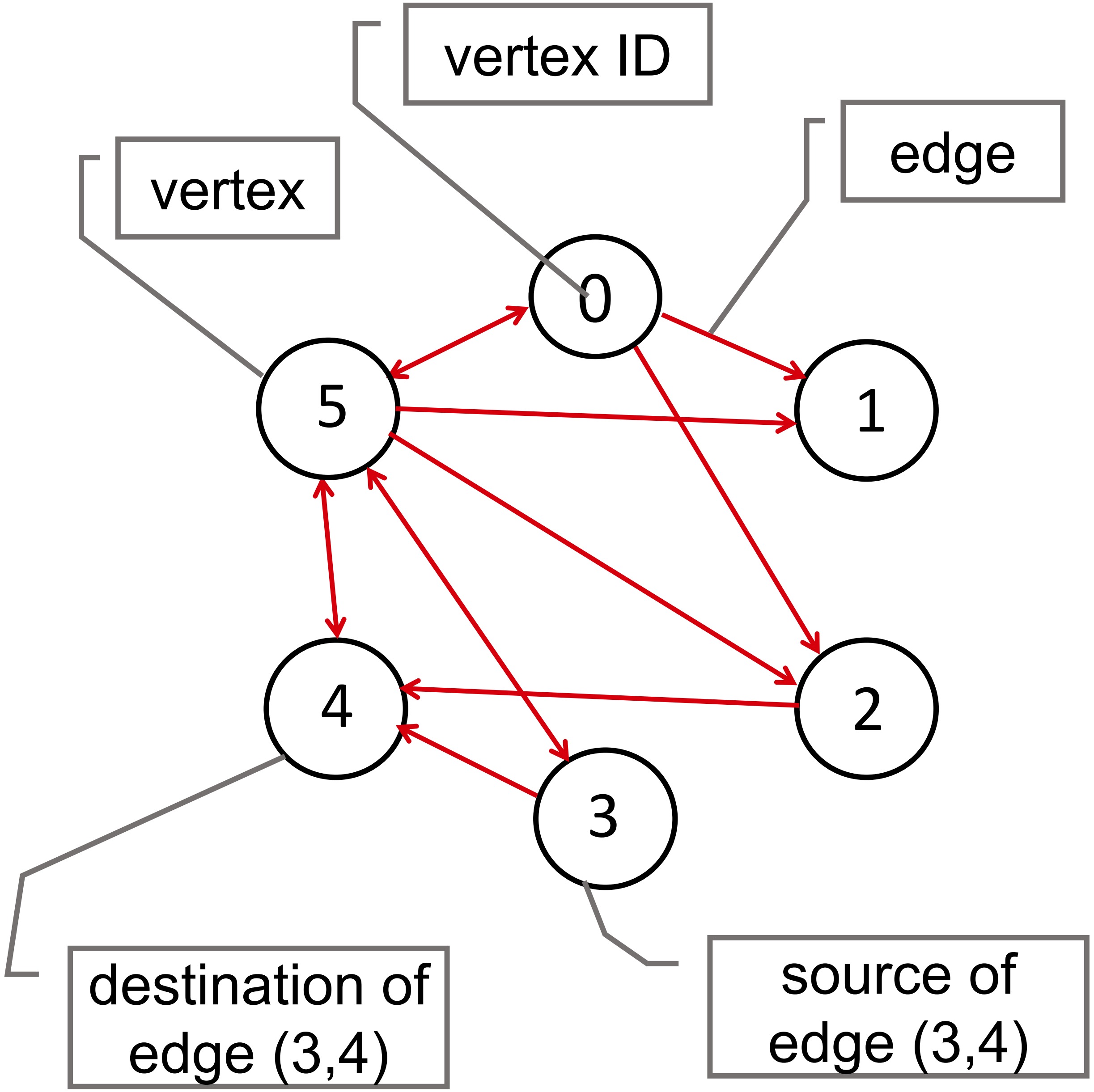
How to map graph analytics over immutable graphs onto a NUMA architecture while minimising execution time?



AGENDA

- Context and Goal
- Preliminaries
- Graph Algorithms
- Graph Analytics Frameworks
- Elements of High-Performance Graph Analytics
- NUMA-awareness
- Graph partitioning
- Load balance
- Conclusion and outlook

PRELIMINARIES



PRELIMINARIES

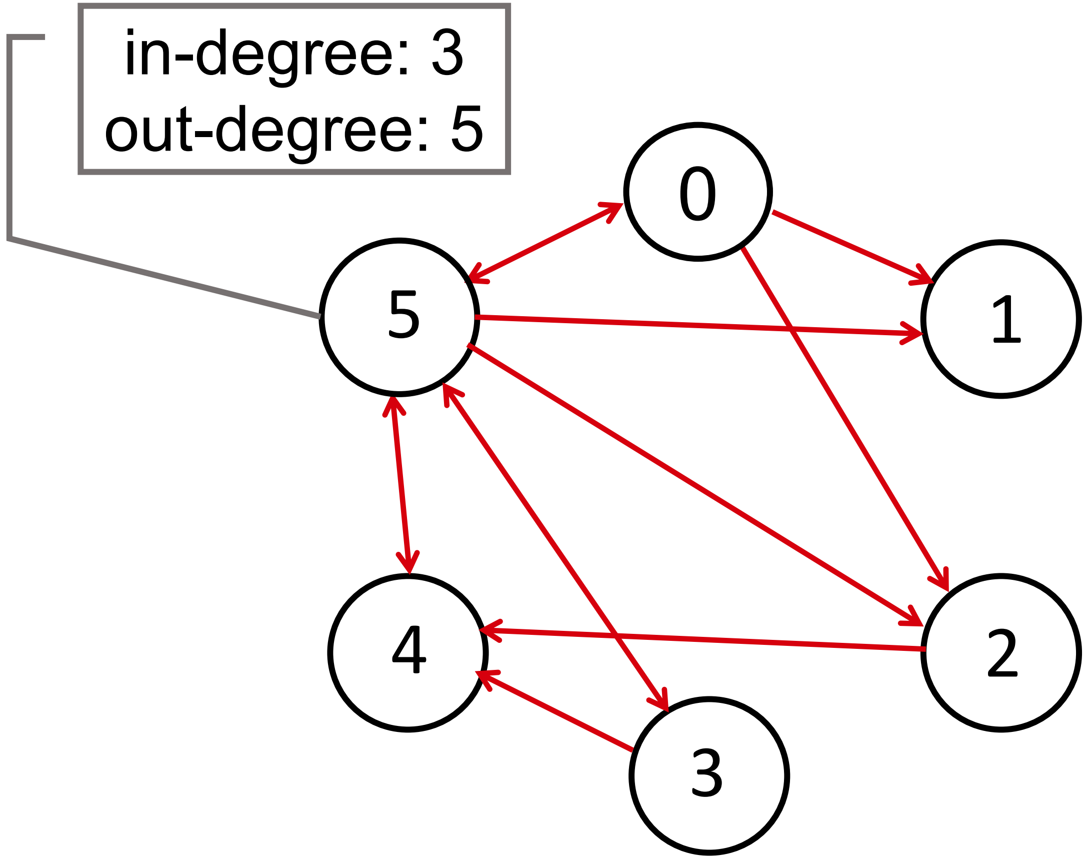
Graph G=(V,E) where

V: set of vertex labels

E $\subseteq$ V×V: set of pairs of vertices

Frontier F is a set of active vertices with F $\subseteq$ V

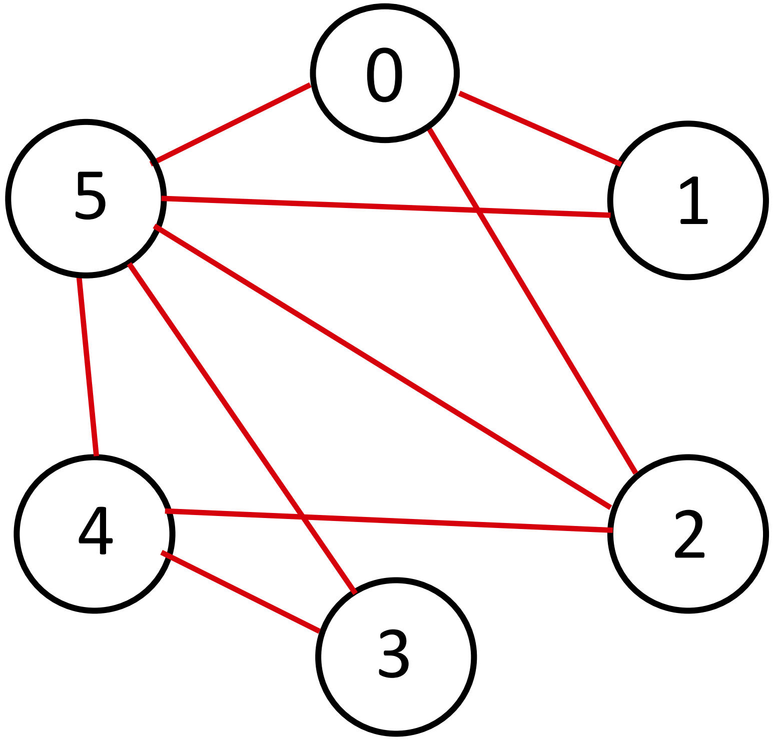
PRELIMINARIES



in-degree: #incoming edges

out-degree: #outgoing edges

PRELIMINARIES

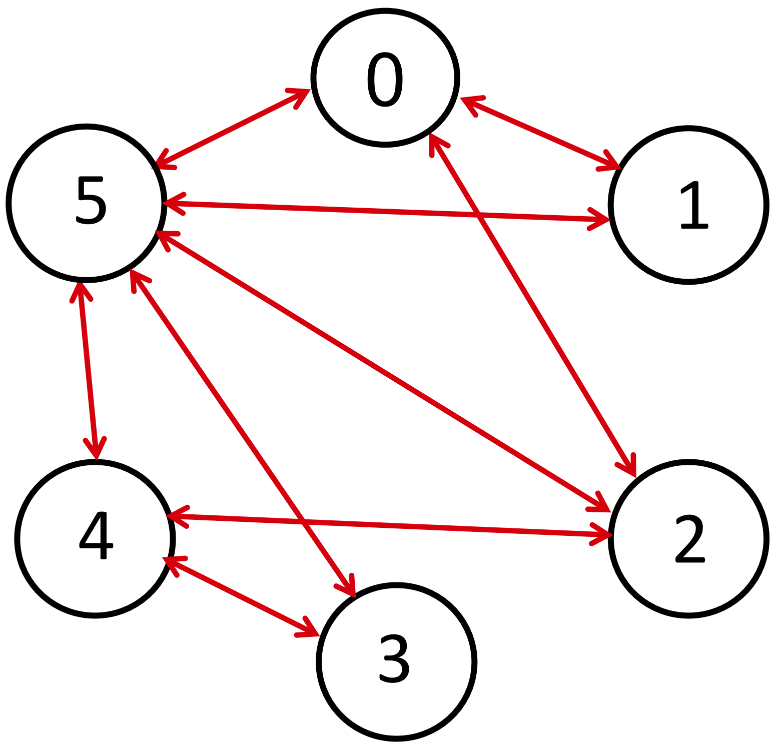


Directed graph: edges have a direction (source, destination)

Undirected graph: edges have no direction

if (u,v) ϵ E then (v,u) ϵ E

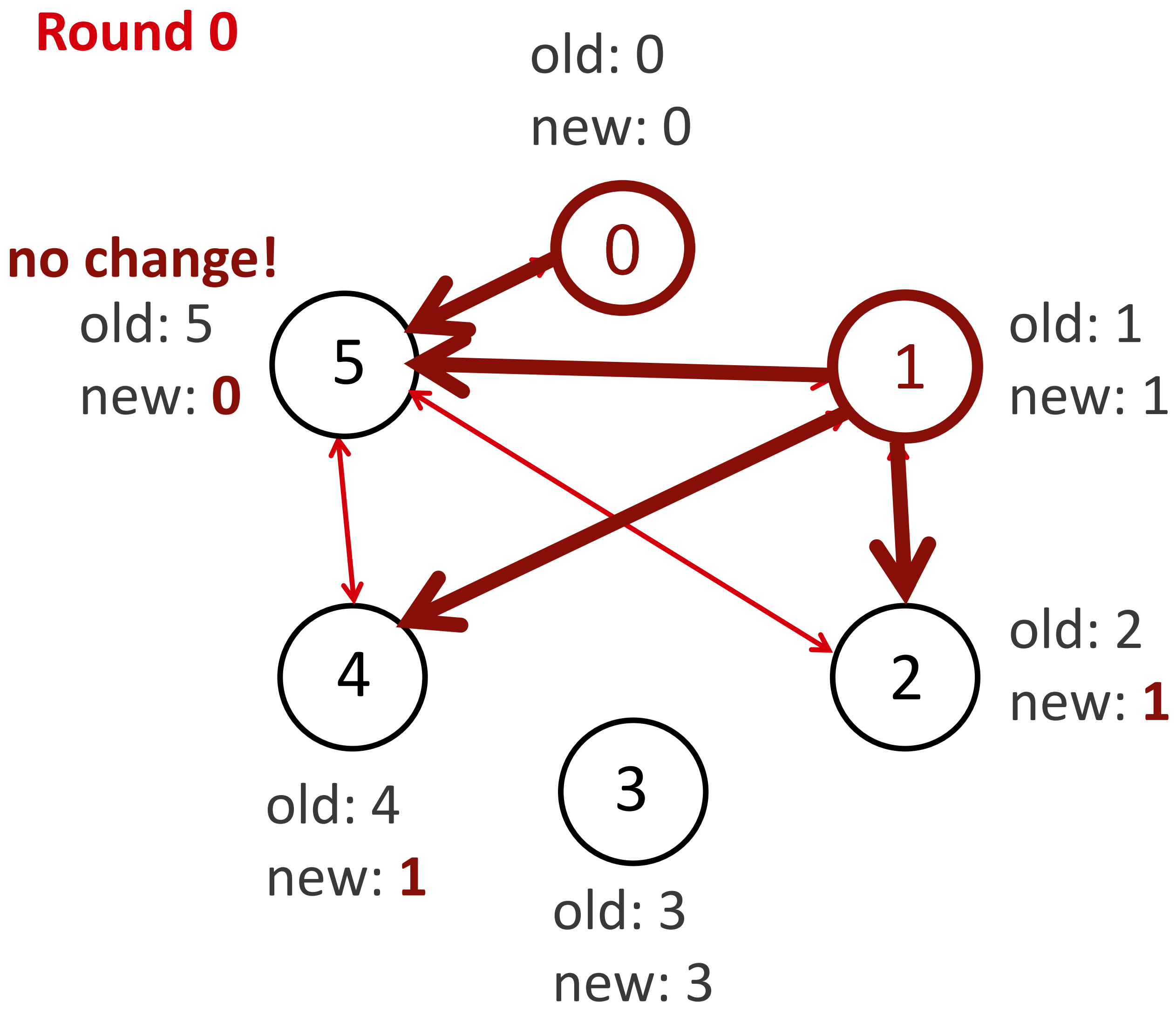
PRELIMINARIES



Undirected graphs are commonly represented such that every edge occurs twice

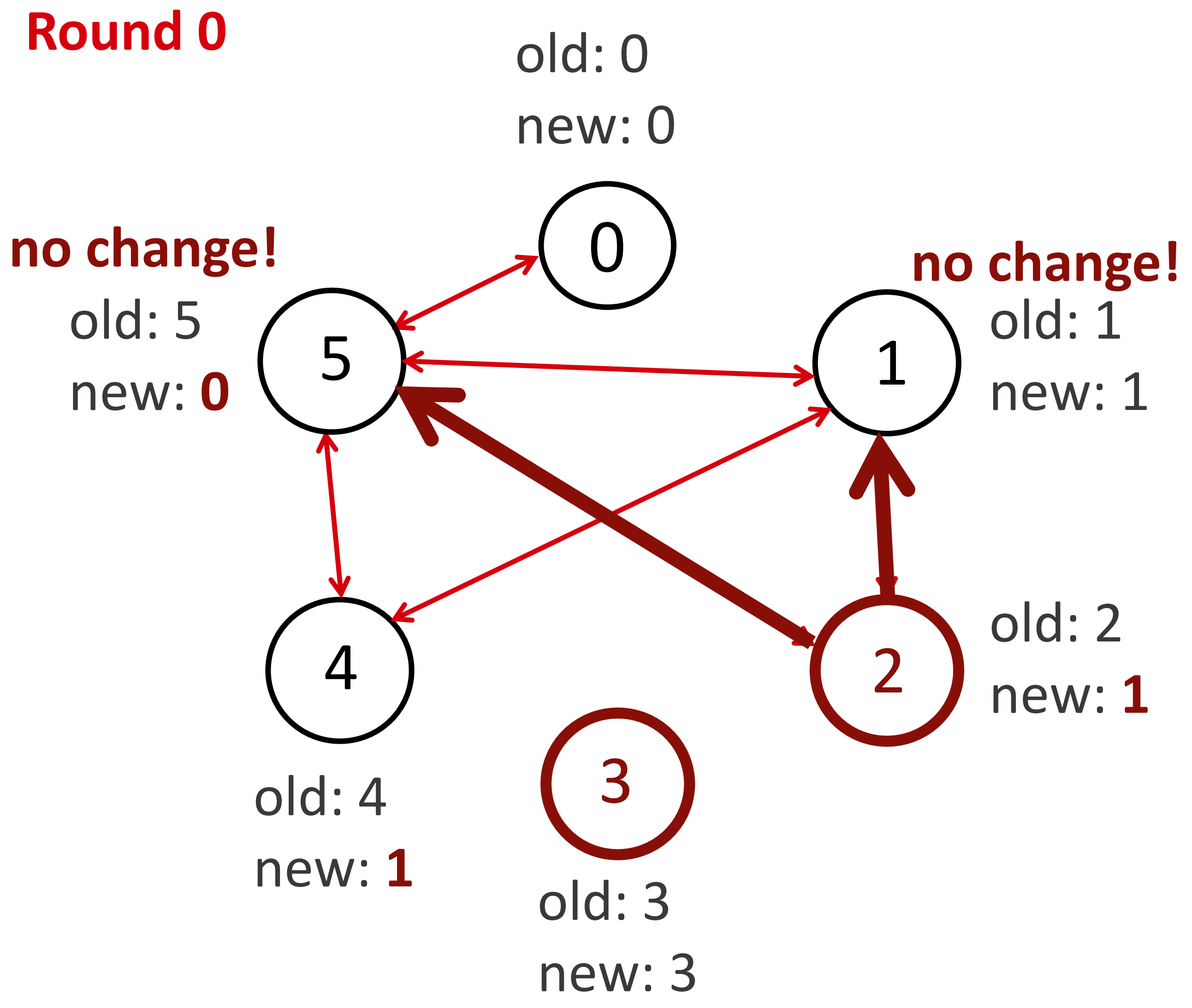
GRAPH ALGORITHMS

LABEL PROPAGATION



- Strongly connected components
- Initial label assignment of ”old” label, copied to “new” label
- Update rule: for (u,v) in E:
  - `new[v] = min(new[v],old[u])`
- Copy “new” to “old” and repeat update phase until no more changes made

