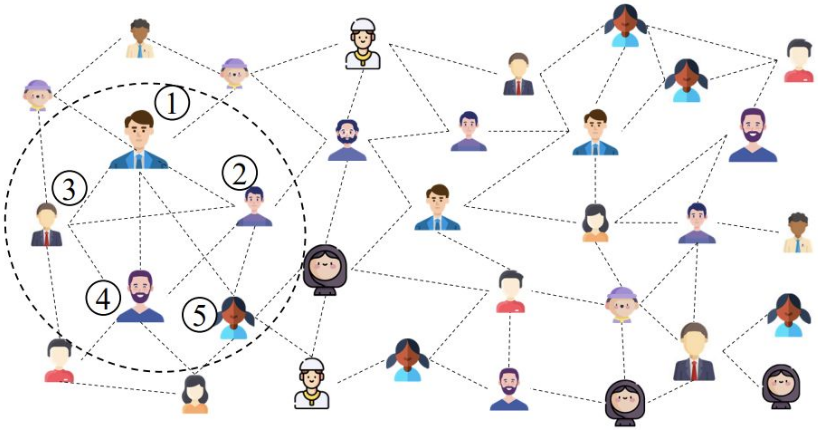
CompressGraph: Efficient Parallel Graph Analytics with Rule-Based Compression

Zheng Chen et. al
Review by Anika Cheerla

Introduction

Real world graphs are gigantic and redundant!



How can we use redundancy to have smaller compressed graphs and faster graph analytics?

The CompressGraph Approach

1) Compressing graphs through rule-based abstraction saves time by removing the need for decompressing.
2) CompressGraph is general and supports a wide-range of graph applications.
3) CompressGraph scales well under high-parallelism.

Existing Graph Compressions

Adjacency matrices and lists

Graph encoding:

1) Variable-length encoding
2) Reference encoding
3) Interval encoding
4) Gap encoding

On the fly decompression difficult to parallelize

Text Analytics Directly on Compression (TADOC)

Rule-based compression that uses context-free grammar rules to represent text data

rule **R** represents a subsequence that happens multiple times in the data

Text Analytics Directly on Compression (TADOC)

rule **R** represents a subsequence that happens multiple times in the data

Input:

123124123124121

Rules:

R2 → 1 2

Text Analytics Directly on Compression (TADOC)

rule **R** represents a subsequence that happens multiple times in the data

Input:

123124123124121

Rules:

R1 → R2 3 R2 4
R2 → 1 2

Text Analytics Directly on Compression (TADOC)

rule **R** represents a subsequence that happens multiple times in the data

Input:

123124123124121

Rules:

$R0 \rightarrow R1 \quad R1 \quad R2 \quad 1$

$R1 \rightarrow R2 \quad 3 \quad R2 \quad 4$

$R2 \rightarrow 1 \quad 2$

Text Analytics Directly on Compression (TADOC)

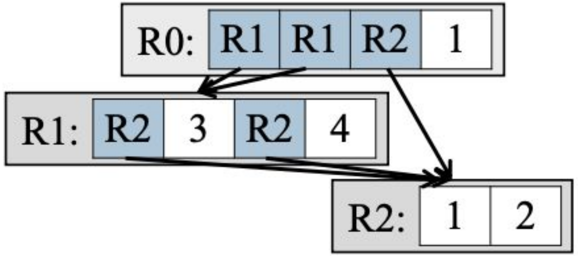
Representation

Input:

123124123124121

Rules:

