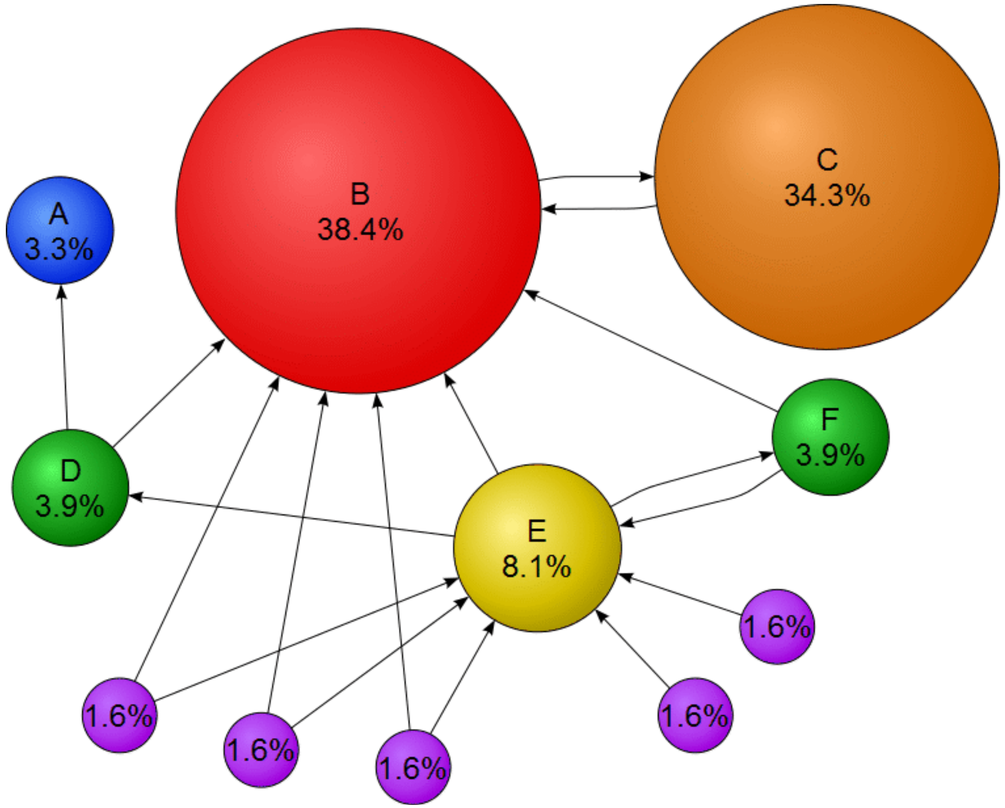
GraphIt: A DSL for High-Performance Graph Analytics

Julian Shun (MIT)

Joint work with Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, and Saman Amarasinghe

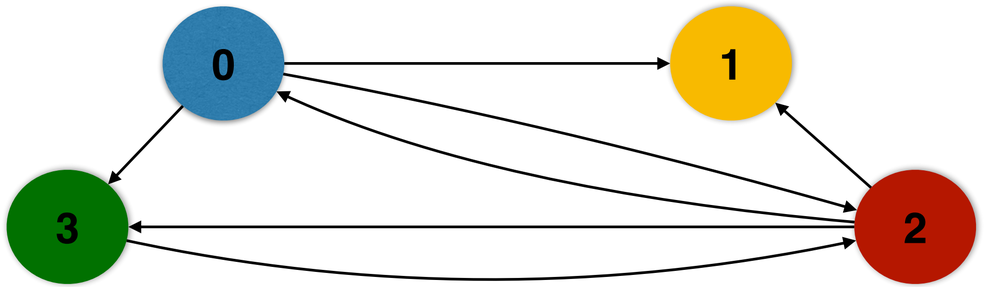
PageRank



$$\mathrm{PR}[v] = \frac{1-\gamma}{|V|} + \gamma \sum_{u \in N^-(v)} \frac{\mathrm{PR}[u]}{deg^+(u)}$$

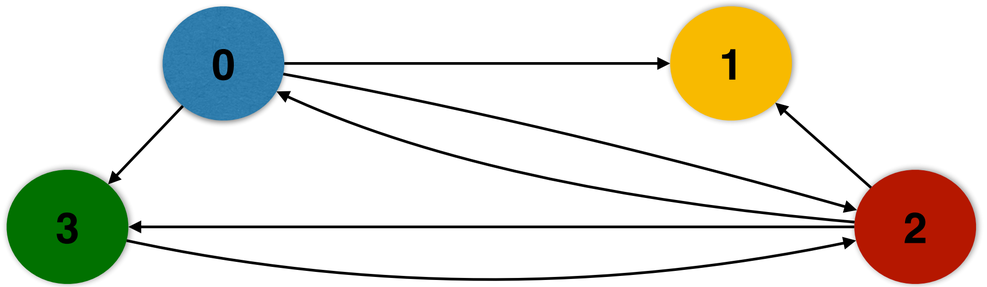
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getOutNeighbors(node)
            newRanks[ngh] += ranks[node]/outDegree[node];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



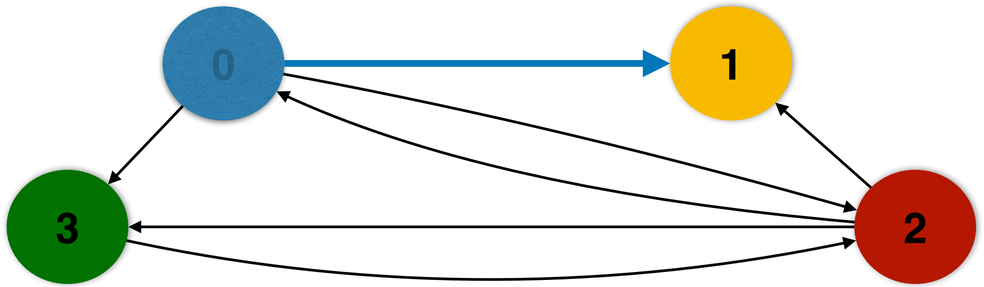
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getOutNeighbors(node)
            newRanks[ngh] += ranks[node]/outDegree[node];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



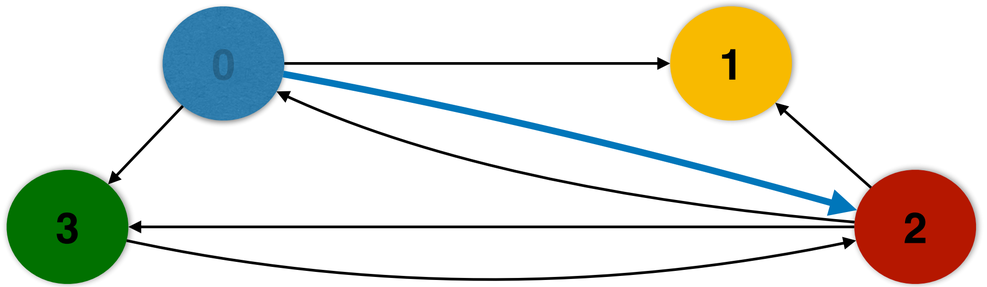
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getOutNeighbors(node)
            newRanks[ngh] += ranks[node]/outDegree[node];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



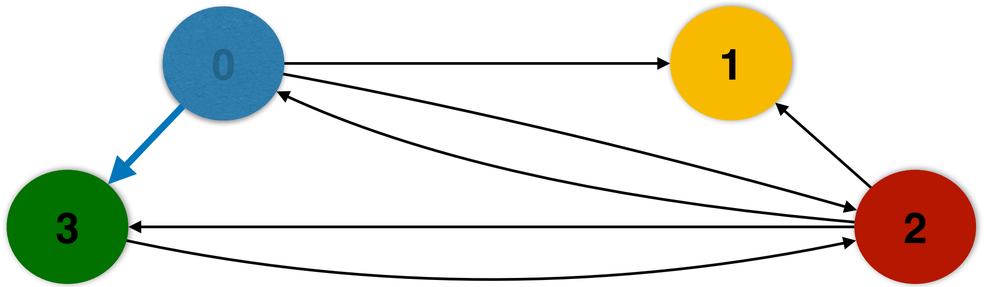
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getOutNeighbors(node)
            newRanks[ngh] += ranks[node]/outDegree[node];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



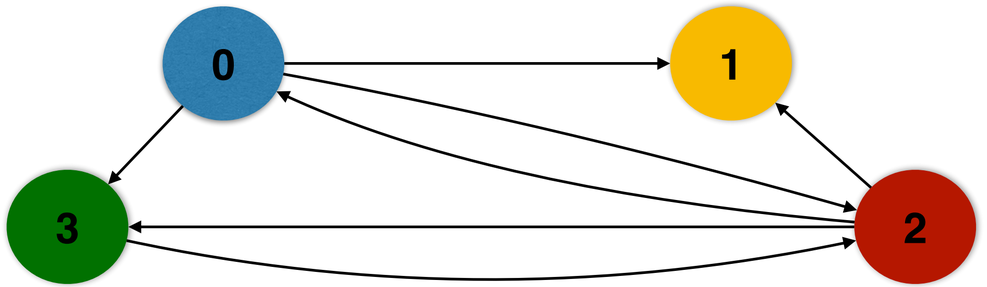
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getOutNeighbors(node)
            newRanks[ngh] += ranks[node]/outDegree[node];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



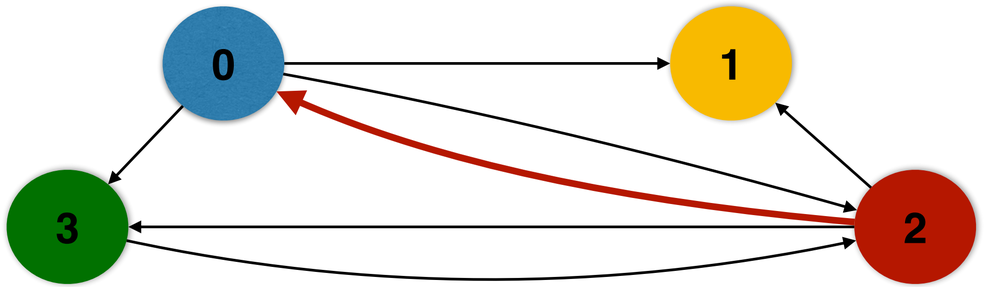
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getOutNeighbors(node)
            newRanks[ngh] += ranks[node]/outDegree[node];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



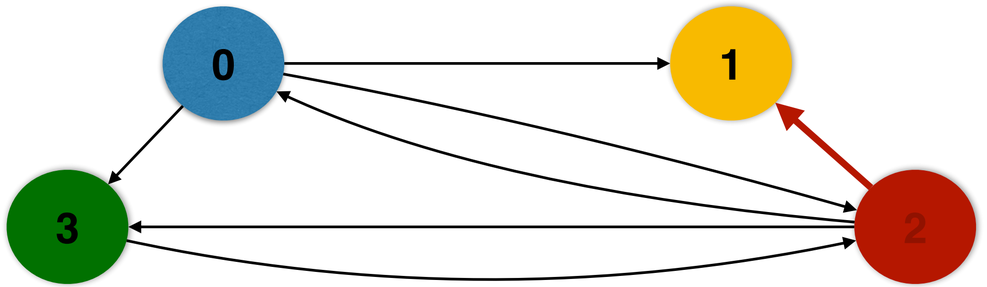
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getOutNeighbors(node)
            newRanks[ngh] += ranks[node]/outDegree[node];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



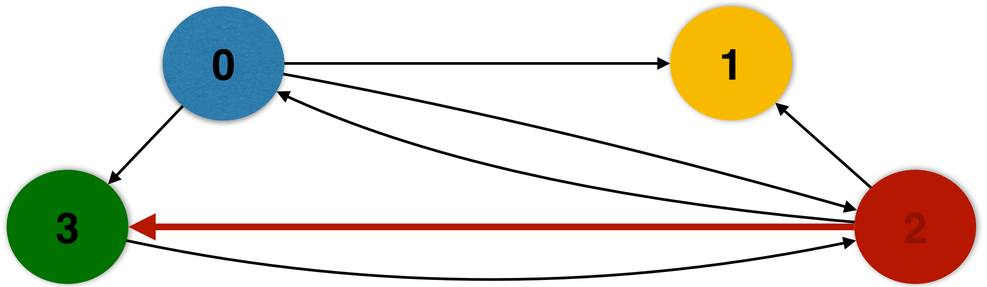
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getOutNeighbors(node)
            newRanks[ngh] += ranks[node]/outDegree[node];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



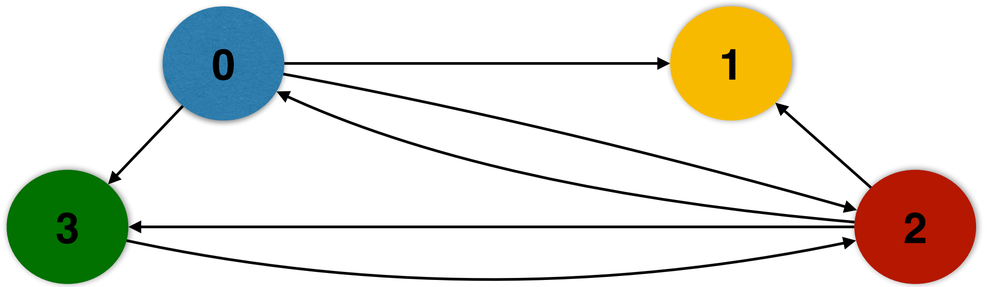
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getOutNeighbors(node)
            newRanks[ngh] += ranks[node]/outDegree[node];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



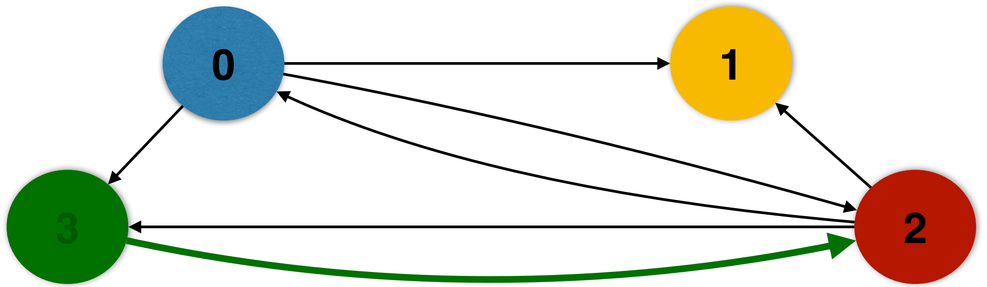
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getOutNeighbors(node)
            newRanks[ngh] += ranks[node]/outDegree[node];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



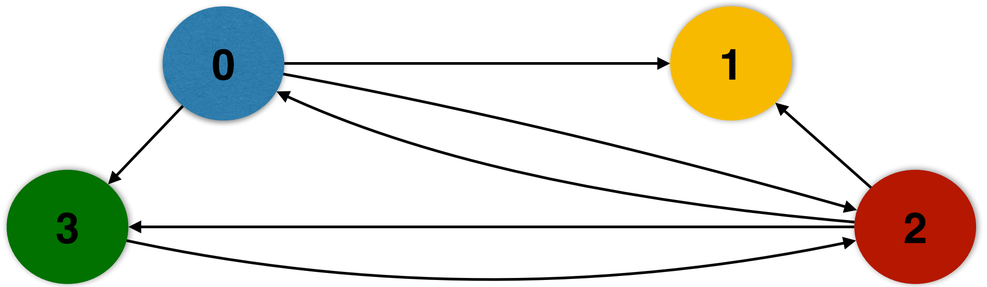
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getOutNeighbors(node)
            newRanks[ngh] += ranks[node]/outDegree[node];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getOutNeighbors(node)
            newRanks[ngh] += ranks[node]/outDegree[node];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



PageRank Example in C++

```
void pagerank(Graph &graph, double * new_rank, double * old_rank, int * out_degree, int max_iter){
   for (i = 0; i < max_iter; i++) {
      for (src : graph.vertices()) {
         for (dst : graph.getOutgoingNeighbors(node)) {
            new_rank[dst] += old_rank[src]/out_degree[src]; } }
      for (node : graph.vertices()) {
         new_rank[node] = base_score + damping*new_rank[node]; }
      swap (old_rank, new_rank); }
}
```

Hand-Optimized C++

```
template<typename APPLY_FUNC>
void edgeset_apply_pull_parallel(Graph &g, APPLY_FUNC apply_func) {
    int64_t numVertices = g.num_nodes(), numEdges = g.num_edges();
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            local_new_rank[socketId][n] = new_rank[n]; } }
    int numPlaces = omp_get_num_places();
    int numSegments = g.getNumSegments("s1");
    int segmentsPerSocket = (numSegments + numPlaces - 1) / numPlaces;
        #pragma omp parallel num_threads(numPlaces) proc_bind(spread){
        int socketId = omp_get_place_num();
        for (int i = 0; i < segmentsPerSocket; i++) {
            int segmentId = socketId + i * numPlaces;
            if (segmentId >= numSegments) break;
            auto sg = g.getSegmentedGraph(std::string("s1"), segmentId);
            #pragma omp parallel num_threads(omp_get_place_num_procs(socketId)) proc_bind(close){
                #pragma omp for schedule(dynamic, 1024)
                for (NodeID localId = 0; localId < sg->numVertices; localId++) {
                    NodeID d = sg->graphId[localId];
                    for (int64_t ngh = sg->vertexArray[localId]; ngh < sg->vertexArray[localId + 1]; ngh++) {
                        NodeID s = sg->edgeArray[ngh];
                        local_new_rank[socketId][d] += contrib[s]; }}}}}
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            new_rank[n] += local_new_rank[socketId][n]; }}}
struct updateVertex {
    void operator() (NodeID v) {
        double old_score = old_rank[v];
        new_rank[v] = (beta_score + (damp * new_rank[v]));
        error[v] = fabs((new_rank[v] - old_rank[v])) ;
        old_rank[v] = new_rank[v];
        new_rank[v] = ((float) 0) ; }; };
void pagerank(Graph &g, double *new_rank, double *old_rank, int *out_degree, int max_iter) {
    for (int i = (0); i < (max_iter); i++) {
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            contrib[v] = (old_rank[v] / out_degree[v]);};
        edgeset_apply_pull_parallel(edges, updateEdge());
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            updateVertex()(v_iter); }; }
```

More than 23x faster

Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores.

Hand-Optimized C++

```
template<typename APPLY_FUNC>
void edgeset_apply_pull_parallel(Graph &g, APPLY_FUNC apply_func) {
    int64_t numVertices = g.num_nodes(), numEdges = g.num_edges();
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            local_new_rank[socketId][n] = new_rank[n]; } }
    int numPlaces = omp_get_num_places();
    int numSegments = g.getNumSegments("s1");
    int segmentsPerSocket = (numSegments + numPlaces - 1) / numPlaces;
        #pragma omp parallel num_threads(numPlaces) proc_bind(spread){
        int socketId = omp_get_place_num();
        for (int i = 0; i < segmentsPerSocket; i++) {
            int segmentId = socketId + i * numPlaces;
            if (segmentId >= numSegments) break;
            auto sg = g.getSegmentedGraph(std::string("s1"), segmentId);
            #pragma omp parallel num_threads(omp_get_place_num_procs(socketId)) proc_bind(close){
                #pragma omp for schedule(dynamic, 1024)
                for (NodeID localId = 0; localId < sg->numVertices; localId++) {
                    NodeID d = sg->graphId[localId];
                    for (int64_t ngh = sg->vertexArray[localId]; ngh < sg->vertexArray[localId + 1]; ngh++) {
                        NodeID s = sg->edgeArray[ngh];
                        local_new_rank[socketId][d] += contrib[s]; }}}}}
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            new_rank[n] += local_new_rank[socketId][n]; }}}
struct updateVertex {
    void operator() (NodeID v) {
        double old_score = old_rank[v];
        new_rank[v] = (beta_score + (damp * new_rank[v]));
        error[v] = fabs((new_rank[v] - old_rank[v])) ;
        old_rank[v] = new_rank[v];
        new_rank[v] = ((float) 0) ; }; };
void pagerank(Graph &g, double *new_rank, double *old_rank, int *out_degree, int max_iter) {
    for (int i = (0); i < (max_iter); i++) {
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            contrib[v] = (old_rank[v] / out_degree[v]);};
        edgeset_apply_pull_parallel(edges, updateEdge());
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            updateVertex()(v_iter); }; }
```

More than 23x faster

Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores.

Multi-Threaded

Load Balanced

NUMA Optimized

Cache Optimized

Hand-Optimized C++

```
template<typename APPLY_FUNC>
void edgeset_apply_pull_parallel(Graph &g, APPLY_FUNC apply_func) {
    int64_t numVertices = g.num_nodes(), numEdges = g.num_edges();
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            local_new_rank[socketId][n] = new_rank[n]; } }
    int numPlaces = omp_get_num_places();
    int numSegments = g.getNumSegments("s1");
    int segmentsPerSocket = (numSegments + numPlaces - 1) / numPlaces;
        #pragma omp parallel num_threads(numPlaces) proc_bind(spread){
        int socketId = omp_get_place_num();
        for (int i = 0; i < segmentsPerSocket; i++) {
            int segmentId = socketId + i * numPlaces;
            if (segmentId >= numSegments) break;
            auto sg = g.getSegmentedGraph(std::string("s1"), segmentId);
            #pragma omp parallel num_threads(omp_get_place_num_procs(socketId)) proc_bind(close){
                #pragma omp for schedule(dynamic, 1024)
                for (NodeID localId = 0; localId < sg->numVertices; localId++) {
                    NodeID d = sg->graphId[localId];
                    for (int64_t ngh = sg->vertexArray[localId]; ngh < sg->vertexArray[localId + 1]; ngh++) {
                        NodeID s = sg->edgeArray[ngh];
                        local_new_rank[socketId][d] += contrib[s]; }}}}}
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            new_rank[n] += local_new_rank[socketId][n]; }}}
struct updateVertex {
    void operator() (NodeID v) {
        double old_score = old_rank[v];
        new_rank[v] = (beta_score + (damp * new_rank[v]));
        error[v] = fabs((new_rank[v] - old_rank[v])) ;
        old_rank[v] = new_rank[v];
        new_rank[v] = ((float) 0) ; }; };
void pagerank(Graph &g, double *new_rank, double *old_rank, int *out_degree, int max_iter) {
    for (int i = (0); i < (max_iter); i++) {
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            contrib[v] = (old_rank[v] / out_degree[v]);};
        edgeset_apply_pull_parallel(edges, updateEdge());
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            updateVertex()(v_iter); }; }
```

More than 23x faster

Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores.

Multi-Threaded

Load Balanced

NUMA Optimized

Cache Optimized

(1) Hard to write correctly

(2) Extremely difficult to experiment with different combinations of optimizations

GraphIt

A Domain-Specific Language for Graph Analytics

- **Decouple algorithm from optimization for graph programs**
  - **Algorithm**: What to Compute
  - **Optimization (schedule)**: How to Compute

GraphIt

A Domain-Specific Language for Graph Analytics

- **Decouple algorithm from optimization for graph programs**
  - **Algorithm**: What to Compute
  - **Optimization (schedule)**: How to Compute
- **Optimization (schedule) representation**
  - **Easy to use** for users to try different combinations
  - **Powerful** enough to beat hand-optimized libraries by up to 4.8x

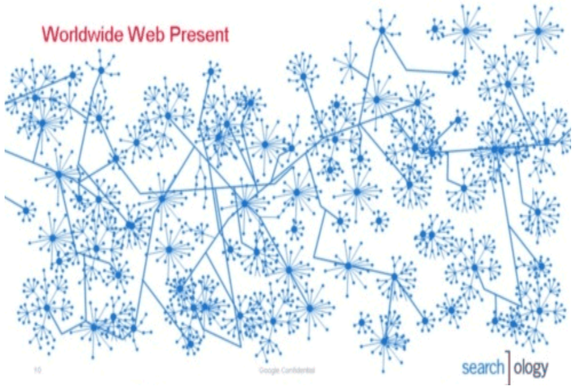
Outline

- Graph Algorithms Overview
- Optimization Tradeoff Space
- GraphIt DSL
- Evaluation

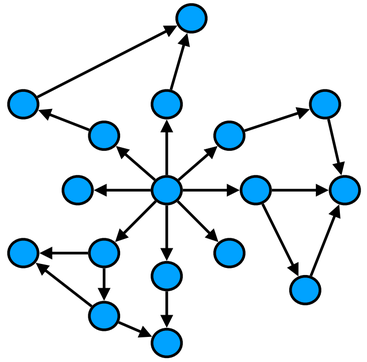
Outline

- Graph Algorithms Overview
- Optimization Tradeoff Space
- GraphIt DSL
- Evaluation

Power-Law Graphs

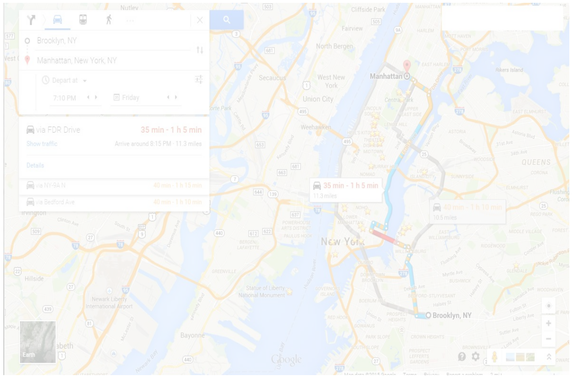


World Wide Web

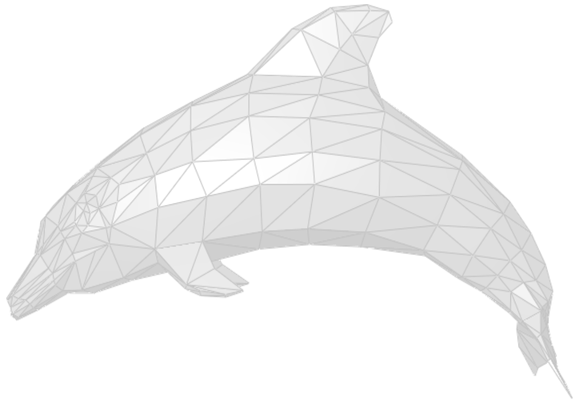


Power-Law Degree Distribution, Small Diameter, Poor Locality

Social Networks



Maps

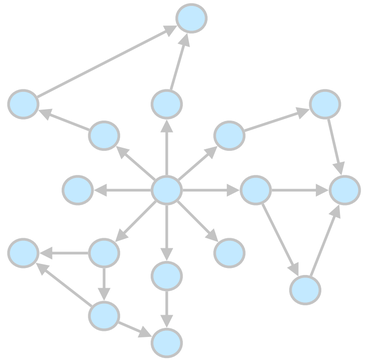


Engineering Meshes

1. http://googlesystem.blogspot.com/2007/05/world-wide-web-as-seen-by-google.html 2. http://www.facebookfever.com/introducing-facebook-new-graph-api-explorer-features/ 3. http://maps.google.com **4. https://en.wikipedia.org/wiki/Polygon_mesh#/media/File:Dolphin_triangle_mesh.png**

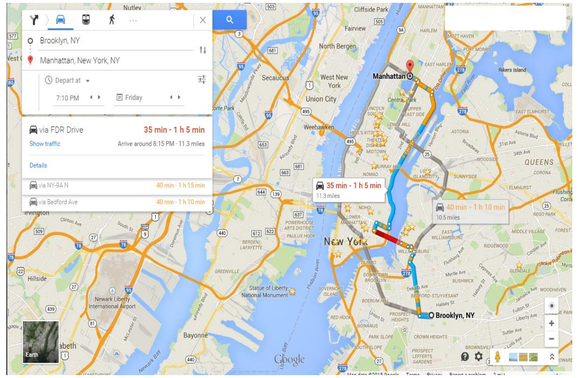
Bounded-Degree Graphs

World Wide Web

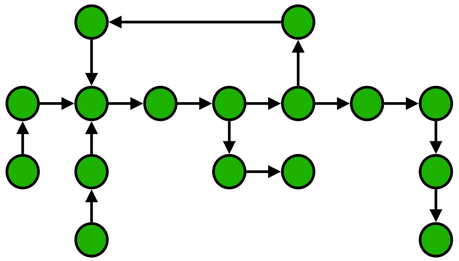


Power-Law Degree Distribution,
Small Diameter, Poor Locality

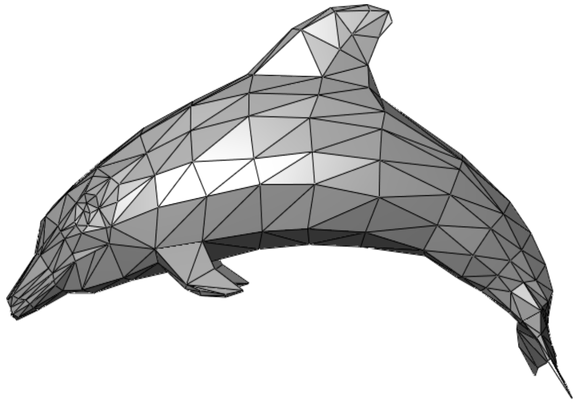
Social Networks



Maps



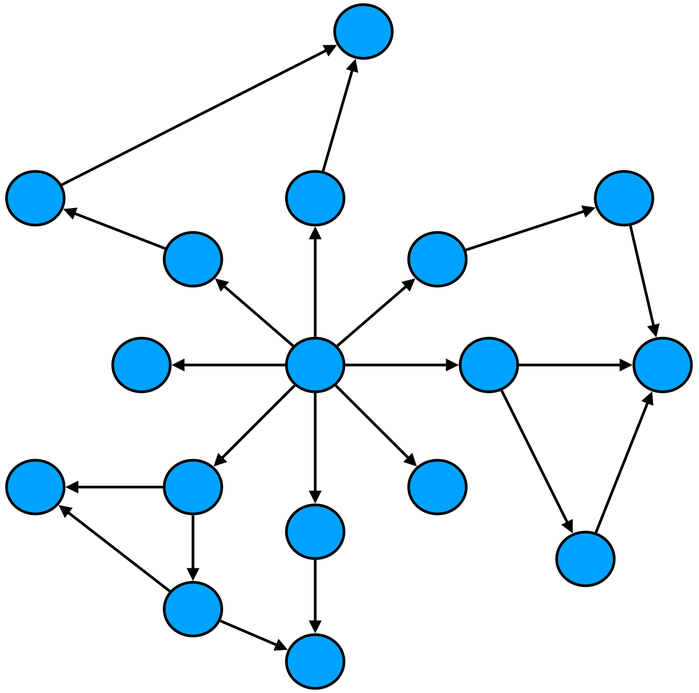
Bounded Degree Distribution
Large Diameter, Excellent Locality