LABEL PROPAGATION

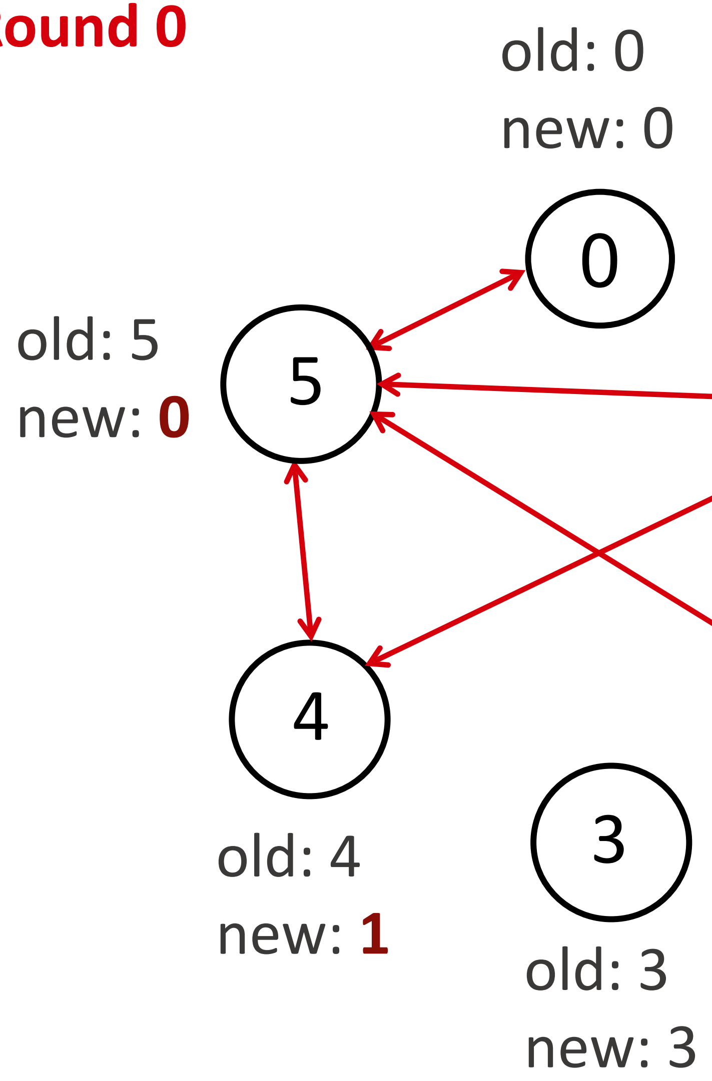


- Strongly connected components
- Initial label assignment of ”old” label, copied to “new” label

- Update rule: for (u,v) in E:
  - `new[v] = min(new[v],old[u])`
- Copy “new” to “old” and repeat update rule on all edges until no more changes made

LABEL PROPAGATION

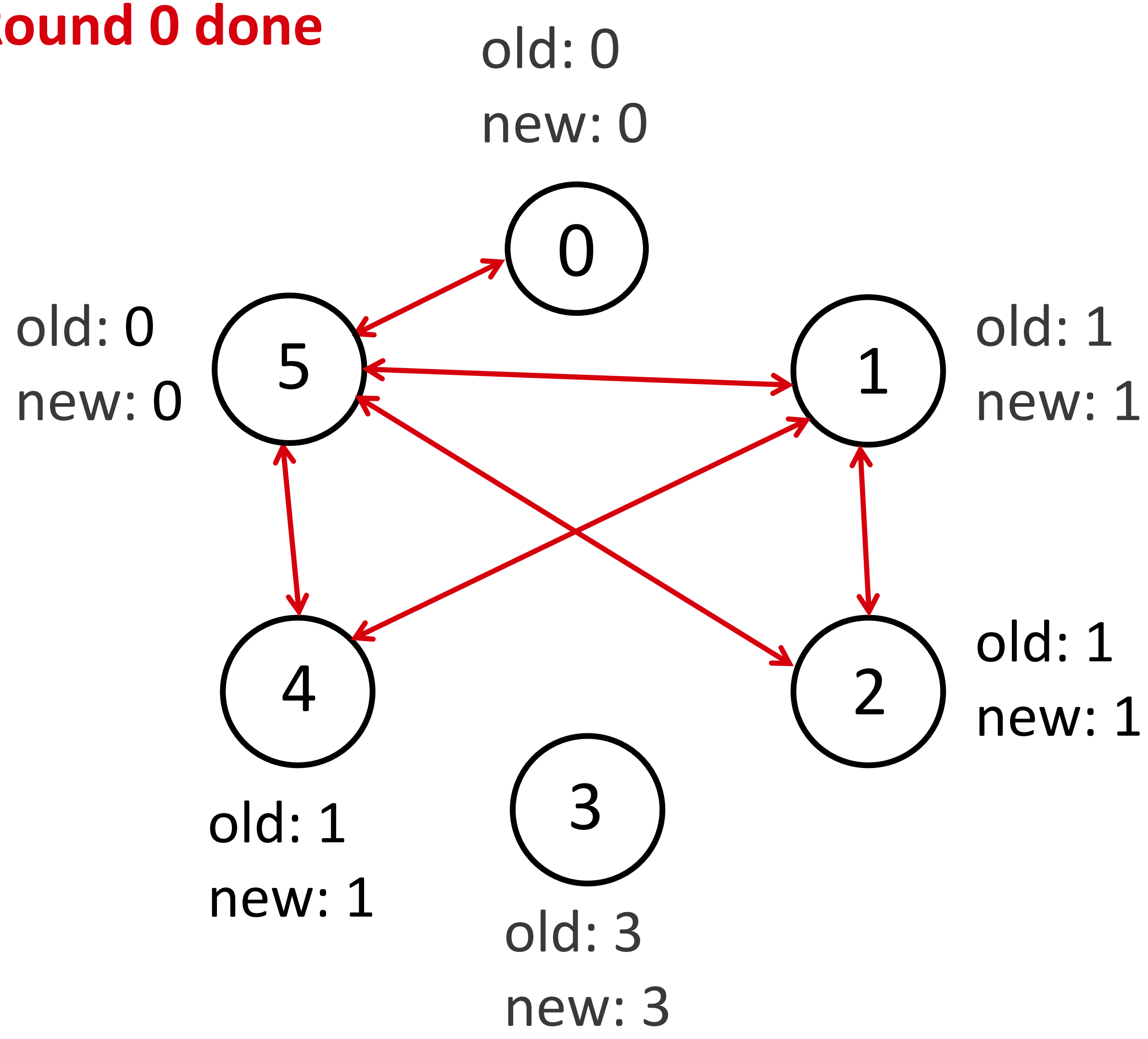
Round 0



Further propagating the labels “4” and “5” held by vertices 4 and 5 incurs no changes in the labels of other vertices

LABEL PROPAGATION

Round 0 done

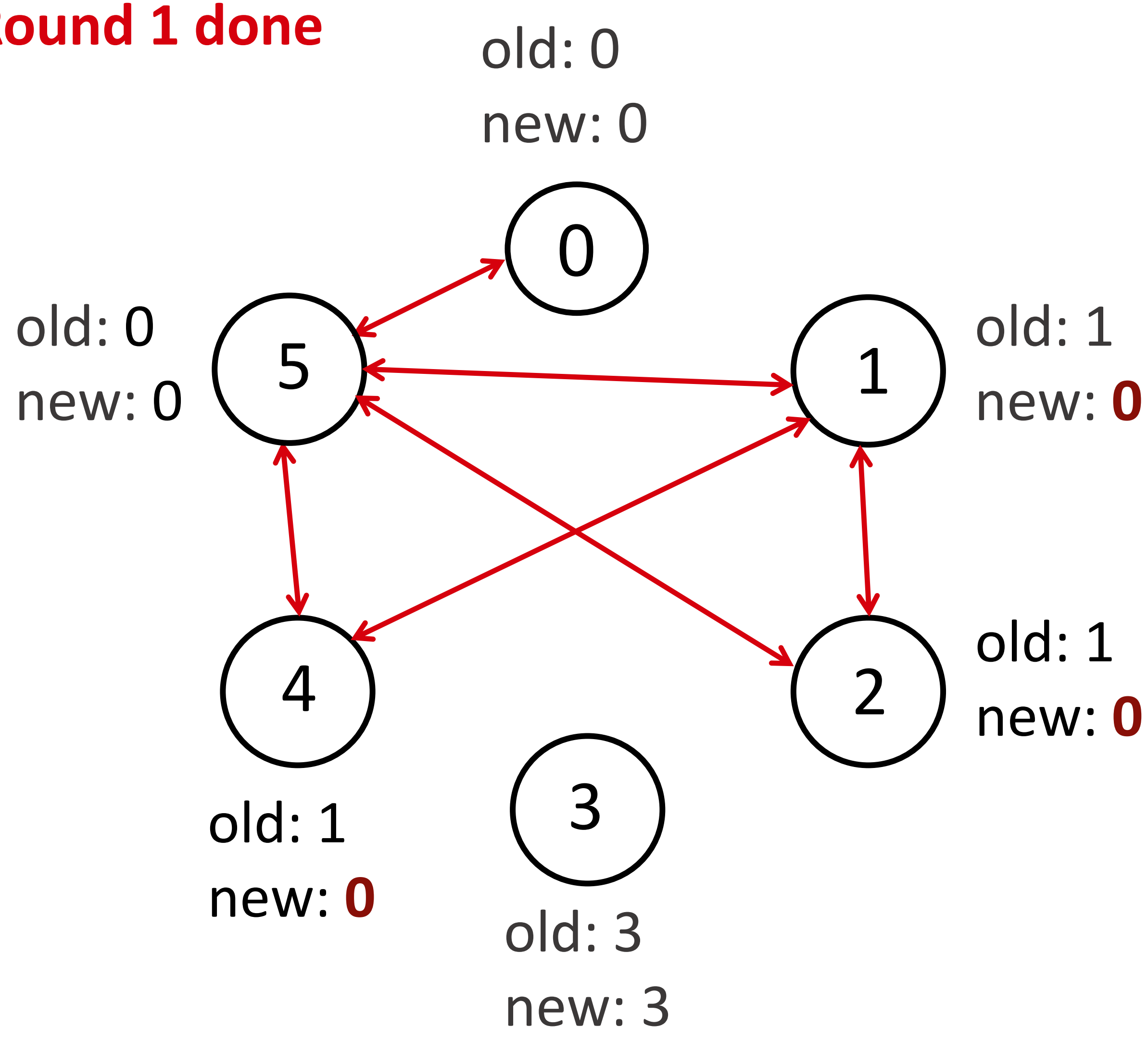


Round 0 of propagating labels has finished

Copy "new" to "old" and go again…

LABEL PROPAGATION

Round 1 done

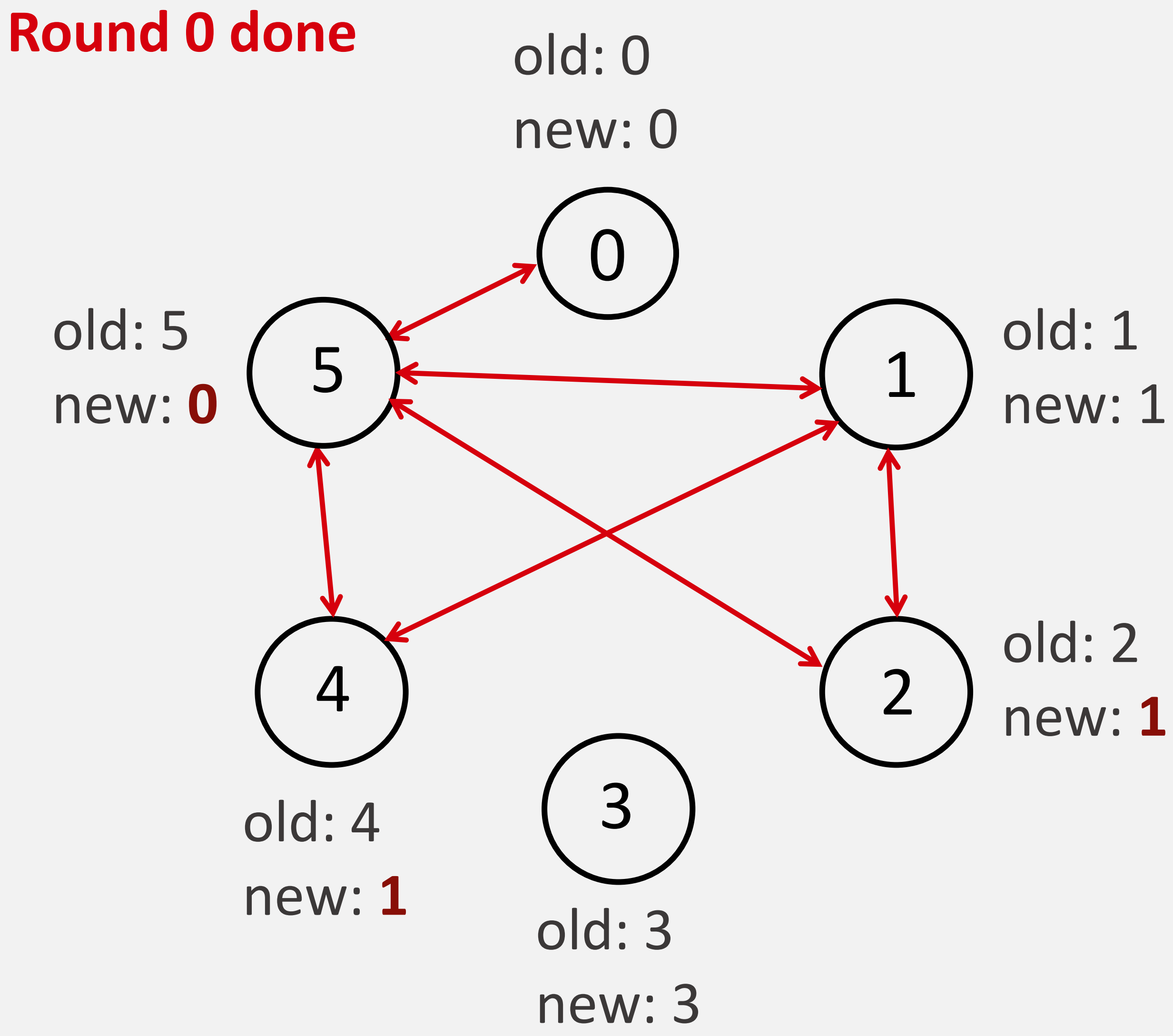


After round 1, label “0” has propagated to more vertices

We need to do one more round to ensure no further changes occur

QUEEN'S UNIVERSITY BELFAST

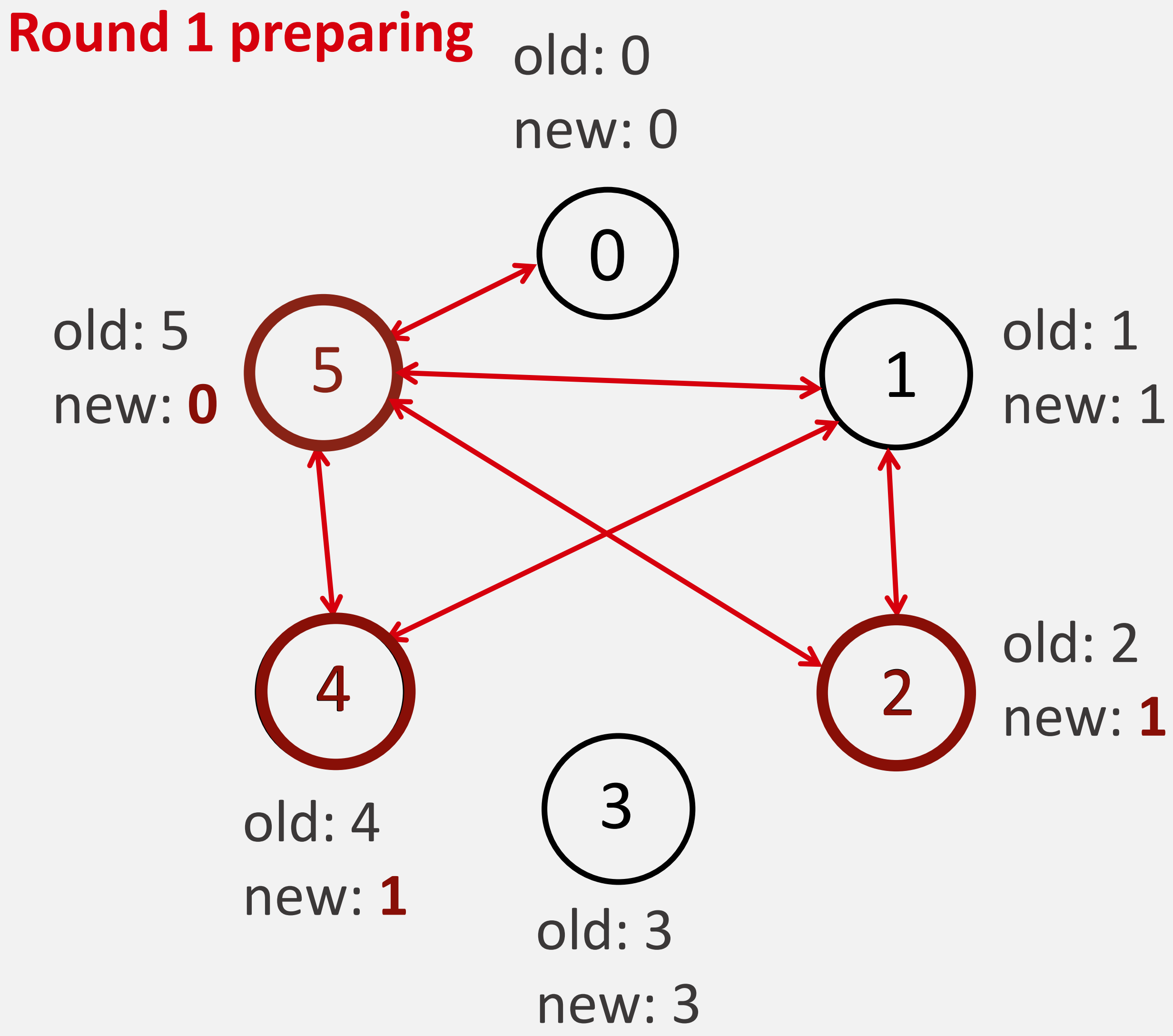
LABEL PROPAGATION WITH FRONTIER



Let's return to the state at the end of round 0

If in any round the label did not change, then there is no point in trying to propagate the label again

LABEL PROPAGATION WITH FRONTIER



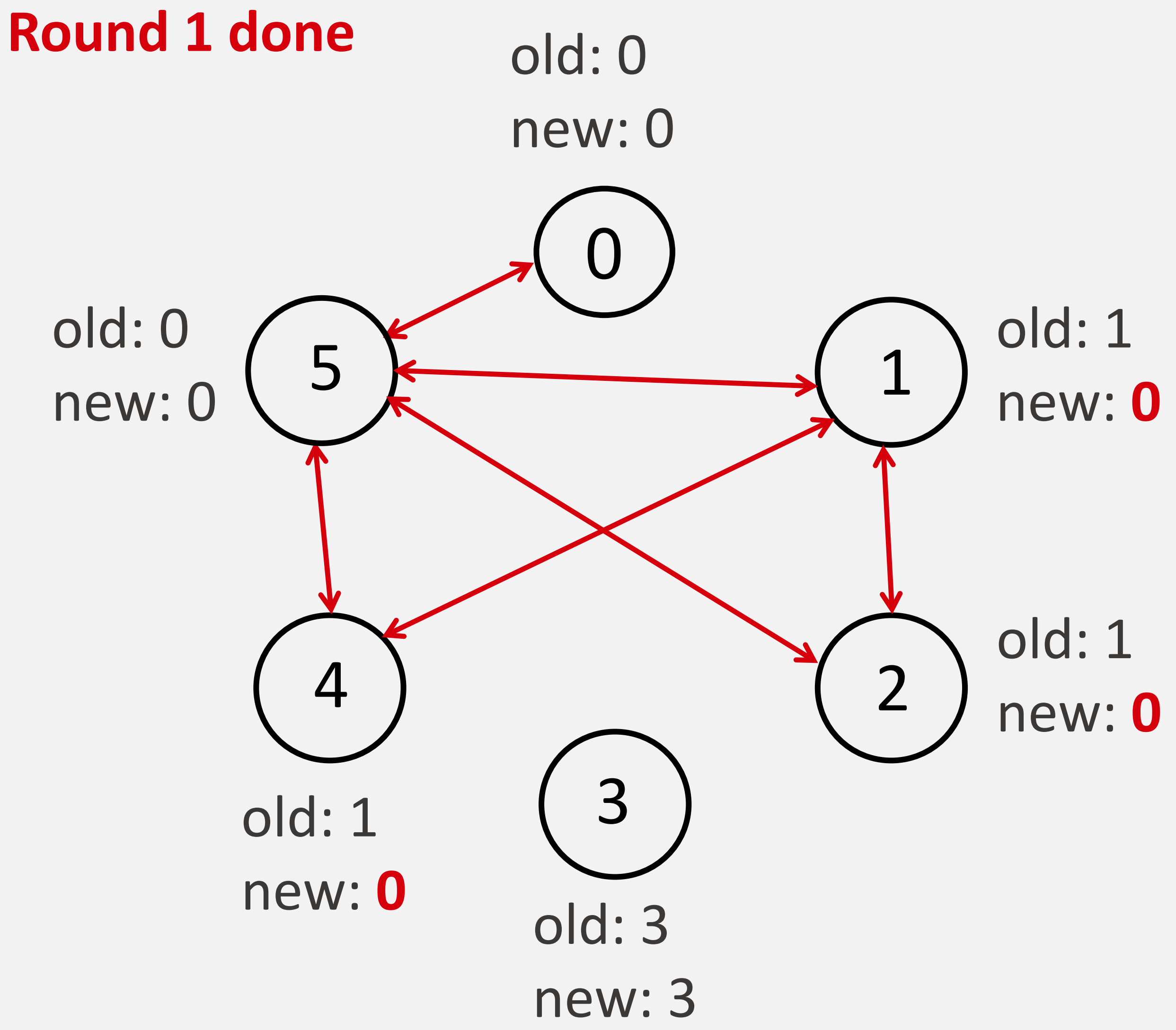
Let's return to the state at the end of round 0