R0 → R1 R1 R2 1
R1 → R2 3 R2 4
R2 → 1 2



Word frequencies computation

R2

<1,1> <2,1>

R1

<1,2> <2,2> <3,1> <4,1>

Step 2
1: $1\times2 = 2$ 3: 1
2: $1\times2 = 2$ 4: 1

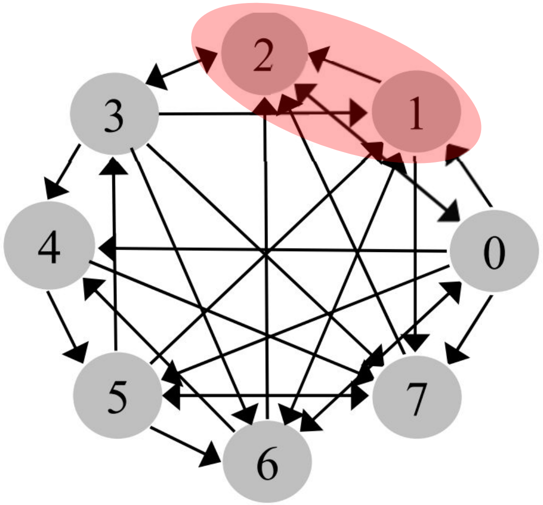
R0

<1,6> <2,5> <3,2> <4,2>

Step 3
1: $2\times2 + 1 + 1 = 6$
2: $2\times2 + 1 = 5$
3: $1\times2 = 2$
4: $1\times2 = 2$

CompressGraph takes inspiration from TADOC

rule defined as a repeated set of neighbors

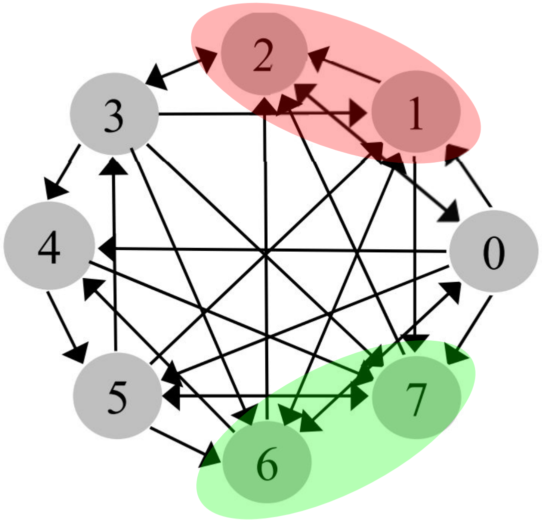


Vertex	Neighbors
0	R1, 5, R2
1	2, R2
2	0, 3
3	R1, R2
4	5, 7
5	1, 3, R2
6	0, R1
7	2, 5

Rule	Content
R1	R3, 4
R2	6, 7
R3	1, 2

CompressGraph takes inspiration from TADOC

rule defined as set of neighbors

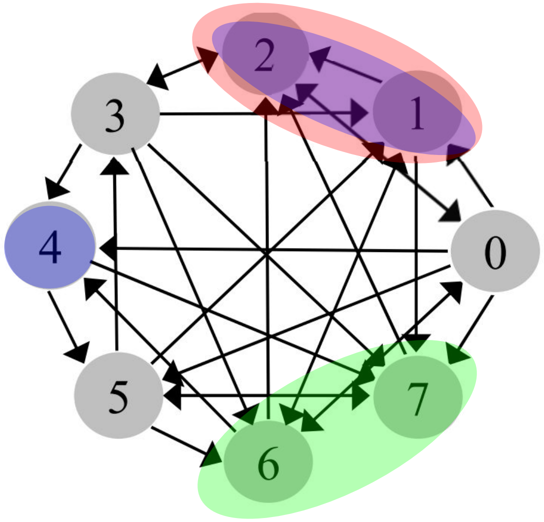


Vertex	Neighbors
0	R1, 5, R2
1	2, R2
2	0, 3
3	R1, R2
4	5, 7
5	1, 3, R2
6	0, R1
7	2, 5

Rule	Content
R1	R3, 4
R2	6, 7
R3	1, 2

CompressGraph takes inspiration from TADOC

rule defined as set of neighbors



Vertex	Neighbors
0	R1, 5, R2
1	2, R2
2	0, 3
3	R1, R2
4	5, 7
5	1, 3, R2
6	0, R1
7	2, 5