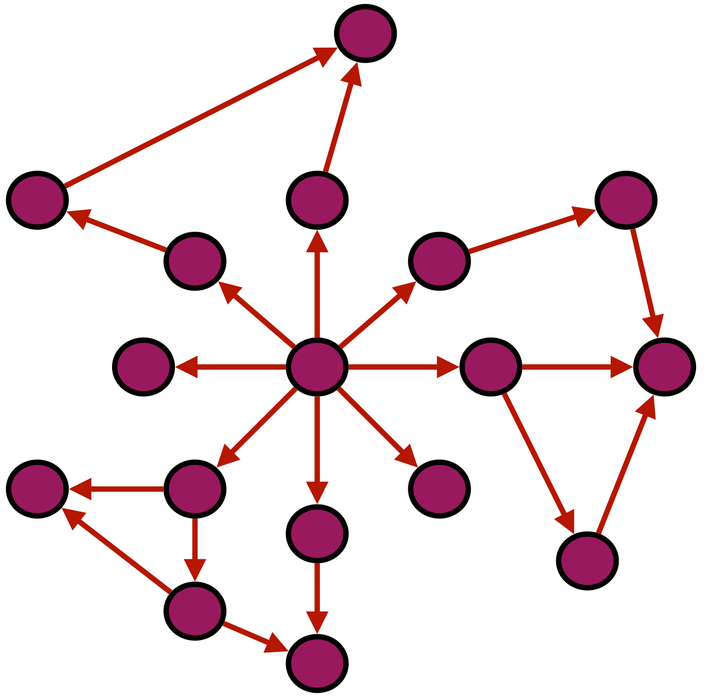
Engineering Meshes

1. http://googlesystem.blogspot.com/2007/05/world-wide-web-as-seen-by-google.html 2. http://www.facebookfever.com/introducing-facebook-new-graph-api-explorer-features/ 3. http://maps.google.com **4. https://en.wikipedia.org/wiki/Polygon_mesh#/media/File:Dolphin_triangle_mesh.png**

Graph Algorithms

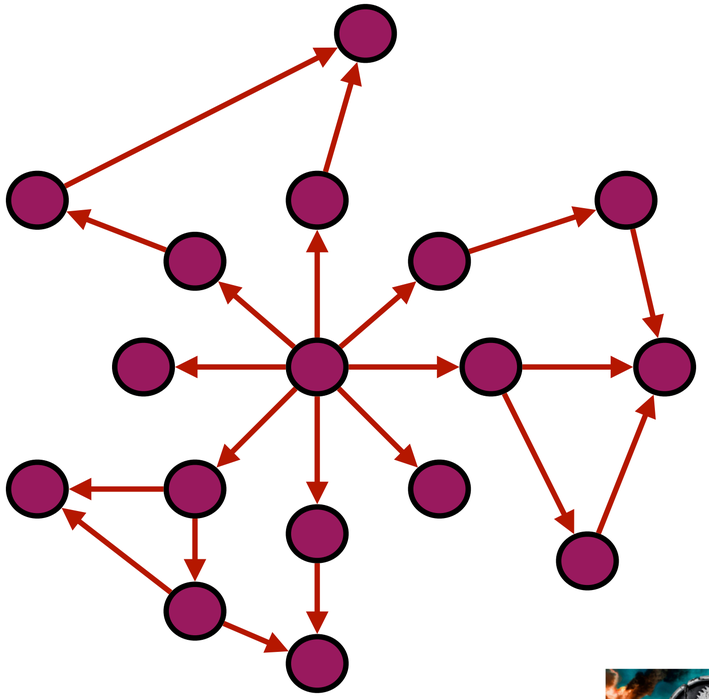


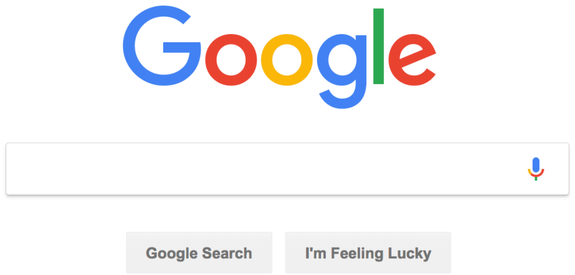
Topology-Driven Algorithms



Work on All Edges and Vertices

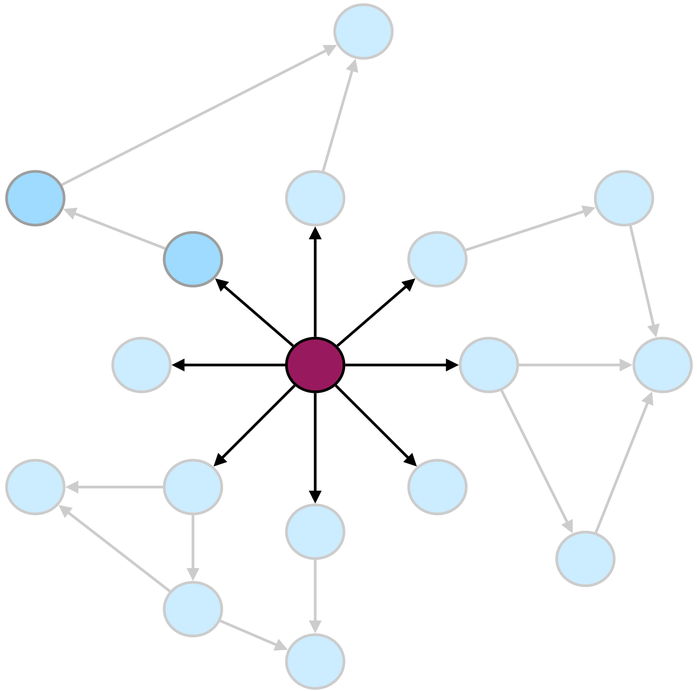
Topology-Driven Algorithms



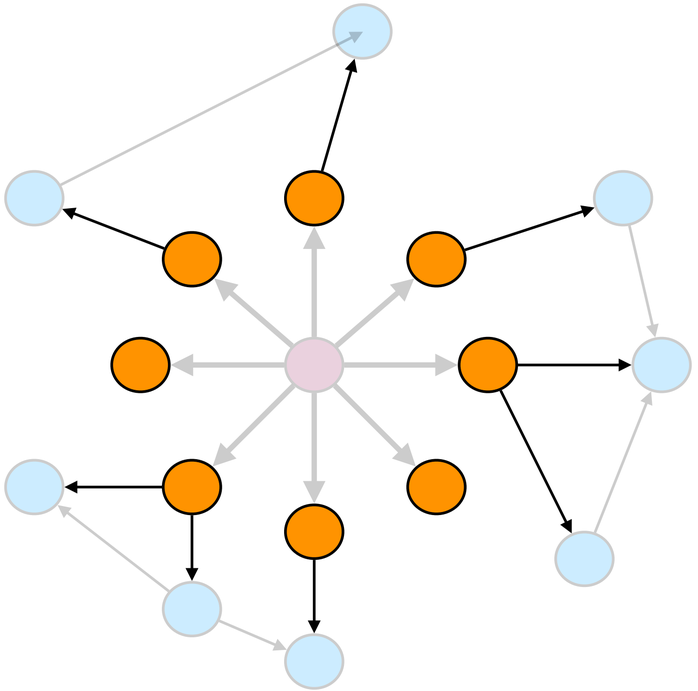


Recommendations for You, Yunming

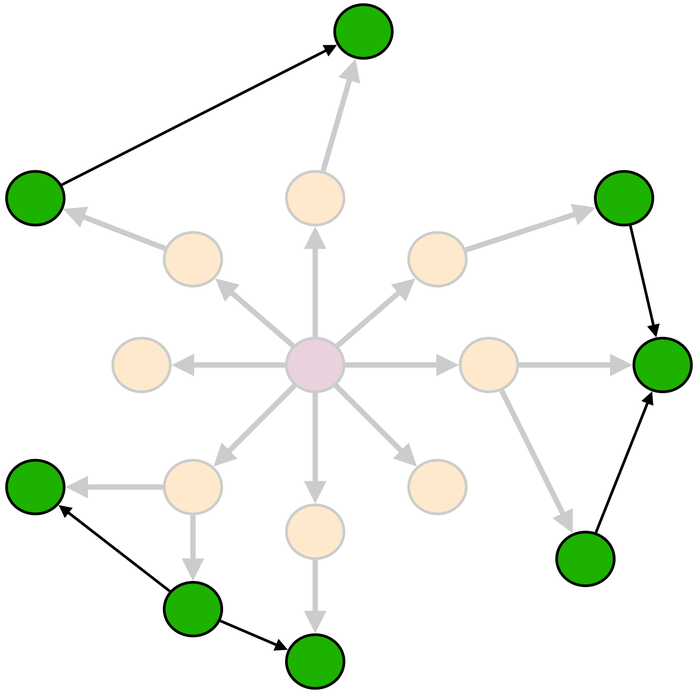
Data-Driven Algorithms



Data-Driven Algorithms

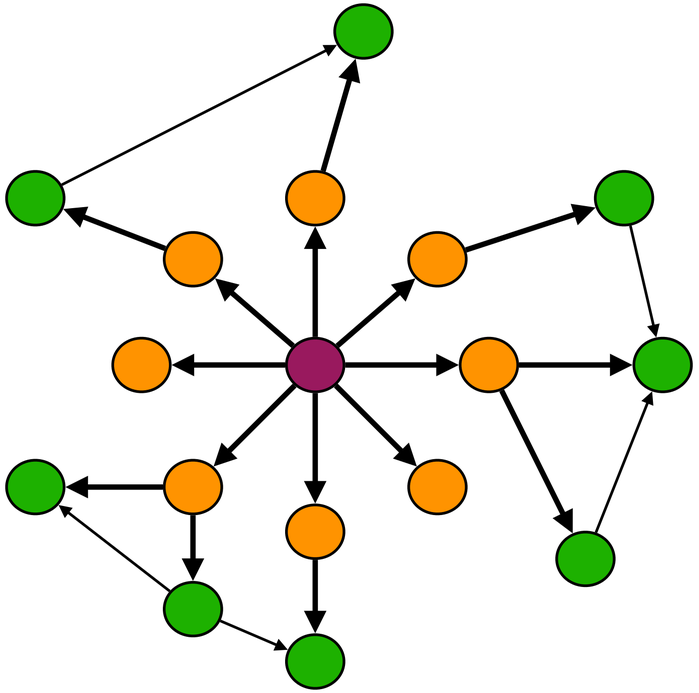


Data-Driven Algorithms



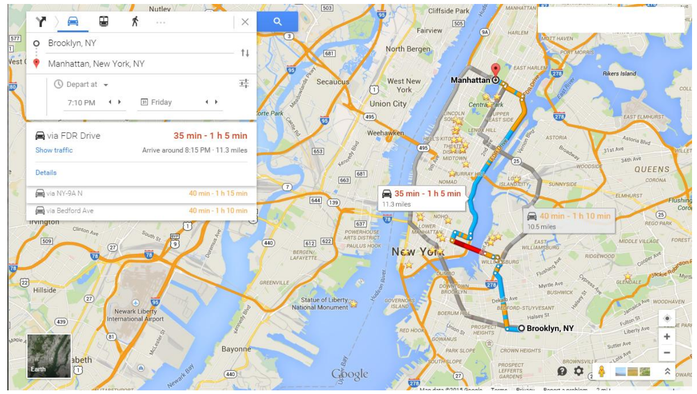
Work on a subset of vertices and edges
(Data-Driven)

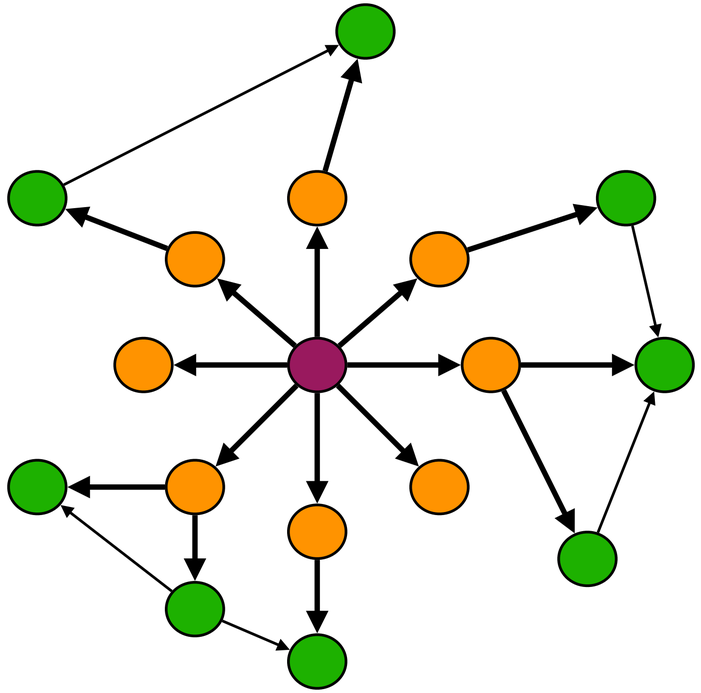
Data-Driven Algorithms

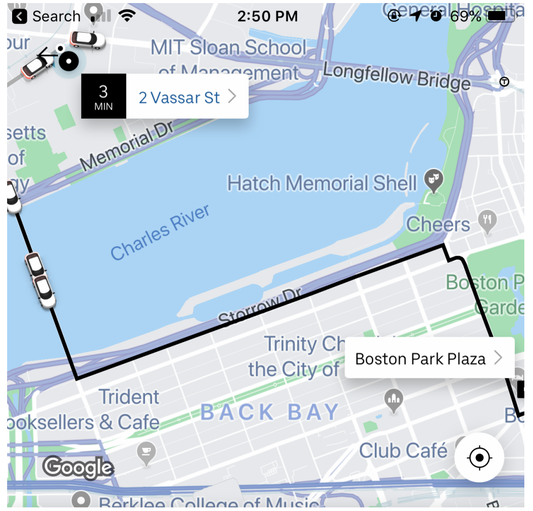


Work on a subset of vertices and edges
(Data-Driven)

Data-Driven Algorithms







Economy

Affordable rides, all to yourself

Premiu

Pool Options

$5.70

3:16pm

UberX

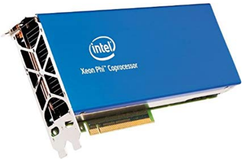
$9.71

3:06pm

Graph Execution Hardware

CPU

GPU



Xeon Phi

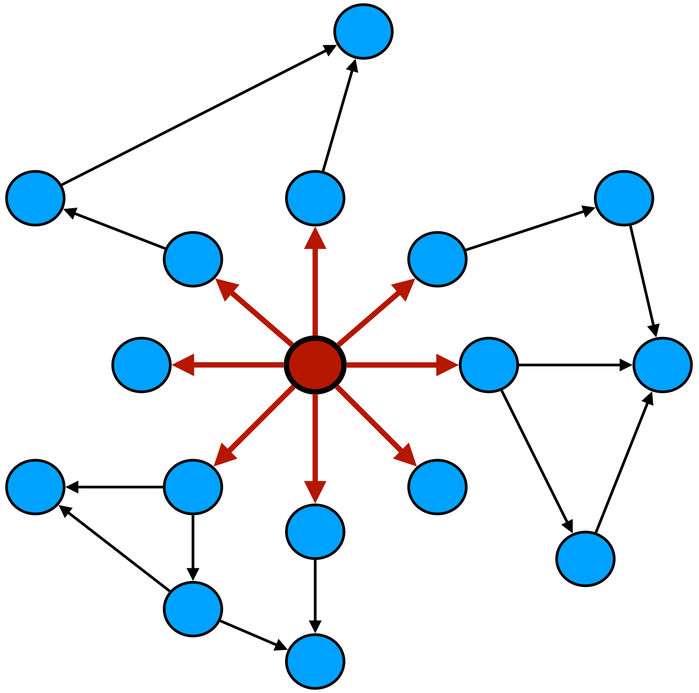
Distributed Cluster

External Memory

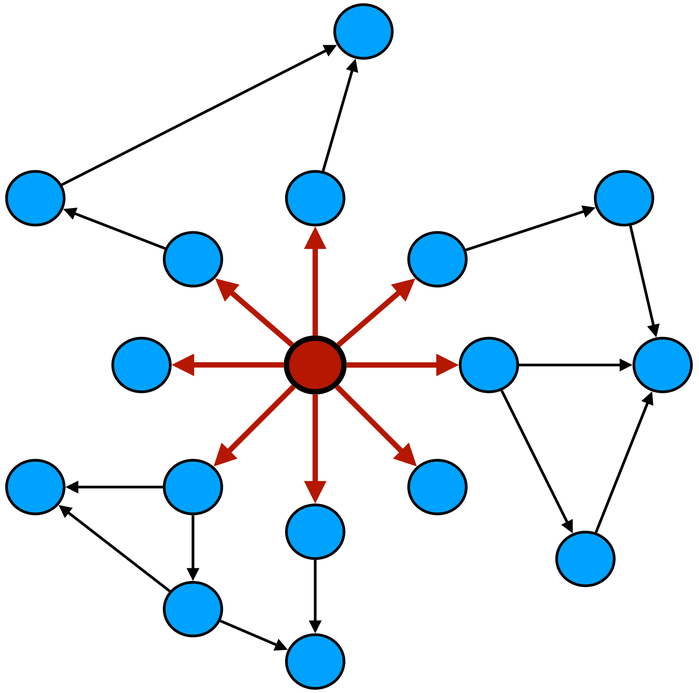
Outline

- Graph Applications Overview
- Optimization Tradeoff Space
- GraphIt DSL
- Evaluation