Vertices 2, 4, 5 have changes in their label (`old[v]!=new[v]`)

Frontier = {2, 4, 5}

Only vertices 2, 4 and 5 are visited in round 1

LABEL PROPAGATION WITH FRONTIER



Let's return to the state at the end of round 0

Vertices 2, 4, 5 have changes in their label (`old[v]!=new[v]`)

Frontier = {2, 4, 5}

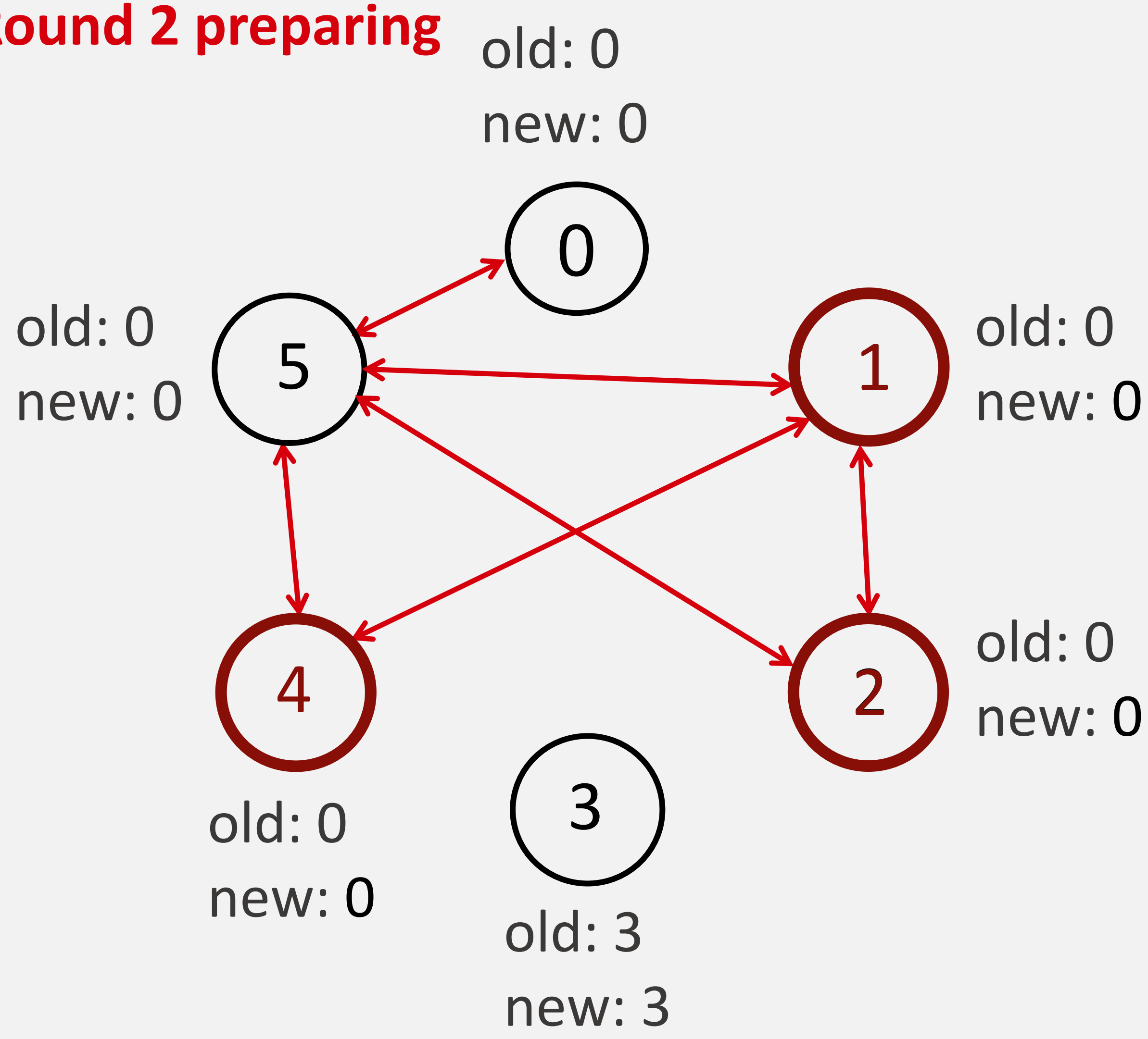
Only vertices 2, 4 and 5 are visited in round 1

In round 2:

Frontier = {1, 2, 4}

LABEL PROPAGATION WITH FRONTIER

Round 2 preparing



Let's return to the state at the end of round 0

Vertices 2, 4, 5 have changes in their label (`old[v]!=new[v]`)

Frontier = {2, 4, 5}

Only vertices 2, 4 and 5 are visited in round 1

In round 2:

Frontier = {1, 2, 4}

Nothing changes

GRAPH ANALYTICS FRAMEWORKS

QUEEN'S UNIVERSITY BELFAST

LIGRA

size(U : frontier) : N

returns |U|

EdgeMap(G : graph,

U : frontier,

F : (vertex × vertex) → bool,

C : vertex → bool) : frontier

VertexMap(U : frontier,

F : vertex → bool) : frontier

[Shun PPoPP'13]

Assume graph G=(V,E)

EdgeMap applies an operation F to each edge (u,v) ∈ E where u ∈ U and C(v) = true. It returns a frontier that contains all v where any call to F(u,v) returned true

VertexMap applies an operation F to each vertex v ∈ U and returns a frontier that contains v iff v ∈ U and F(v) = true

In both cases, F may have side effects, e.g., updating properties for the vertices

QUEEN'S UNIVERSITY BELFAST

LABEL PROPAGATION IN LIGRA

Algorithm 8 Connected Components

```
IDs = {0, . . . , |V| − 1}                ▷ initialized such that IDs[i] = i
prevIDs = {0, . . . , |V| − 1}            ▷ initialized such that prevIDs[i] = i

procedure CCUPDATE(s, d)
    origID = IDs[d]
    if (WRITEMIN(&IDs[d], IDs[s])) then
        return (origID == prevIDs[d])
    return 0

procedure COPY(i)
    prevIDs[i] = IDs[i]
    return 1

procedure CC(G)
    Frontier = {0, . . . , |V| − 1}       ▷ vertexSubset initialized to V
    while (SIZE(Frontier) ≠ 0) do
        Frontier = VERTEXMAP(Frontier, COPY)
        Frontier = EDGEMAP(G, Frontier, CCUPDATE, C_true)
    return IDs
```

Source: Shun PPoPP’13

writeMin is an atomic “fetch_and_min” operation

Like compare-and-set, returns true if destination is successfully modified

QUEEN’S UNIVERSITY BELFAST

POWERGRAPH

```
interface GASVertexProgram(u) {
  // Run on gather_nbrs(u)
  gather(D_u, D_(u,v), D_v) → Accum
  sum(Accum left, Accum right) → Accum
  apply(D_u, Accum) → D_u^new
  // Run on scatter_nbrs(u)
  scatter(D_u^new, D_(u,v), D_v) → (D_(u,v)^new, Accum)
}
```

Figure 2: All PowerGraph programs must implement the stateless gather, sum, apply, and scatter functions.

Algorithm 1: Vertex-Program Execution Semantics

Input: Center vertex u

if *cached accumulator* a_u *is empty* **then**
    foreach *neighbor* v *in gather_nbrs(u)* **do**
        $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{(u,v)}, D_v))$
    end
end
$D_u \leftarrow \text{apply}(D_u, a_u)$
foreach *neighbor* v *scatter_nbrs(u)* **do**
    $(D_{(u,v)}, \Delta a) \leftarrow \text{scatter}(D_u, D_{(u,v)}, D_v)$
    if a_v *and* Δa *are not Empty* **then** $a_v \leftarrow \text{sum}(a_v, \Delta a)$
    else $a_v \leftarrow$ Empty
end

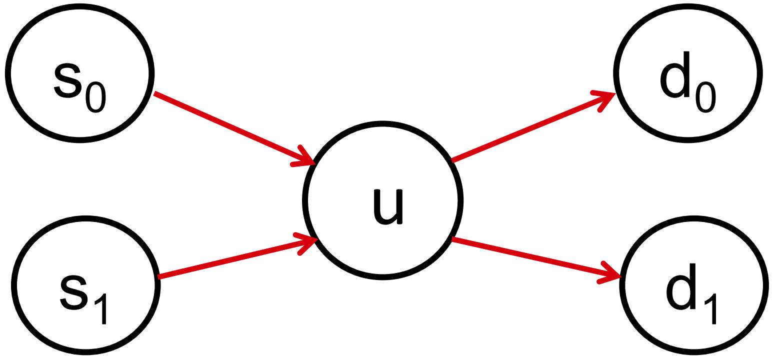
[Gonzalez OSDI'12]

Similar concepts, presented differently

Vertices 'activated' by explicit call as opposed to recording frontier

Needs to maintain state on vertices and on edges

Distributed framework



QUEEN'S UNIVERSITY BELFAST

PEGASUS

GIM-V, or 'Generalized Iterative Matrix-Vector multiplication' is a generalization of normal matrix-vector multiplication. Suppose we have a n by n matrix M and a vector v of size n. Let $m_{i,j}$ denote the (i,j)-th element of M. Then the usual matrix-vector multiplication is

$$M \times v = v' \text{ where } v'_i = \sum_{j=1}^{n} m_{i,j} v_j.$$

There are three operations in the previous formula, which, if customized separately, will give a surprising number of useful graph mining algorithms:

1) combine2: multiply $m_{i,j}$ and v_j.
2) combineAll: sum n multiplication results for node i.
3) assign: overwrite previous value of v_i with new result to make v'_i.

In GIM-V, let's define the operator $\times_G$, where the three operations can be defined arbitrarily. Formally, we have:

$$v' = M \times_G v$$

where $v'_i = \text{assign}(v_i, \text{combineAll}_i(\{x_j \mid j = 1..n, \text{ and } x_j = \text{combine2}(m_{i,j}, v_j)\}))$.

[Kang ICDM'09]

Similar concepts, presented differently

Uses connection between graphs and their adjacency matrix

Generalized matrix-vector multiplication captures 'accumulation' concept

Essentially says that graph algorithms may be represented as semi-rings

ELEMENTS OF HIGH-PERFORMANCE GRAPH ANALYTICS

QUEEN'S UNIVERSITY BELFAST

GRAPH ANALYTICS STRUCTURE

```
frontier F := …;
frontier newF := { };
for edge (u,v) ∈ E do
    if u ∈ F then
        if C(v) and op(u,v) then
            newF = newF ∪ { v };
        fi
    fi
od
```

- ***op*** implements the update of vertex properties
- ***op, C*** are algorithm-specific
- ***op*** returns true if destination should be considered in the next round
- ***op***(u,v) is usually of the form

  `new[v]=new[v]⊕old[v]`

  where ⊕ is a commutative and associative binary operation (reduction)
- ***C***(v) checks convergence

CONVERGENCE

```
frontier F := …;
frontier newF := { };
for edge (u,v) ∈ E do
    if u ∈ F then
        if C(v) and op(u,v) then
            newF = newF ∪ { v };
        fi
    fi
od
```

Shun PPoPP'13:

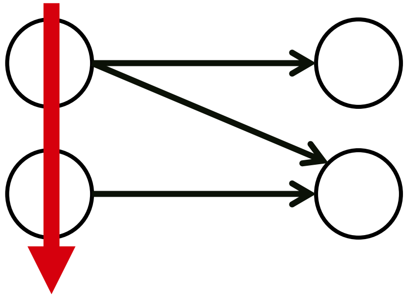
"The function C is useful in algorithms where a value associated with a vertex only needs to be updated once (i.e. breadth-first search)."

The paper also checks convergence for betweenness centrality

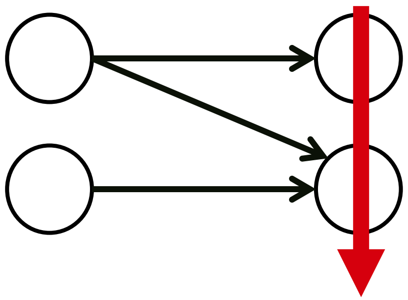
Real usefulness depends on how the graph is traversed

GRAPH DATA STRUCTURES

- **Compressed Sparse Rows (CSR)**
  - List outgoing edges for each vertex
  - “Forward” traversal
  - “Top-down” traversal
  - “Push”
  - “Vertex-centric”



- **Compressed Sparse Columns (CSC)**
  - List incoming edges for each vertex
  - “Backward” traversal
  - “Bottom-up” traversal
  - “Pull”
  - “Vertex-centric”



Frontier: CSR allows to skip edges for inactive vertices (u∉F)

Pruning: CSC allows to skip edges for pruned vertices (C(v)=false)

CSR-BASED EDGEMAP

`frontier F := …;`

`frontier newF := { };`

for vertex u ∈ V **do**

    if u ∈ F **then**

        for vertex v ∈ out(u) **do**

            if ~~*C*(v) **and**~~ *op*(u,v) **then**

                newF = newF ∪ { v };

fi od fi od

- Checking frontier is compulsory: **op**(u,v) may be called only if u∈F
- Pruning is not compulsory: may call **op**(u,v) if **C**(v)=false
- As **op**(u,v) is only a hand-full of instructions, there is little benefit in using **C**(v) in CSR

QUEEN'S UNIVERSITY BELFAST

CSC-BASED EDGEMAP

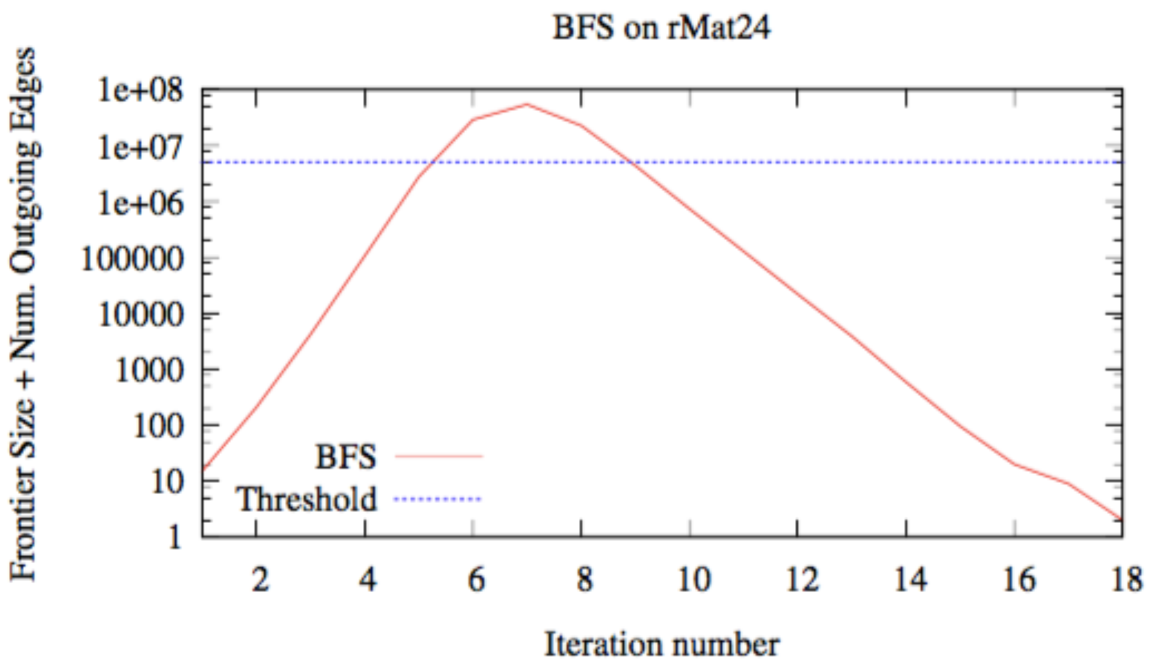
```
frontier F := …;
frontier newF := { };
for vertex v ∈ V do
        if C(v) then
             for vertex u ∈ in(v) do
                   if u ∈ F then
                           if op(u,v) then
                                newF = newF ∪ { v };
                          if not C(v) then break; fi
fi fi fi od od
```

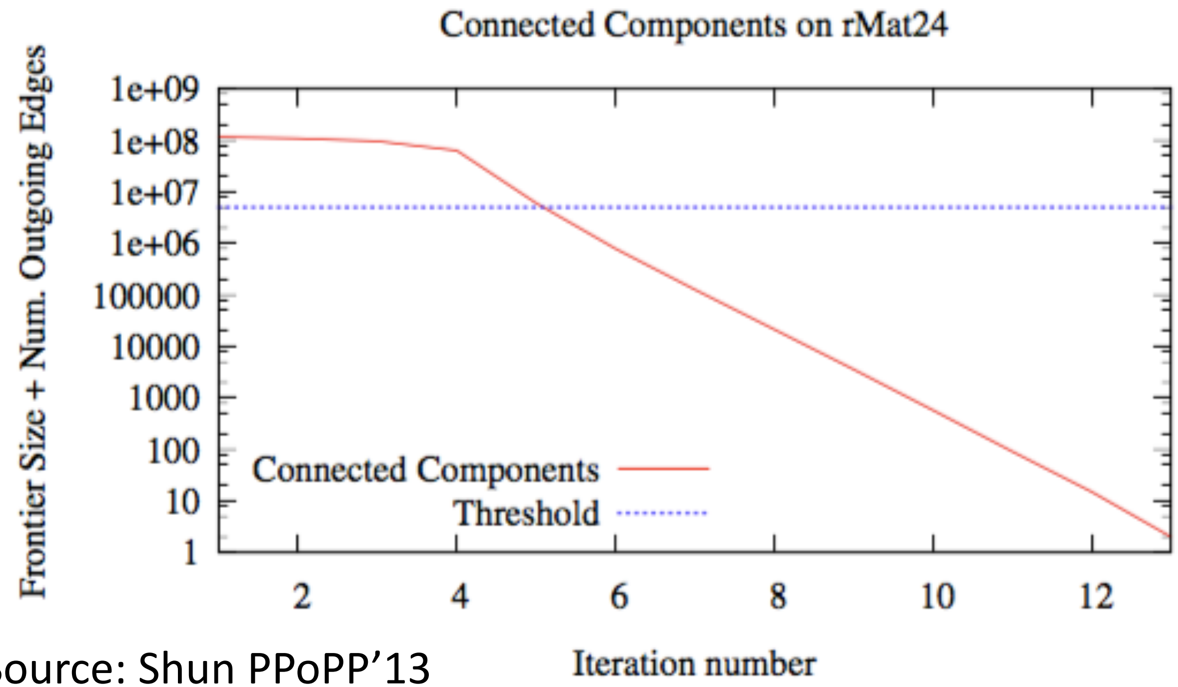
- Pruning is highly effective in CSC
- Allows to early terminate visiting the in-edges of u
- Or skip in-edges alltogether