Rule	Content
R1	R3, 4
R2	6, 7
R3	1, 2

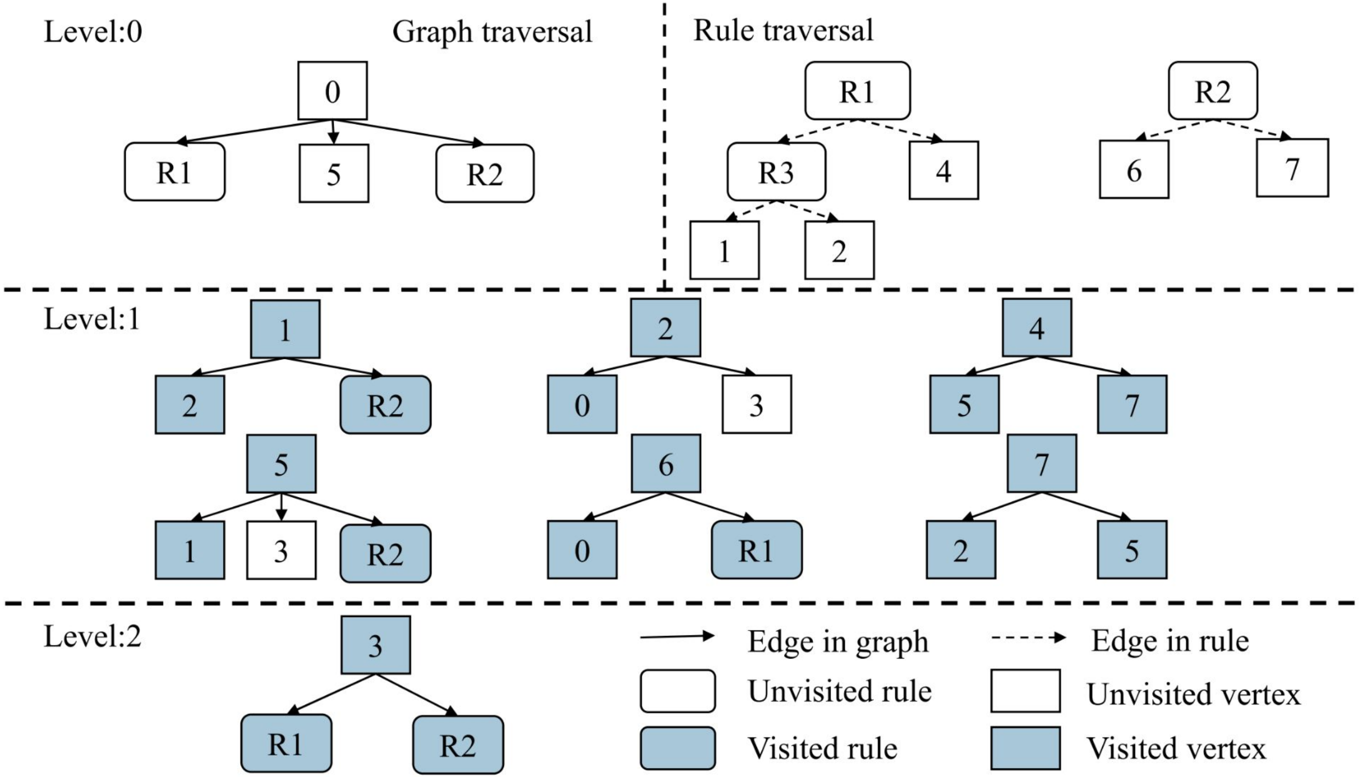
BFS on CompressGraph

1) Put initial vertex in queue

2) *Graph traversal*: Visit unvisited neighbors and put in queue

3) *Rule traversal*: Expand newly encountered rules into vertices and put in queue

4) Pop first element of queue, then 2)