Push Traversal



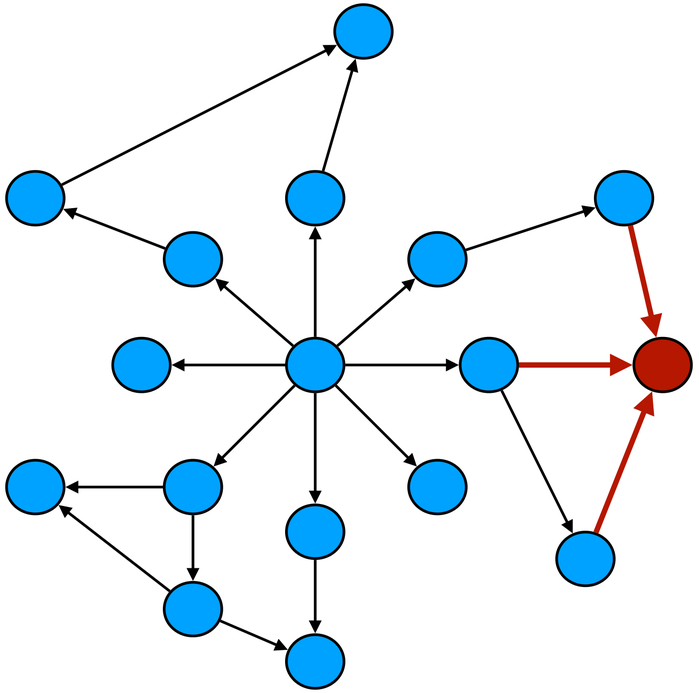
Push Traversal



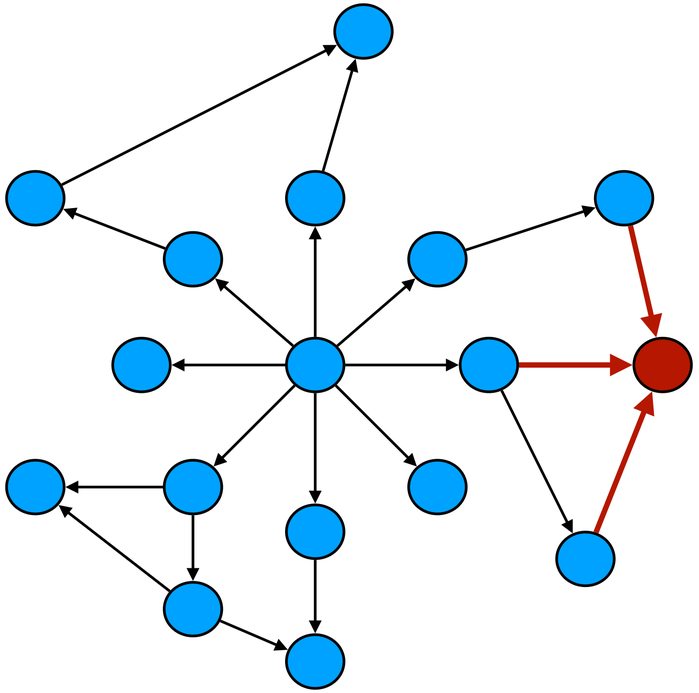
Incurs overhead with atomics

Traverses no extra edges

Pull Traversal



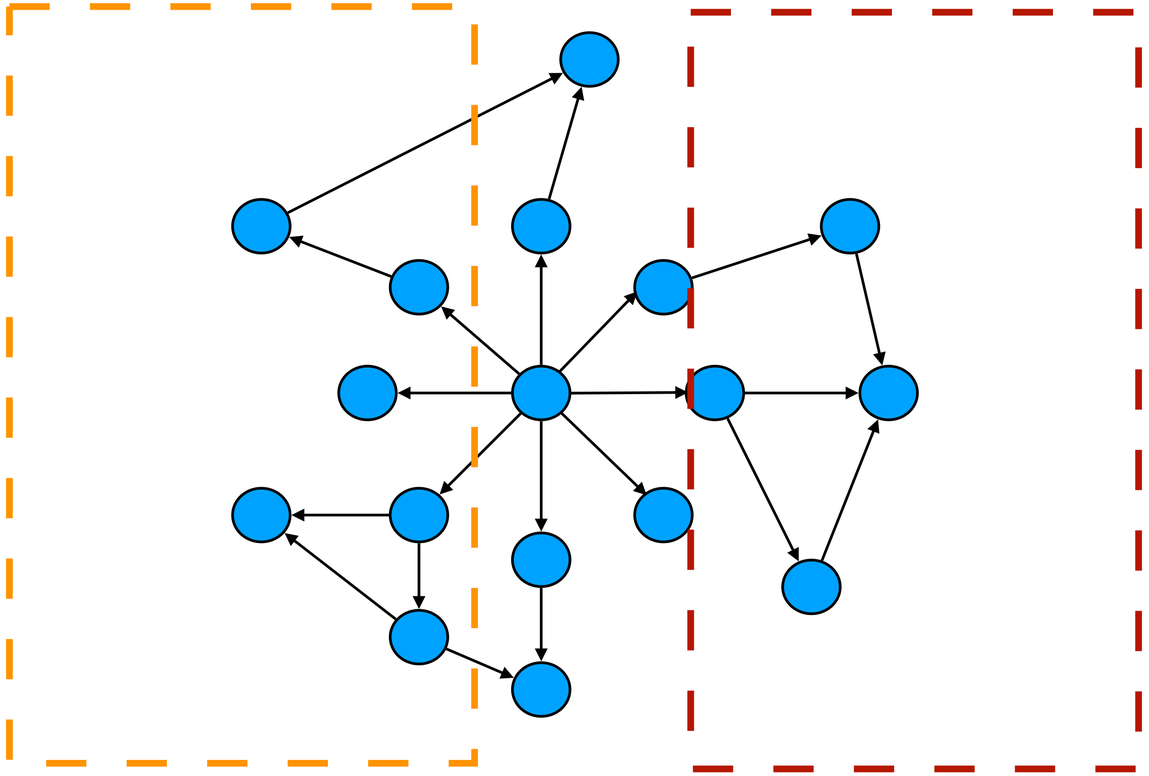
Pull Traversal



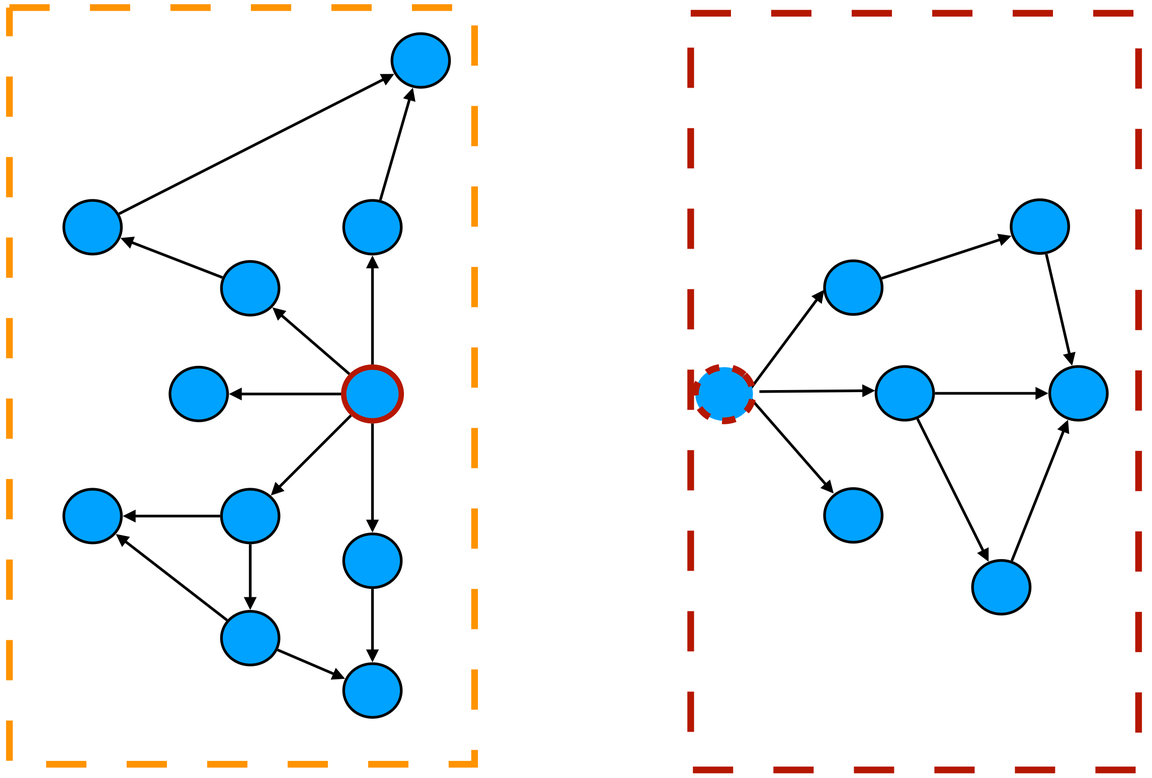
Incurs no overhead from atomics

Traverses extra edges

Partitioning



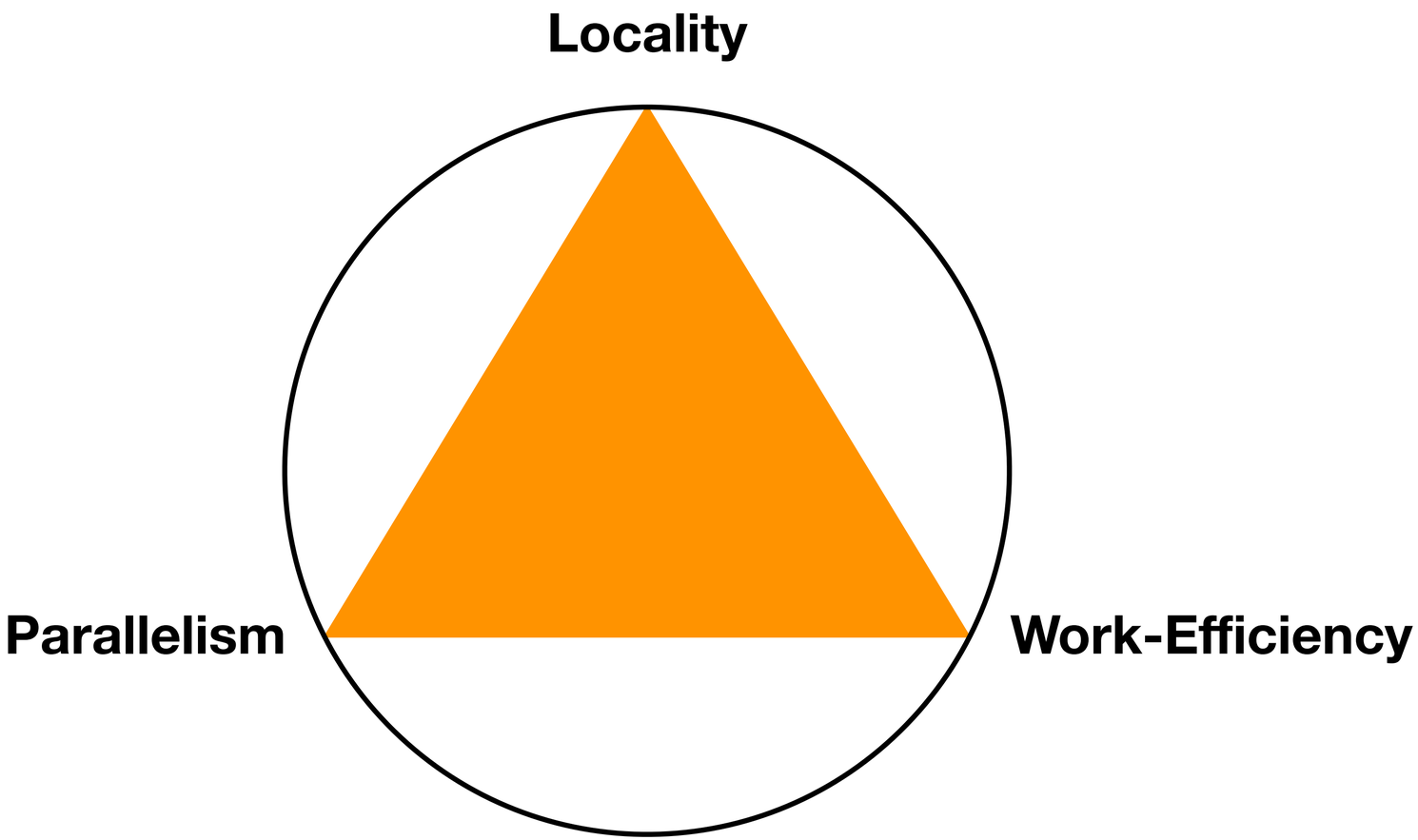
Partitioning



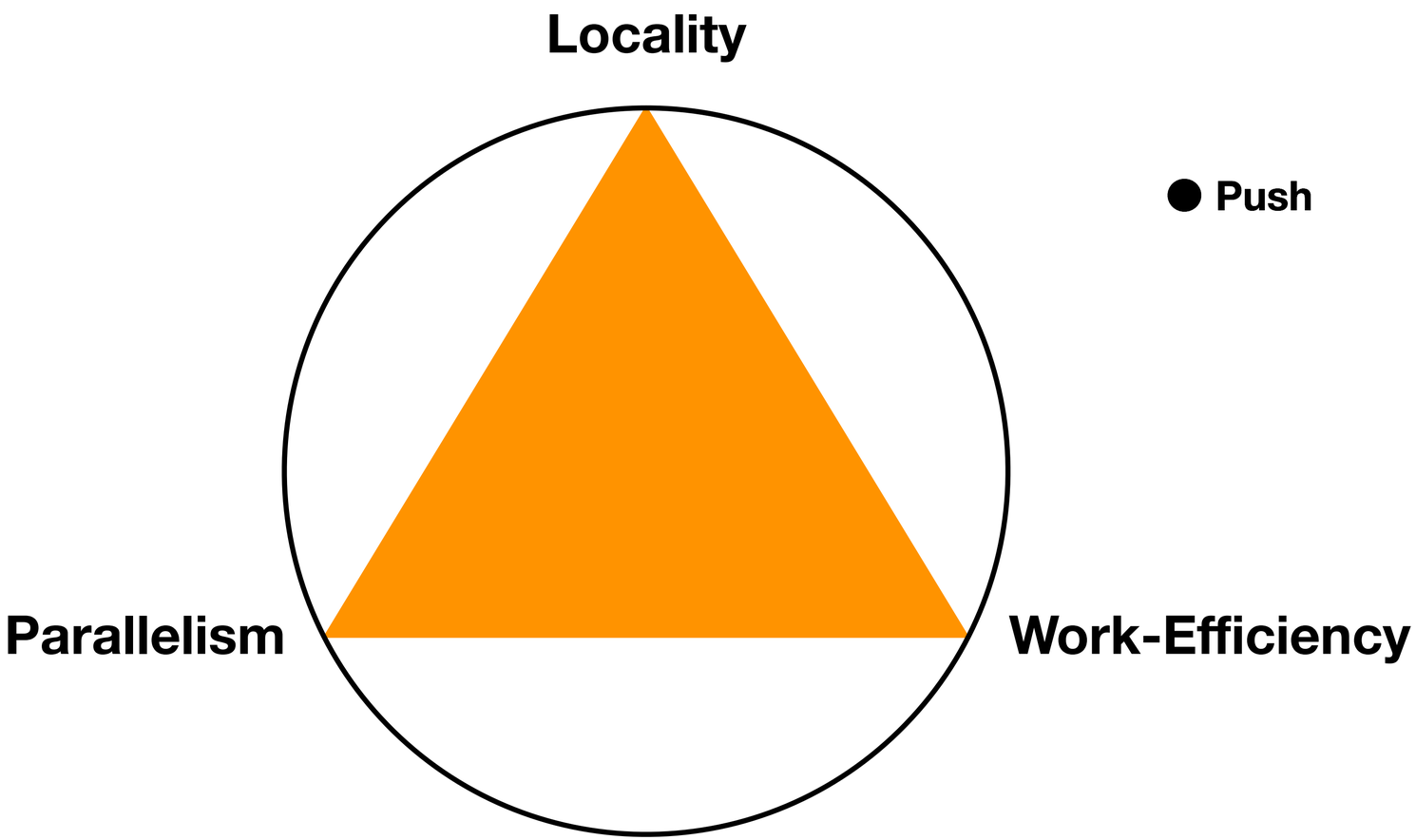
Improves locality

Needs extra instructions to traverse two graphs

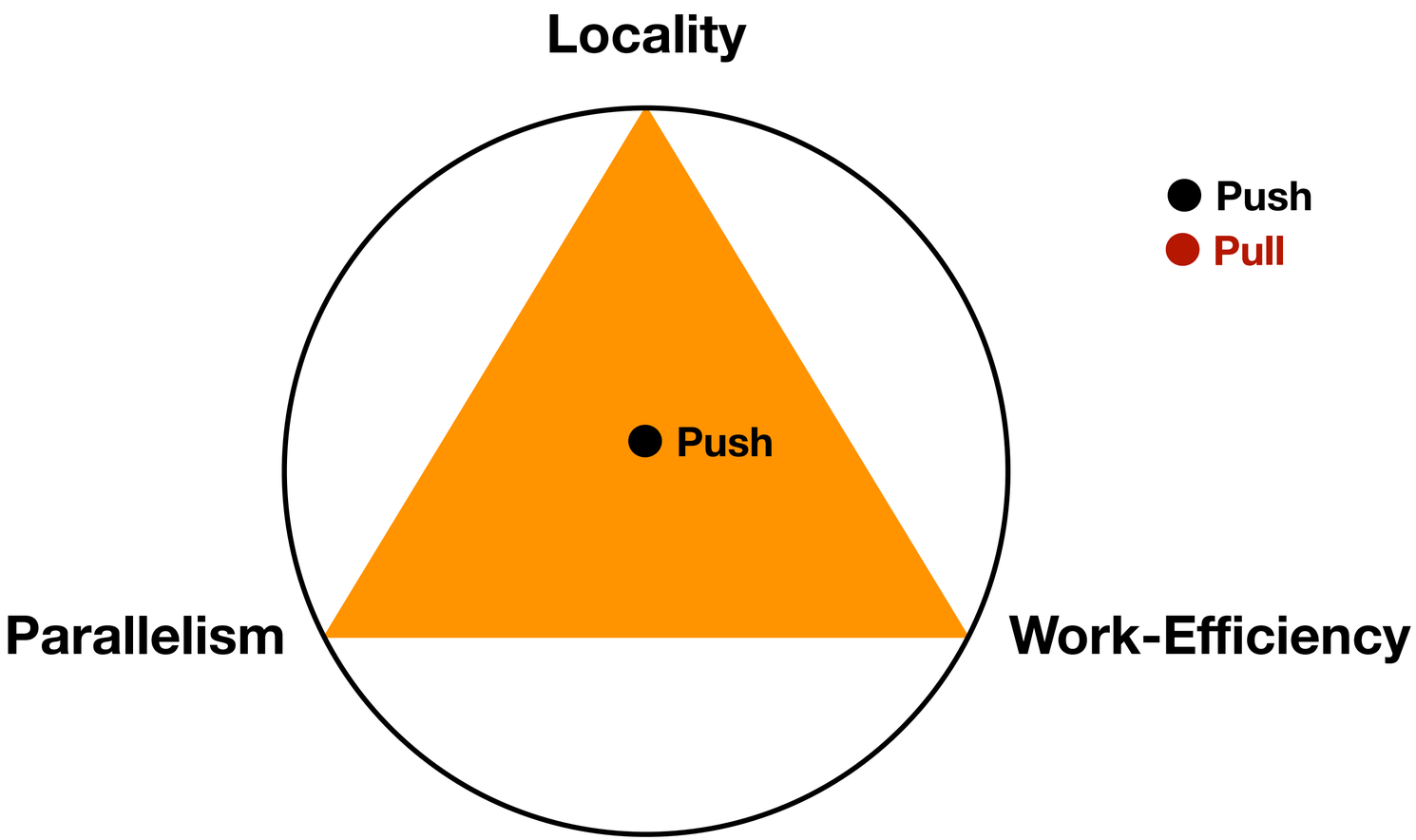
Optimization Tradeoff Space



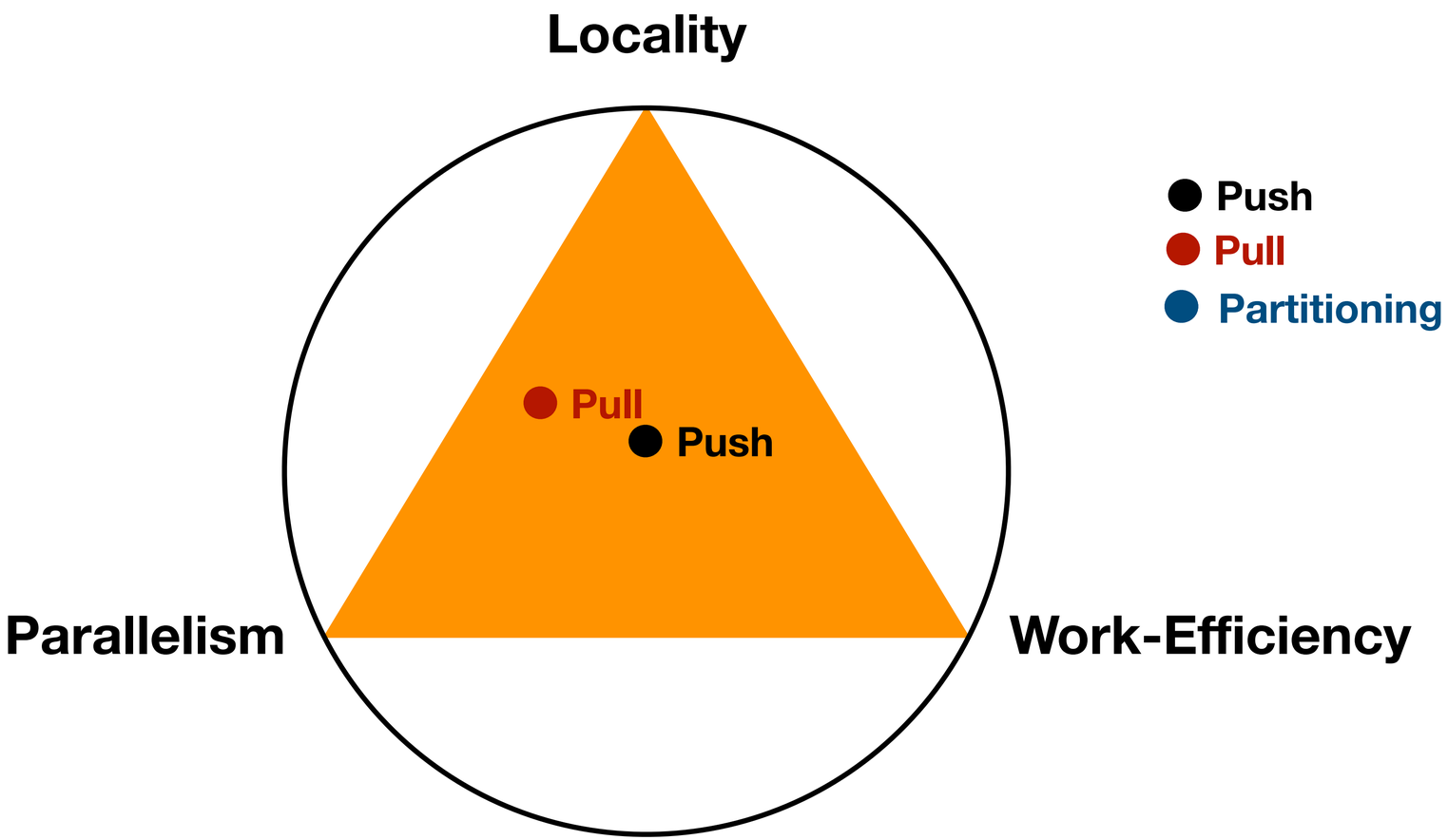
Optimization Tradeoff Space



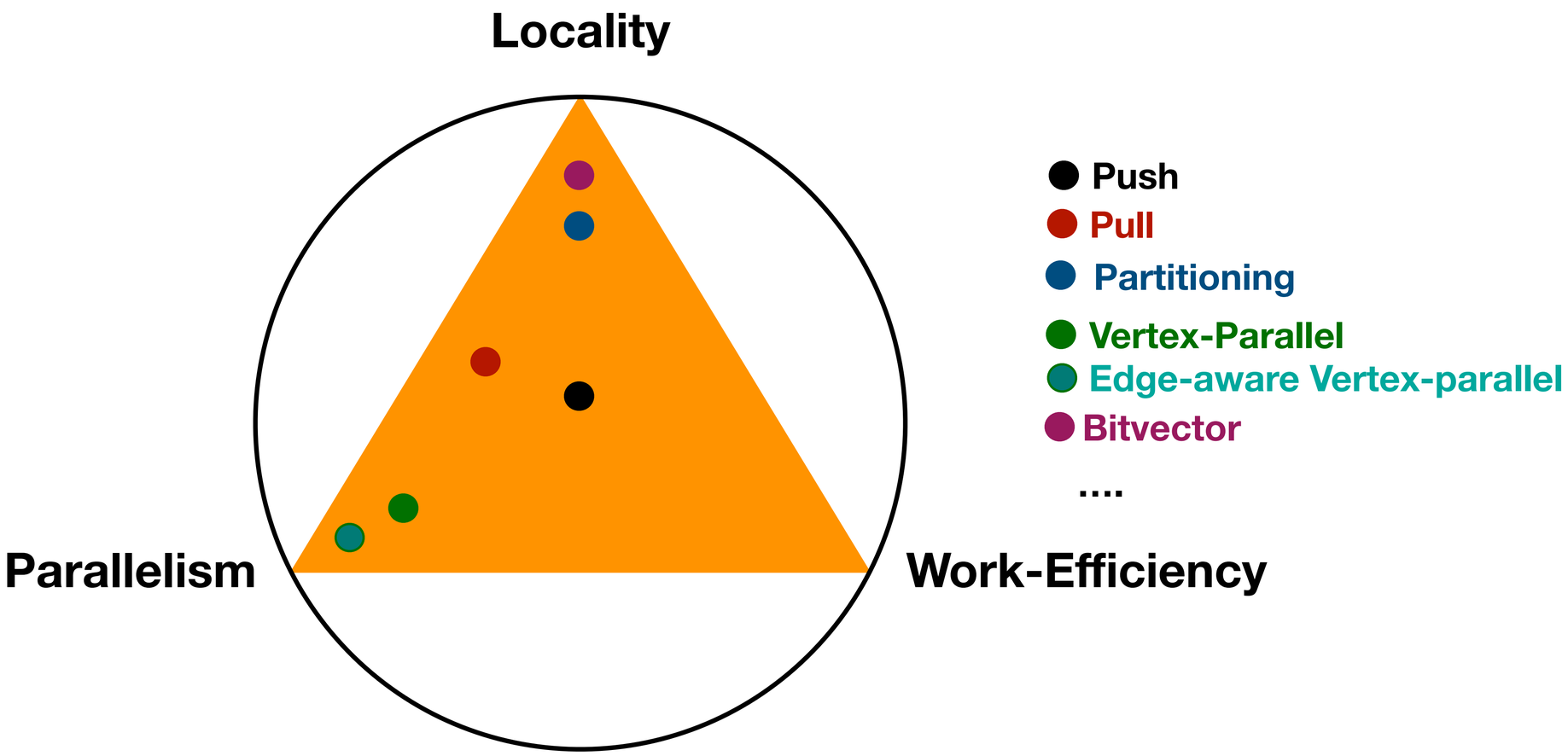
Optimization Tradeoff Space

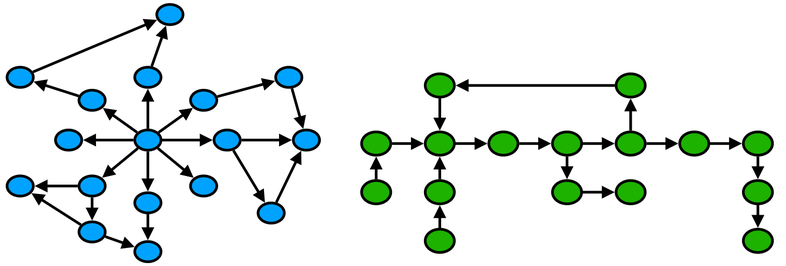


Optimization Tradeoff Space

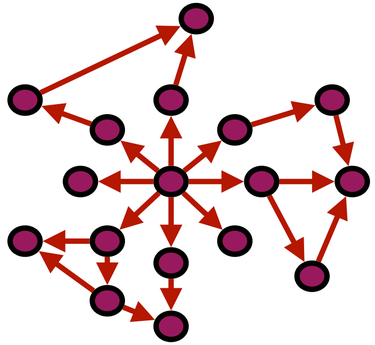


Optimization Tradeoff Space

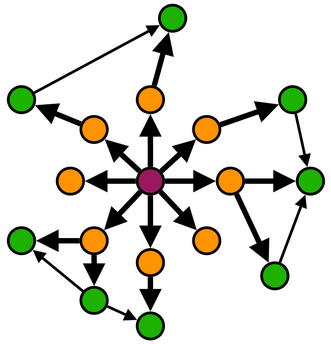


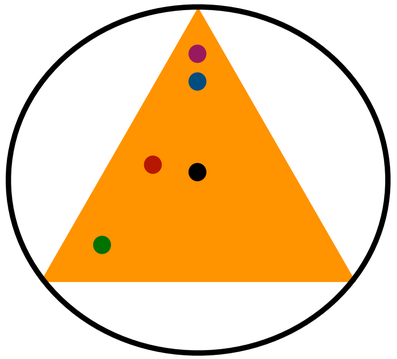


Graphs



Algorithms





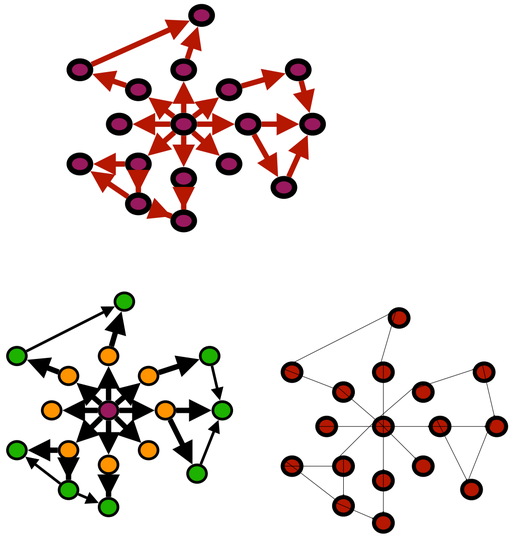
Optimizations

Hardware

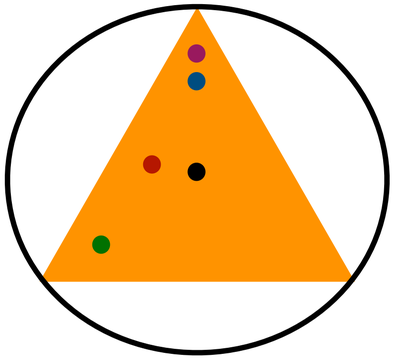
Outline

- Graph Applications Overview
- Optimization Tradeoff Space
- GraphIt DSL
- Evaluation

GraphIt DSL

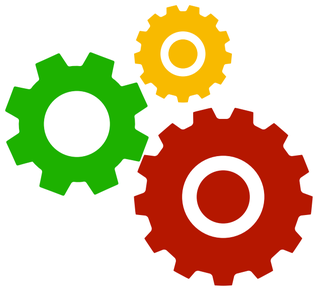


Algorithm Representation (Algorithm Language)



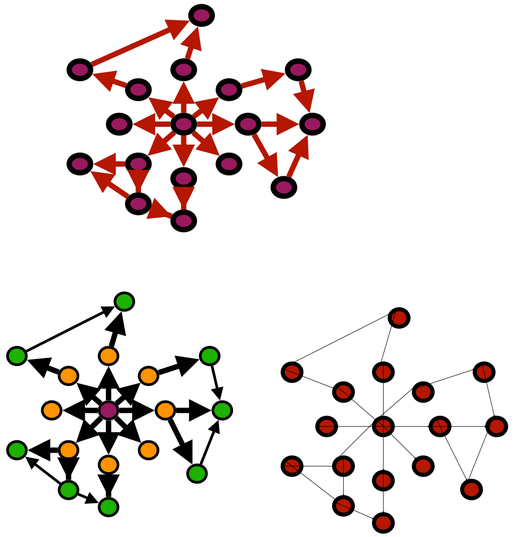
Optimization Representation

- **Scheduling Language**
- **Schedule Representation (e.g. Graph Iteration Space)**



Autotuner

GraphIt DSL



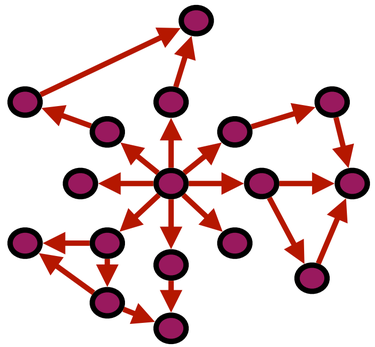
Algorithm Representation
(Algorithm Language)

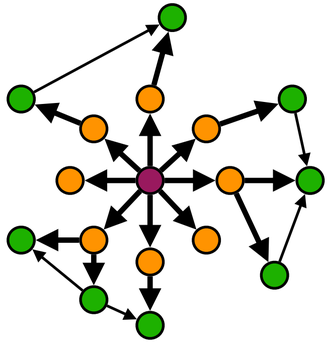
Optimization Representation

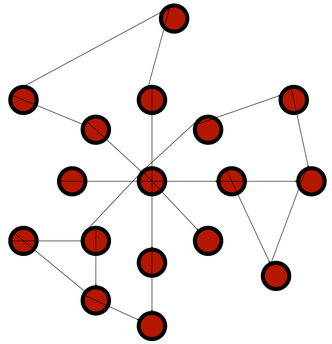
- **Scheduling Language**
- **Schedule Representation**
  (e.g. Graph Iteration Space)

Autotuner

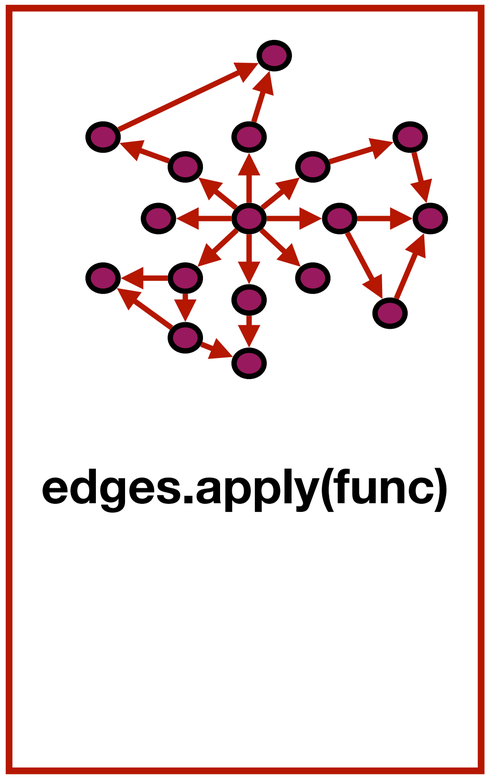
Algorithm Language

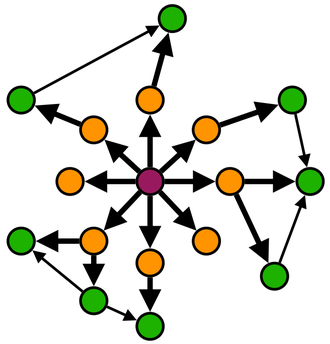


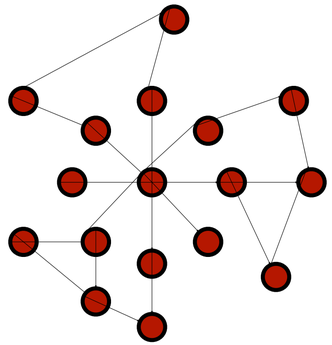




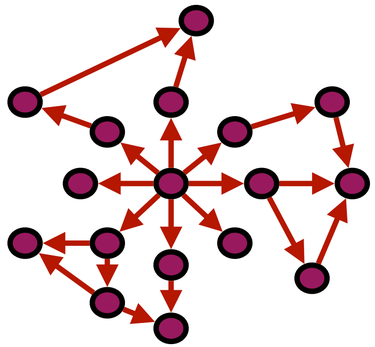
Algorithm Language



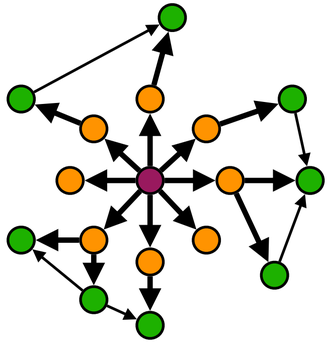




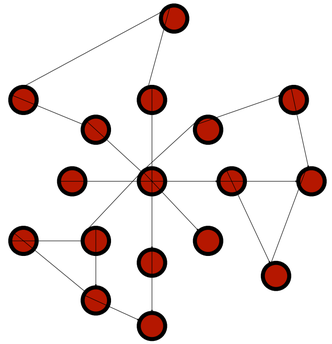
Algorithm Language



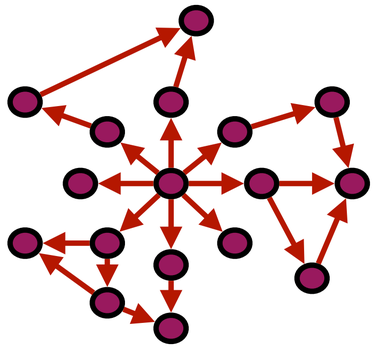
edges.apply(func)



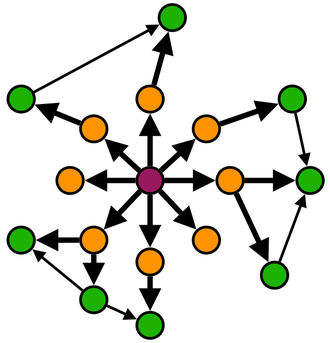
edges.from(vertexset)
.to(vertexset)
.srcFilter(filtF)
.dstFilter(filtF)
.apply(func)



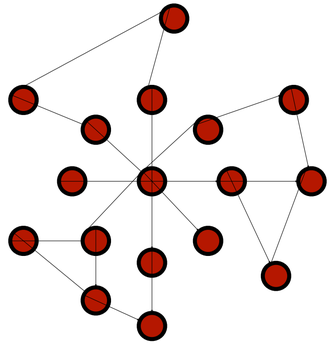
Algorithm Language



edges.apply(func)

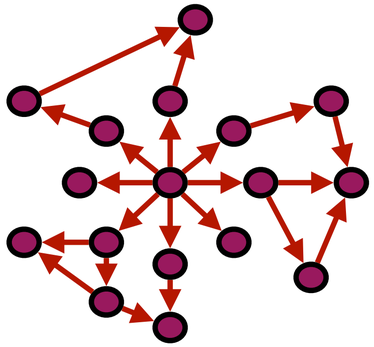


edges.from(vertexset)
.to(vertexset)
.srcFilter(filtF)
.dstFilter(filtF)
.apply(func)



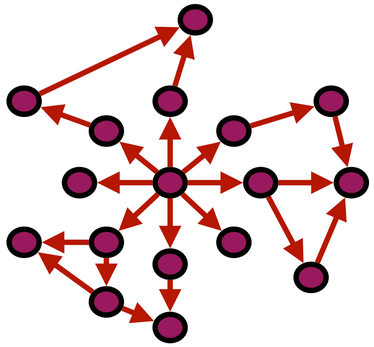
vertices.apply(func)

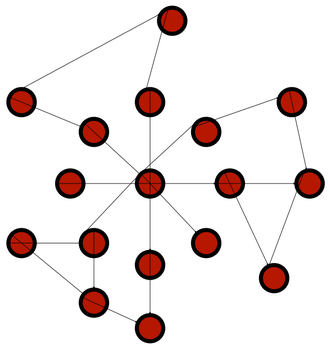
PageRank Example



```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end
```

PageRank Example

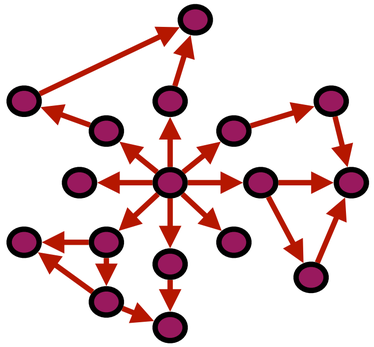


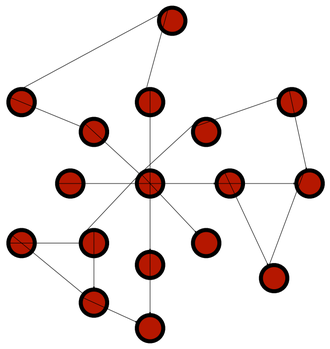


```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end
```

PageRank Example



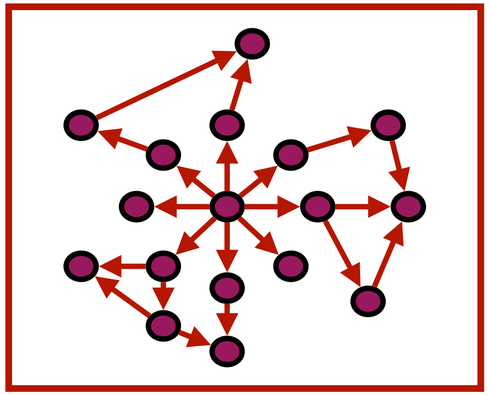


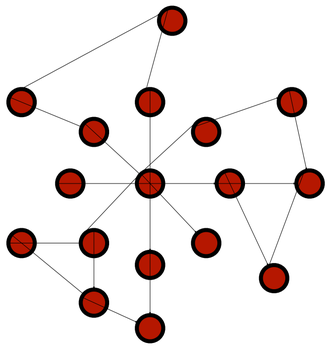
```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end


func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
    end
 end
```

PageRank Example



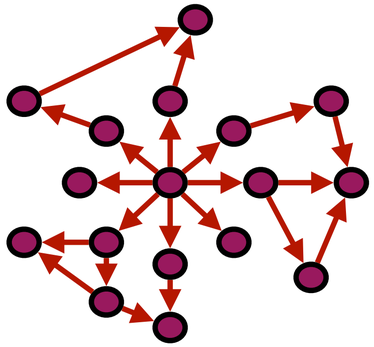


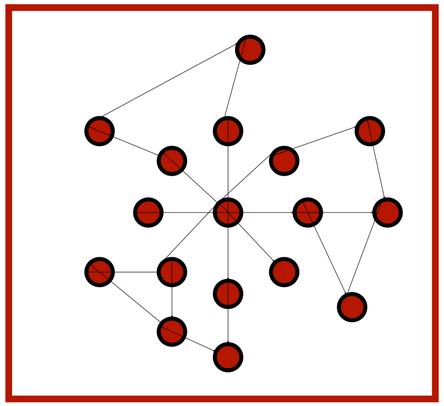
```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
    end
 end
```

PageRank Example





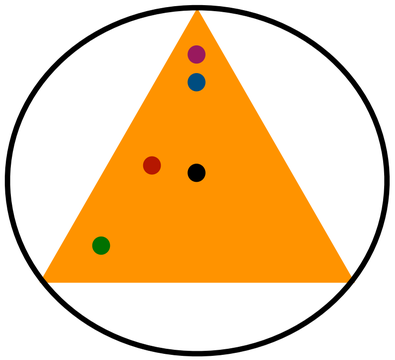
```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end
```

```
func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end
```

```
func main()
    for i in 1:max_iter
      #s1# edges.apply(updateEdge);
      vertices.apply(updateVertex);
    end
end
```

GraphIt DSL

Algorithm Representation
(Algorithm Language)



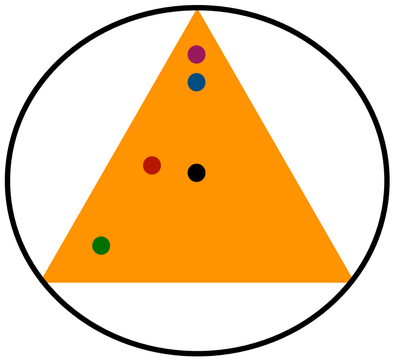
Optimization Representation

- **Scheduling Language**
- **Schedule Representation (e.g. Graph Iteration Space)**

Autotuner

GraphIt DSL

Algorithm Representation
(Algorithm Language)



Optimization Representation

- **Scheduling Language**
- **Schedule Representation (e.g. Graph Iteration Space)**

Autotuner

Scheduling Language

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end


func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
    end
 end
```

Scheduling Language

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
     end
 end
```

Schedule 1

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
     end
 end
```

Scheduling Functions

```
schedule:
    program->configApplyDirection("s1", "SparsePush");
```

Schedule 1

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
     end
 end
```

Scheduling Functions

```
schedule:
    program->configApplyDirection("s1", "SparsePush");
```

Schedule 1

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
     end
 end
```

Pseudo Generated Code

```
double * new_rank = new double[num_verts];
double * old_rank = new double[num_verts];
int * out_degree = new int[num_verts];