EVOLUTION OF FRONTIER SIZE

Algorithms exhibit one of three primary patterns:

- flat
- shrink
- grow then shrink

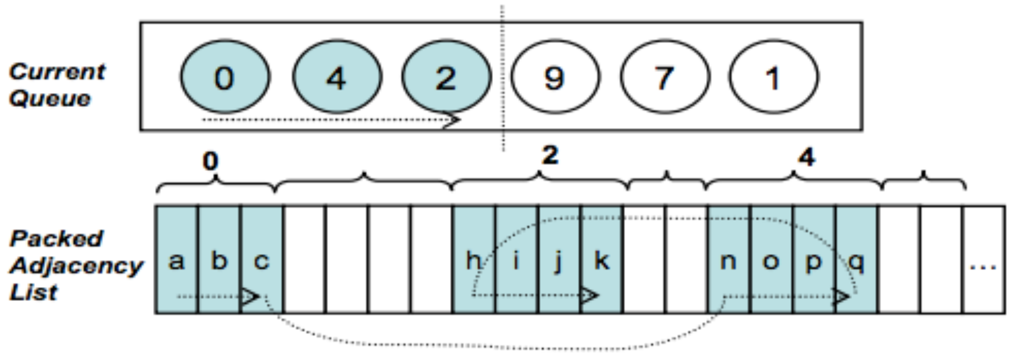




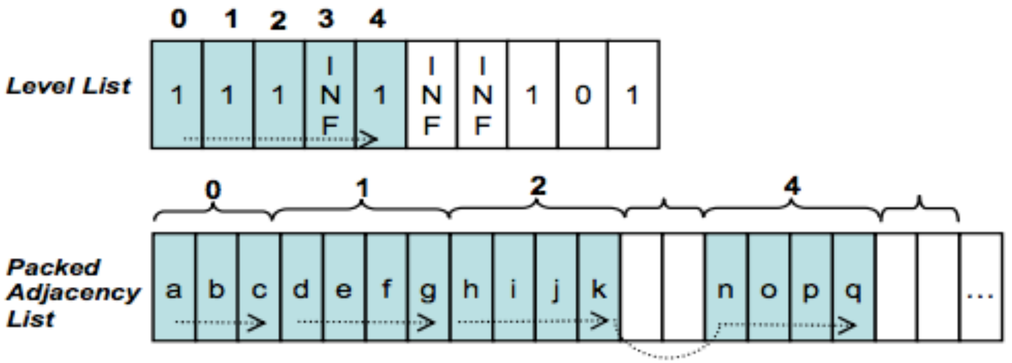
Source: Shun PPoPP'13

QUEEN'S UNIVERSITY BELFAST

FRONTIER REPRESENTATION



(a) Data-Access Pattern of Queue-Based Method



(b) Data-Access Pattern of Read-Based Method

"Sparse" frontiers

- Few bits set
- Queue of active vertex IDs

"Dense" frontiers

- Many bits set
- Bitmap or array of booleans

Dynamically switch as frontier size changes

[Hong et al, PACT'11]

CSR-BASED EDGEMAP WITH SPARSE FRONTIER

frontier F := …; // queue

frontier newF := { }; // queue

for vertex u ∈ F **do**

    for vertex v ∈ out(u) **do**

        if ~~C(v) **and**~~ *op*(u,v) **then**

            newF = newF ∪ { v }; // append

fi od od

- Reminder: dense frontiers:

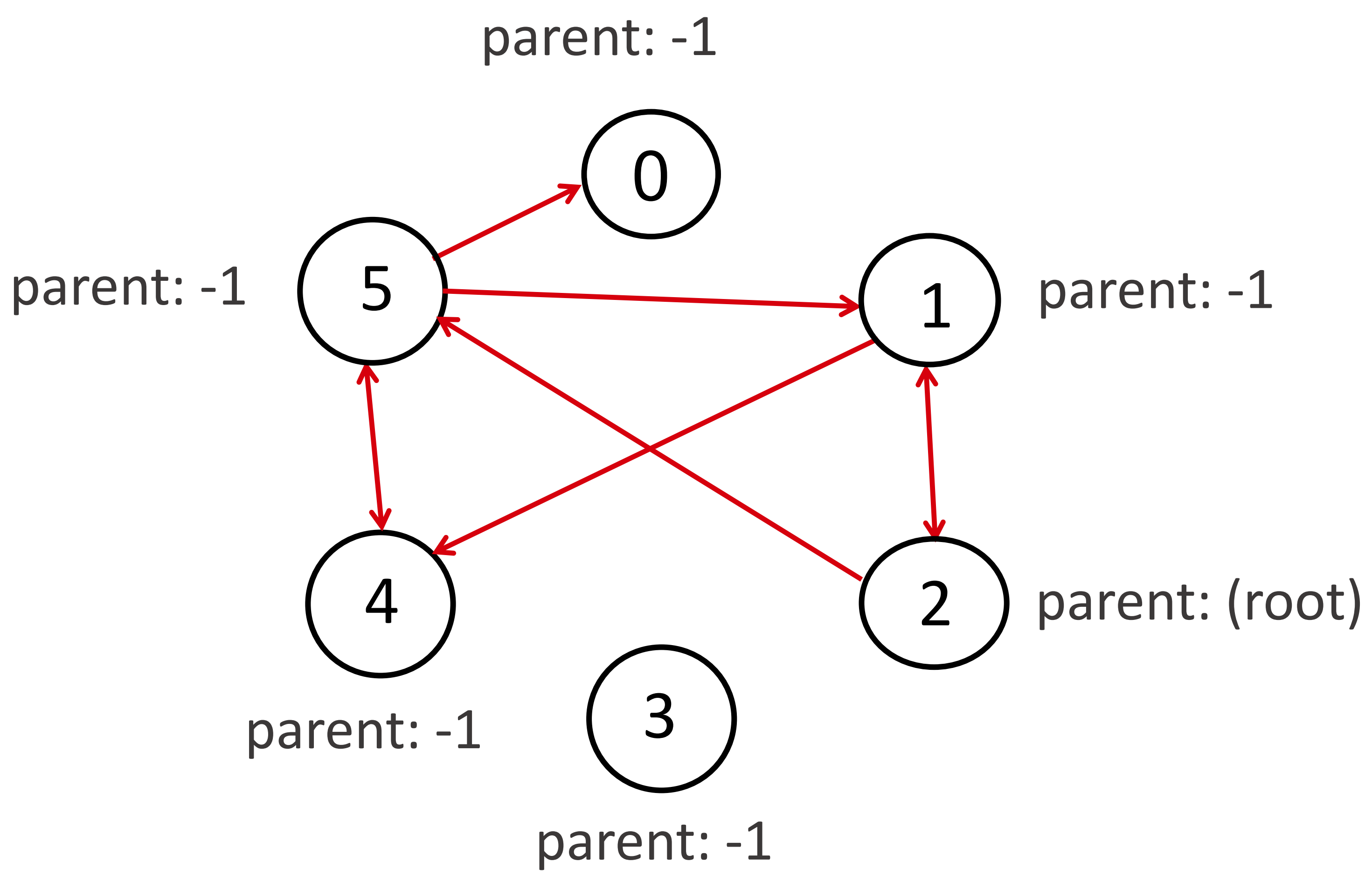
for vertex u ∈ V **do**

    if u ∈ F **then**

…

- Iteration over F is efficient when stored as a queue
- When F is stored as a queue, only CSR is efficient
- new frontier may contain duplicates!

QUEEN'S UNIVERSITY BELFAST



BREADTH-FIRST SEARCH

Starting from a root vertex, identify a shortest path to all other vertices

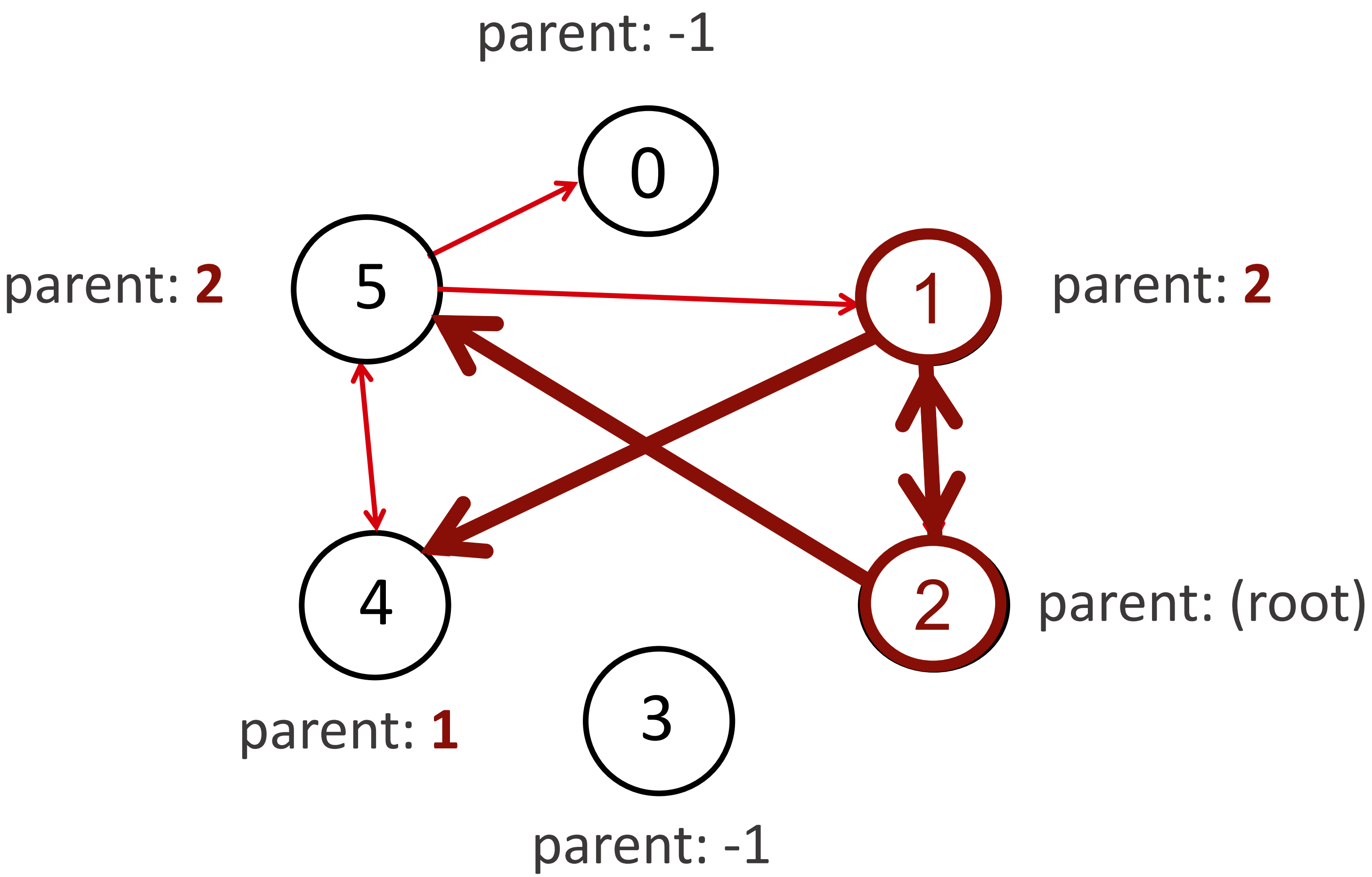
Construct a spanning tree

-1 means parent unknown

In this case we start from vertex 2

Requires a frontier: all vertices that received a parent in the previous round

BREADTH-FIRST SEARCH



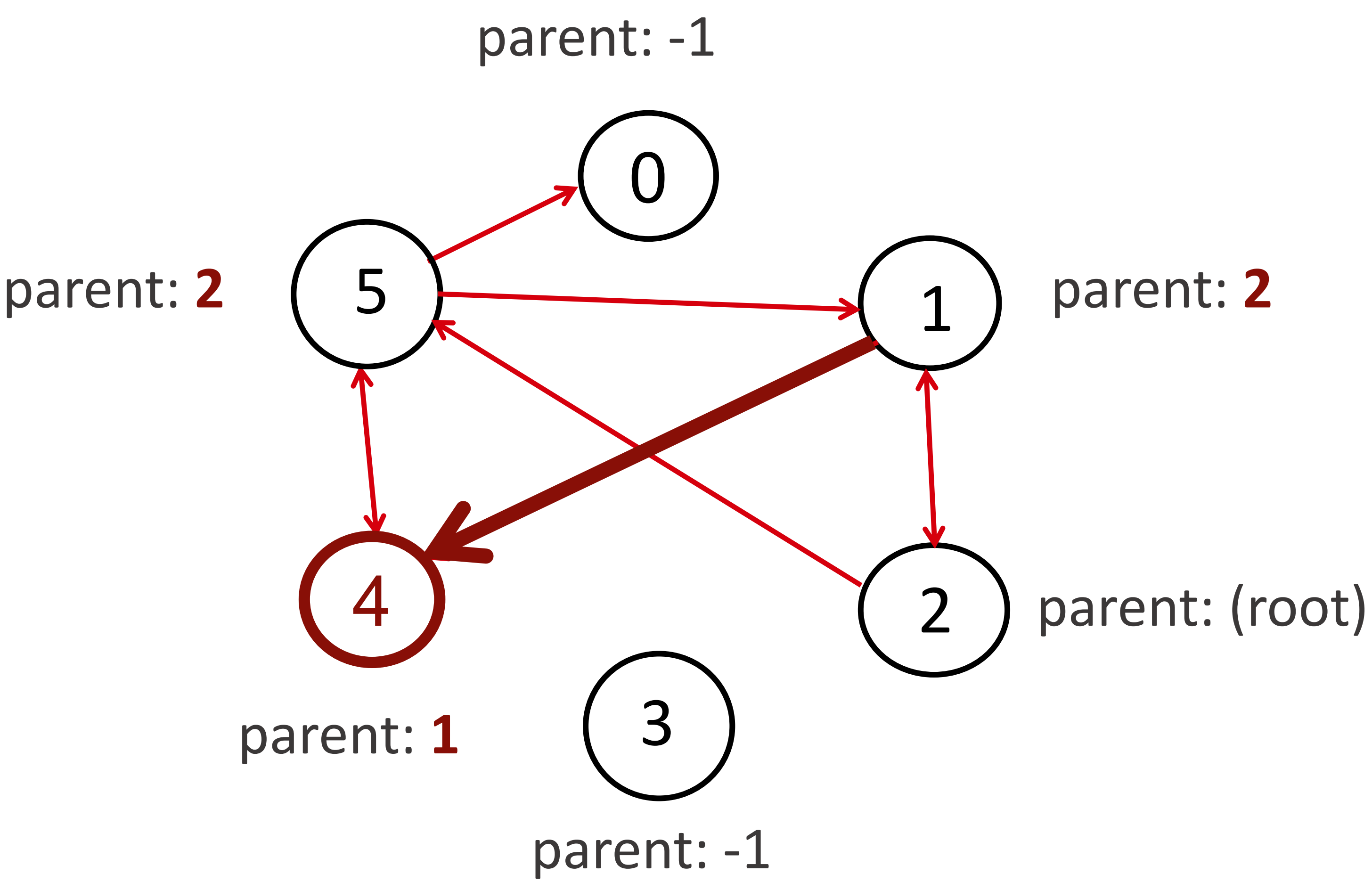
”Top-down” traversal, “push”

Focus on out-edges

Old frontier: **1 5**

New frontier: **4 5**

BREADTH-FIRST SEARCH



"Bottom-up" traversal, "pull"

Focus on in-edges

Vertex complete as soon as parent updated

Old frontier: **1 5**

New frontier:

IMPACT OF CONVERGENCE

[Beamer, SC12]

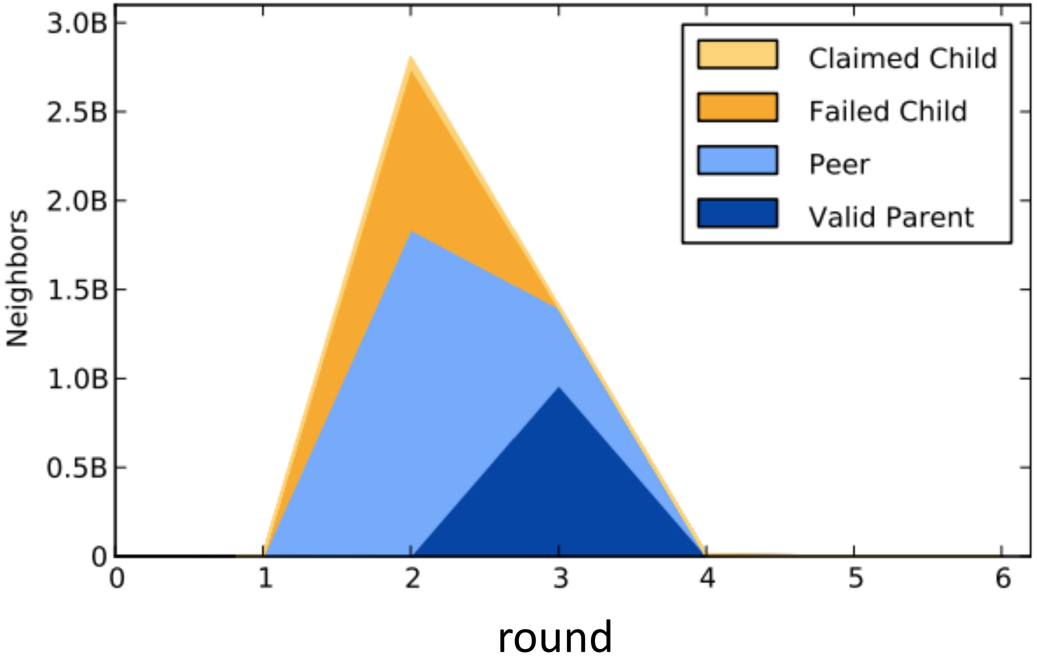


Fig. 3. Breakdown of edges in the frontier for a sample search on kron27 (Kronecker generated 128M vertices with 2B undirected edges) on the 16-core system.