BFS Program

vertex to vertex and vertex to rule increment distance by 1

rule to vertex or rule doesn’t change distance

INIT: visited

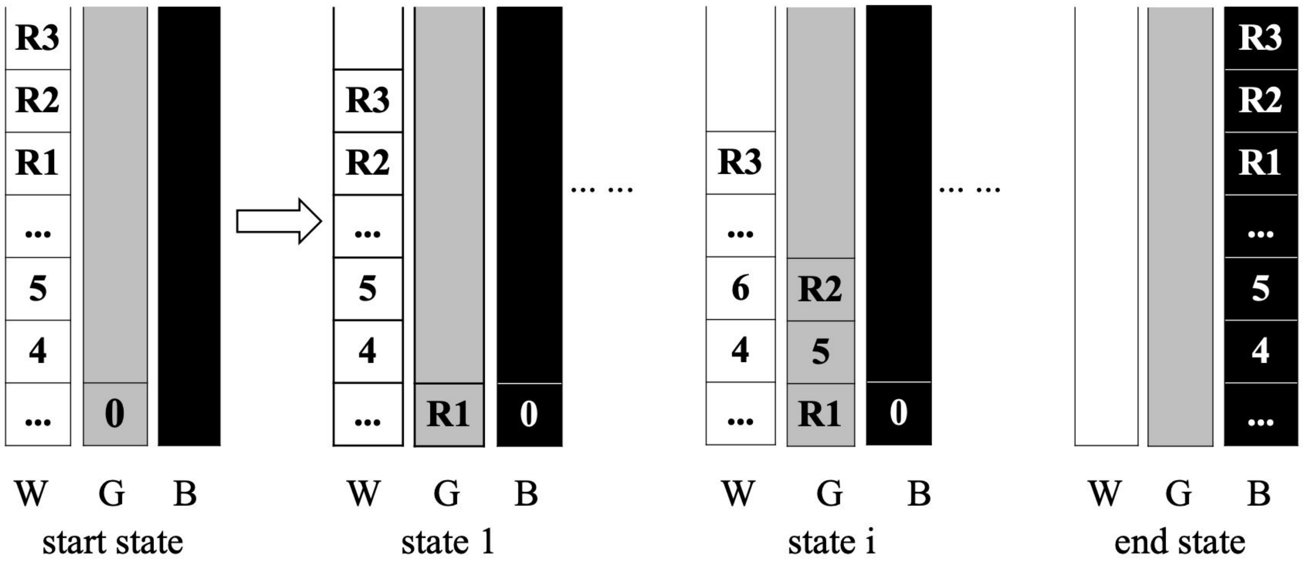
```
CompressGraph = {Graph; Operation; Condition; Result, State_start, State_end};
Graph = {V, R, E};
class Operation{
    void Ov2v(vertex src, vertex dst){
        if(dst.distance == INIT) { dst.distance = src.distance+1; }
    }
    void Ov2r(vertex src, rule dst){
        if(dst.distance == INIT) { dst.distance = src.distance+1; }
    }
    void Or2v(rule src, vertex dst){
        if(dst.distance == INIT) { dst.distance = src.distance; }
    }
    void Or2r(rule src, rule dst){
        if(dst.distance == INIT) { dst.distance = src.distance; }
    }
};
class Condition{
    bool Cv(vertex V) { return V.distance == INIT; }
    bool Cr(rule R) { return R.distance == INIT; }
};
class Result{
    int distance;
    Result(Graph G){
        distance = INIT;
    }
};
State_start = {V&R-{root}, {root}, null};
State_end = {U1, null, U2};
State_cur = State_start;
```

Finite State Machine

FSM w/ states defined by W (unprocessed), G (processing), B(done)

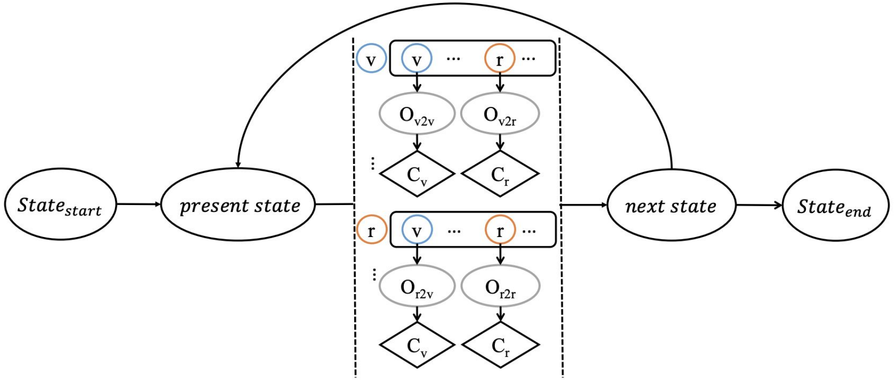
In each state transition:

take out *v* or *r* in G, traverse neighbors and add to G, put element into B



CompressGraph can handle any vertex and its neighbors

Given graph vertex *v* with neighbor set {*u*1, *u*2, ..., *un*}:



If Edge <*v*, *ui*> Exists in Compressed Graph:

- Process <*v*, *ui*> with operation *Ov*2*v*
- Process *ui* with operation *Cv*
- Determine whether to add *ui* to *State*.*G*