…

for (NodeID src : vertices) {
  for(NodeID dst : G.getOutNgh(src)){
     new_rank[dst]  +=  old_rank[src] / out_degree[src];
   }
}

….
```

Scheduling Functions

```
schedule:
    program->configApplyDirection("s1", "SparsePush");
```

Schedule 2

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
     end
 end
```

Pseudo Generated Code

```
double * new_rank = new double[num_verts];
double * old_rank = new double[num_verts];
int * out_degree = new int[num_verts];

…

parallel_for (NodeID src : vertices) {
  for(NodeID dst : G.getOutNgh(src)){
      atomic_add (new_rank[dst],
                        old_rank[src] / out_degree[src] );
   }
}

….
```

Scheduling Functions

```
schedule:
    program->configApplyDirection("s1", "SparsePush");
    program->configApplyParallelization("s1", "dynamic-vertex-parallel");
```

Schedule 3

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
     end
 end
```

Pseudo Generated Code

```
double * new_rank = new double[num_verts];
double * old_rank = new double[num_verts];
int * out_degree = new int[num_verts];

…

parallel_for (NodeID dst : vertices) {
  for(NodeID src : G.getInNgh(dst)){
     new_rank[dst]  += old_rank[src] / out_degree[src];
   }
}

….
```

Scheduling Functions

```
schedule:
    program->configApplyDirection("s1", "DensePull");
    program->configApplyParallelization("s1", "dynamic-vertex-parallel");
```

Schedule 4

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
     end
 end
```

Pseudo Generated Code

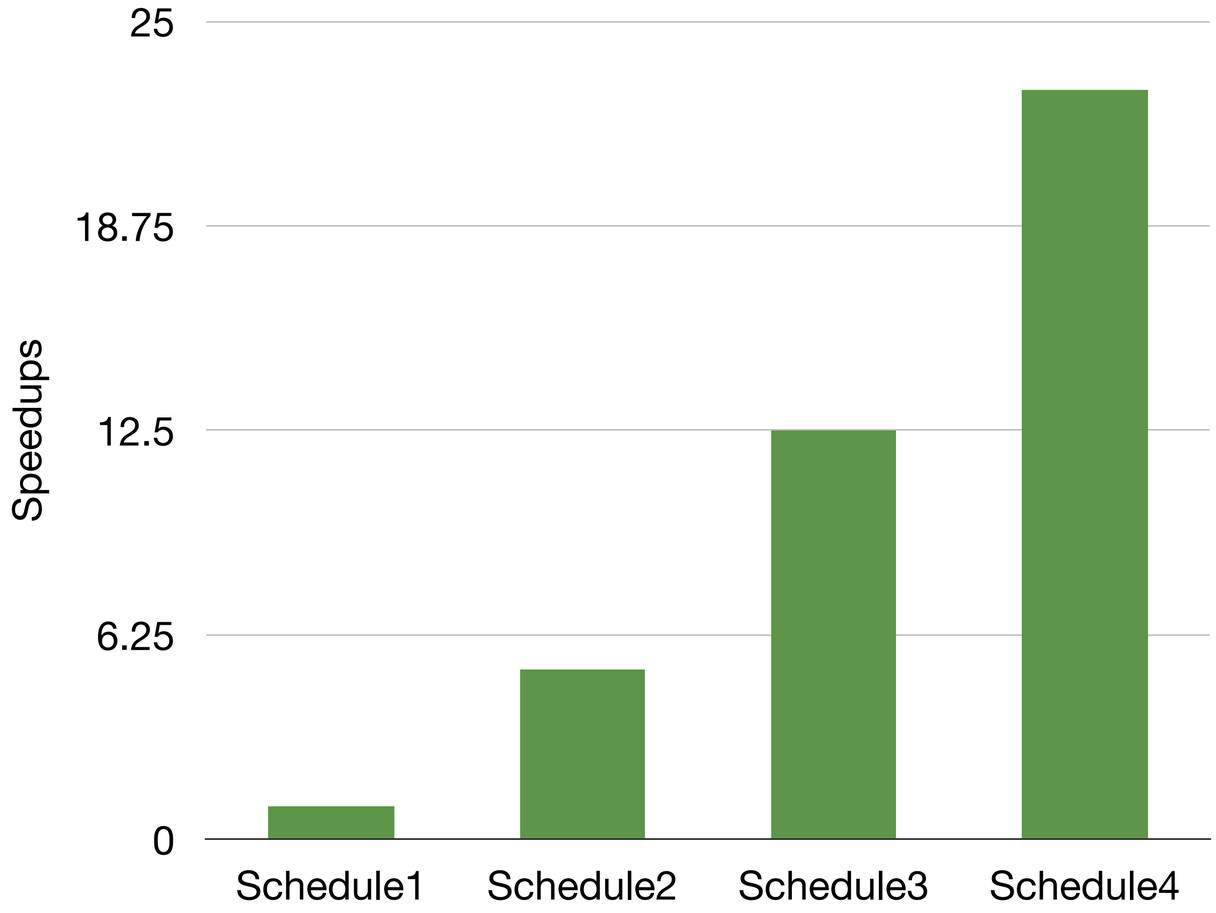
```
double * new_rank = new double[num_verts];
double * old_rank = new double[num_verts];
int * out_degree = new int[num_verts];

…
for (Subgraph sg : G.subgraphs) {
  parallel_for (NodeID dst : vertices) {
    for(NodeID src : G.getInNgh(dst)){
     new_rank[dst]  += old_rank[src] / out_degree[src];
    }
  }
}
….
```

Scheduling Functions

```
schedule:
    program->configApplyDirection("s1", "DensePull");
    program->configApplyParallelization("s1", "dynamic-vertex-parallel");
    program->configApplyNumSSG("s1", "fixed-vertex-count", 10);
```

Speedups of Schedules



Twitter graph with 41M vertices and 1.47B edges
Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores

Many More Optimizations

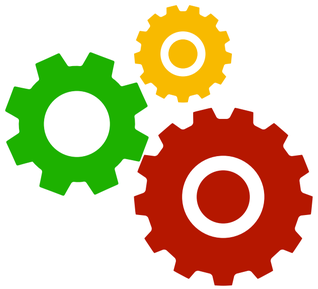
- Direction optimizations (configApplyDirection),
  - SparsePush, DensePush, DensePull, DensePull-SparsePush, DensePush-SparsePush
- Parallelization strategies (configApplyParallelization)
  - serial, dynamic-vertex-parallel, static-vertex-parallel, edge-aware-dynamic-vertex-parallel, edge-parallel
- Cache (configApplyNumSSG)
  - fixed-vertex-count, edge-aware-vertex-count
- NUMA (configApplyNUMA)
  - serial, static-parallel, dynamic-parallel
- AoS, SoA (fuseFields)
- Vertexset data layout (configApplyDenseVertexSet)
  - bitvector, boolean array

GraphIt DSL

Algorithm
Representation
(Algorithm Language)

Optimization Representation

- Scheduling Language
- Schedule Representation
(e.g. Graph Iteration Space)



Autotuner

Schedule 4

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:11
         #s1# edges.apply(updateEdge);
         vertices.apply(updateVertex);
     end
 end
```

Scheduling Functions

```
schedule:
    program->configApplyDirection("s1", "DensePull");
    program->configApplyParallelization("s1", "dynamic-vertex-parallel");
    program->configApplyNumSSG("s1", "fixed-vertex-count", 10);
```

Schedule 4

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:11
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
     end
 end
```

Finding the best schedule can be hard for non-experts.

Scheduling Functions

```
schedule:
    program->configApplyDirection("s1", "DensePull");
    program->configApplyParallelization("s1", "dynamic-vertex-parallel");
    program->configApplyNumSSG("s1", "fixed-vertex-count", 10);
```