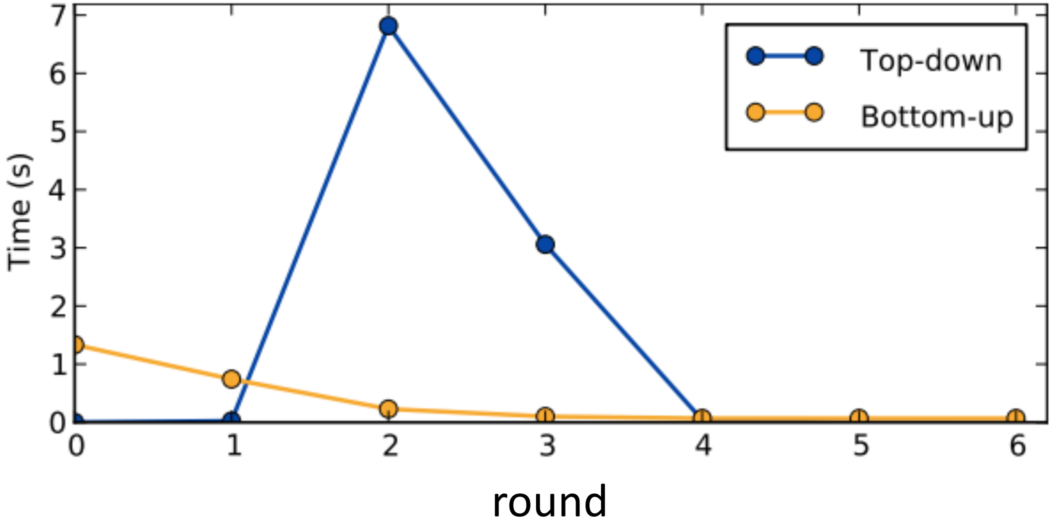


Fig. 6. Sample search on kron27 (Kronecker 128M vertices with 2B undirected edges) on the 16-core system.

Bottom-up/pull is faster in the middle rounds

In those rounds, many vertices are active

Claimed child: parent[v] updated from -1 to a vertex ID

Failed child: parent[v] updated in same round by parent

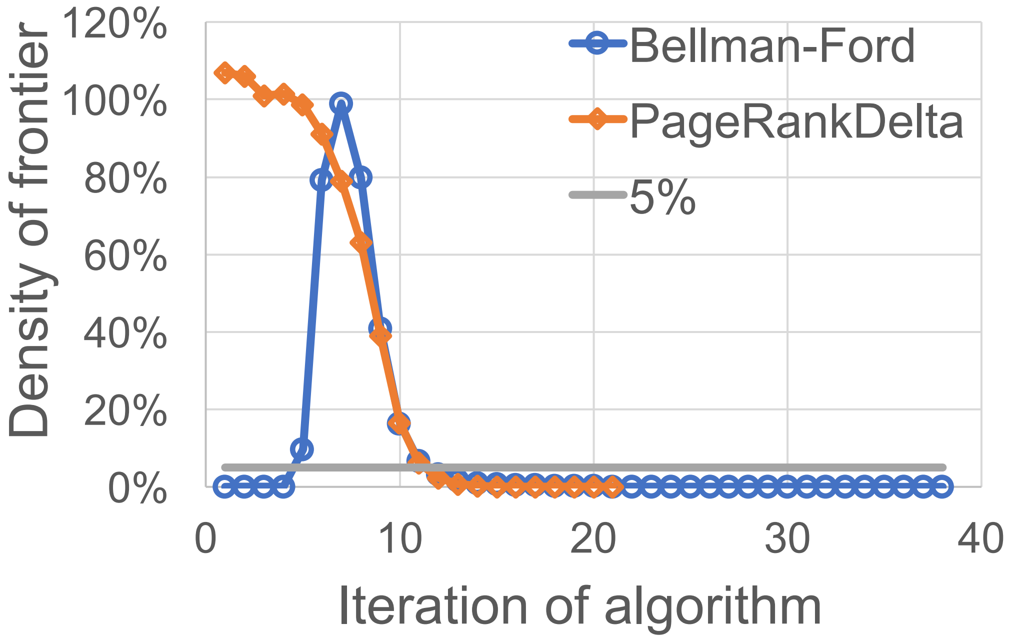
Peer: parent[v] updated in same round by sibling

Valid parent: v was encountered in a round prior to u

DIRECTION-OPTIMISATION

[Beamer SC'12] [Shun SPAA'13]

```
d = (#active vertices +
#active edges) / #edges

if d > 5% then
   # dense frontier
   if algorithm prefers
      forward then
      traverse CSR
   else
      traverse CSC
   endif
else # d <= 5%
   # sparse frontier
   traverse CSR
endif
```



Programmer's choice
Little is known to guide this

We will shed some light on this
… work in progress

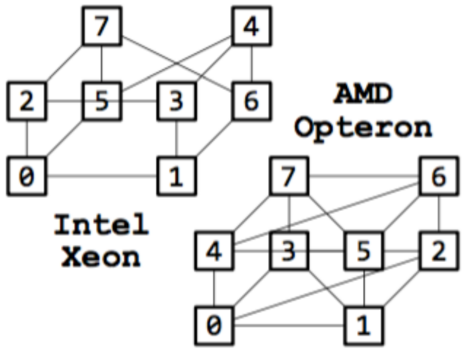
Requires storage of both
CSC and CSR

QUEEN'S UNIVERSITY BELFAST

NUMA-AWARENESS

QUEEN'S UNIVERSITY BELFAST

POLYMER



(a) System topology

Inst.	0-hop	1-hop	2-hop
80-core Intel Xeon machine			
Load	117	271	372
Store	108	304	409
64-core AMD Opteron machine			
Load	228	419	498
Store	256	463	544

(b) Latencies (cycles) on the distance

Figure 3. The characteristics of NUMA machines for experiments.

Access	0-hop	1-hop	2-hop	Interleaved
80-core Intel Xeon machine				
Sequential	3207	2455	2101	2333
Random	720	348	307	344
64-core AMD Opteron machine				
Sequential	3241	2806/2406	1997	2509
Random	533	509/487	415	466

Figure 4. The bandwidth (MB/s) of memory access on the distance.

[Zhang PPoPP'15]

Remote access has higher latency, lower bandwidth than local access

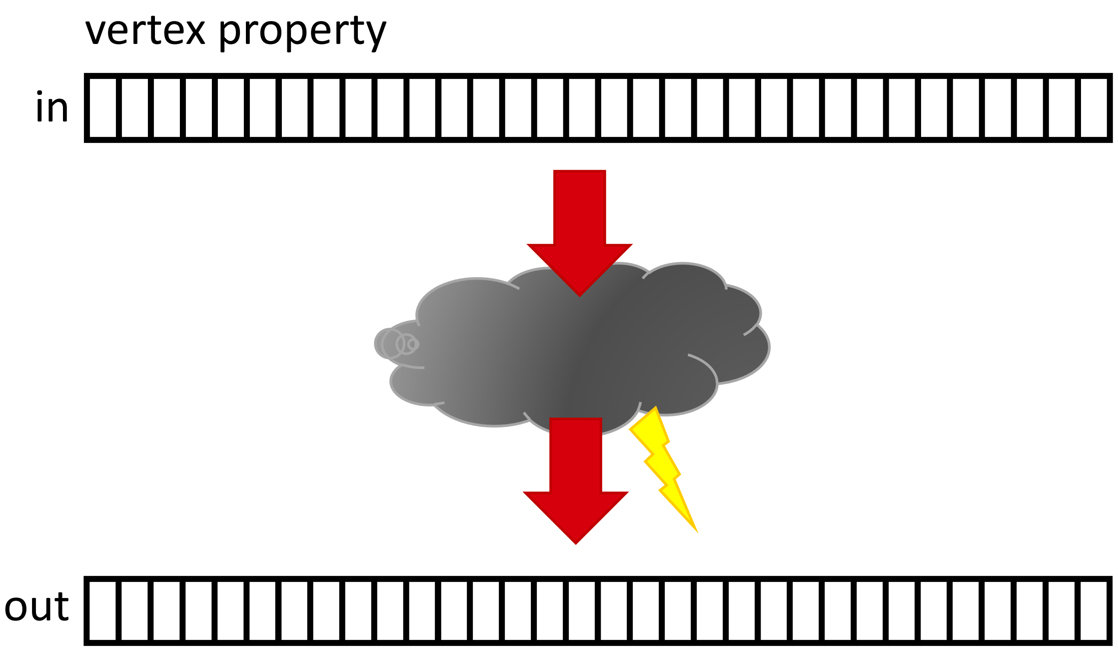
Stores are more affected than loads

Designed a scheme using graph partitioning [Kyrola OSDI'12] and privatization of vertex properties

We will discuss how their ideas were rehashed in GraphGrind

QUEEN'S UNIVERSITY BELFAST

EDGEMAP, VERTEXMAP AND NUMA-AWARENESS



Goal: map code and data to NUMA nodes

One type of arrays

- Properties (per vertex)

Two types of loops

- Loops over edges
- Loops over vertices

Two types of iteration

- Sparse frontier
- Dense frontier

QUEEN'S UNIVERSITY BELFAST

RECAP: RACE CONDITIONS

A pair of load and store instructions, at least one of which is a store, that access the same memory location

In a concurrent program with race conditions, the outcome of the program may differ depending on the relative execution speed of threads

Typical solutions:

- mutual exclusion
- atomic memory operations
- owner-computes

OWNER-COMPUTES

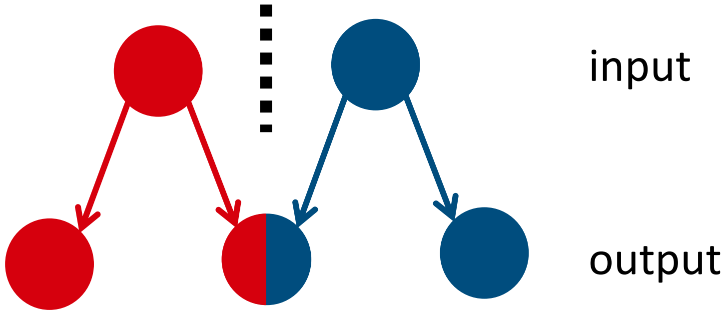
Decomposition based on partitioning input/output data is referred to as the owner computes rule

Each partition performs all the computations involving data that it owns

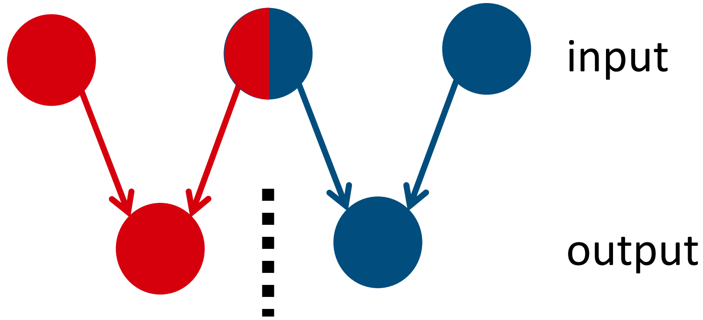
- **Input data decomposition:** A task performs all the computations that can be done using these input data
- **Output data decomposition:** A task computes all the results in the partition assigned to it

Input partitioning

Red and blue processors

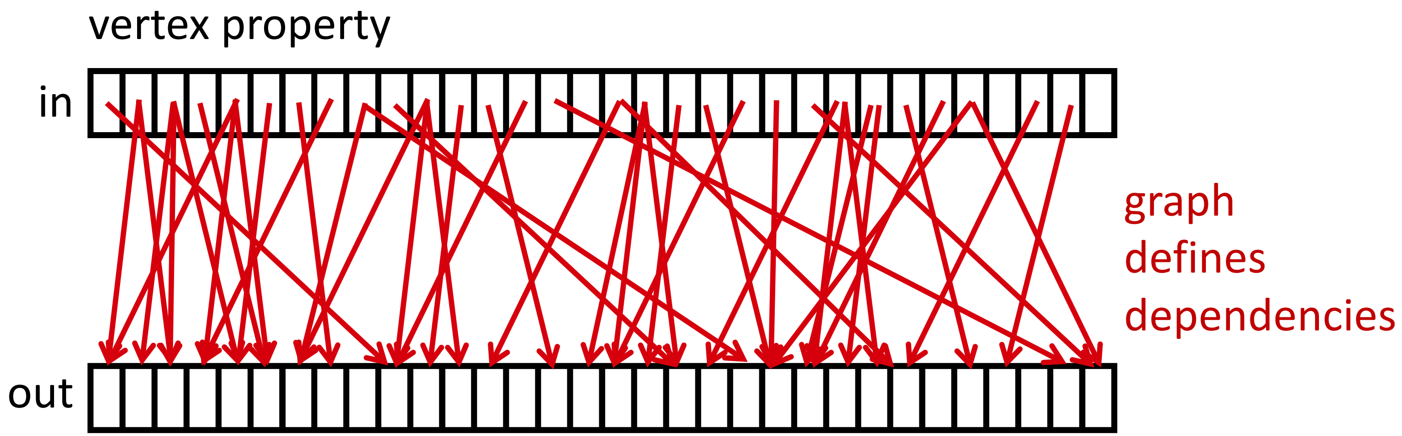


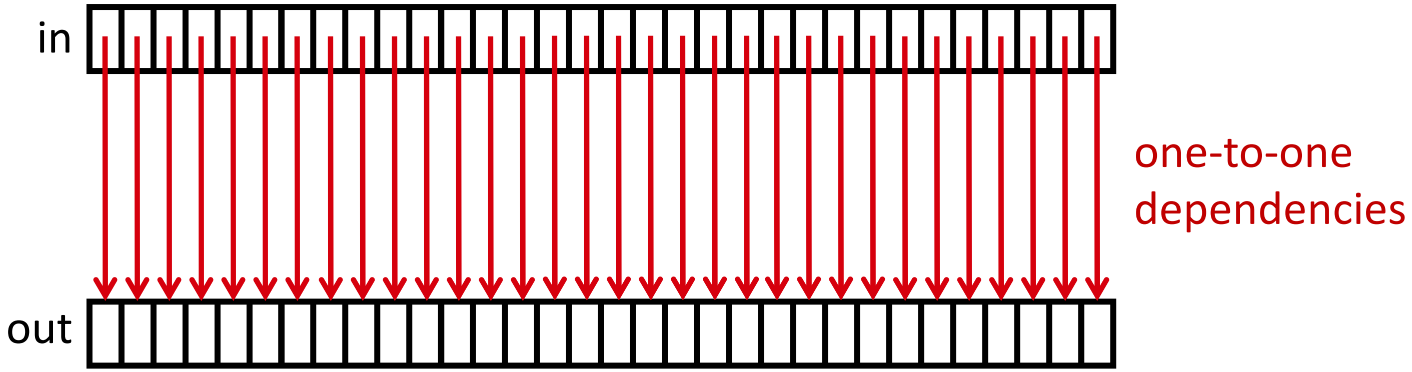
Output partitioning



QUEEN'S UNIVERSITY BELFAST

THE "MAP" IN EDGEMAP AND VERTEXMAP





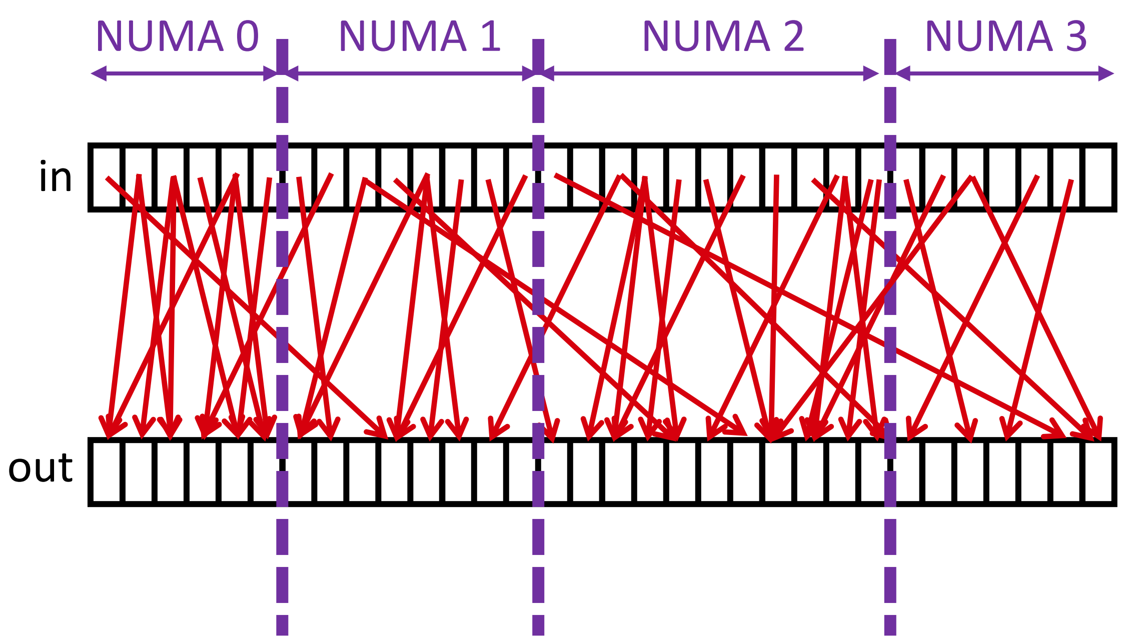
Edgemap:

- Iteration space: (u,v) $\in$ E where u, v $\in$ V
- Dependencies are determined by graph topology

Vertexmap:

- Iteration space: v $\in$ V

QUEEN'S UNIVERSITY BELFAST



NUMA-AWARE LAYOUT FOR EDGEMAP

How to split edgemap over NUMA nodes?

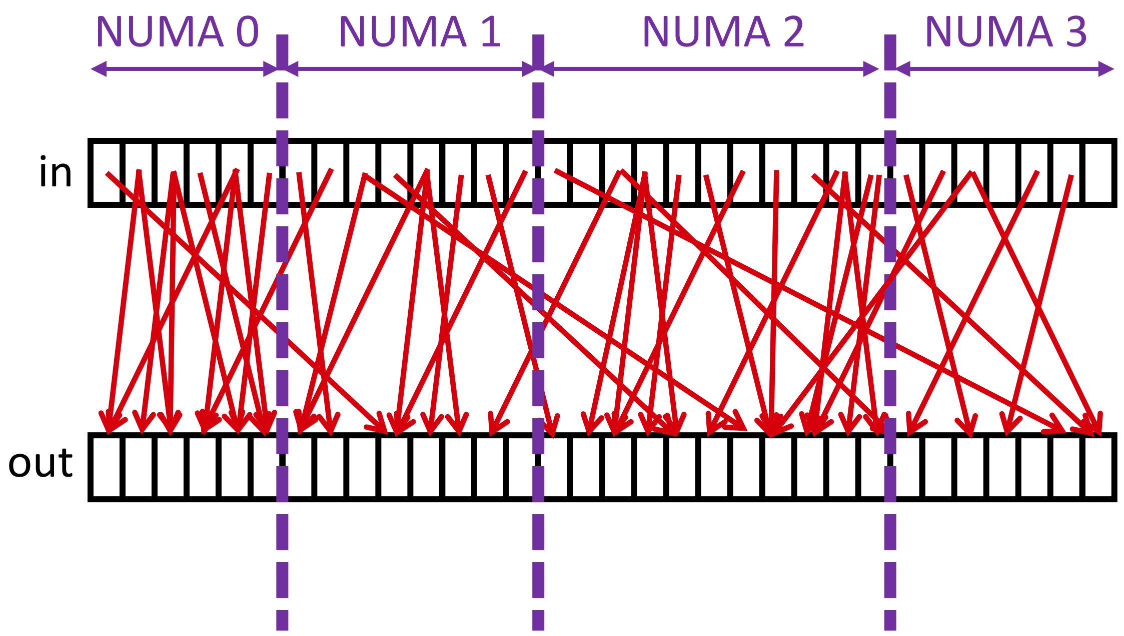
- code
- data

Observation

Remote stores are more expensive than remote loads [Zhang PPoPP ‘15]

Need to co-locate code with the updated data

Edges are processed by CPUs attached to the NUMA node that holds the destination’s property



NUMA-AWARE LAYOUT FOR EDGEMAP

Goal

Determine cuts of { code, data } such that performance is maximised

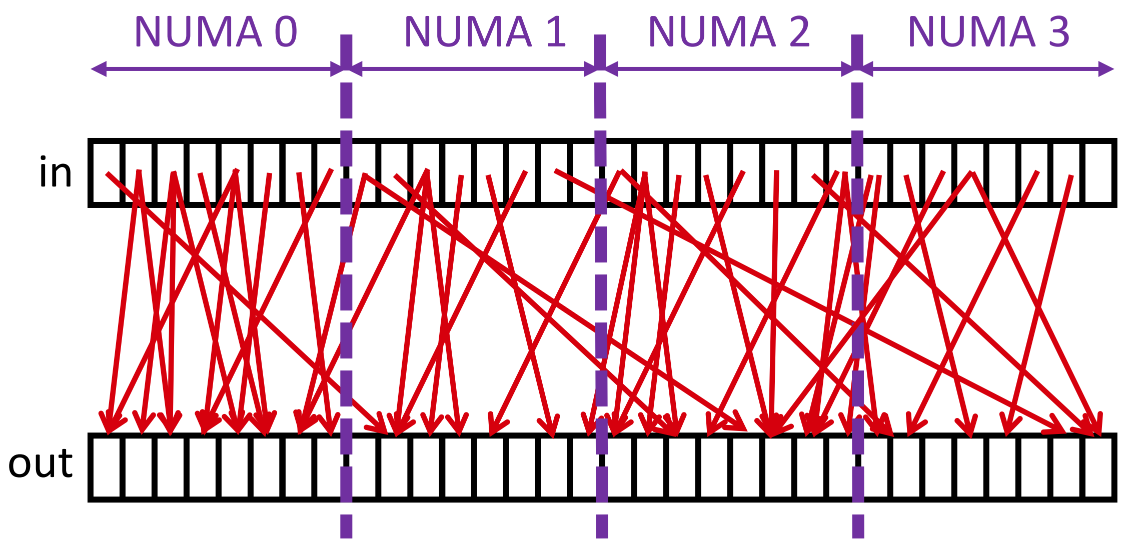
How?