If Edge <*v*, *ui*> Does Not Exist in Compressed Graph:

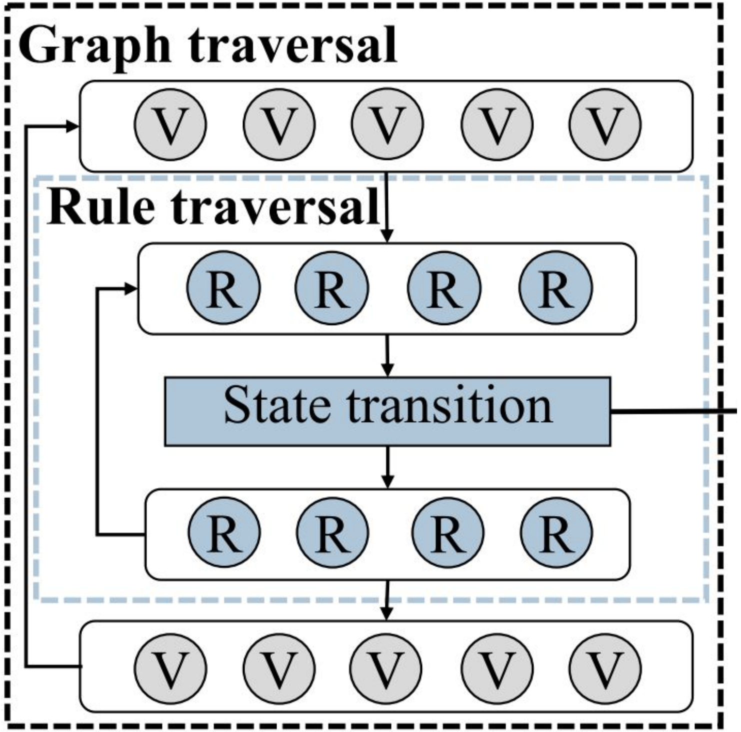
- Path <*v*, *r*1, ..., *rm*, *ui*> exists in compressed graph using rules only
- Perform rule traversal to process <*v*, *ui*> in original graph
- Use *Ov*2*r* to process <*v*, *r*1>, *Or*2*r* to process <*ri*, *ri*+1>, and *Or*2*v* to process <*rm*, *ui*>
- Use *Cr* to determine whether to add {*r*1, *r*2, ..., *rm*} to *State*.*G*

Rule Traversal: Use the *Result* field of rules to store intermediate results

Key takeaways

1) The repeated sequence of neighboring vertices is represented by a rule that takes less space.
2) Rule-based compression reduces redundant computations by caching and reusing results for rules.
3) Rule-based compression allows processing directly on the compressed graph, avoiding expensive decoding operations.