Goal

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:11
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
     end
 end
```

Ideally, the user only need to write the algorithm

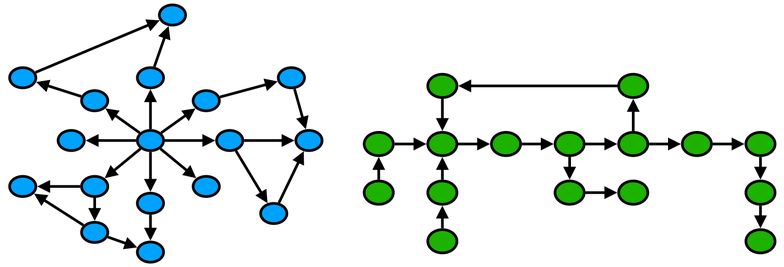
Autotuner

Algorithm Specification

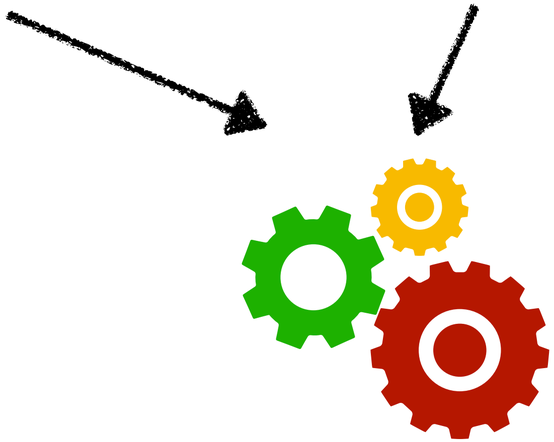
```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:11
         #s1# edges.apply(updateEdge);
         vertices.apply(updateVertex);
     end
 end
```



Input Graphs



Autotuner

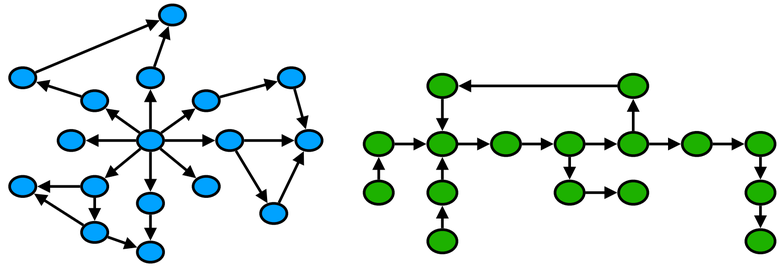
Autotuner

Algorithm Specification

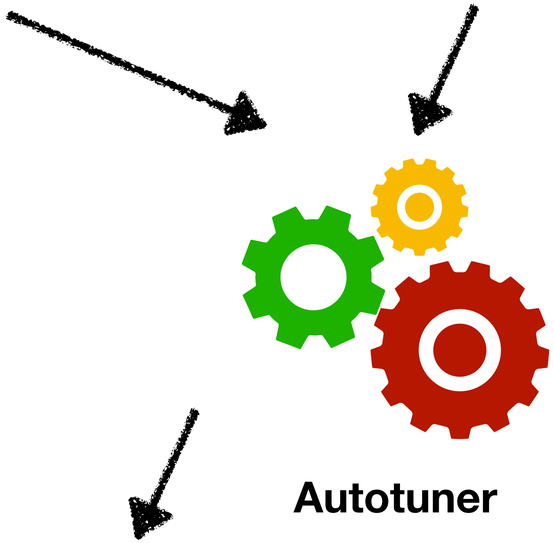
```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
    new_rank[v] = beta_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:11
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
     end
 end
```



Input Graphs

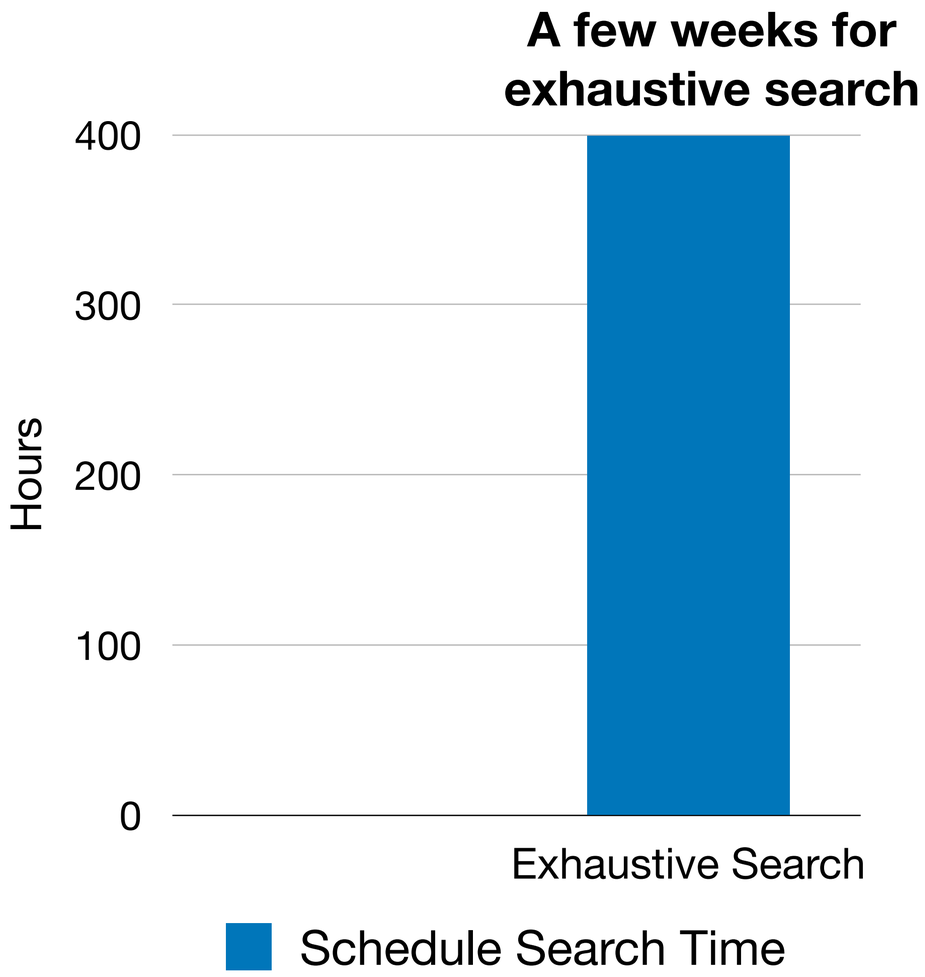


Autotuner

Scheduling Functions

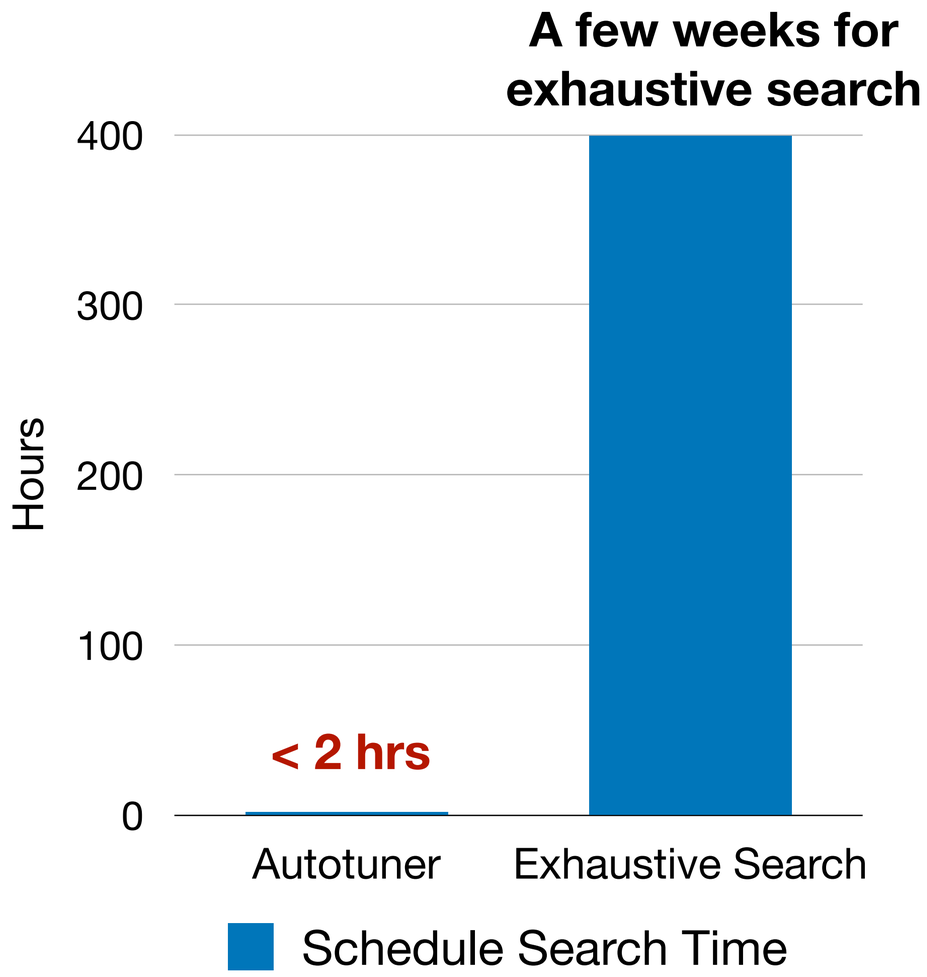
```
schedule:
    program->configApplyDirection("s1", "DensePull");
    program->configApplyParallelization("s1", "dynamic-vertex-parallel");
    program->configApplyNumSSG("s1", "fixed-vertex-count", 10);
```

Autotuner


A few weeks for exhaustive search
400
300
200
100
0
Hours
Exhaustive Search
Schedule Search Time

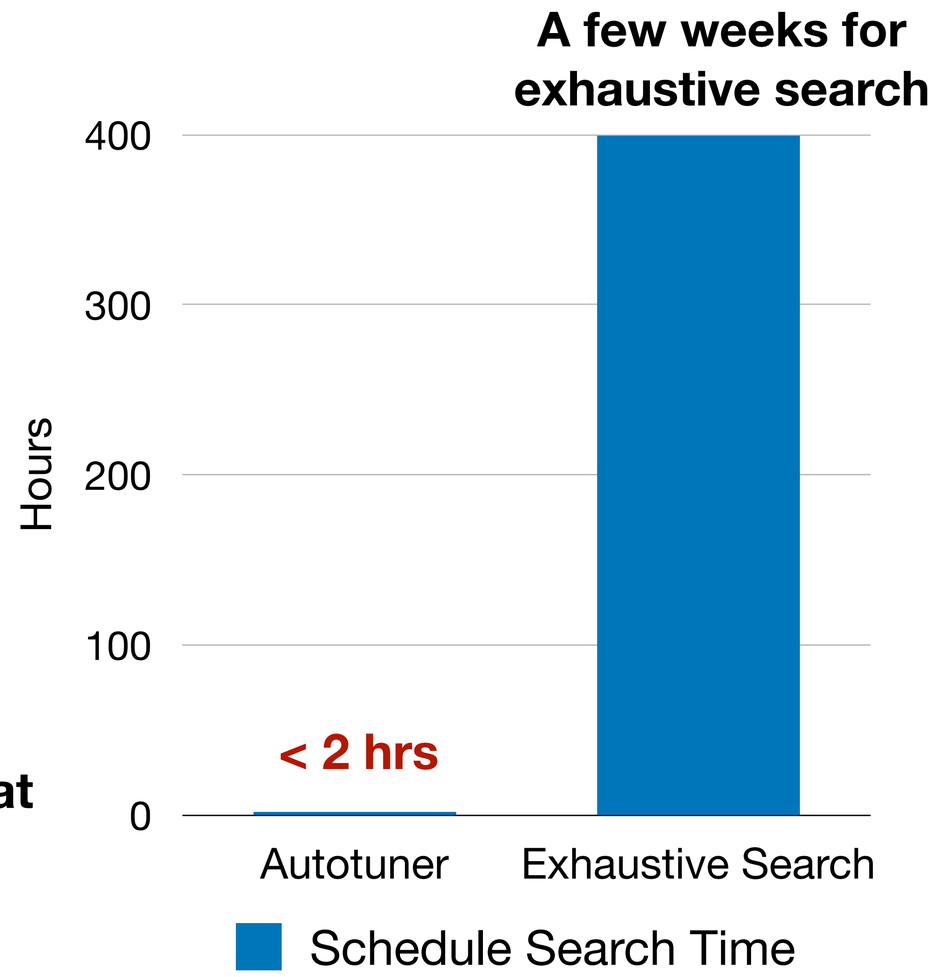
Autotuner

Uses an ensemble of search methods. Build on top of OpenTuner [PACT14]



Autotuner

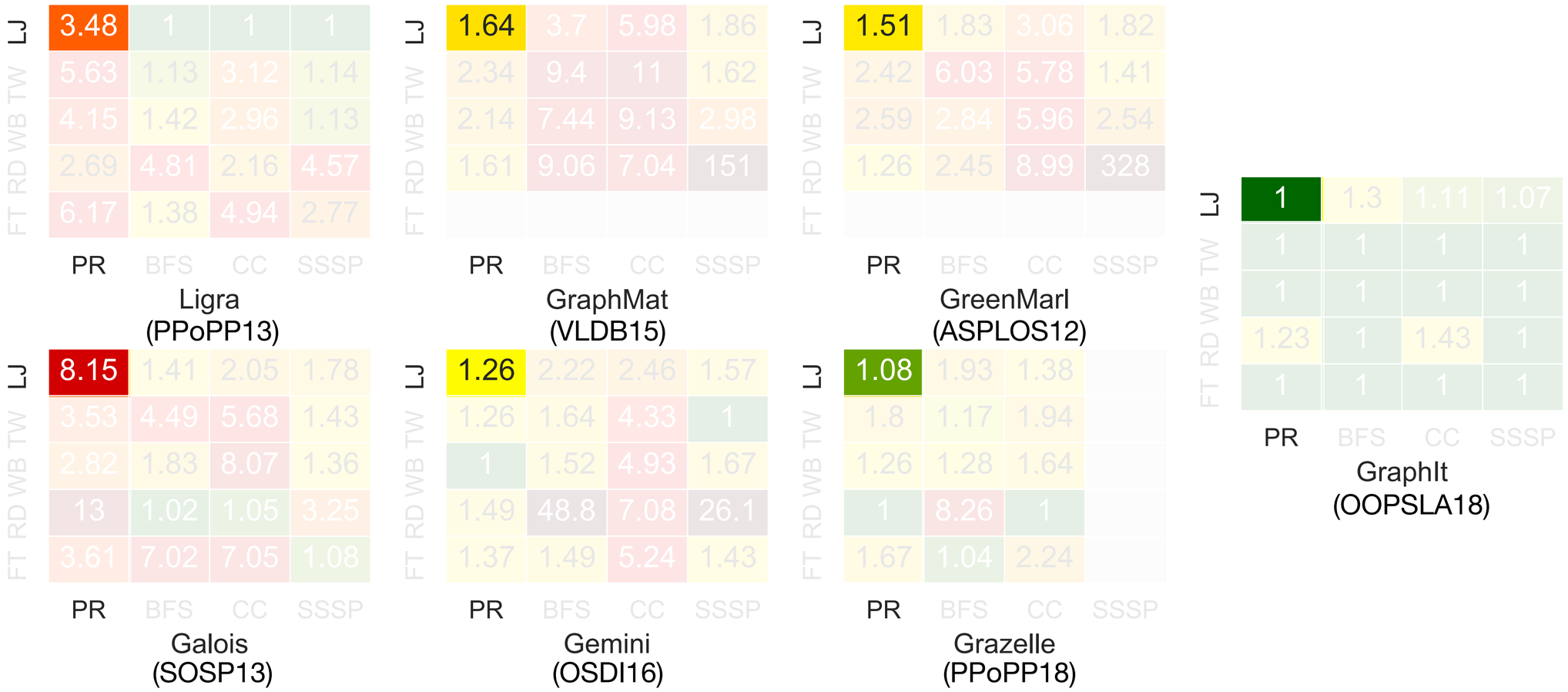
Finds a few schedules that outperform hand-tuned schedules



Outline

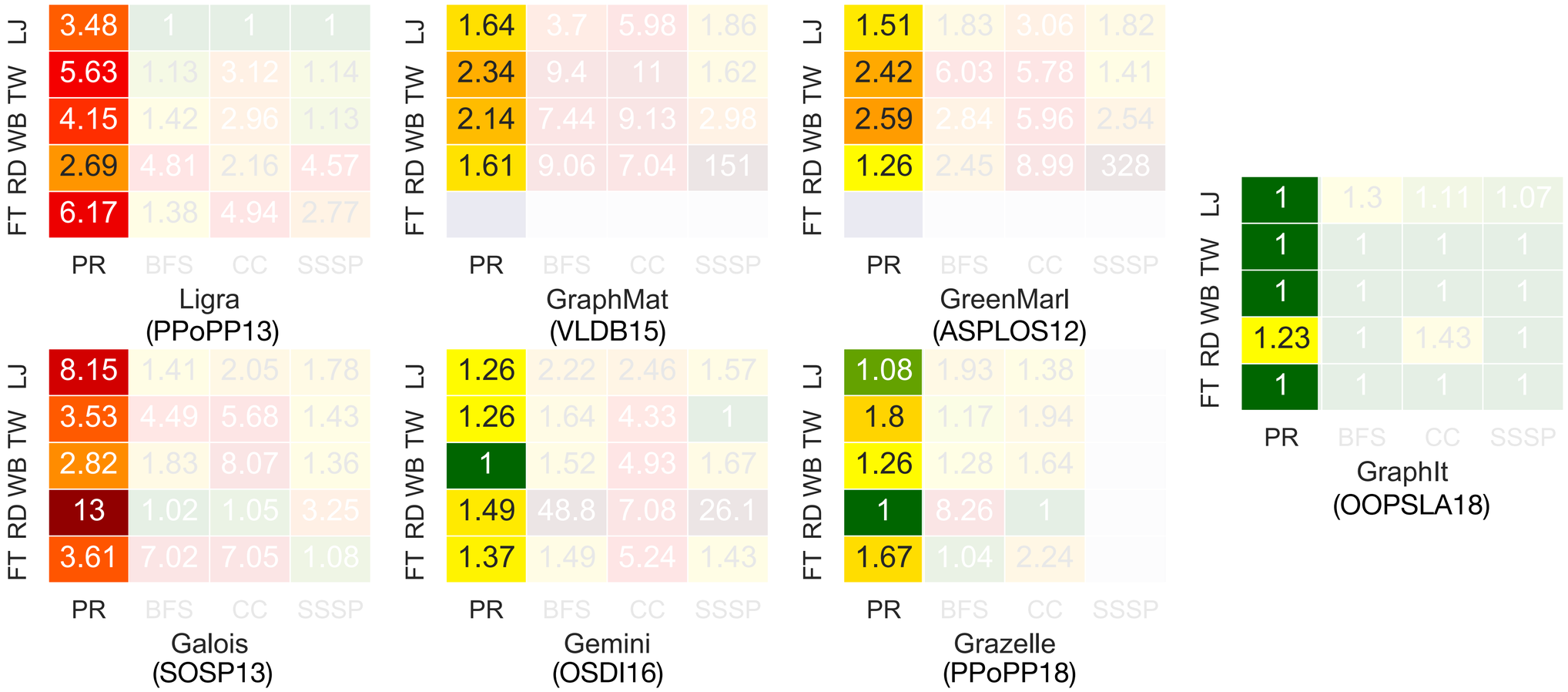
- Graph Applications Overview
- Optimization Tradeoff Space
- GraphIt DSL
- Evaluation

State of the Art and GraphIt



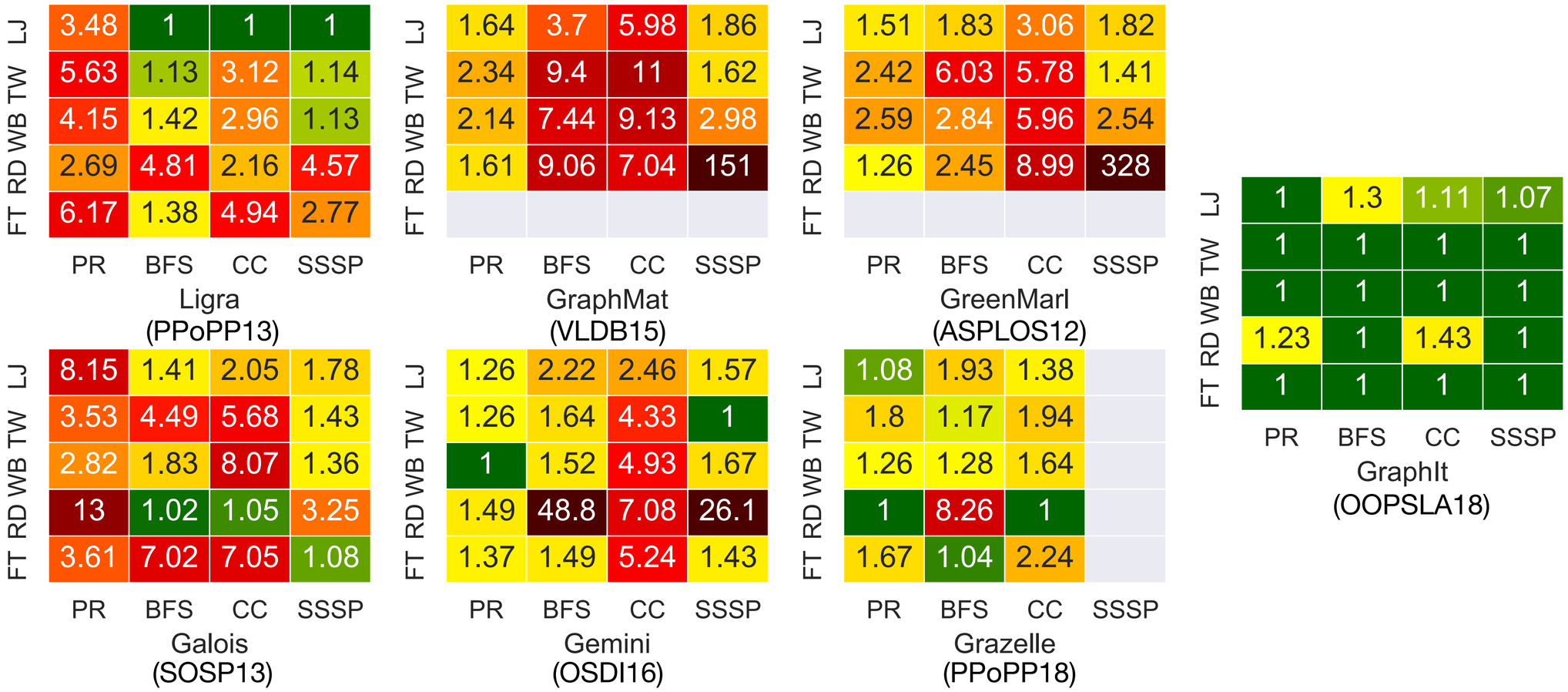
Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores

State of the Art and GraphIt



Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores

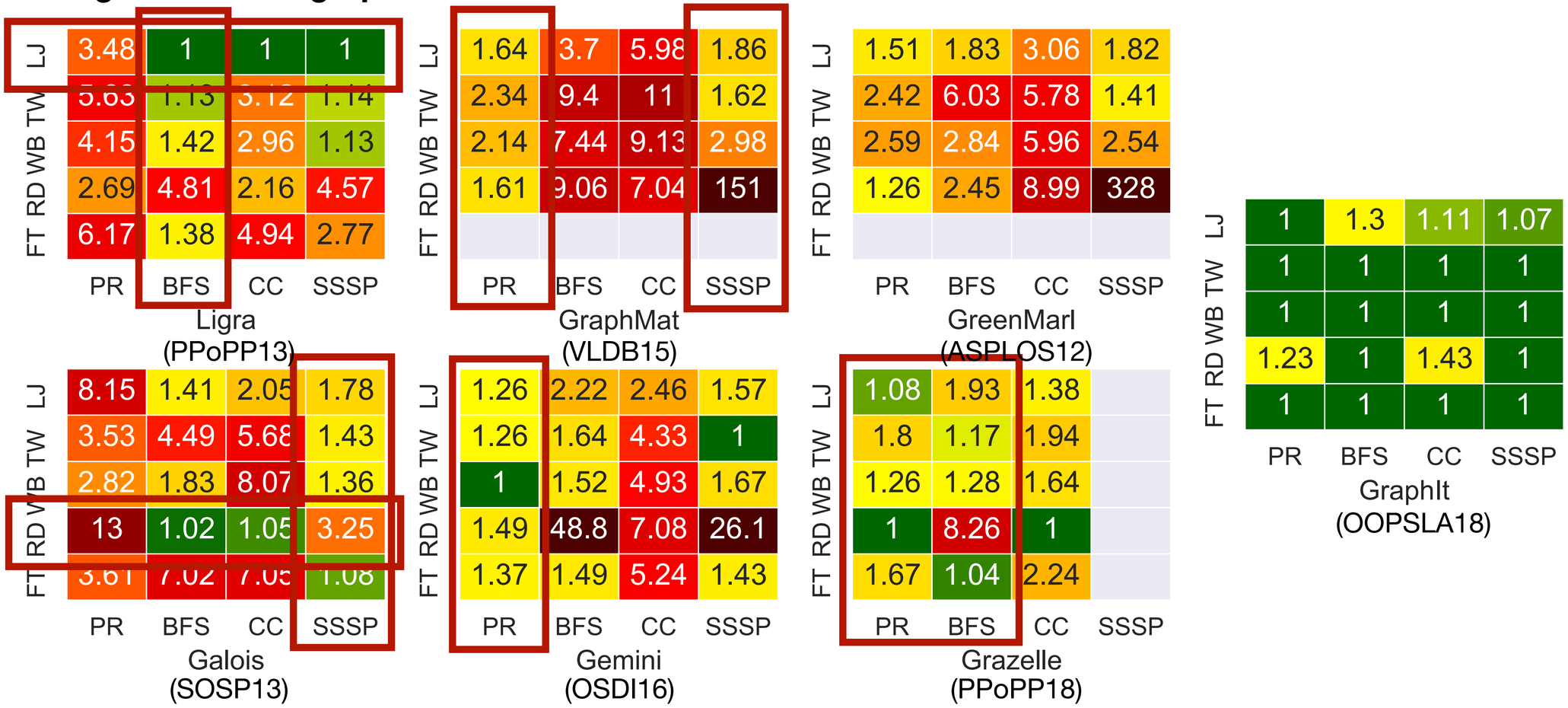
State of the Art and GraphIt



Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores

State of the Art and GraphIt

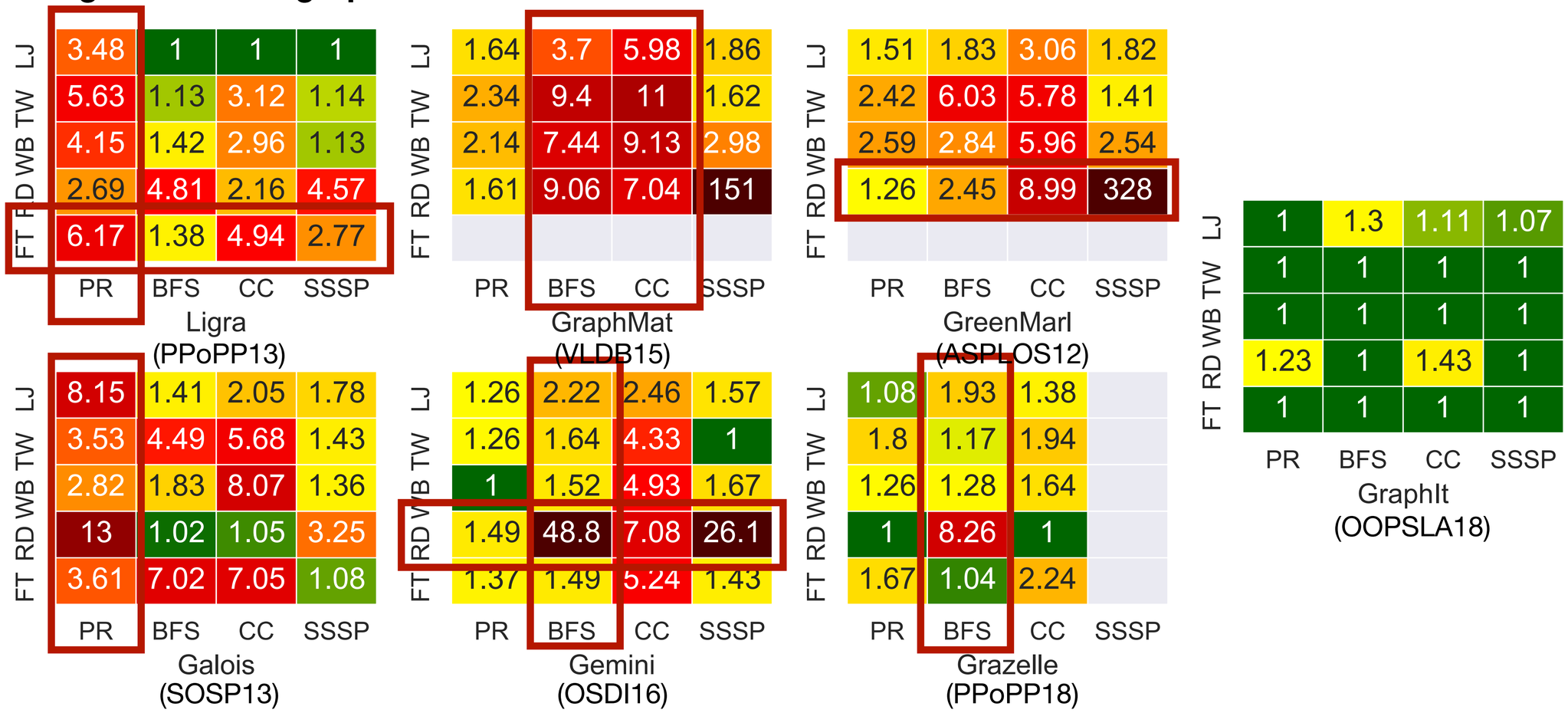
Most frameworks are good at certain algorithms and graphs



Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores

State of the Art and GraphIt

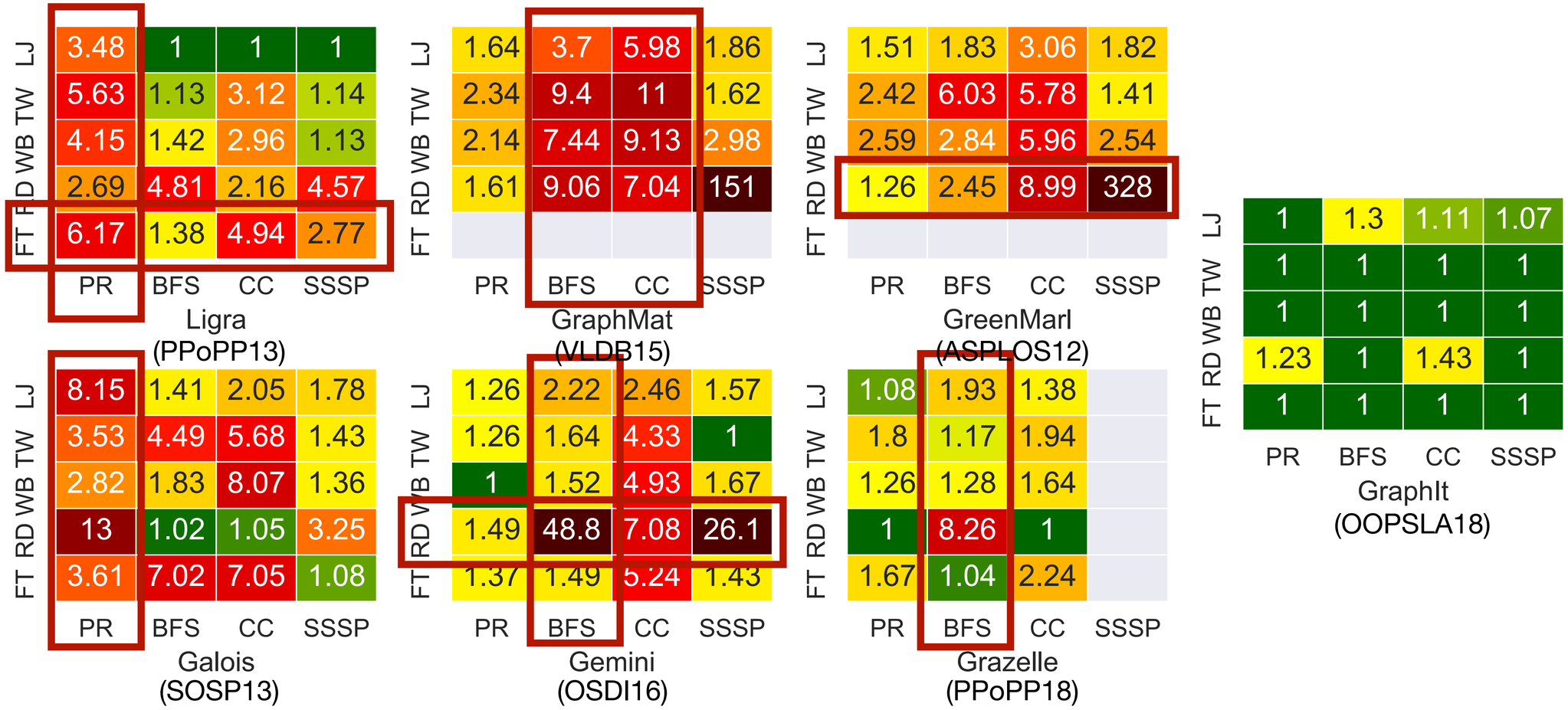
Most frameworks are bad at certain algorithms and graphs



Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores

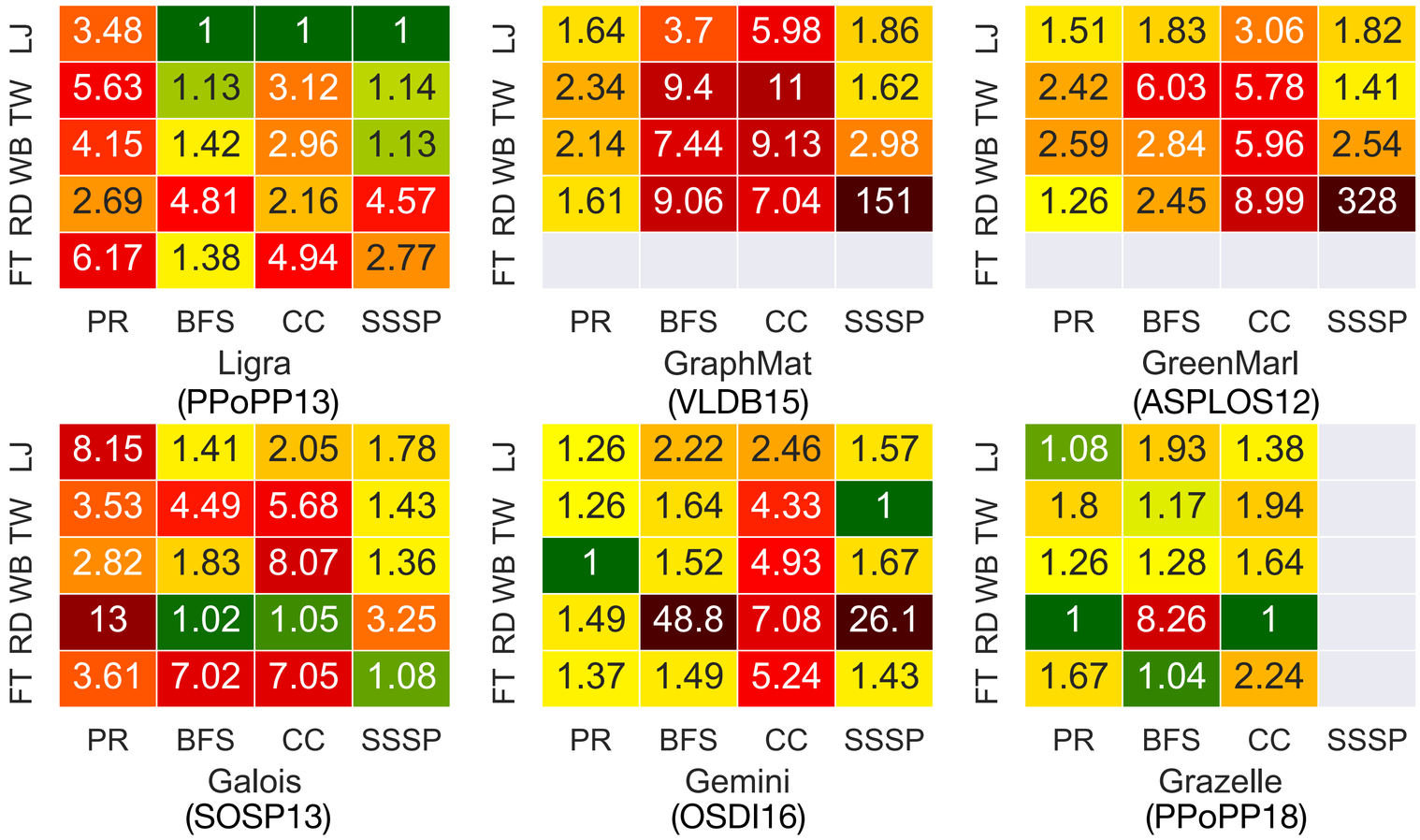
State of the Art and GraphIt

Previous work support a subset of optimizations

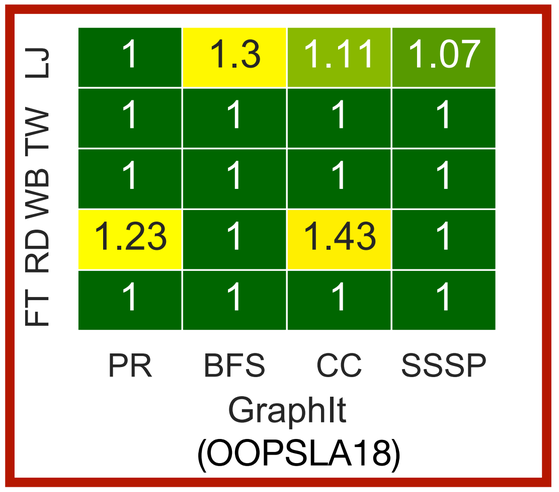


Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores

Consistent High-Performance

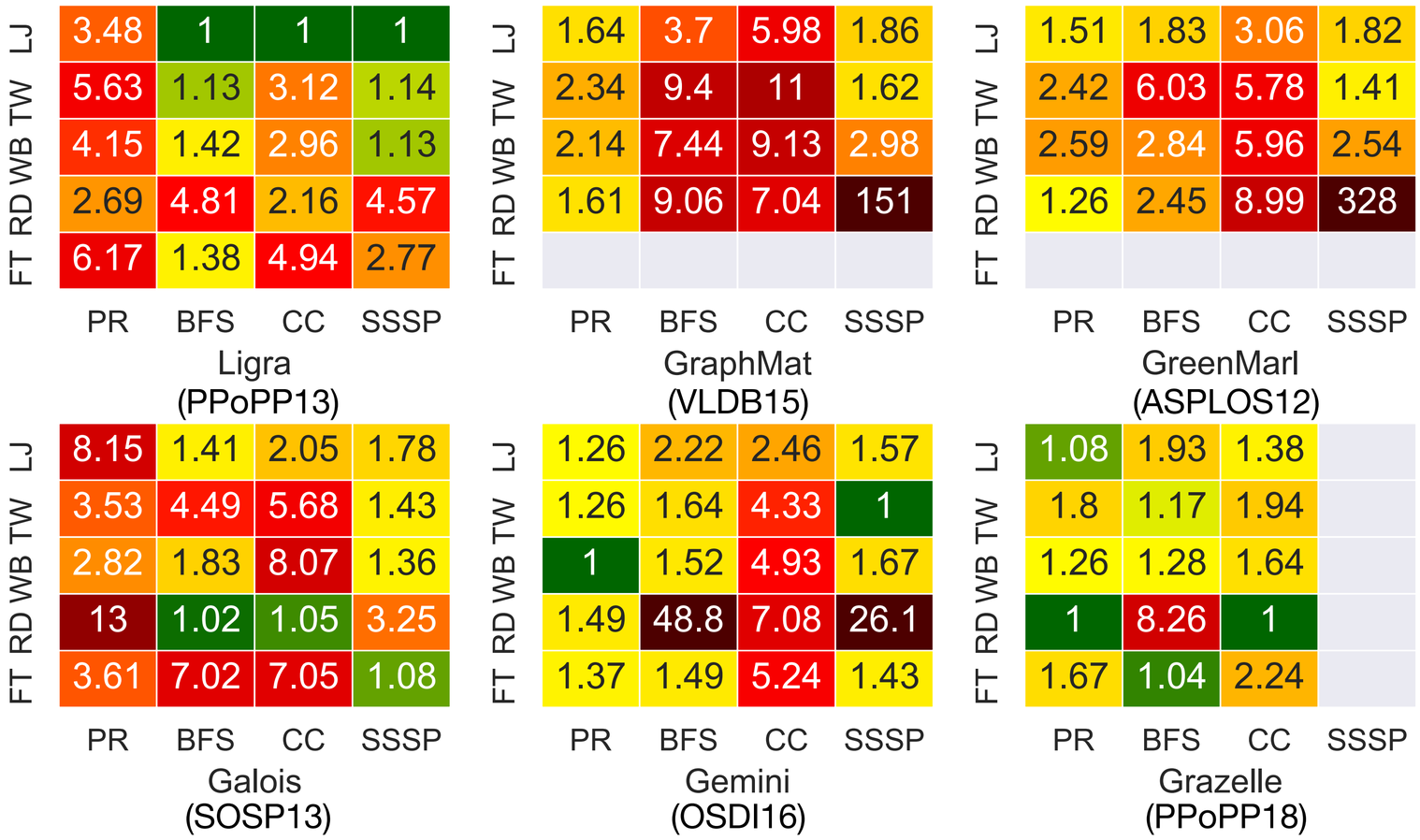


Good across different algorithms and graphs

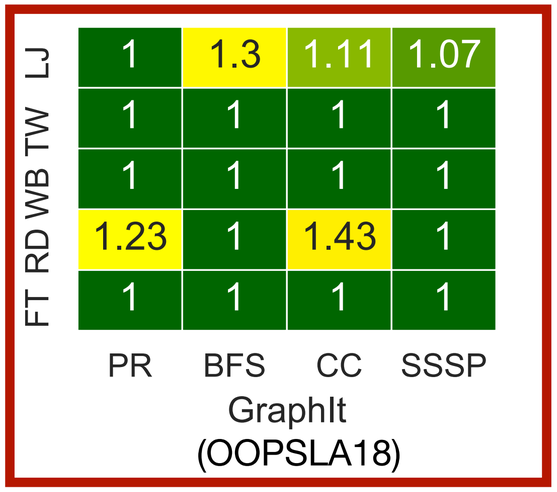


Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores

Speedup over State of the Art

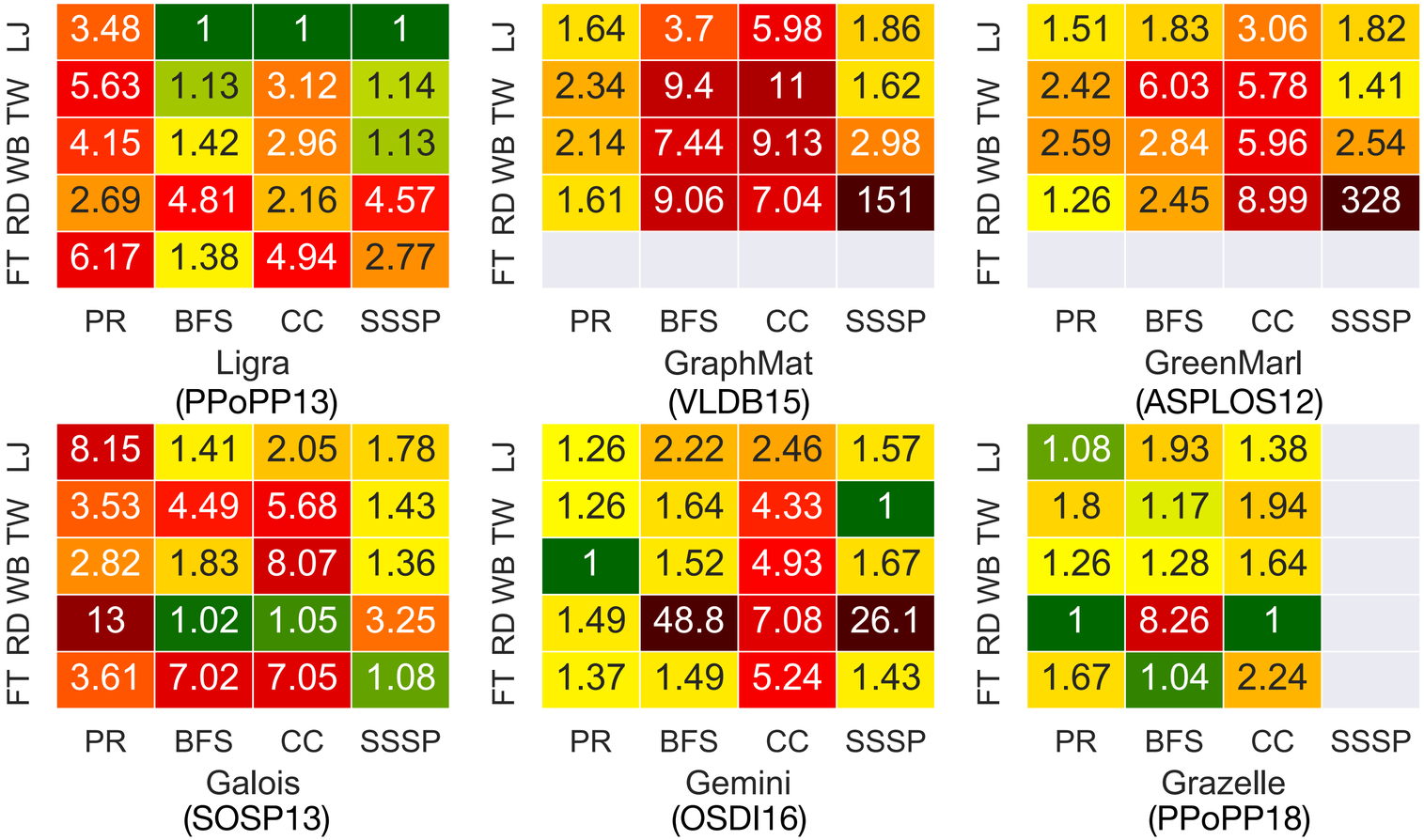


Finds previously unexplored combinations of optimizations

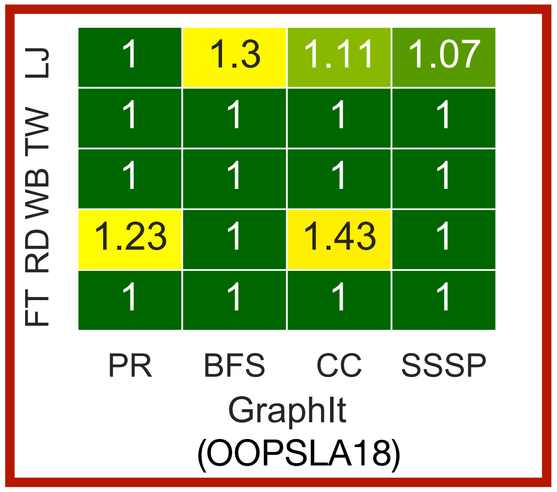


Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores

Ease-of-Use

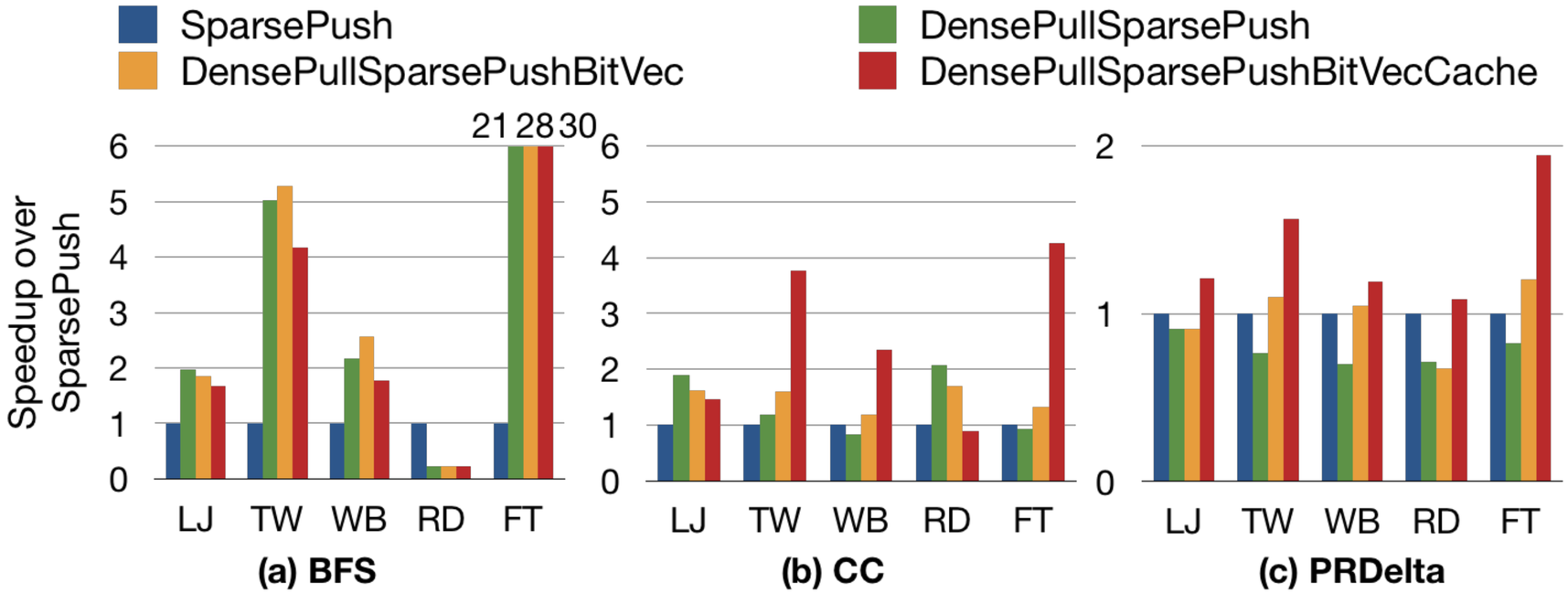


Reduces the lines of code by up to an order of magnitude compared to the next fastest framework

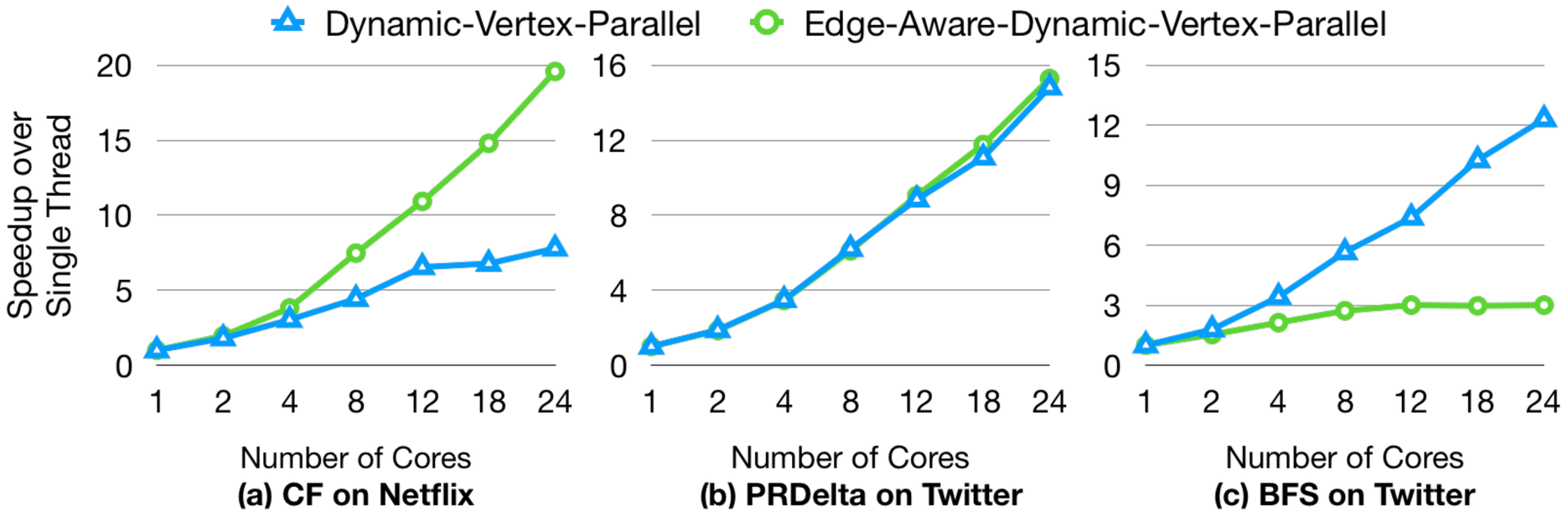


Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores

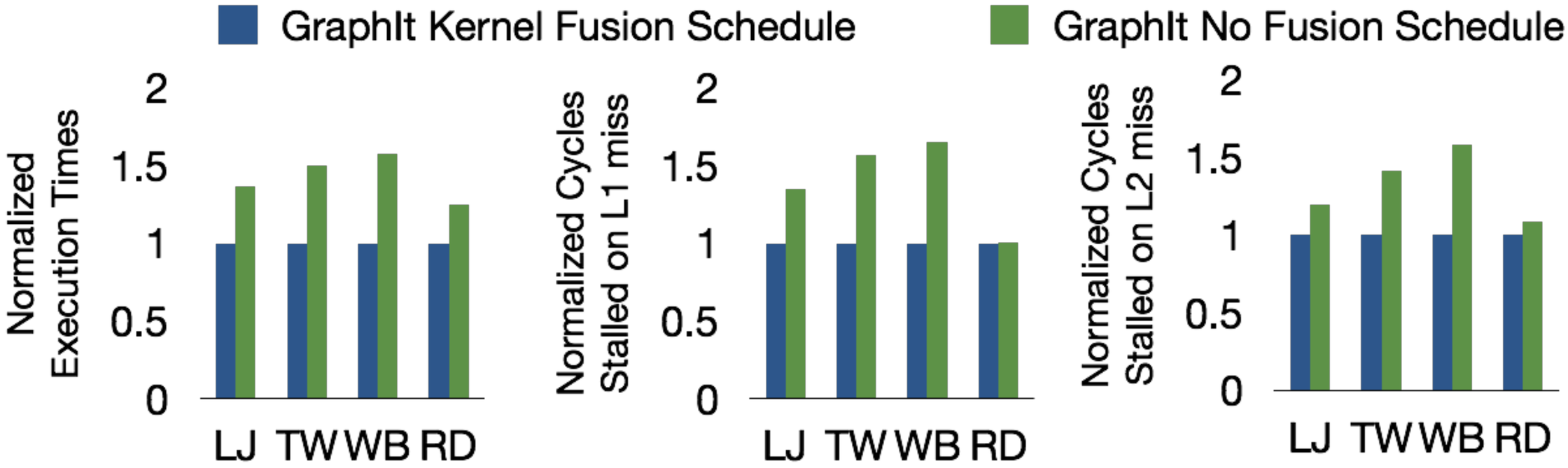
Different Schedules



Different Schedules



Kernel Fusion



Related Work

Graph optimizations:

Ligra [PPoPP13], Galois [SOSP13], Propagation Blocking [IPDPS17], CSR Segmentation [BigData 17], Grazelle [PPoPP18] …

- We focus on composing the optimizations

Graph DSLs:

GreenMarl [ASPLOS12], EmptyHeaded [SIGMOD16], Elixir [OOPSLA12], Gluon [PLDI18], Abelian [EuroPar18]…

- We expose extensive performance tuning capabilities

Summary

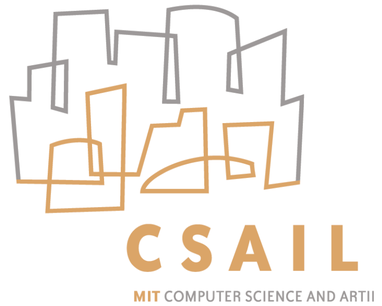
- Performance of graph programs depends highly on data, algorithm, and hardware

- GraphIt decouples algorithm from optimization to achieve high-performance and programmability

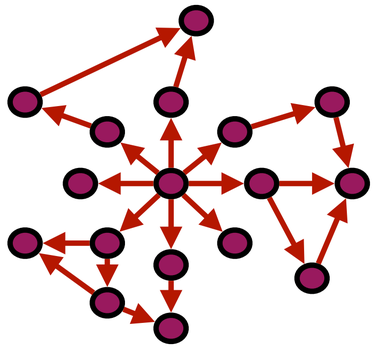
  Open source (**graphit-lang.org**)

Optimizing Ordered Graph Algorithms with GraphIt

Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, Julian Shun

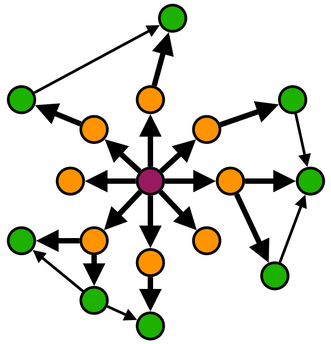


Topology-Driven

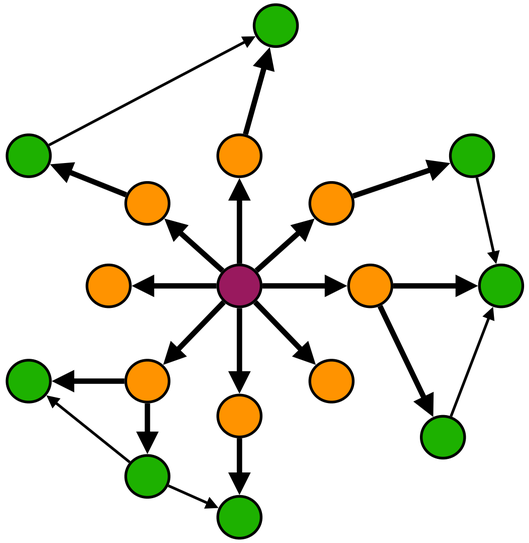


Algorithms

Data-Driven

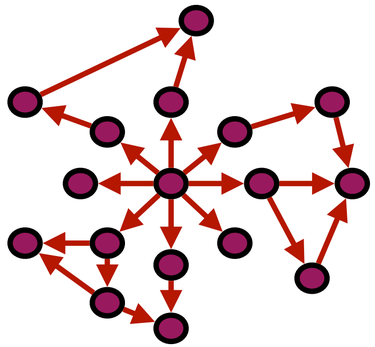


Unordered



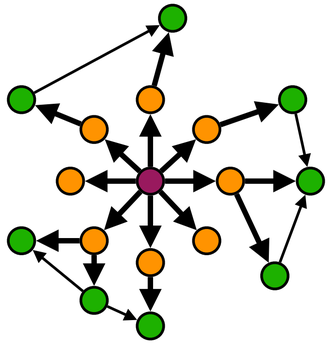
Active Vertices can be processed in parallel in arbitrary order

Topology-Driven

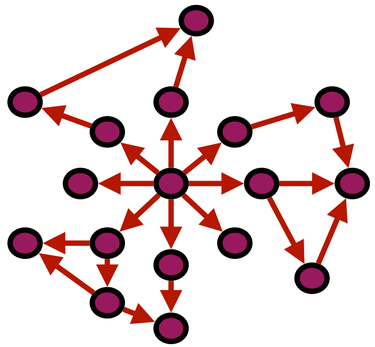


Algorithms

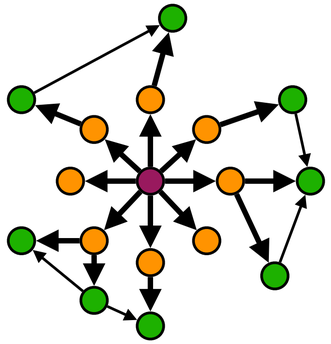
Data-Driven



Topology-Driven

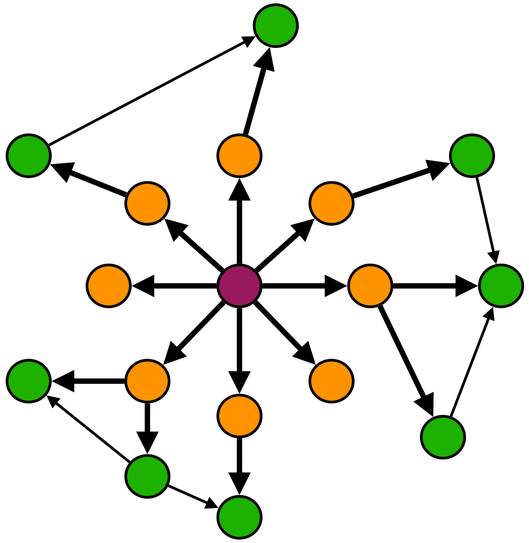


Data-Driven



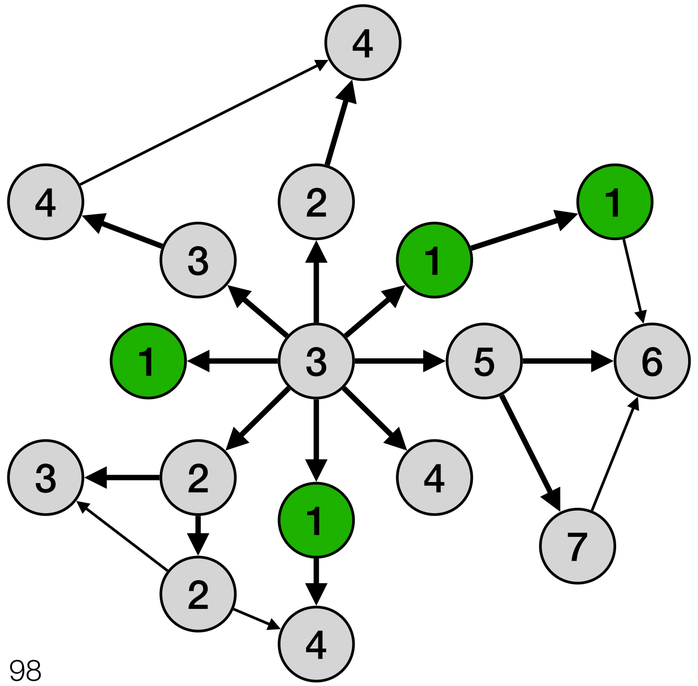
Algorithms

Unordered



Active Vertices can be processed in parallel in arbitrary order

Ordered



Vertices are processed according to priorities

Priorities can change dynamically

Vertices of the same priority are processed in parallel

Topology-Driven

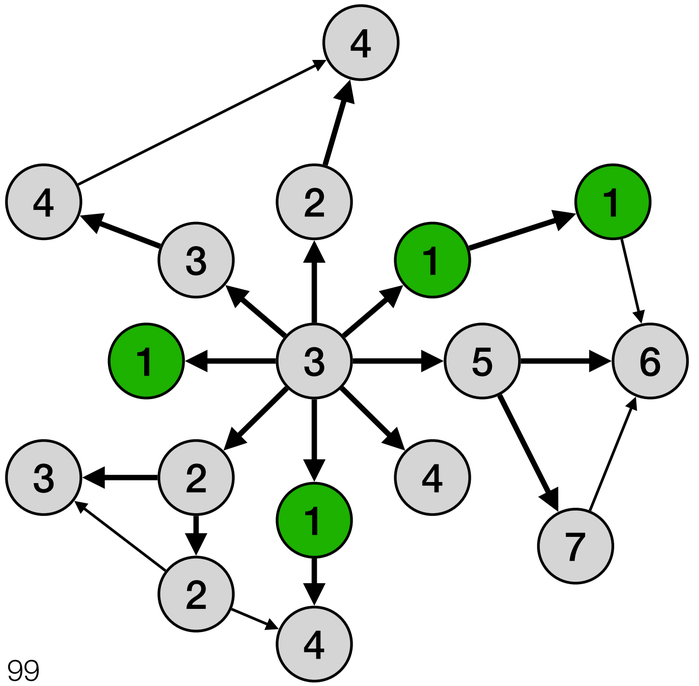
Algorithms

Data-Driven

Unordered

Active Vertices can be processed in parallel in arbitrary order

Ordered



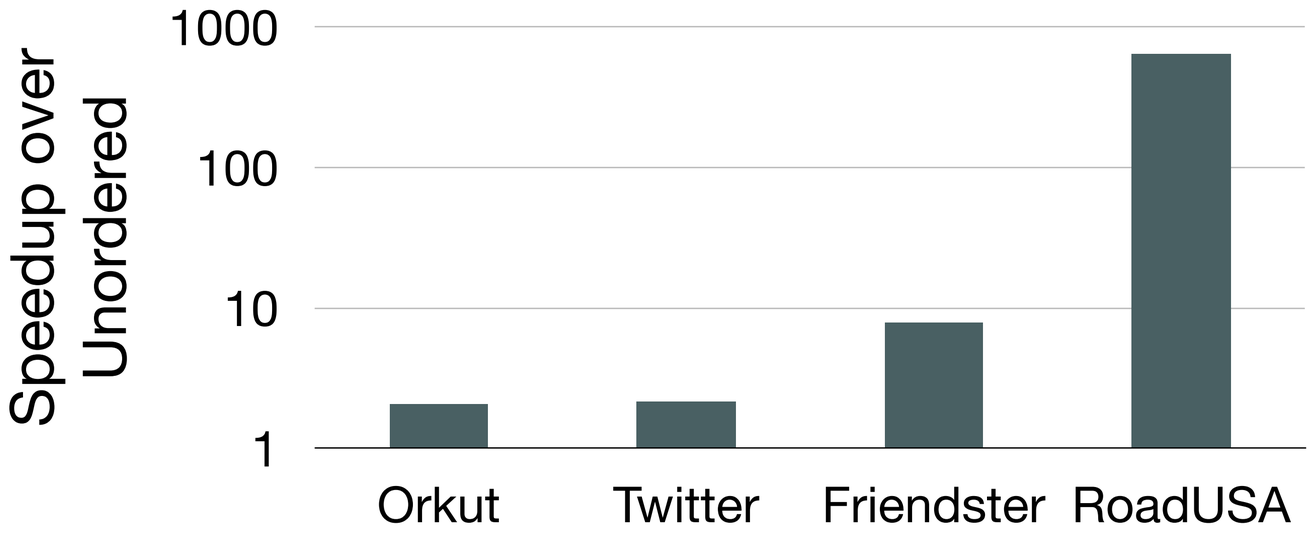
Vertices are processed according to priorities

Priorities can change dynamically

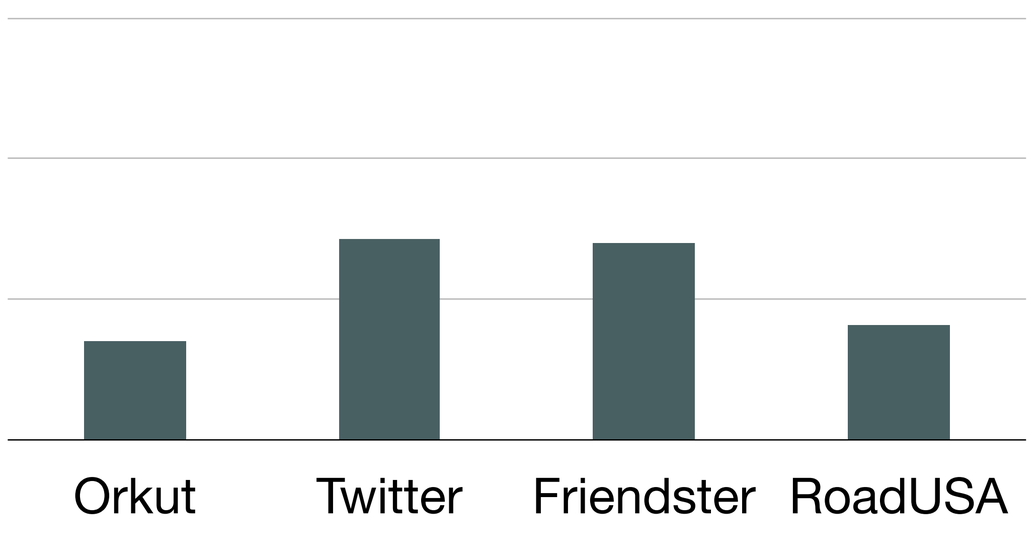
Vertices of the same priority are processed in parallel

Ordered vs Unordered

Ordered algorithms can often achieve 2x to 640x speedup over unordered counterparts



Single Source Shortest Paths



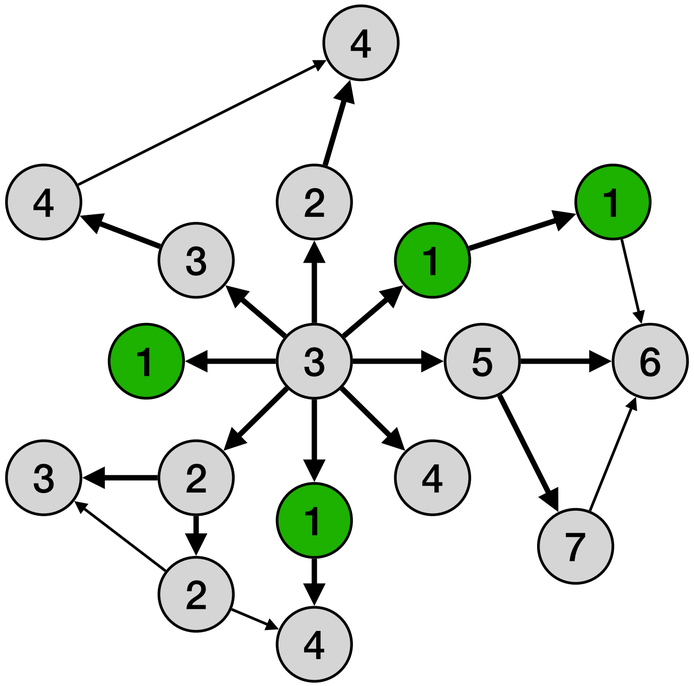
KCore

Speedup of Ordered vs Unordered on a 24-core CPU with GraphIt

Priority-based Extensions to GraphIt

- **Decouple Algorithm from Optimization for Ordered Graph Algorithms**
  - Priority-based Algorithm Language Operators
  - Optimizations for Ordered Parallelism

Priority-Based Extensions



- PriorityQueue
  - dequeueReadySet()
  - getCurrentPriority()
  - finished(), finishedNode()
  - updatePriorityMin, updatePrioritySum, ..

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end



func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
  end
end
```