Partition graph such that each partition (NUMA node) has an equal:

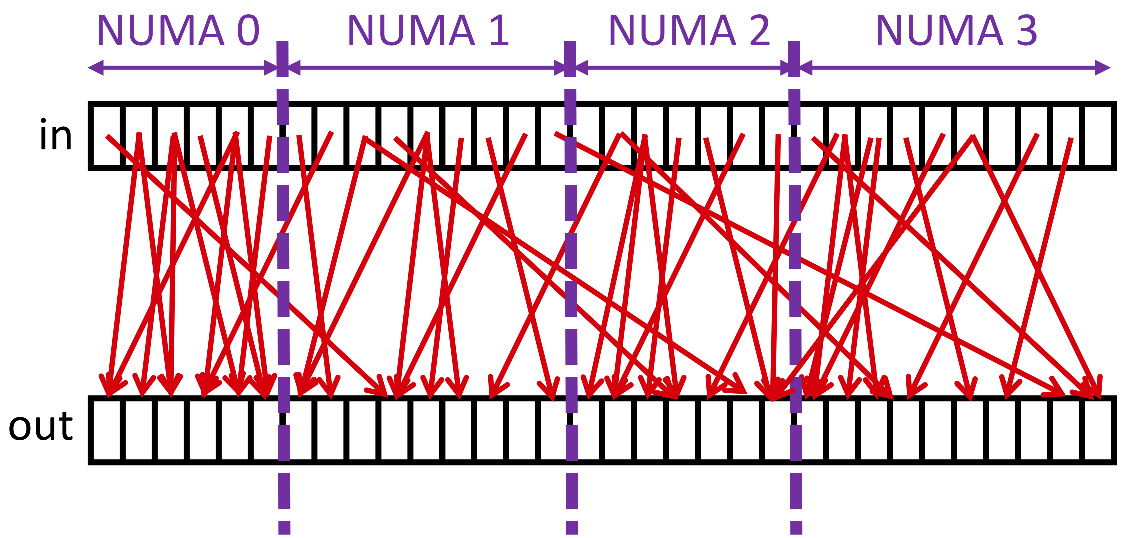
1. #edges, #cuts [PowerGraph OSDI'12] … breaks locality
2. #sources [X-stream SOSP'13] … race conditions
3. #edges [Polymer PPoPP '15]
4. (α #destinations + #edges) [Gemini OSDI'16]

NUMA-AWARE LAYOUT FOR EDGEMAP

Vertex-oriented algorithms



Edge-oriented algorithms



It depends! [GraphGrind ICS'17]

"Vertex-oriented" algorithms

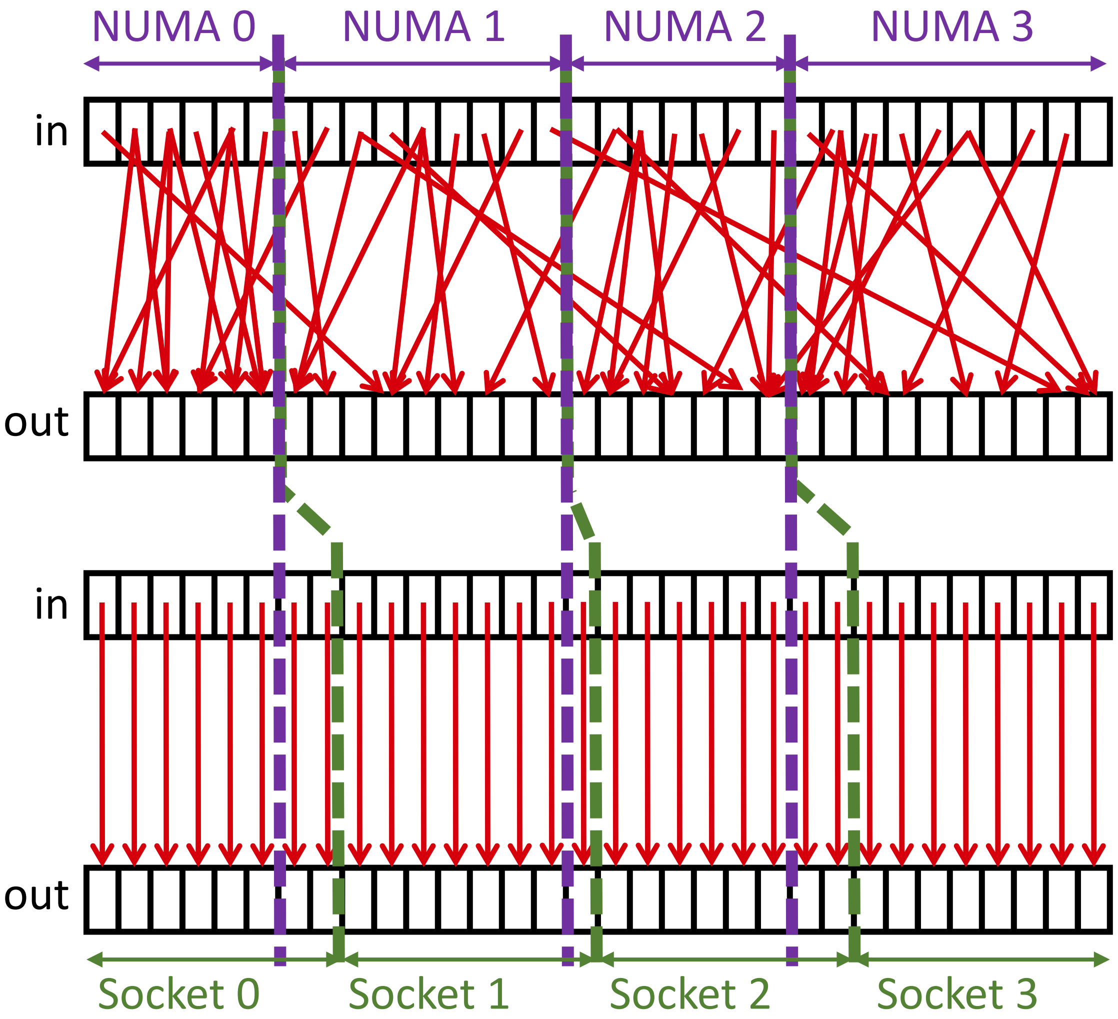
- Best performance with equal #destinations
- Frontier density mostly below 50%
- BFS, Betweenness Centrality, Bellman-Ford

"Edge-oriented" algorithms

- Best performance with equal #edges
- Frontier density mostly close to 100%
- PageRank, SpMV, Belief Prop., PageRankDelta

NUMA-AWARE LAYOUT FOR VERTEXMAP

Edge-oriented algorithms



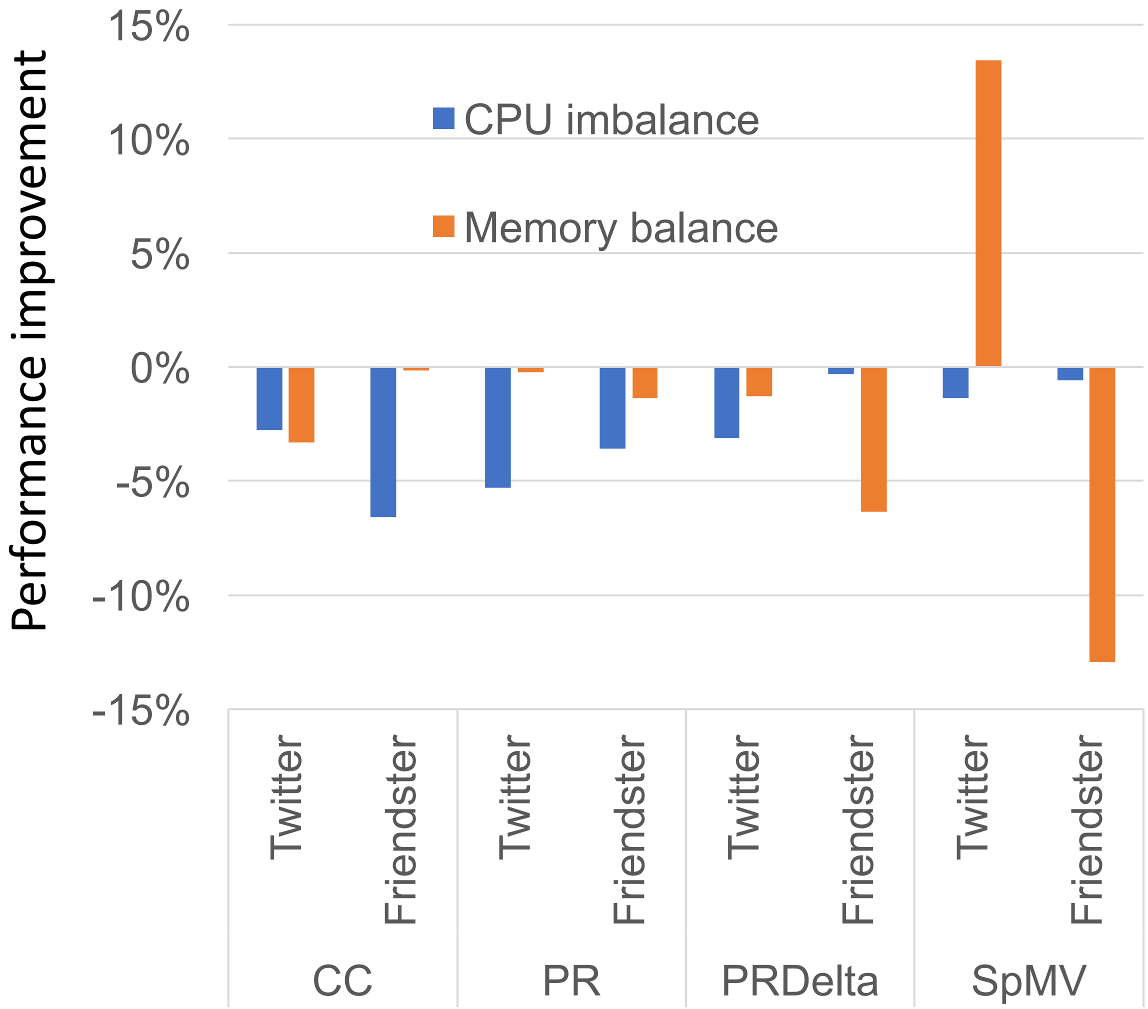
"Vertex-oriented" algorithms

- Trivial

"Edge-oriented" algorithms

- Need to choose between balancing compute and minimising traffic across NUMA nodes
- Better to balance compute and incur additional inter-node traffic [GraphGrind ICS'17]
- Consequently, data is partitioned differently from compute

NUMA-AWARENESS CHOICES



- Baseline is CPU balance and memory imbalance
  - Implies remote accesses during vertex map
- CPU imbalance
  - No remote accesses during vertex map
- Memory balance
  - No remote accesses during vertex map
  - Many remote accesses during edge map

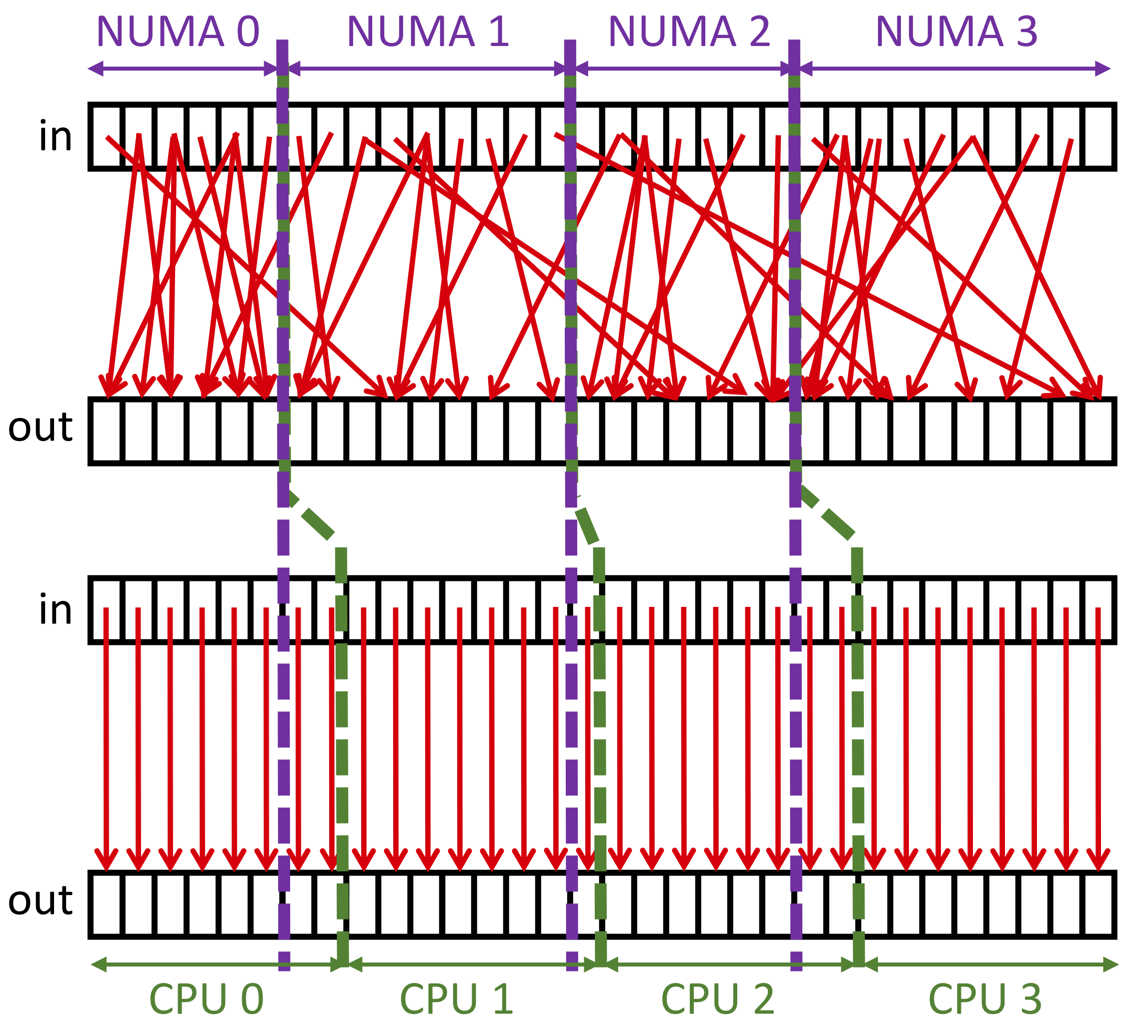
QUEEN'S UNIVERSITY BELFAST

CAN WE MEET BOTH REQUIREMENTS?

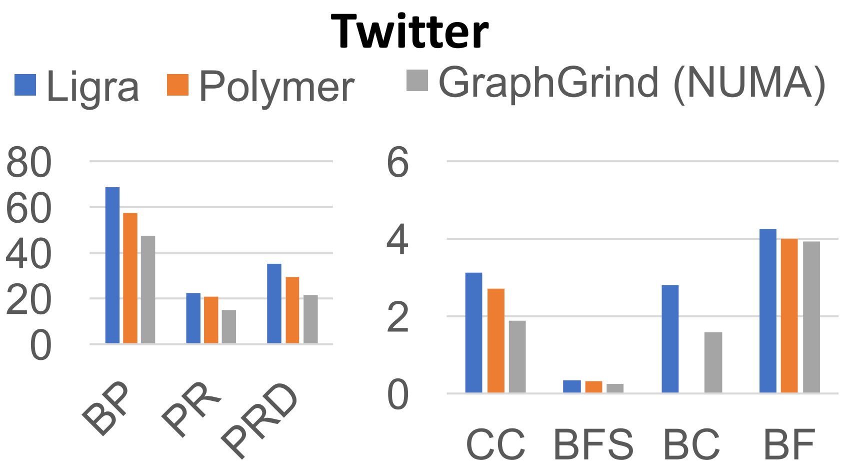
Have our cake and eat it too!