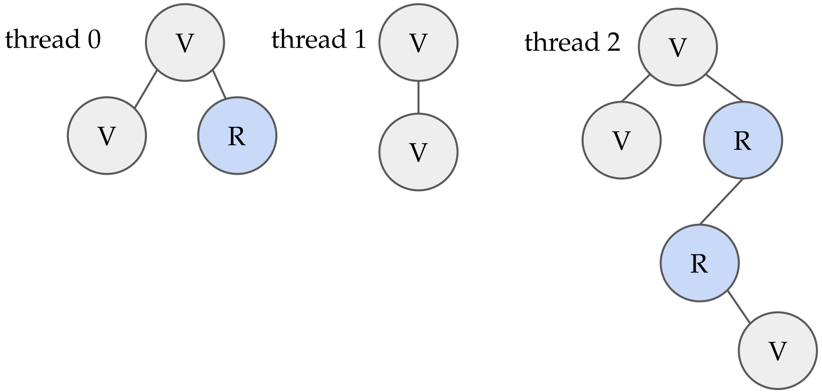
Two-level traversal



Parallel Strategies

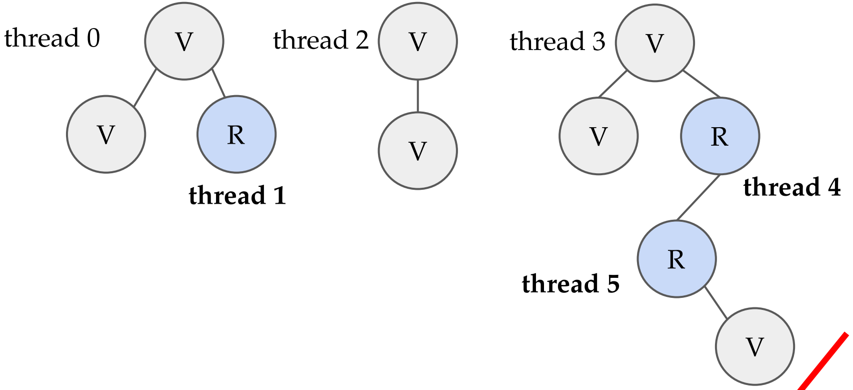
Intra thread

process vertices in parallel



Inter thread

also process rules in parallel



N = average # of iterations to traverse each rule in current state
N < 4: intra-thread
N>= 4 inter-thread

Inter-Level Synchronization-Free Graph Traversal

Avoid rule-level synchronization waiting to make full use of GPU capacity.

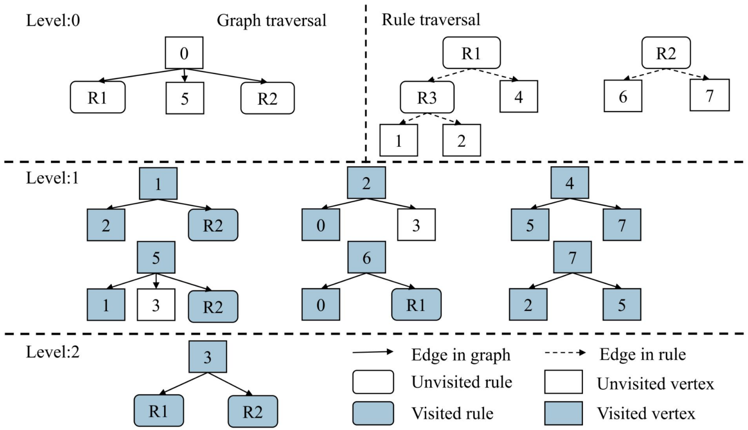
Enables rules and vertices at different graph levels to work simultaneously.

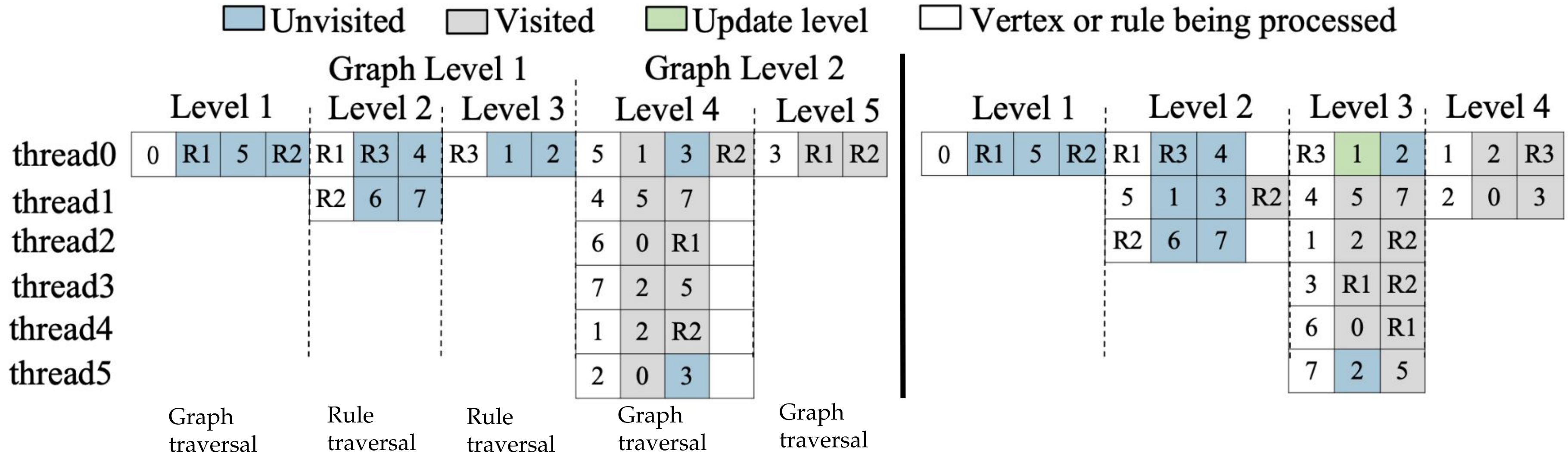
$O((|V|+|E|+|R|)/N)$

Can only be applied to:

1) Result irrelevant to graph level
2) Only one level of graph traversal per round

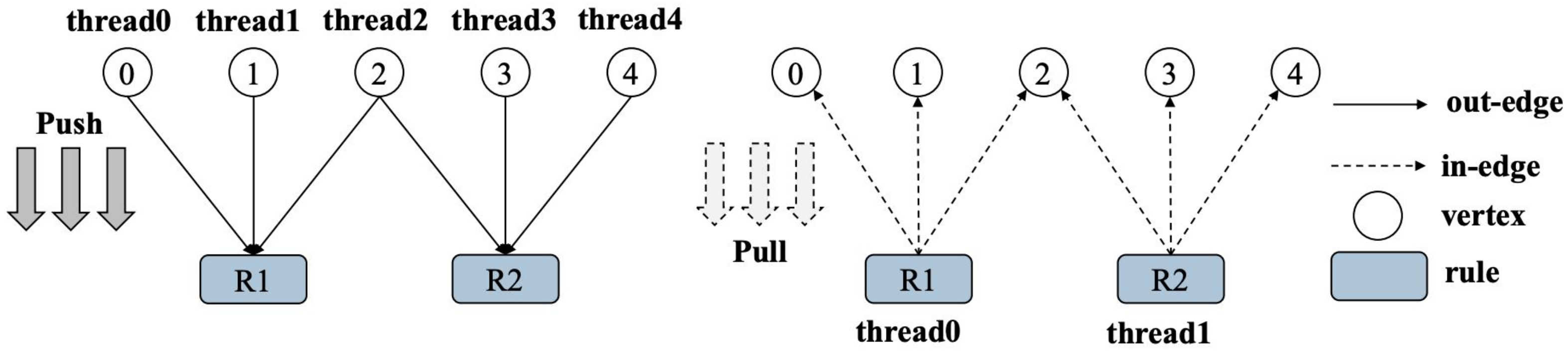
Inter-Level Synchronization-Free Graph Traversal





In-Edge Support Handling Write Conflicts

inverted edges can save $|E|$ atomic operations by *pulling* data from destination rather than *pushing* data to the destination



Speedup results

State-of-the-art compression Ligra+ and Gunrock.

Comparison across 6 common graph application and 12 datasets of various redundancies.

Average of 1.97x speedup over Ligra and 3.95x over Gunrock.

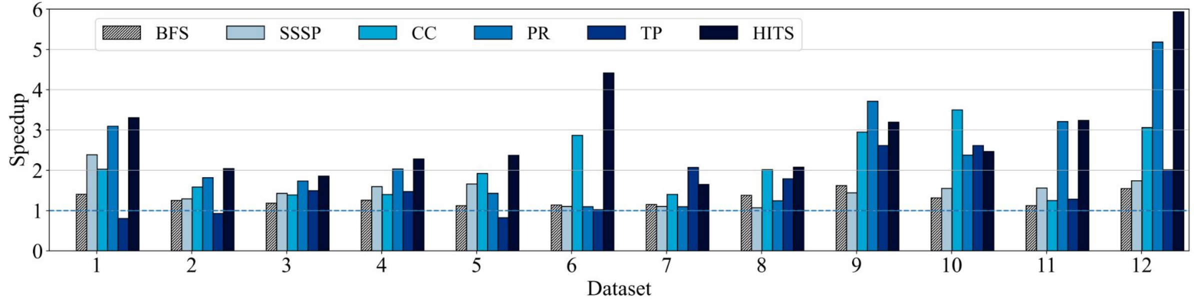


Fig. 11. CPU performance speedup (vs. Ligra+).