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end



func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
  end
end
```

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end



func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
  end
end
```

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end



func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
 end
end
```

Delta-Stepping

Hides Physical Implementation for PriorityQueue

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end



func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
 end
end
```

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end



func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
 end
end
```

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end


func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
  end
end
```

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end


func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
 end
end
```

Schedule:

program

->configApplyPriorityUpdate(“s1”, “lazy”)

->configApplyPriorityUpdateDelta(“s1”, 4)

->configApplyDirection(“s1“, SparsePush”)

->configApplyParallelization(“s1”, “dynamic-vertex-parallel”)

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end


func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
    var frontier: vertexset{Vertex} = pq.dequeueReadySet();
    #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
    delete frontier;
  end
end
```

Schedule:

program

->configApplyPriorityUpdate(“s1”, “lazy”)

->configApplyPriorityUpdateDelta(“s1”, 4)

->configApplyDirection(“s1“, SparsePush”)

->configApplyParallelization(“s1”, “dynamic-vertex-parallel”)

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end


func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
 end
end
```

Schedule:

program

->configApplyPriorityUpdate(“s1”, “lazy”)

->configApplyPriorityUpdateDelta(“s1”, 4)

->configApplyDirection(“s1“, SparsePush”)

->configApplyParallelization(“s1”, “dynamic-vertex-parallel”)

```
1  int * dist = new int[num_verts];
2  LazyPriorityQueue* pq;
3  int delta = 4;
4  WGraph* G = loadGraph(argv[1]);
5
6  //simplified snippets of the generated main function
7  …
8  dist[start_vertex] = 0;
9  pq = new LazyPriorityQueue(true, “lower”, dist, delta);
10 While (pq.finished()){
11   VertexSubset *  frontier = getNextBucket(pq);
12   uint* outEdges = setupOutputBuffer(g, frontier);
13   uint* offsets = setupOutputBufferOffsets(g, frontier);
14   parallel_for (uint s : frontier.vert_array) {
15     int j = 0;
16     uint offset = offsets[i];
17     for(WNode d : G.getOutNgh(s)){
18       bool tracking_var = false;
19       int new_dist = dist[s.v] + d.weight;
20       tracking_var = atomicWriteMin(&dist[d.v], new_dist);
21       If (tracking_var && CAS(dedup_flags[d.v],0,1)){
22         outEdges[offset + j] = d.v;
23       } else { outEdges[offset + j] = UINT_MAX; }
24       j++;
25    }}
26    VertexSubset* nextFrontier = setupFrontier(outEdges);
27    updateBuckets(nextFrontier, pq, delta);
28    …
29 }
30 …
```

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end


func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
 end
end
```

Schedule:

program

->configApplyPriorityUpdate(“s1”, “eager”)

->configApplyPriorityUpdateDelta(“s1”, 4)

->configApplyDirection(“s1“, SparsePush”)

->configApplyParallelization(“s1”, “dynamic-vertex-parallel”)

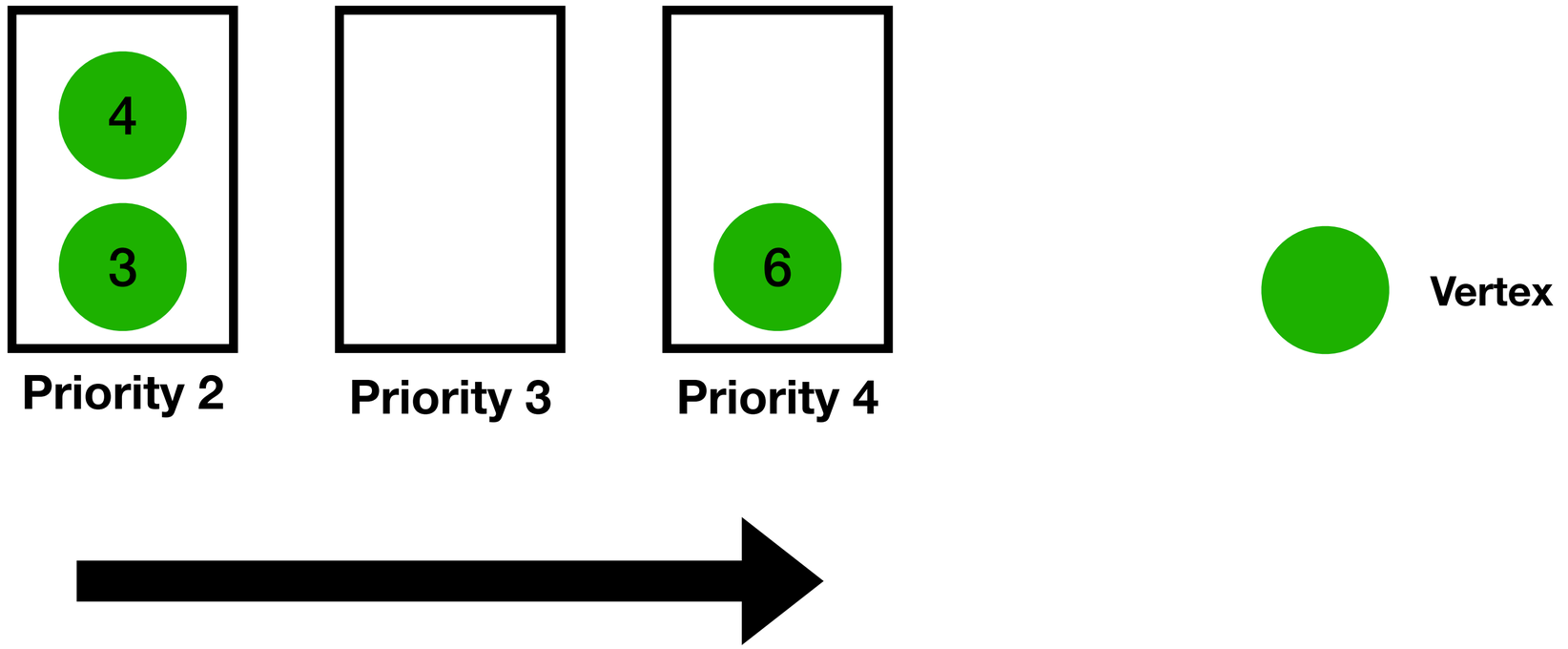
```
1  int * dist = new int[num_verts];
2  EagerPriorityQueue* pq;
3  int delta = 4;
4  WGraph* G = loadGraph(argv[1]);
5
6  //simplified snippets of the generated main function
7  …
8  dist[start_vertex] = 0;
9  frontier[0] = start_vertex;
10 pq = new EagerPriorityQueue(true, “lower”, dist, delta);
11 uint* frontier =  new uint[G.num_edges()];
12 #pragma omp parallel
13 {   vector<vector<uint> > local_bins(0);
14    while (pq.finished()) {
15      #pragma omp for nowait schedule(dynamic, 64)
16      for (size_t i = 0; i < frontier.size(); i++) {
17        uint s = frontier[i];
18        for (WNode d : G.getOutNgh(s)) {
19          int new_dist = dist[s] + d.weight;
20          bool changed = atomicWriteMin(&dist[d.v],new_dist);
21          if (changed == false) {break;}}
22          if (changed) {
23            size_t dest_bin = new_dist/delta;
24            if (dest_bin >= local_bins.size()) {
25              local_bins.resize(dest_bin+1);}
26            local_bins[dest_bin].push_back(d.v);
27          }}}// end of for frontier for loop
28    … //omitted:find next bucket
29   #pragma omp barrier
30    … //omitted:copy local buckets to global bucket
31   #pragma omp barrier } // end of parallel region
32 …
```

Outline

- Background and Motivation
- Programming Model
- **Optimizations**
- Related Work
- Evaluation

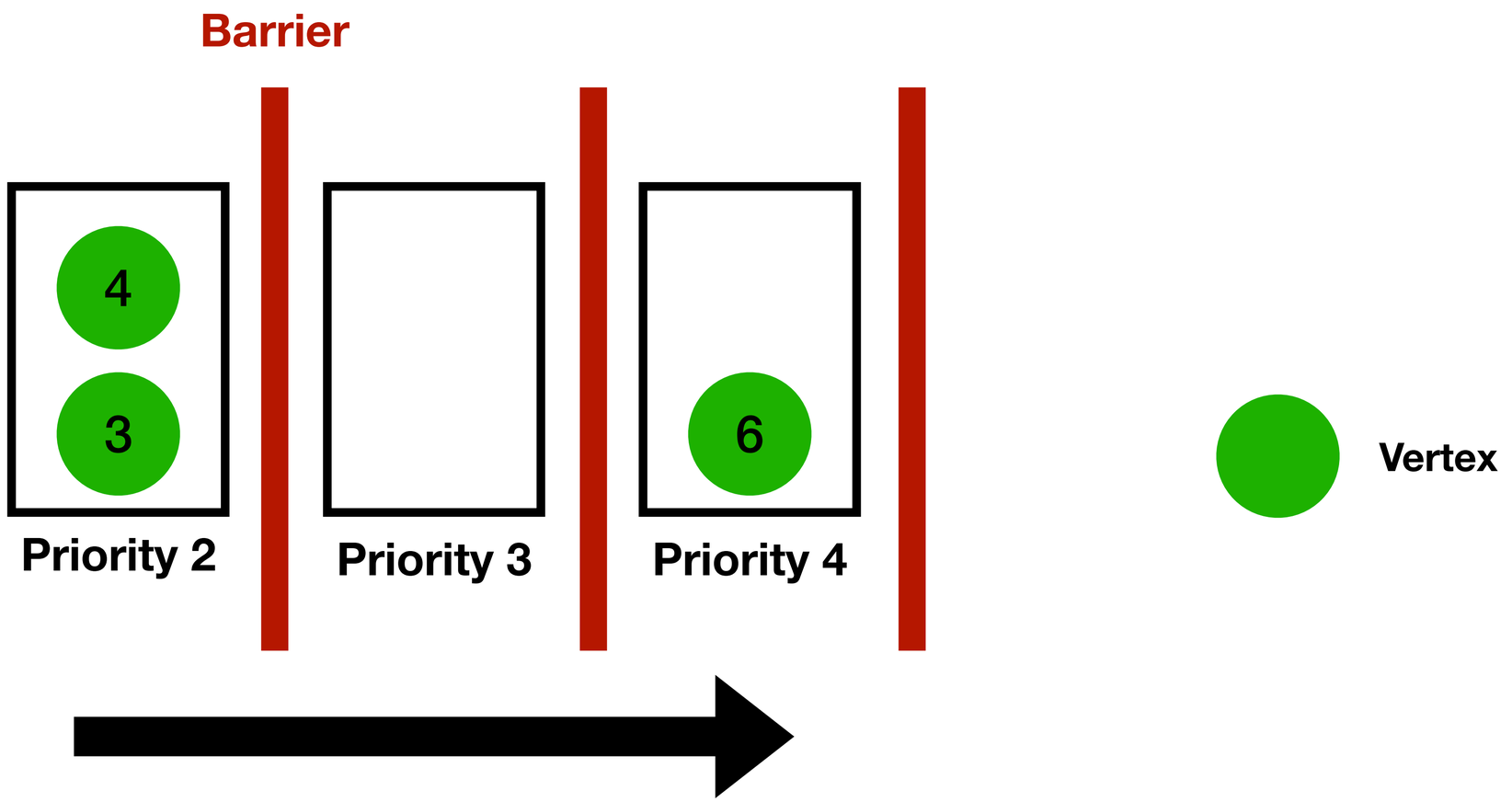
PriorityQueue with Bucketing

Vertices are stored in buckets according to their priority, and are processed in order



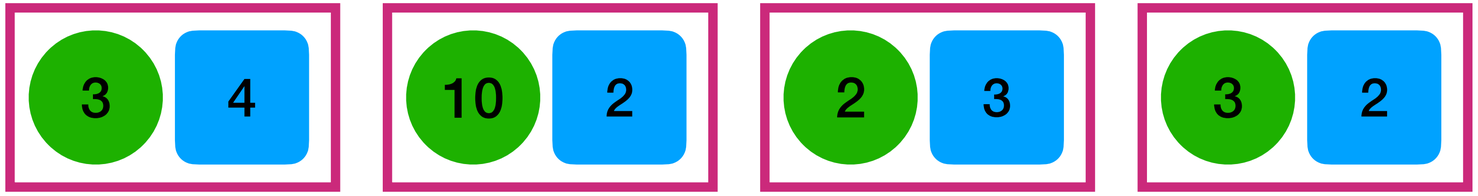
PriorityQueue with Bucketing

Global synchronization after each bucket (barrier)