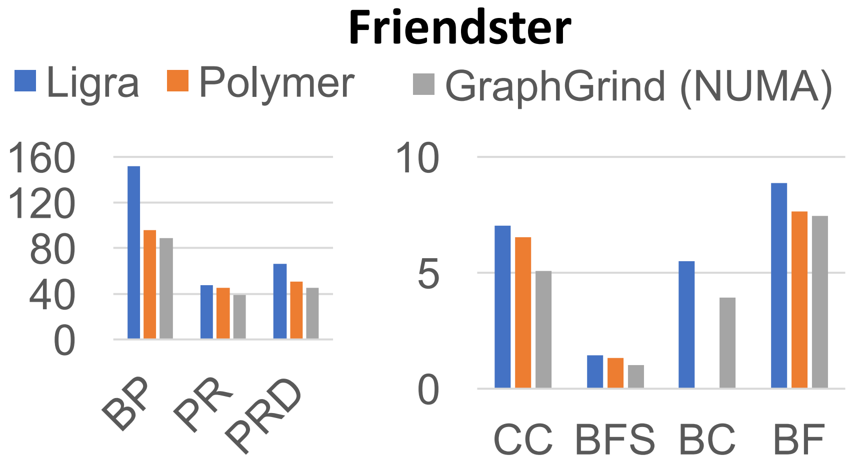
VEBO

Edge-oriented algorithms



PERFORMANCE EVALUATION OF NUMA-AWARENESS





Combination of optimisations [Sun ICS'17]

- Pruned CSC/CSR representation
- Tune partitioning to edge/vertex algorithms
- NUMA-aware layout of vertex arrays
- CSC traversal: "caching" intermediate values to minimise load/stores
- Full frontier: specialised version of code that omits frontier check
- Sparse CSR traversal: no partitioning applied

4-socket 2.6GHz Intel Xeon E7-4860 v2, 48 threads, 256 GiB

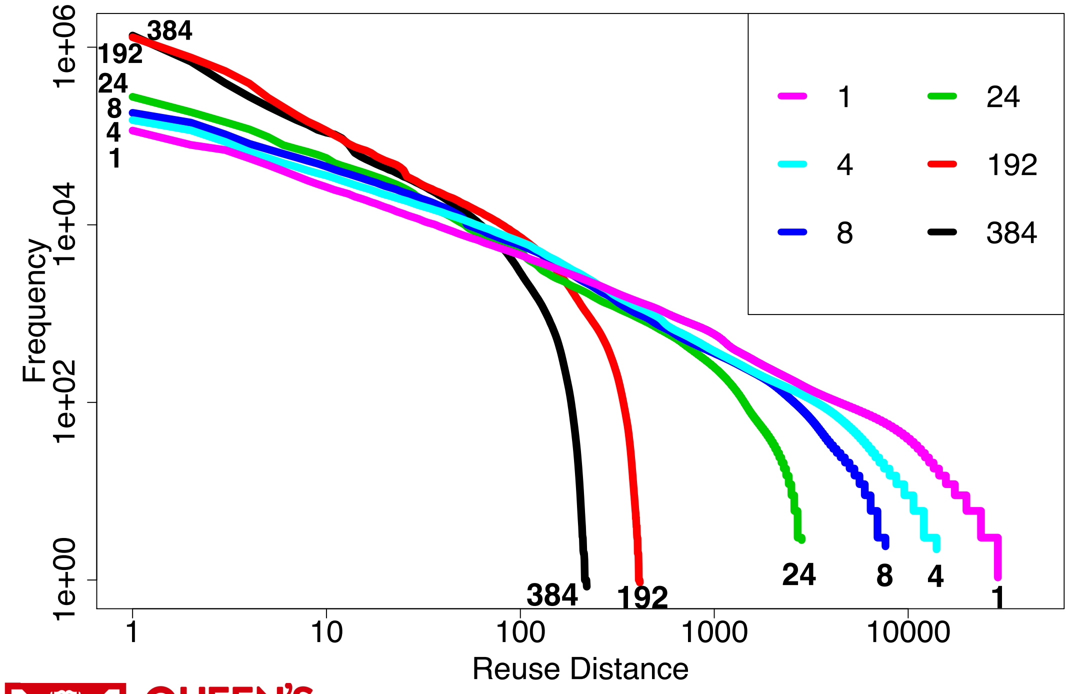
QUEEN'S UNIVERSITY BELFAST

GRAPH PARTITIONING

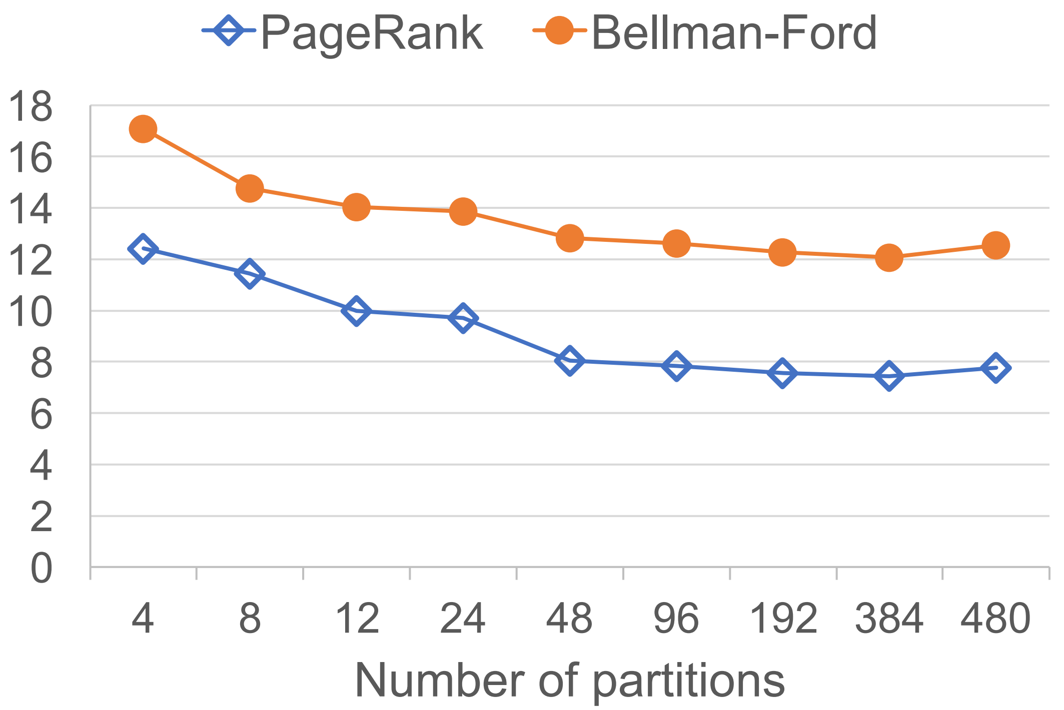
QUEEN'S UNIVERSITY BELFAST

MEMORY LOCALITY

- Reuse Distance Distribution

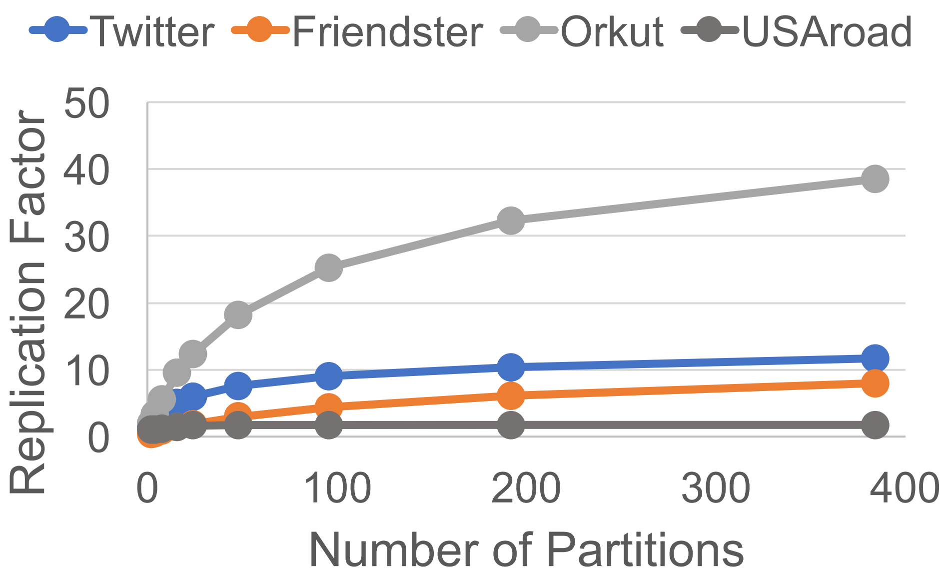


- Misses Per Kilo-Instruction (MPKI) – Twitter graph



QUEEN'S UNIVERSITY BELFAST

VERTEX REPLICATION



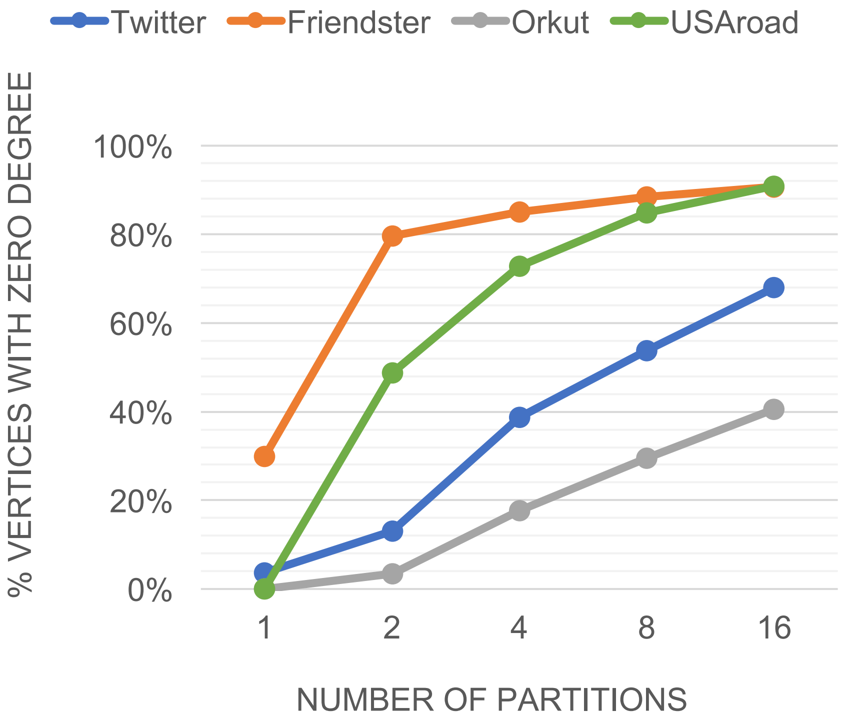
When partitioning the edge set, a vertex may appear in multiple partitions

Replication factor =
#repeated vertices / #unique vertices

Replication factor tends to |E|/|V| as number of partitions grows

Replication implies space and runtime overhead

IMPLICATIONS OF VERTEX REPLICATION



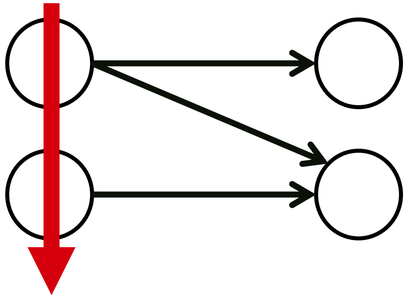
A different view on the same effect:

If we partition the edge set P-way, then a vertex with degree d<P has zero edges in at least P-d partitions. It has some edges in at most d partitions.

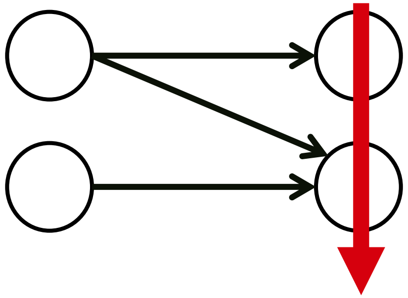
Partitions of a sparse graph are *hyper-sparse*

GRAPH DATA STRUCTURES

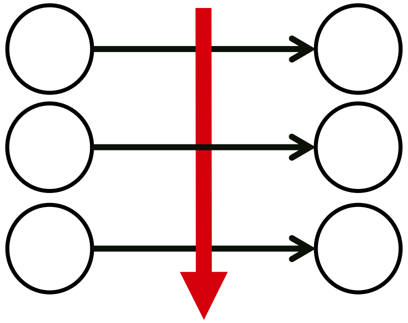
- **Compressed Sparse Rows (CSR)**
  - List outgoing edges for each vertex
  - “Forward” traversal (push)
  - “vertex-centric”



- **Compressed Sparse Columns (CSC)**
  - List incoming edges for each vertex
  - “Backward” traversal (pull)
  - “vertex-centric”

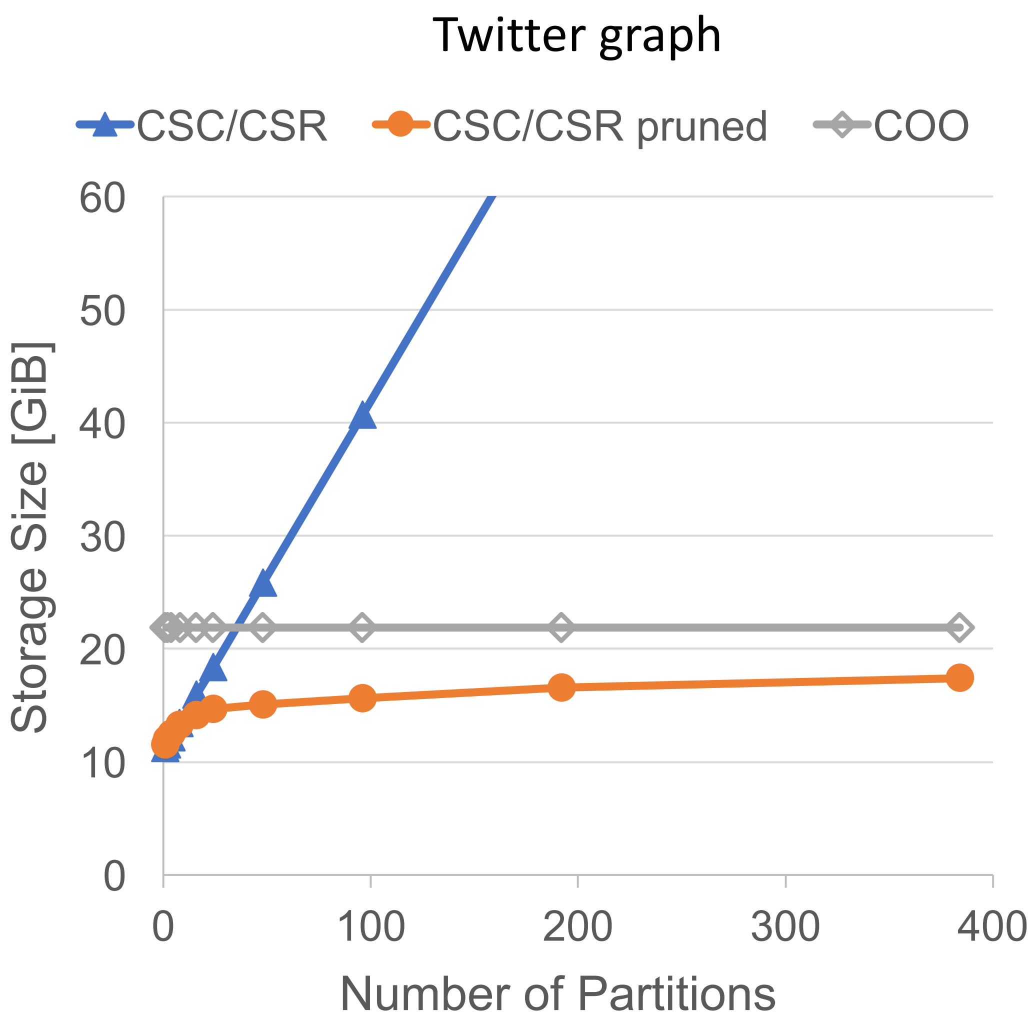


- **Coordinate list (COO)**
  - A list of edges
  - “edge-centric”



No performance boost from frontiers or pruning for COO

QUEEN’S UNIVERSITY BELFAST



IMPLICATIONS OF VERTEX REPLICATION

CSC and CSR are not scalable formats

- Increased storage
- Increased execution time to traverse graph

Pruned CSC/CSR

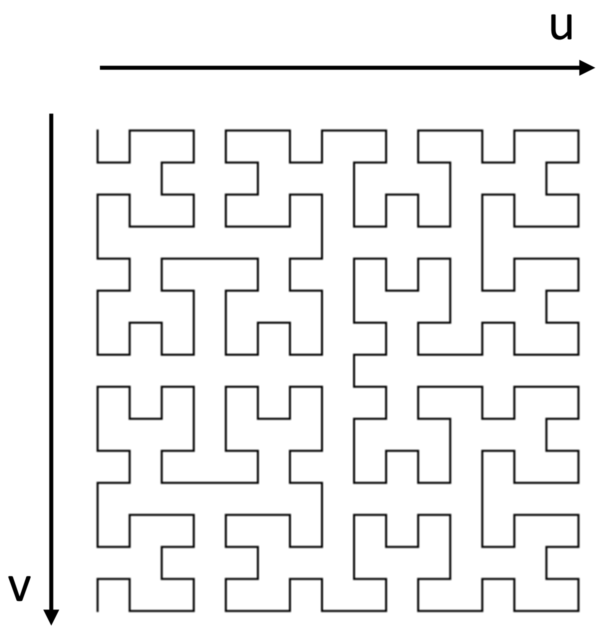
- A.k.a. Compressed Compressed Sparse Rows/Columns
- Omits zero-degree vertices

COO is scalable to any number of partitions

- But inefficient for sparse frontiers

QUEEN'S UNIVERSITY BELFAST

COO ADVANTAGE: SPACE FILLING CURVES



Edges are points in a 2D space:

For u,v in 0,…,|V|-1:

$(u,v) = 1$ if $(u,v) \in E$

$(u,v) = 0$ otherwise

Edgemap: visit all (u,v) in E

Space filling curves define a traversal order through a space that tends to minimise memory locality

Map *n*D order onto 1D order

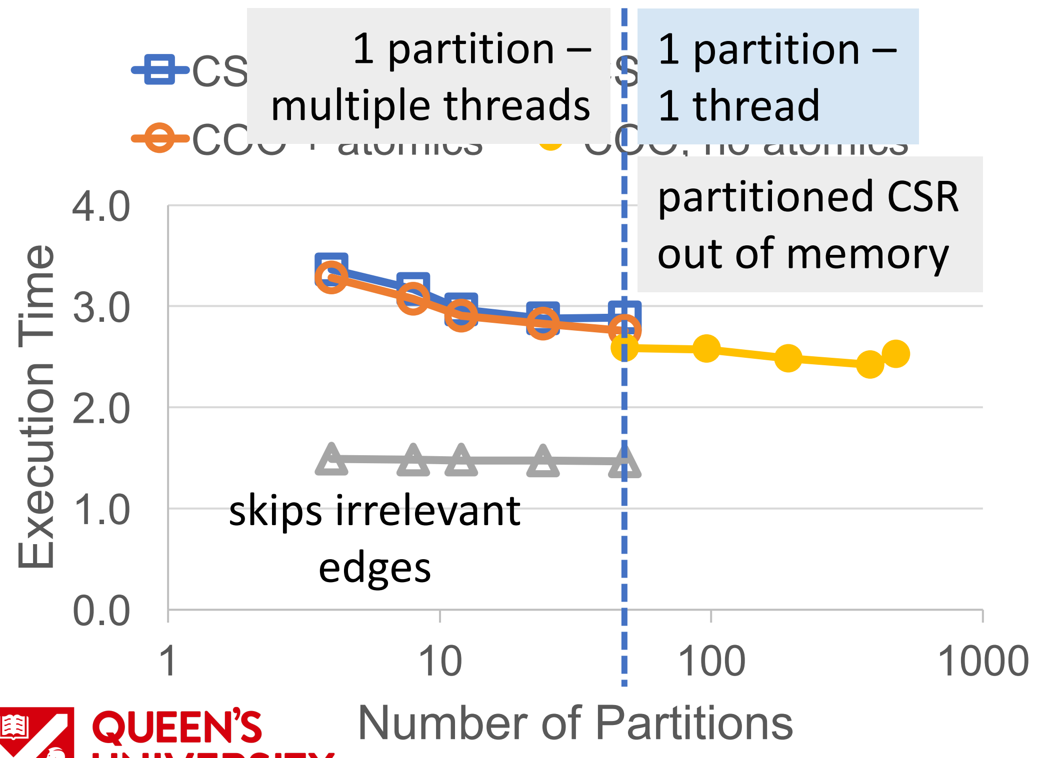
Hilbert curve, Morton order (Z-order), and many others

COO allows edges to be stored in any order:

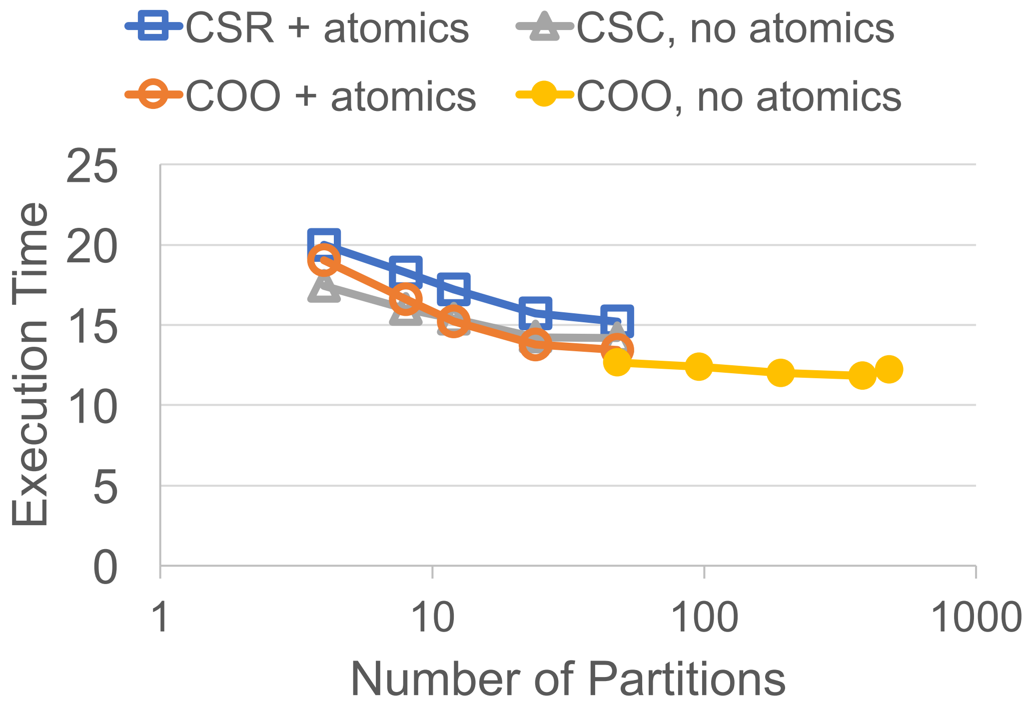
- CSR order
- CSC order
- Space filling curves