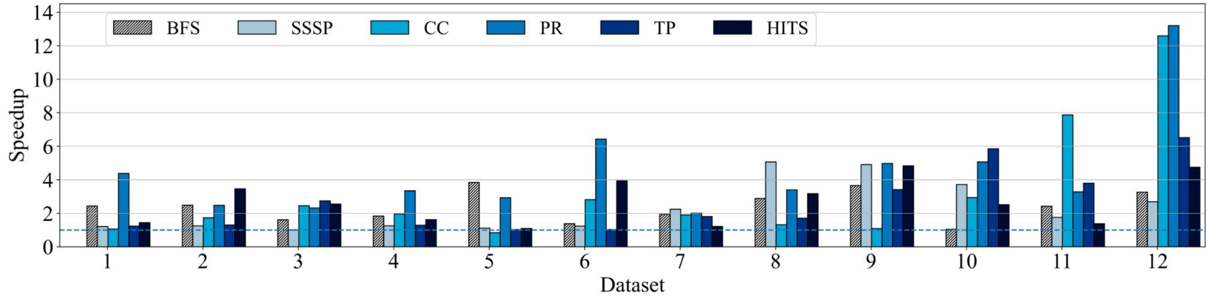
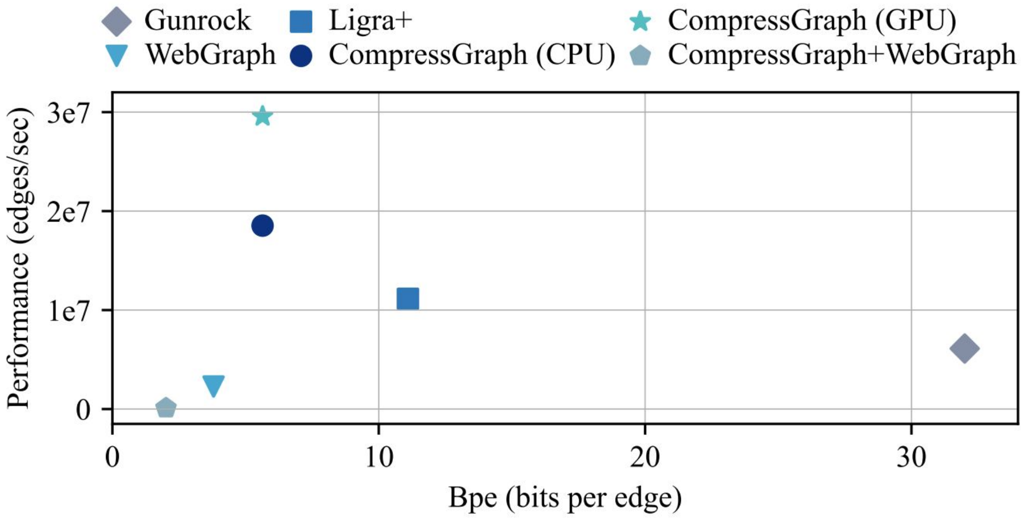


Fig. 12. GPU performance speedup (vs. Gunrock).

Time/space measurement

Compare the number of processed edges per second to the ratio of the size of the graph to the number of edges.



Feature benefit breakdown

Dynamic rule-traversal has ~18% improvement over intra-thread and ~51% improvement over inter-tread.

Synchronization-free traversal gives ~42% improvement. Effective for BFS and HITS.

In-edge has ~28% performance improvement.

Conclusions

Serially,

Enabling direct processing on compressed graphs has large space and time improvements.

Parallelly,

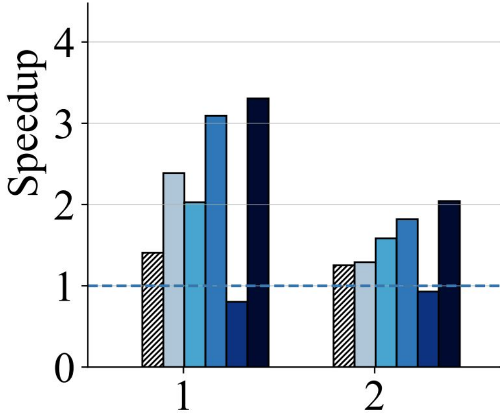
CompressGraph can be optimized to handle parallelism without decompressing the graph.

Strengths and weaknesses, directions for future work

CompressGraph's rule construction expects redundancy in graphs.

In 3 cases, TP sort is slower on CompressGraph than state of the art:

- rule-level synchronization waiting
- smaller, denser graphs with less redundancy



No performance on dynamic graphs.