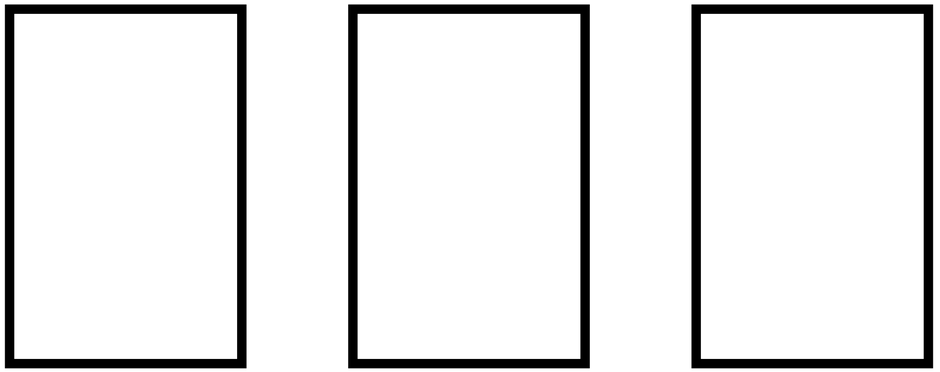


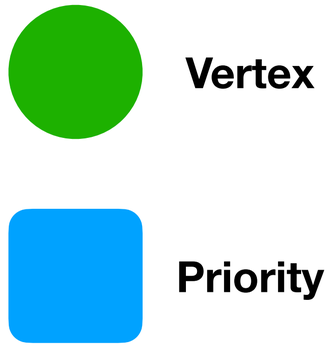
Eager Bucket Update

VertexID, NewPriority



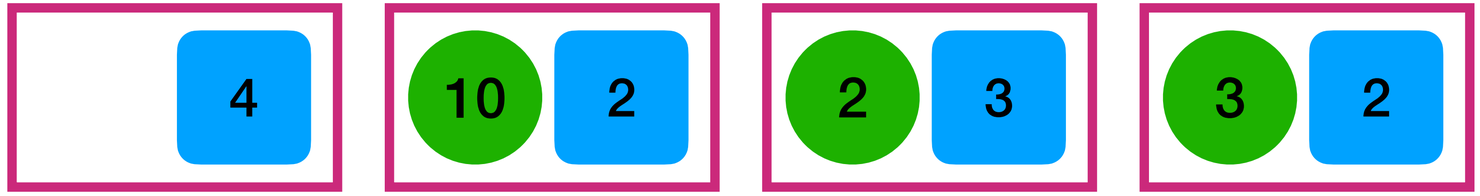
Buckets



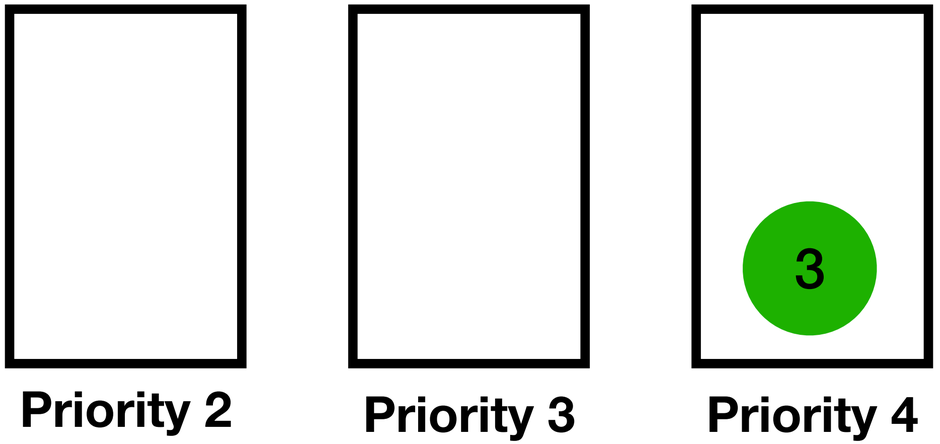


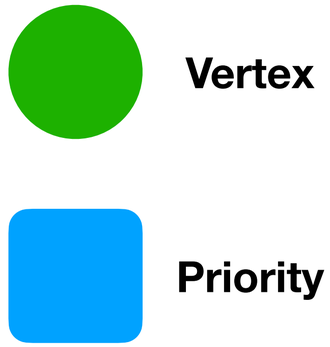
Eager Bucket Update

VertexID, NewPriority



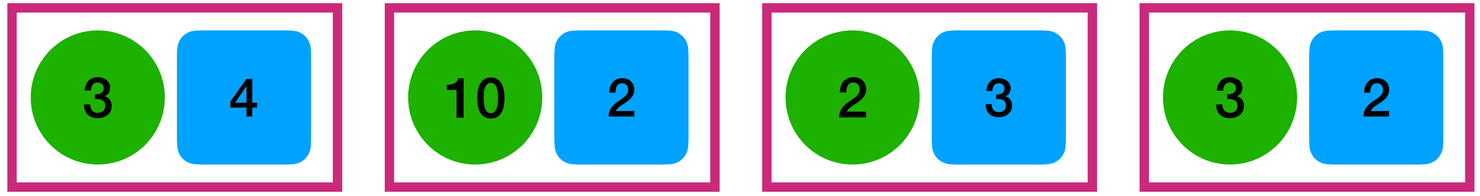
Buckets



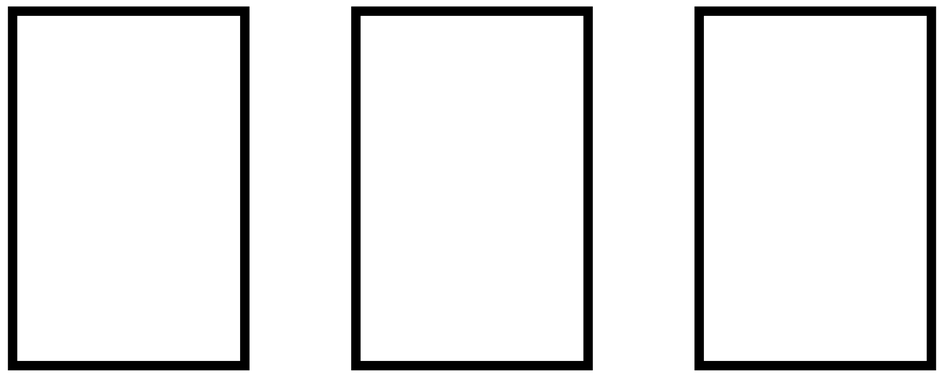


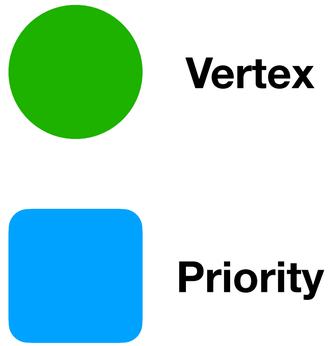
Lazy Bucket Update

VertexID, NewPriority



Buckets

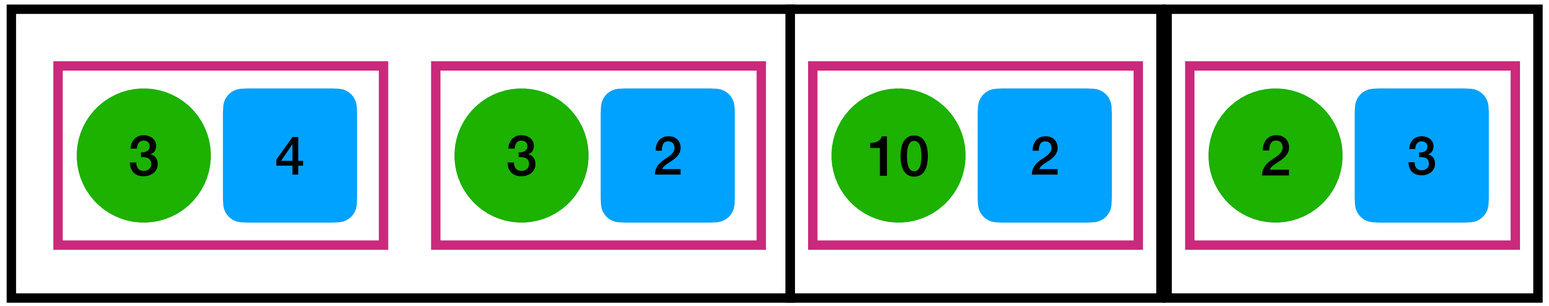




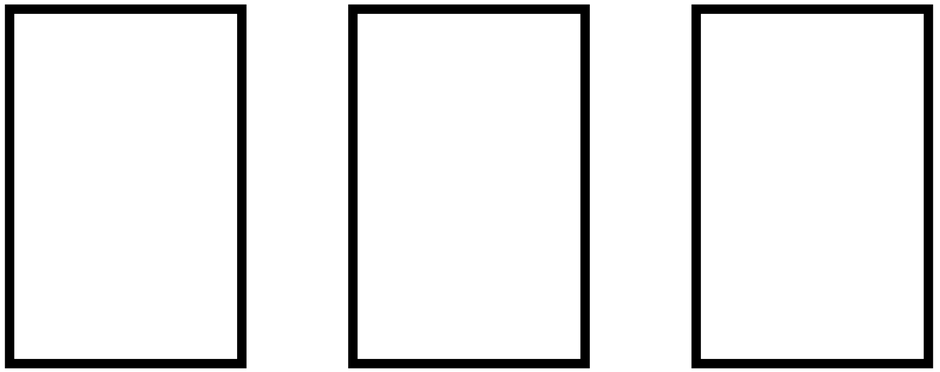
Lazy Bucket Update

VertexID, NewPriority

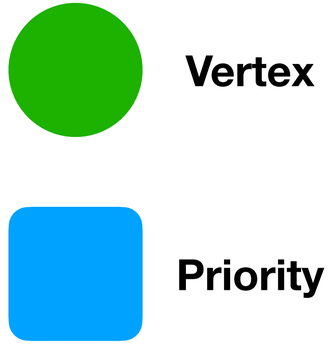
Buffer, Group Bucket Updates by VertexID



Buckets



Priority 2 **Priority 3** **Priority 4**



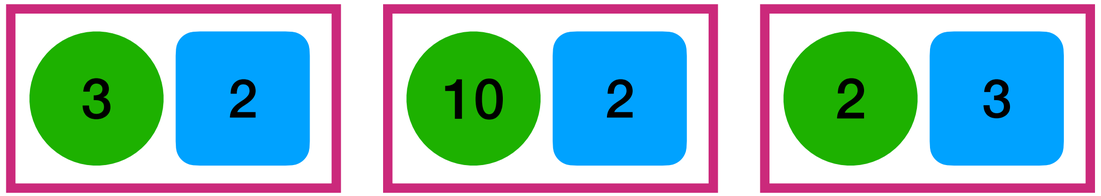
Vertex

Priority

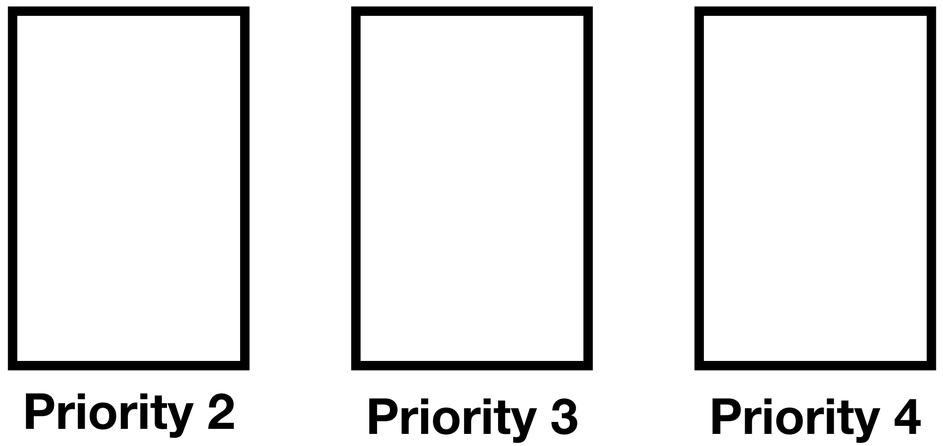
Lazy Bucket Update

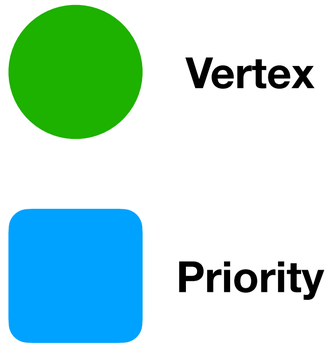
VertexID, NewPriority

Min Reduce Bucket Update by VertexID



Buckets

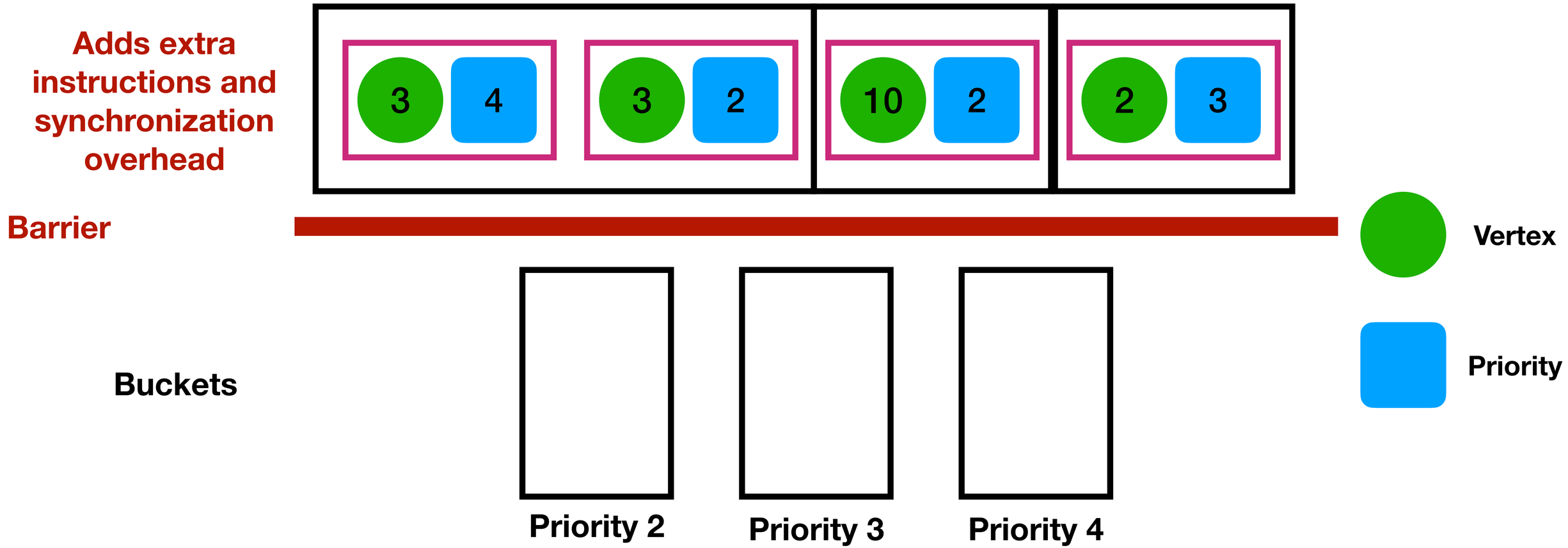




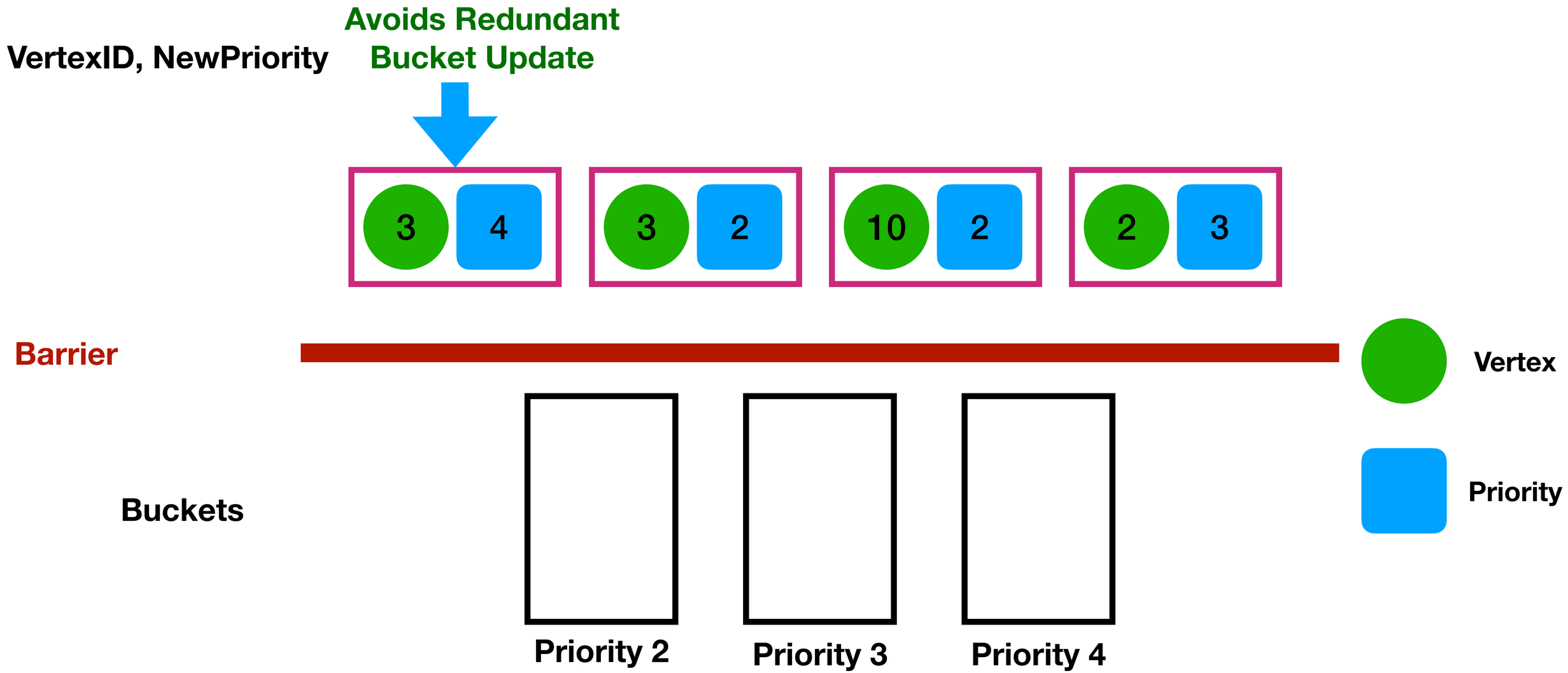
Lazy Bucket Update

VertexID, NewPriority

Adds extra instructions and synchronization overhead



Lazy Bucket Update



Eager vs Lazy

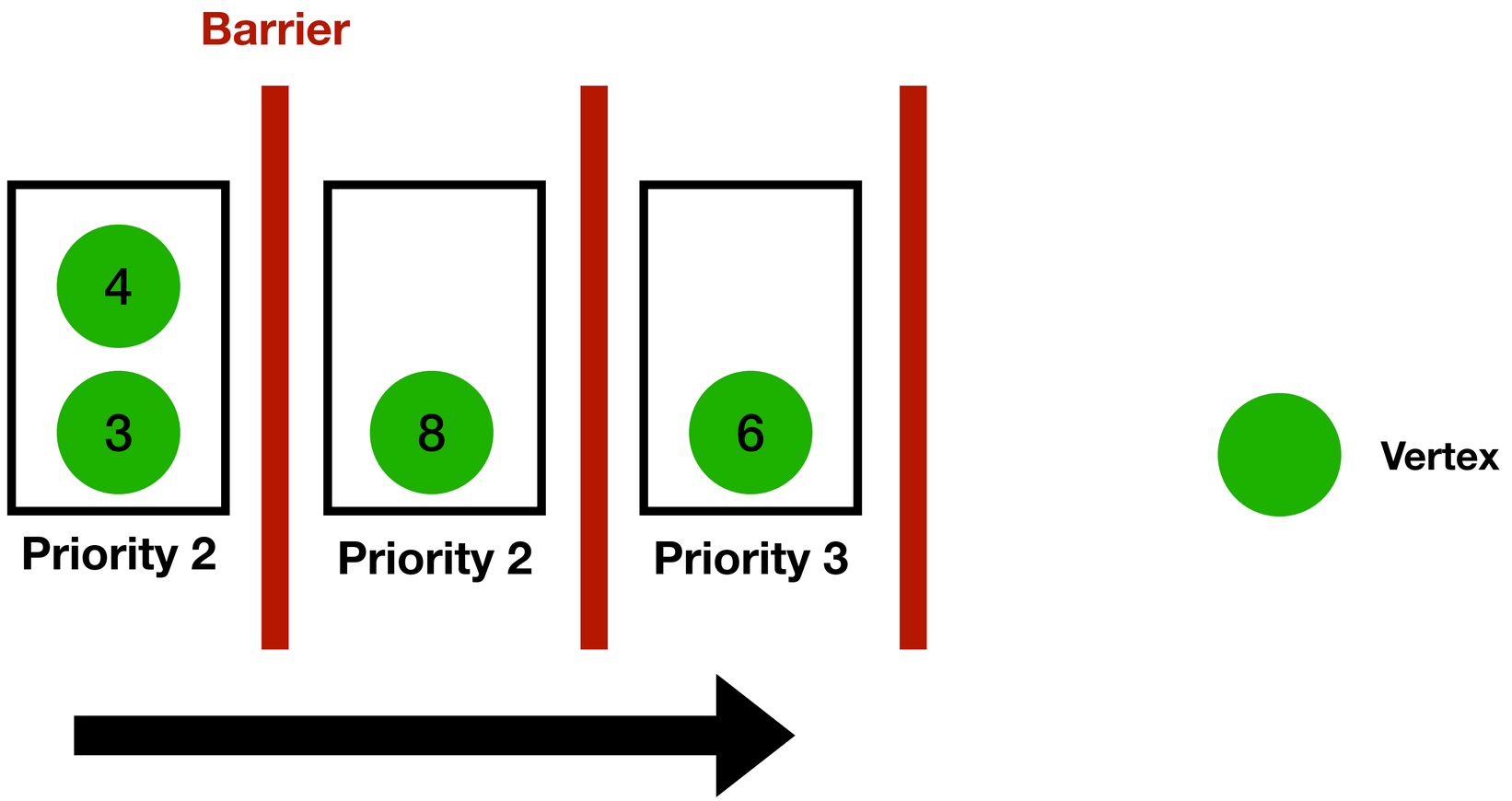
Eager Bucket Update

- Lower Synchronization Overhead (Good for SSSP, PPSP, AStar, and wBFS on Road Networks)
- Redundant Bucket Updates

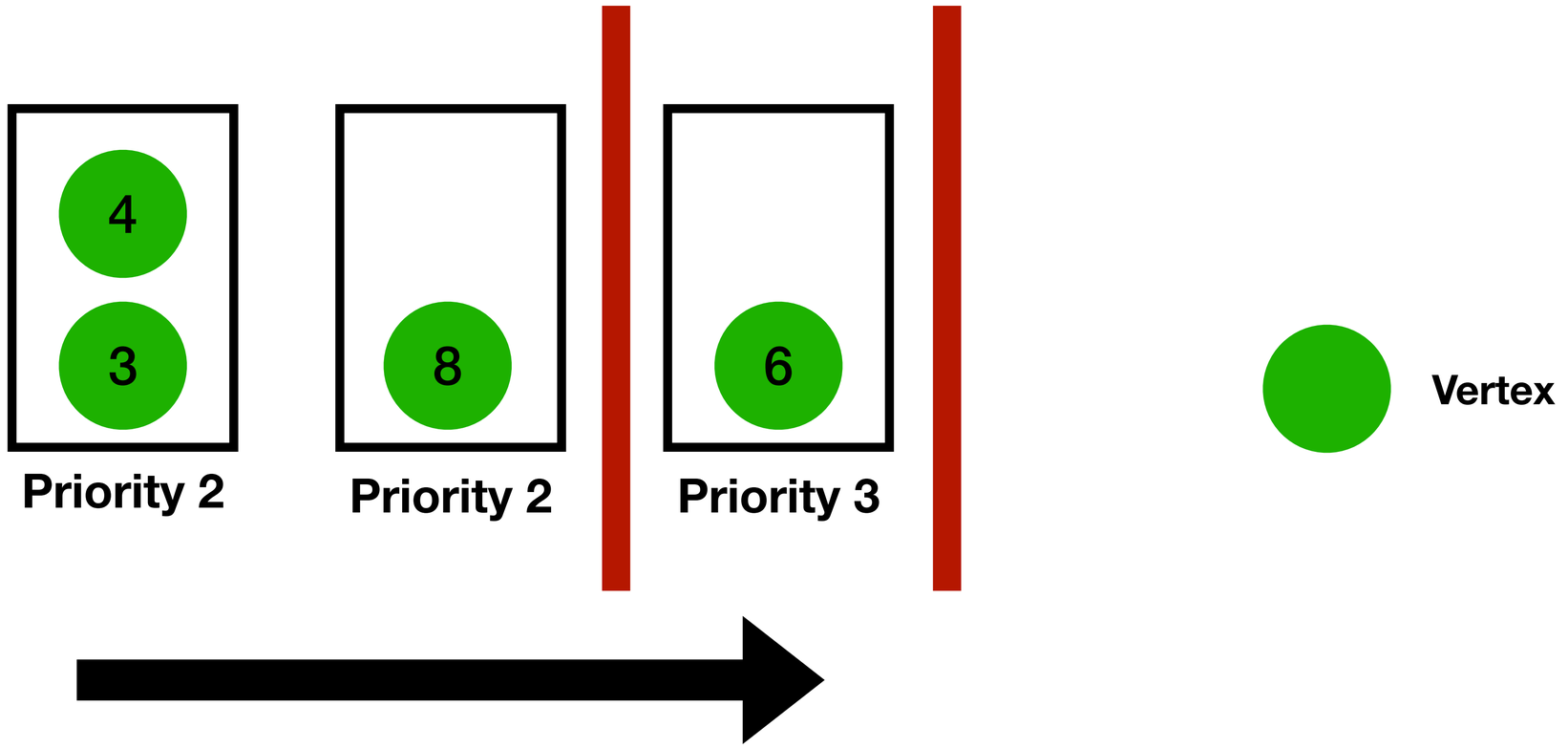
Lazy Bucket Update

- Fewer Redundant Bucket Updates (Good for KCore)
- More Instructions and Higher Synchronization Overhead

Bucket Fusion



Bucket Fusion



Scheduling Space Extensions

- **configApplyPriorityUpdate**
  - lazy, lazy_const_sum, eager_with_fusion, eager_no_fusion
- **configApplyPriorityUpdateDelta**
  - delta parameter for priority coarsening
- **configBucketFusionThreshold**
- **configNumBuckets** (number of materialized buckets)

Compatibility with GraphIt

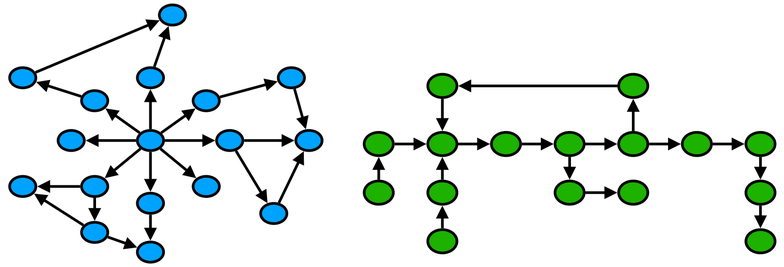
- Direction optimizations (configApplyDirection),
  - SparsePush, DensePush, DensePull, DensePull-SparsePush, DensePush-SparsePush
- Parallelization strategies (configApplyParallelization)
  - serial, dynamic-vertex-parallel, static-vertexparallel, edge-aware-dynamic-vertex-parallel, edge-parallel
- Cache (configApplyNumSSG)
  - fixed-vertex-count, edge-aware-vertexcount
- NUMA (configApplyNUMA)
  - serial, static-parallel, dynamic-parallel
- AoS, SoA (fuseFields)
- Vertexset data layout (configApplyDenseVertexSet)
  - bitvector, boolean array

Autotuning Optimizations

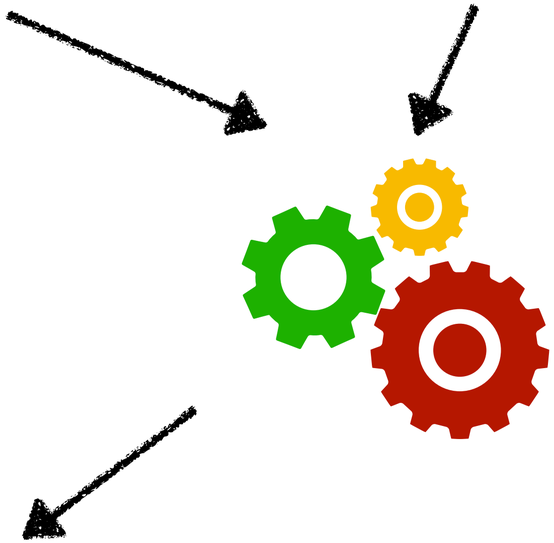
```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end


func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
 end
end
```



Input Graphs



Schedule:

program

->configApplyPriorityUpdate(“s1”, “lazy”)

->configApplyPriorityUpdateDelta(“s1”, 4)

->configApplyDirection(“s1“, SparsePush”)

->configApplyParallelization(“s1”, “dynamic-vertex-parallel”)

Outline

- Background and Motivation
- Programming Model
- Optimizations
- **Related Work**
- Evaluation

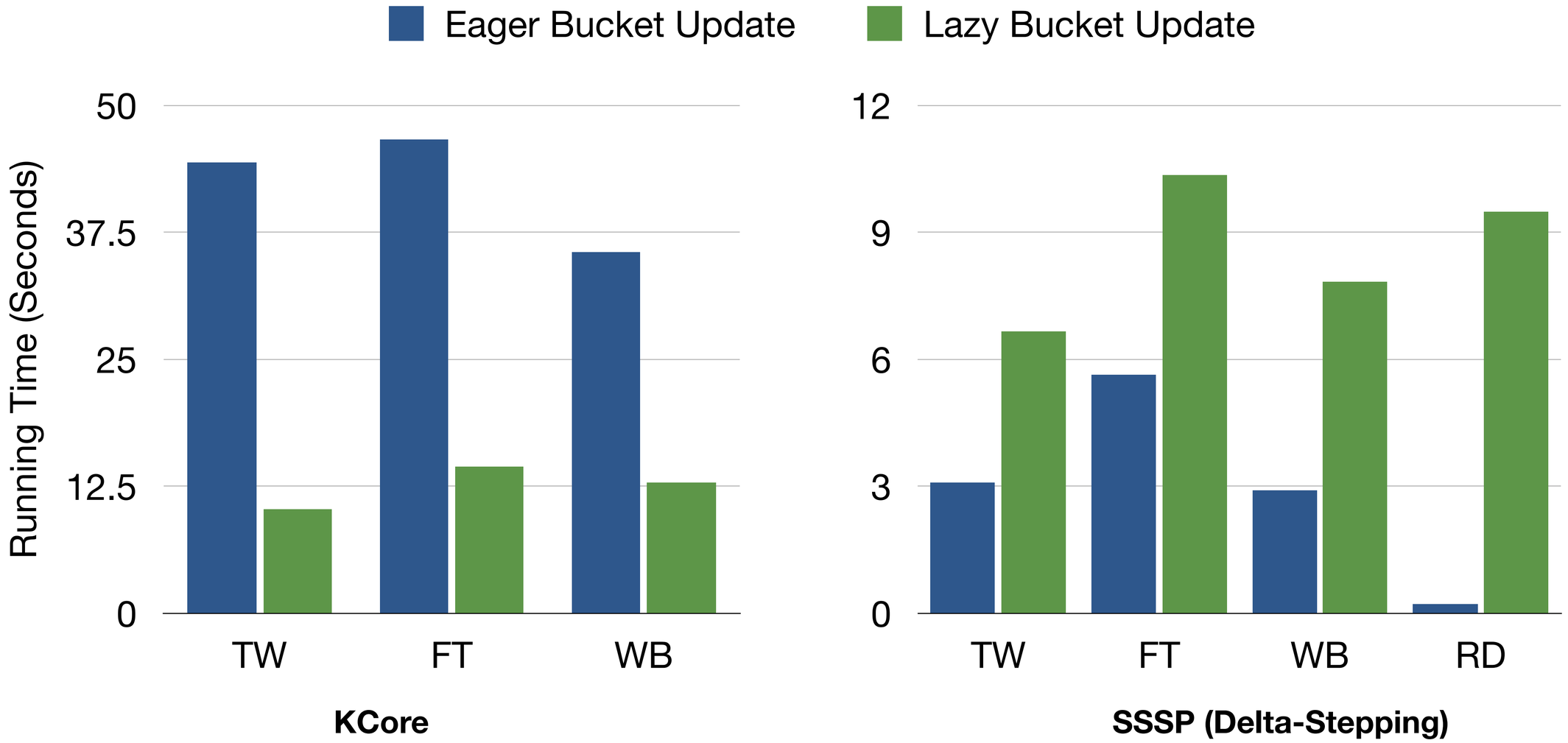
Related Work

- Architecture Support for Speculative Execution
  - Swarm [MICRO15,MICRO16,ISCA17,MICRO18]
- Other Frameworks that Support Ordered Graph Algorithms
  - Julienne [SPAA17], GAPBS [IISWC15] , Galois [PPoPP11,SOSP13,PLDI18]