GRAPH PARTITIONING BENEFITS

- Betweenness Centrality, Twitter



- PageRank, Twitter
10 iterations



QUEEN'S UNIVERSITY BELFAST

DIRECTION-OPTIMIZATION

- Ligra [Shun PPoPP'13]

```
d = (#active vertices +
#active edges) / #edges

if d > 5% then
   # dense frontier
   if algorithm prefers
      forward then
      traverse CSR
   else
      traverse CSC
   endif
else # d <= 5%
   # sparse frontier
   traverse CSR
endif
```

- GraphGrind [Sun ICPP'17]
- 3-way heuristic

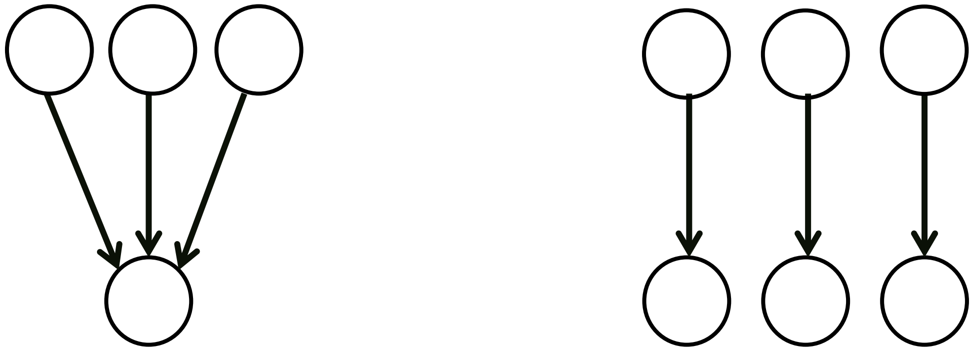
```
d = (#active vertices +
#active edges) / #edges

if d > 50% then
   # dense frontier
   traverse partitioned COO
else if d > 5% then
   # medium-dense case
   # dense frontier
   traverse CSC
else # d <= 5%
   # sparse frontier
   traverse CSR
endif
```

QUEEN'S UNIVERSITY BELFAST

LOAD BALANCE

LOAD BALANCE

Revisting edge balance:
Two partitions with 3 edges
Which partition is processed faster?

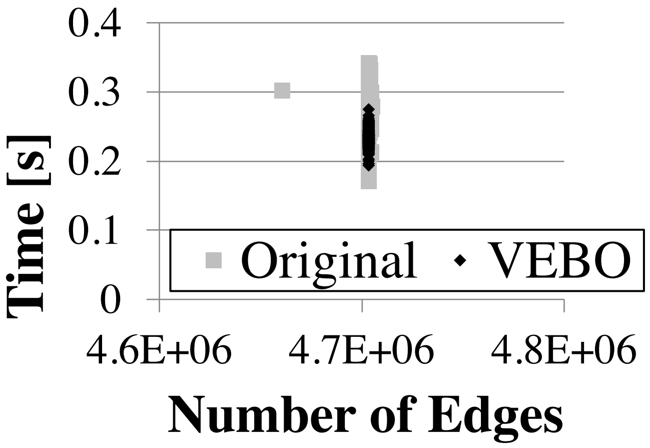


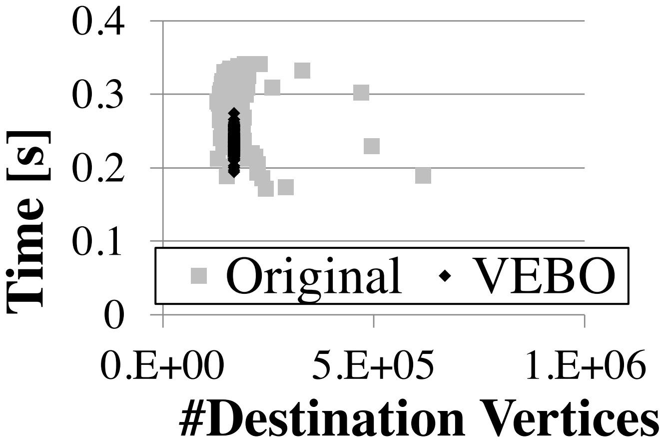
Execution time/partition highly dependent on the degree of vertices

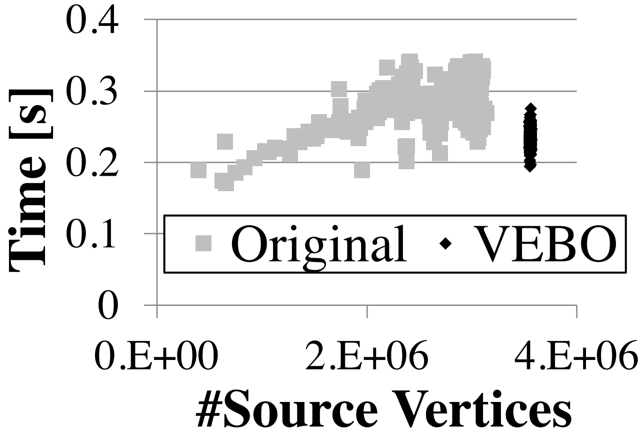
Reorder vertices

- in order of decreasing in-degree
- using list scheduling

VEBO: Vertex and Edge Balanced Partitioning

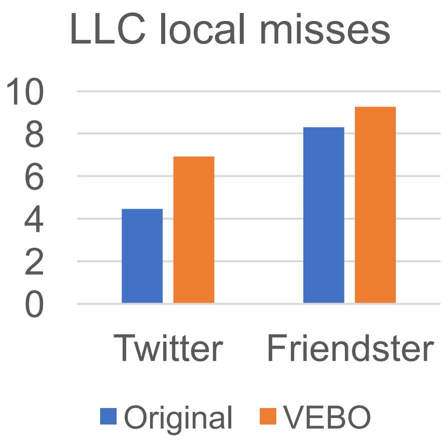


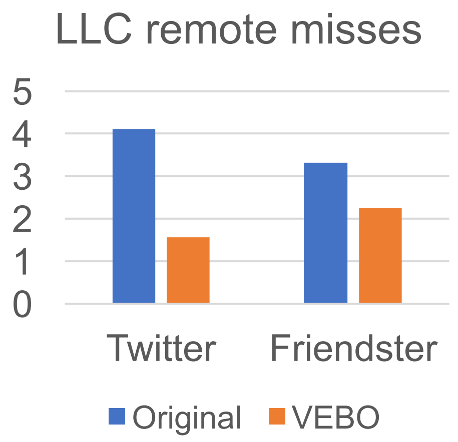


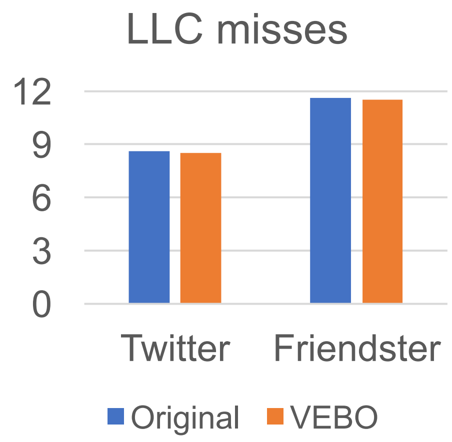


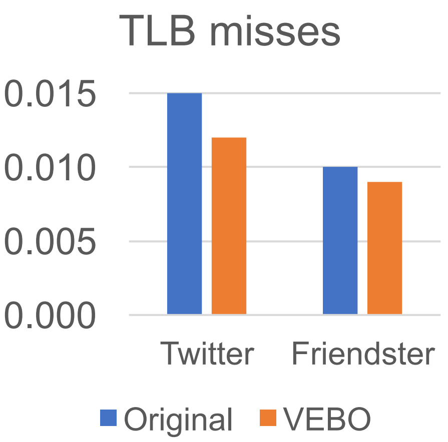
Friendster
PageRank

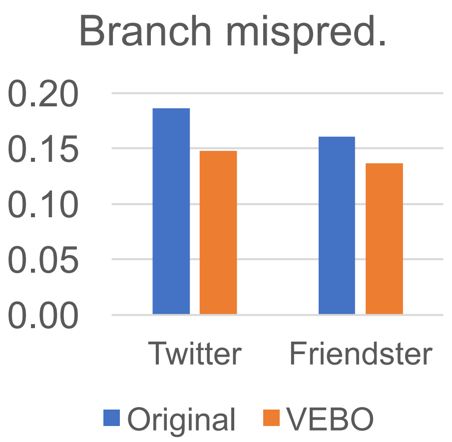
VEBO BENEFITS











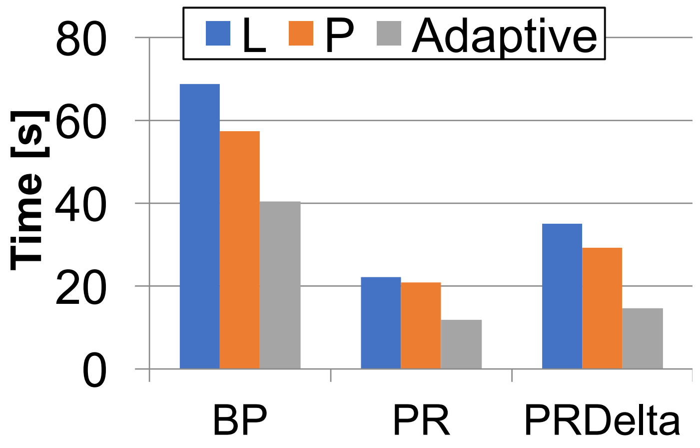
Partitions are processed faster as a side-effect of reordering

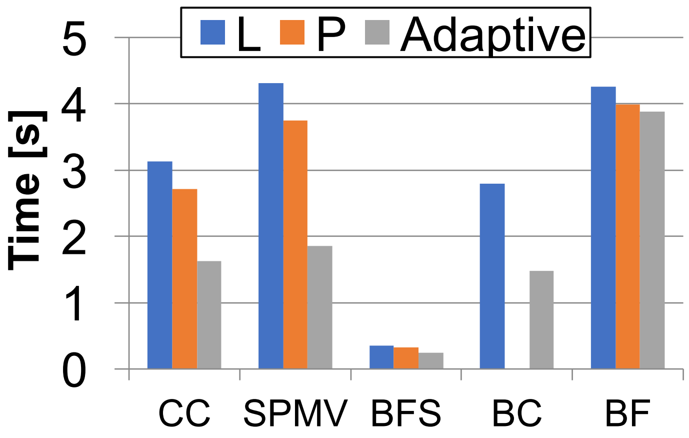
Remote cache misses are traded for local misses

PageRank

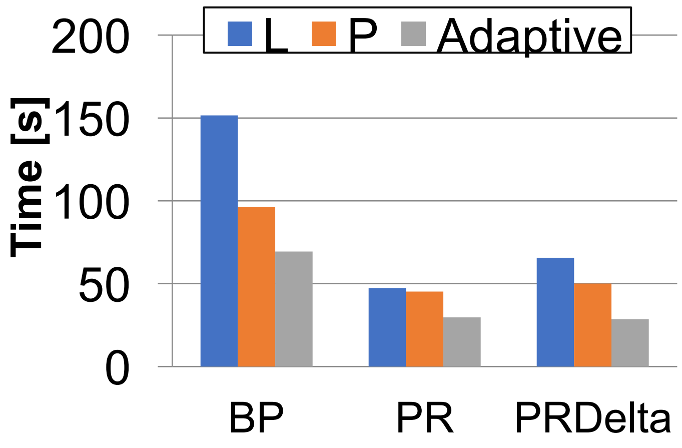
PERFORMANCE

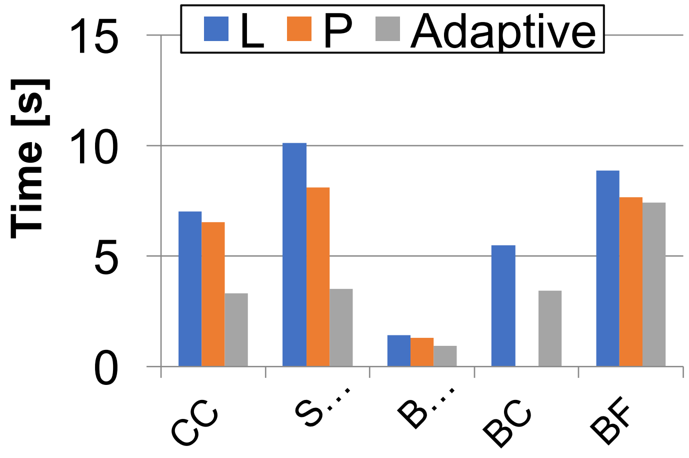
Twitter





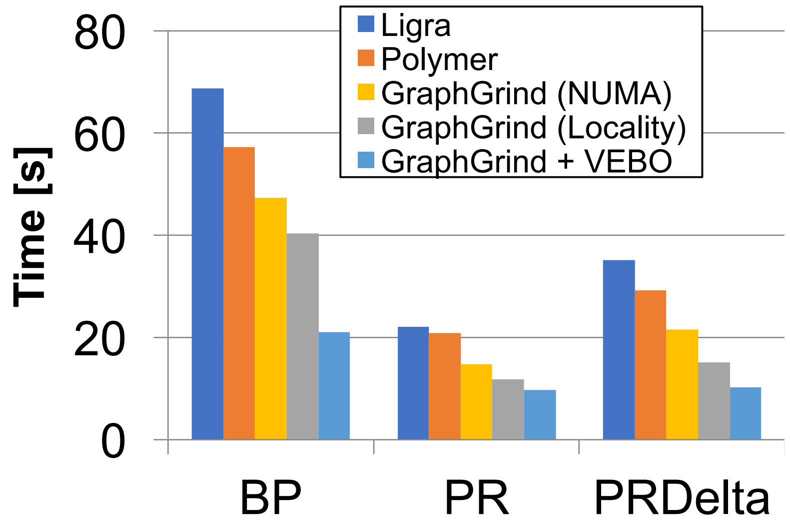
Friendster

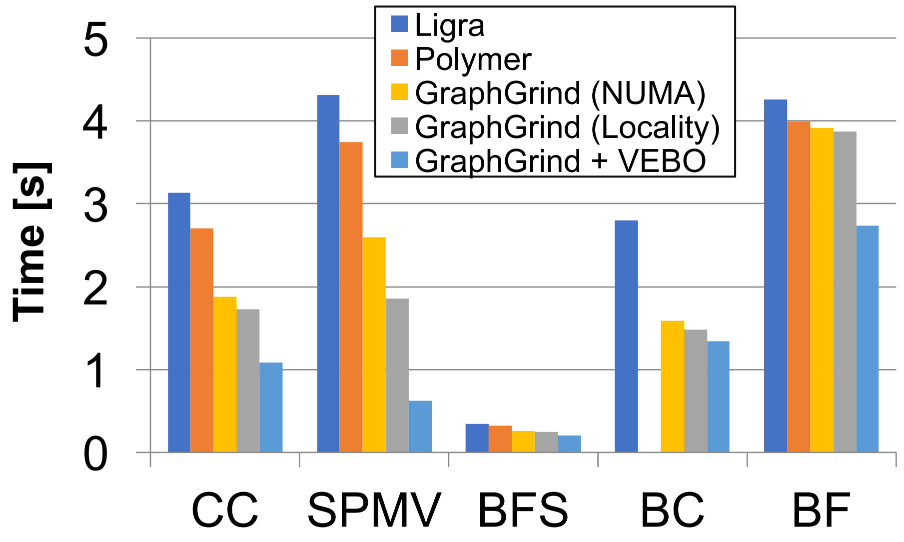




L: Ligra, P: Polymer, Adaptive: GraphGrind with 3-way ”direction-optimization”

QUEEN'S UNIVERSITY BELFAST





PERFORMANCE

Comparing Ligra, Polymer (NUMA-aware), and 3 versions of GraphGrind

Twitter graph

4-socket 2.6GHz Intel Xeon E7-4860 v2, 48 threads, 256 GiB

Similar results hold for other graphs

VEBO relabels vertex IDs to achieve load balance

QUEEN'S UNIVERSITY BELFAST

CONCLUSION AND OUTLOOK

QUEEN'S UNIVERSITY BELFAST

CONCLUSION AND OUTLOOK

Scale-free properties of graphs make it hard to achieve high-performance

Code itself is short – devil is in the detail

Graph partitioning crucial: NUMA-locality; avoiding atomics; improving memory locality

Some open questions:

- What are the limits on memory efficiency?
- What is the cause of performance difference between CSR/CSC/COO?
- Do the principles behind GraphGrind apply to distributed memory systems?
- How well does the programming model capture graph algorithms?

REFERENCES

- Beamer, S., Asanovic, K. and Patterson, D. Direction-Optimizing Breadth-First Search. SC'12
- Besta, M., Podstawski, M., Groner, L., Solomonik, E. and Hoefler, T. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations, IPDPS'17
- Gonzalez, J. E., Low, Y., Gu, H., Bickson, D. and Guestrin, C. PowerGraph: distributed graph-parallel computation on natural graphs, OSDI'12
- Hong, S., Tayo, O. and Olukotun, K. Efficient parallel graph exploration on multi-core CPU and GPU. PACT'11
- Kang, U., Tsourakakis, C. E., and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations ICDM'09
- Kyrola, A., Blelloch, G. E. and Guestrin, C. GraphChi: Large-Scale Graph Computation on Just a PC. OSDI'12
- Roy, A. and Mihailovic, I. and Zwaenepoel, W. X-stream: Edge-centric graph processing using streaming partitions, SOSP'13
- Sharma, A., Jiang, J., Bommannavar, P., Larson, B. and Lin, J. GraphJet: Real-Time Content Recommendations at Twitter. PVLDB'16
- Shun, J. and Blelloch, G. E. Ligra: A Lightweight Graph Processing Framework for Shared Memory. PPoPP'13
- Sun, J., Vandierendonck, H. and Nikolopoulos, D. S. GraphGrind: addressing load imbalance of graph partitioning. ICS'17
- Sun, J., Vandierendonck, H. and Nikolopoulos, D. S. Accelerating Graph Analytics by Utilising the Memory Locality of Graph Partitioning. ICPP'17
- Sun, J., Vandierendonck, H. and Nikolopoulos, D. S. VEBO: A Vertex-and Edge-Balanced Ordering Heuristic to Load Balance Parallel Graph Processing. arXiv:1806.06576. 2018
- Zhang, K., Chen, R. and Chen, H. NUMA-Aware Graph-structured Analysis. PPoPP'15
- Zhu, X. and Chen, W. and Zheng, W. and Ma, X. Gemini: A Computation-Centric Distributed Graph Processing System. OSDI'16

QUEEN'S
UNIVERSITY
BELFAST

ESTD 1845