Outline

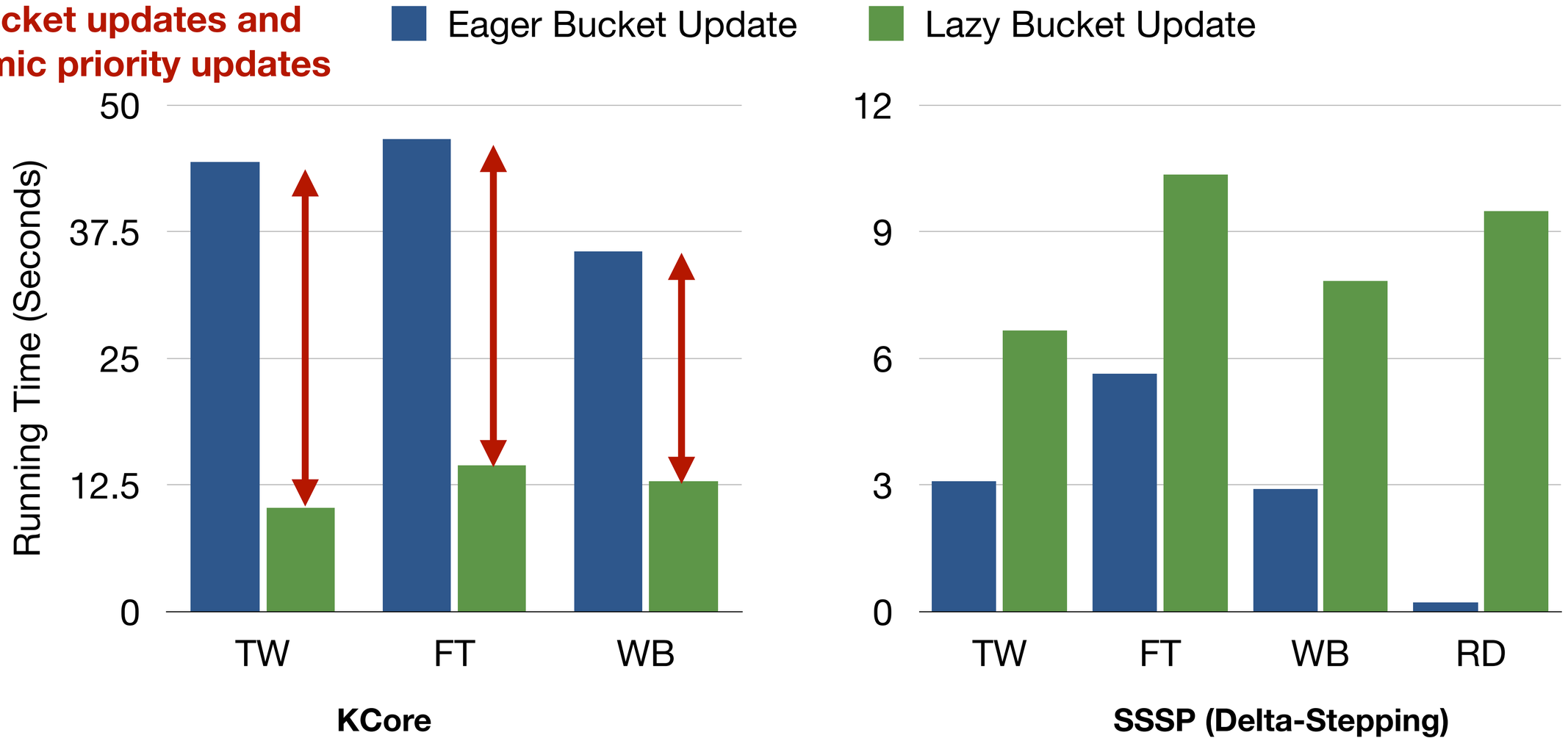
- Background and Motivation
- Programming Model
- Optimizations
- Related Work
- **Evaluation**

Eager vs Lazy

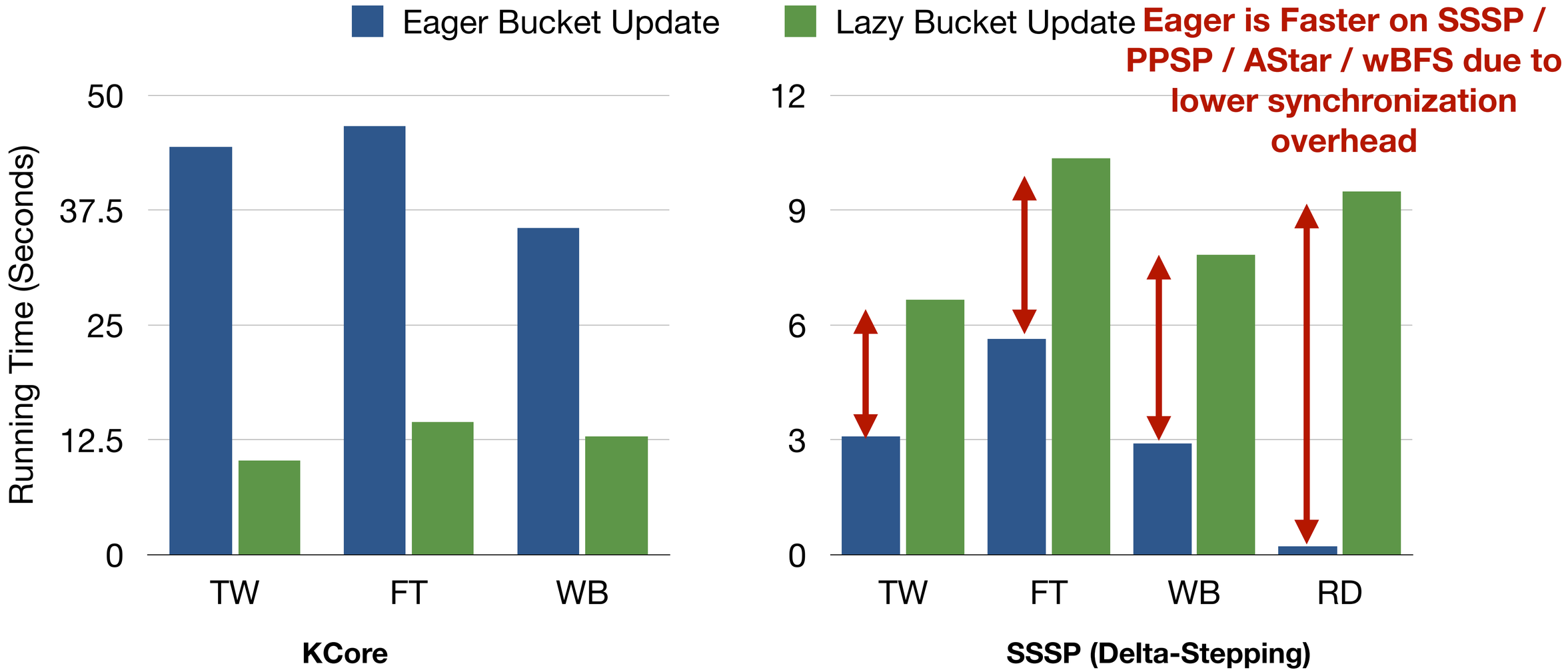


Eager vs Lazy

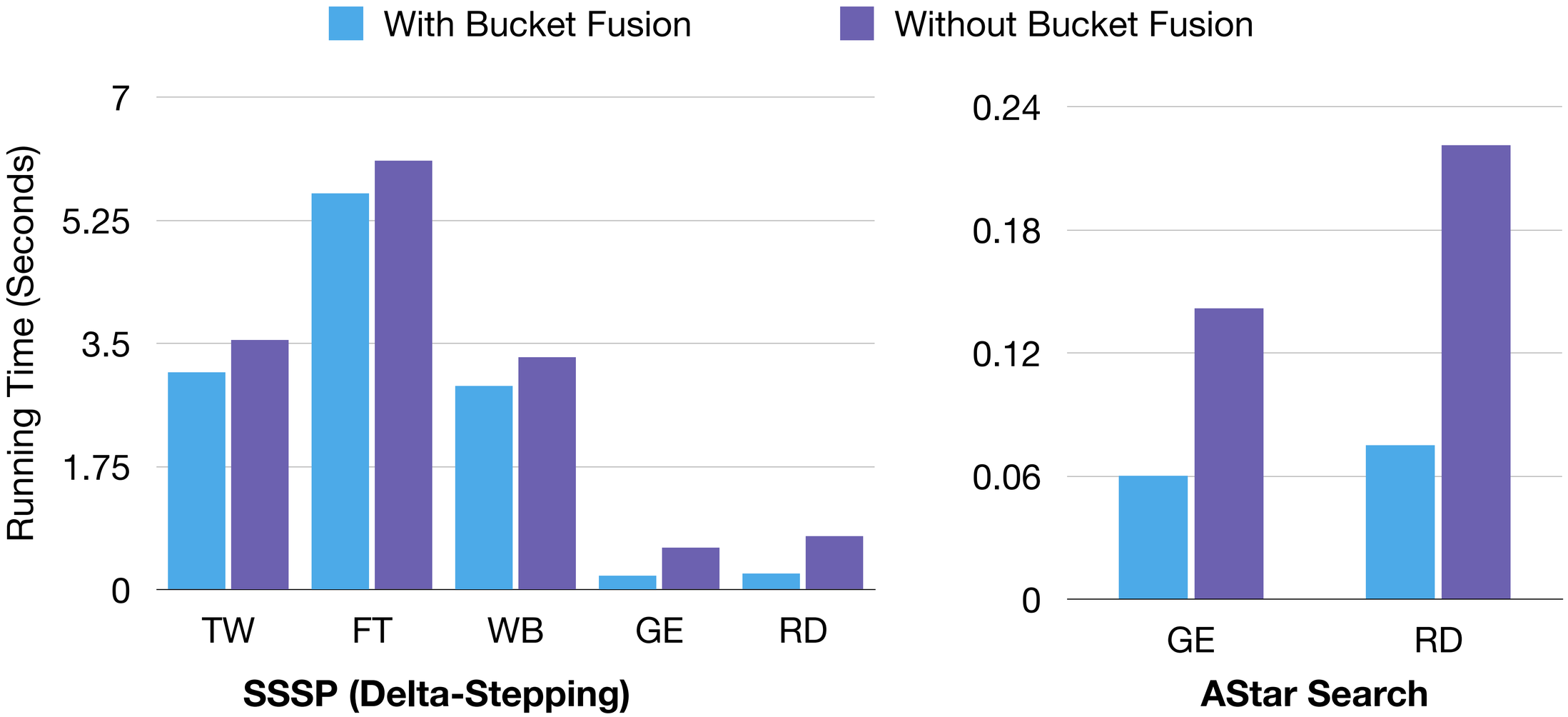
Lazy is Faster on KCore due to fewer redundant bucket updates and atomic priority updates



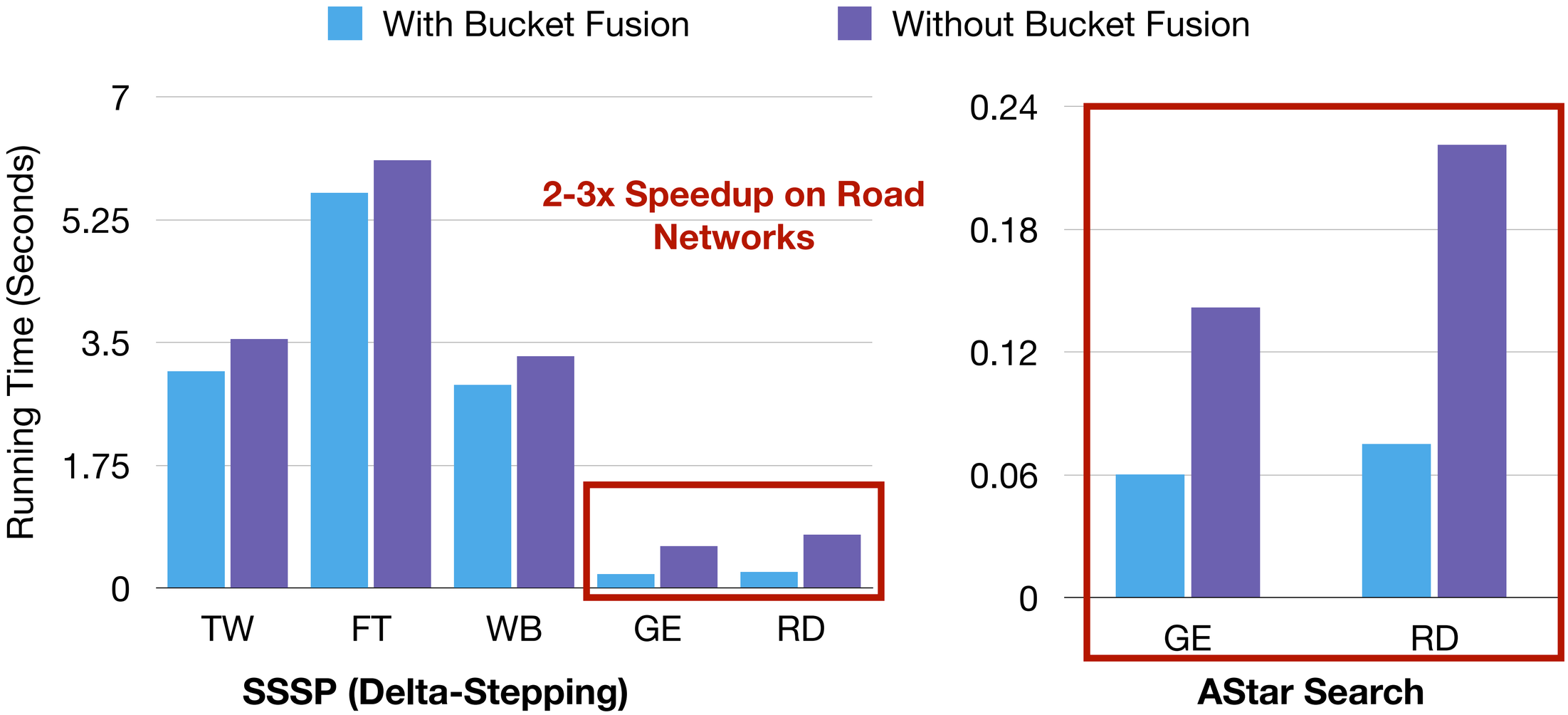
Eager vs Lazy



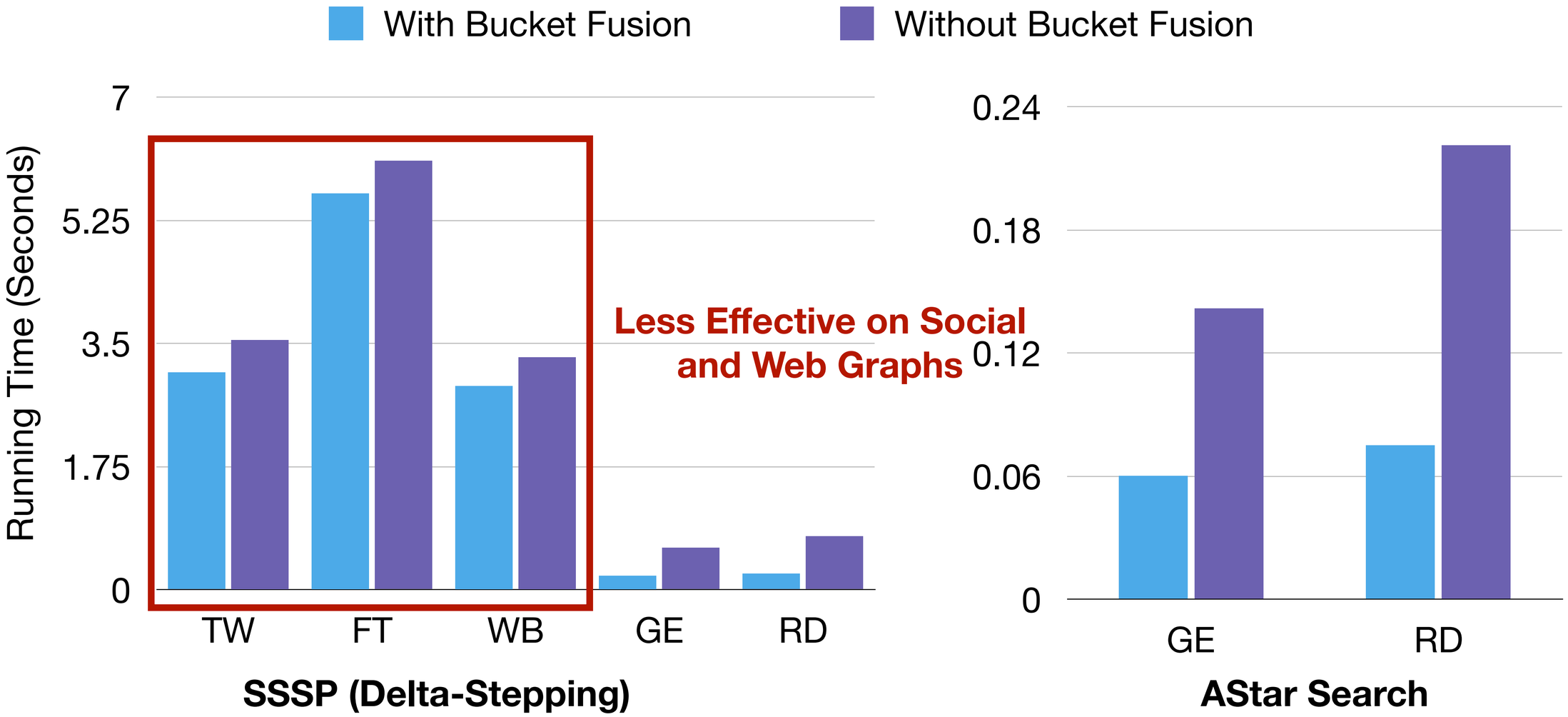
Bucket Fusion (Eager)



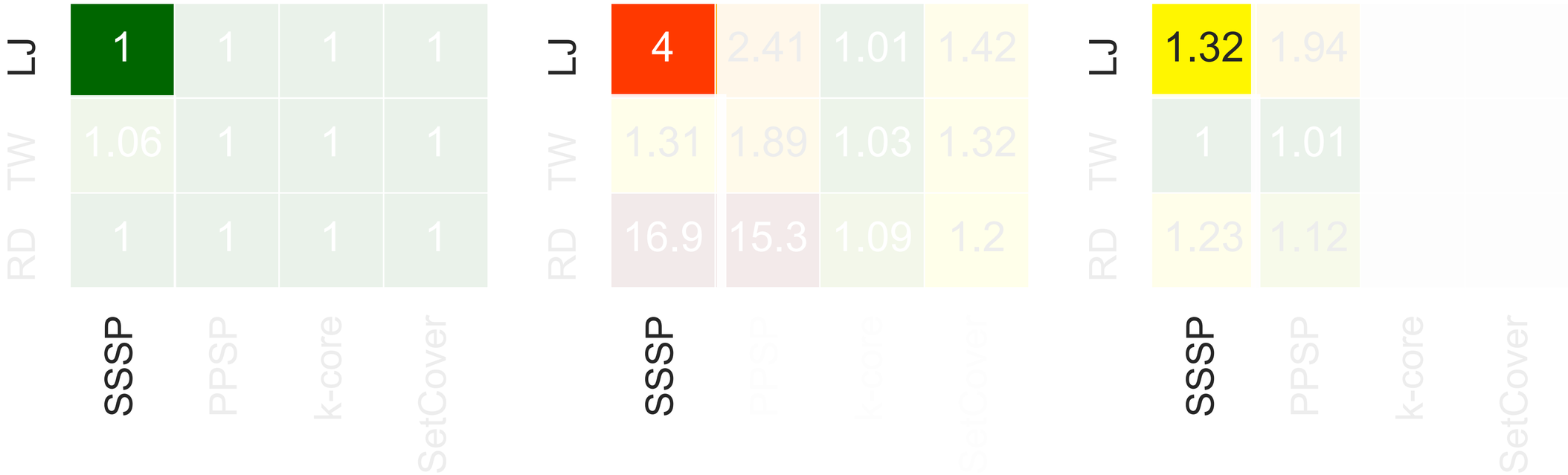
Bucket Fusion (Eager)



Bucket Fusion (Eager)

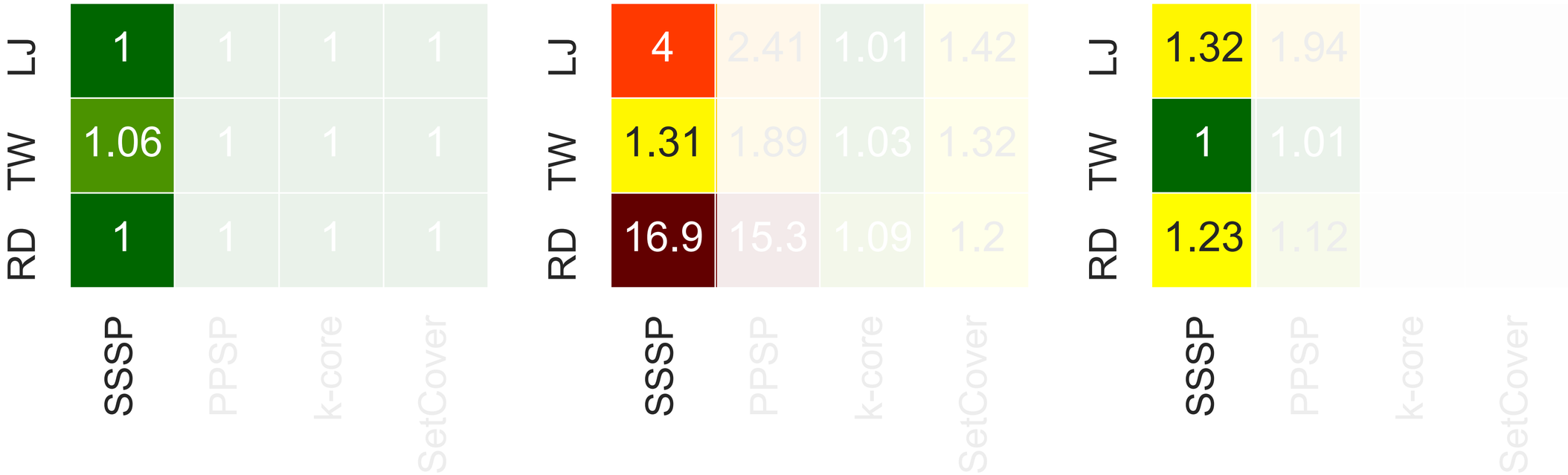


Comparisons with Ordered Frameworks



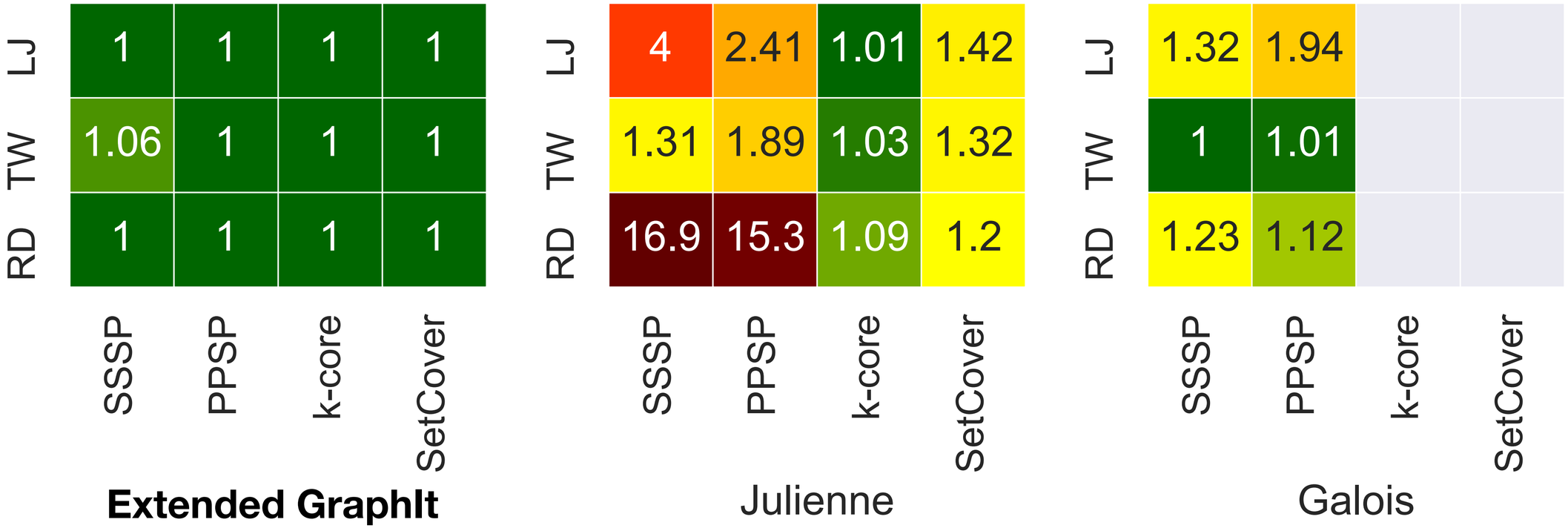
Results for more graphs and algorithms are in the paper
(AStar Search, weighted BFS)

Comparisons with Ordered Frameworks



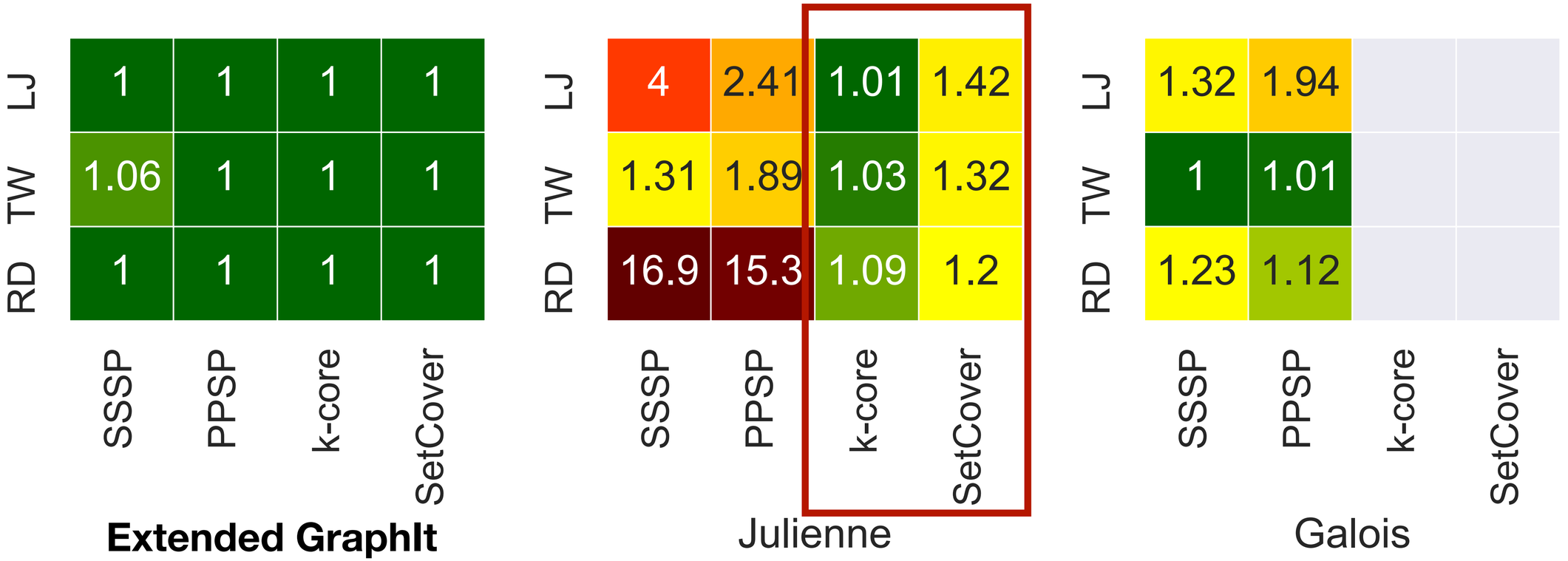
Results for more graphs and algorithms are in the paper (AStar Search, weighted BFS)

Comparisons with Ordered Frameworks

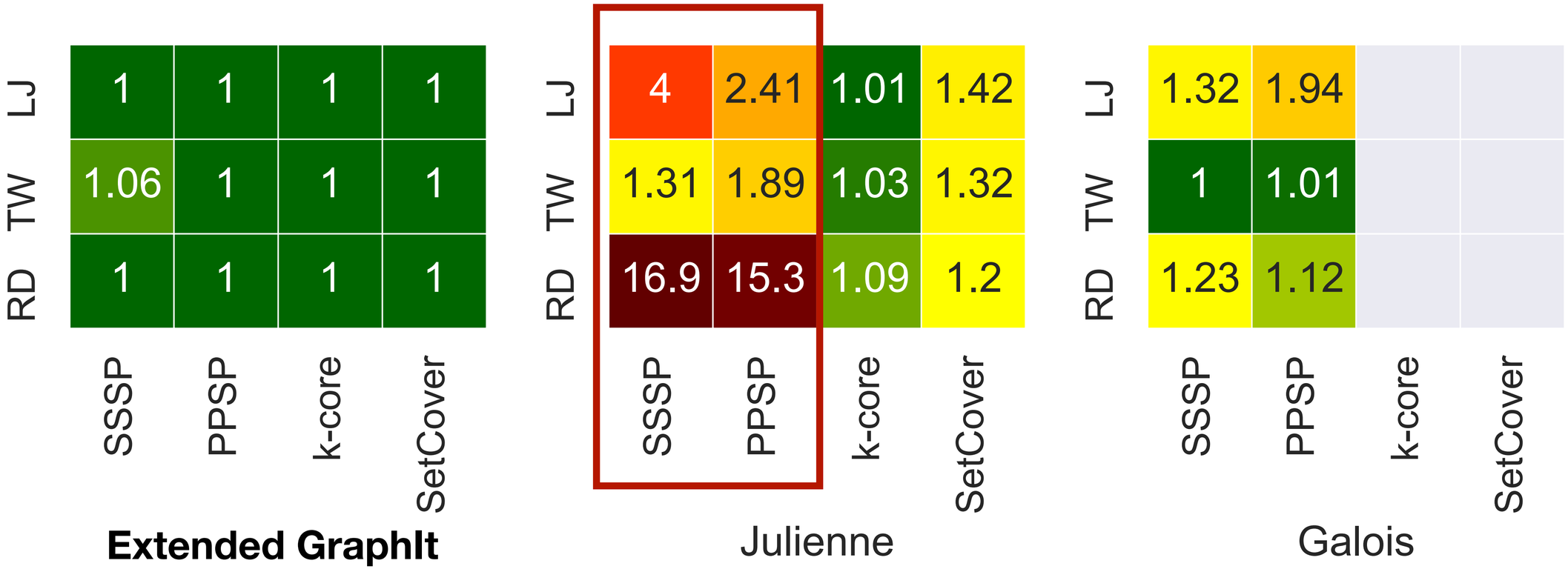


Results for more graphs and algorithms are in the paper
(AStar Search, weighted BFS)

Comparisons with Ordered Frameworks

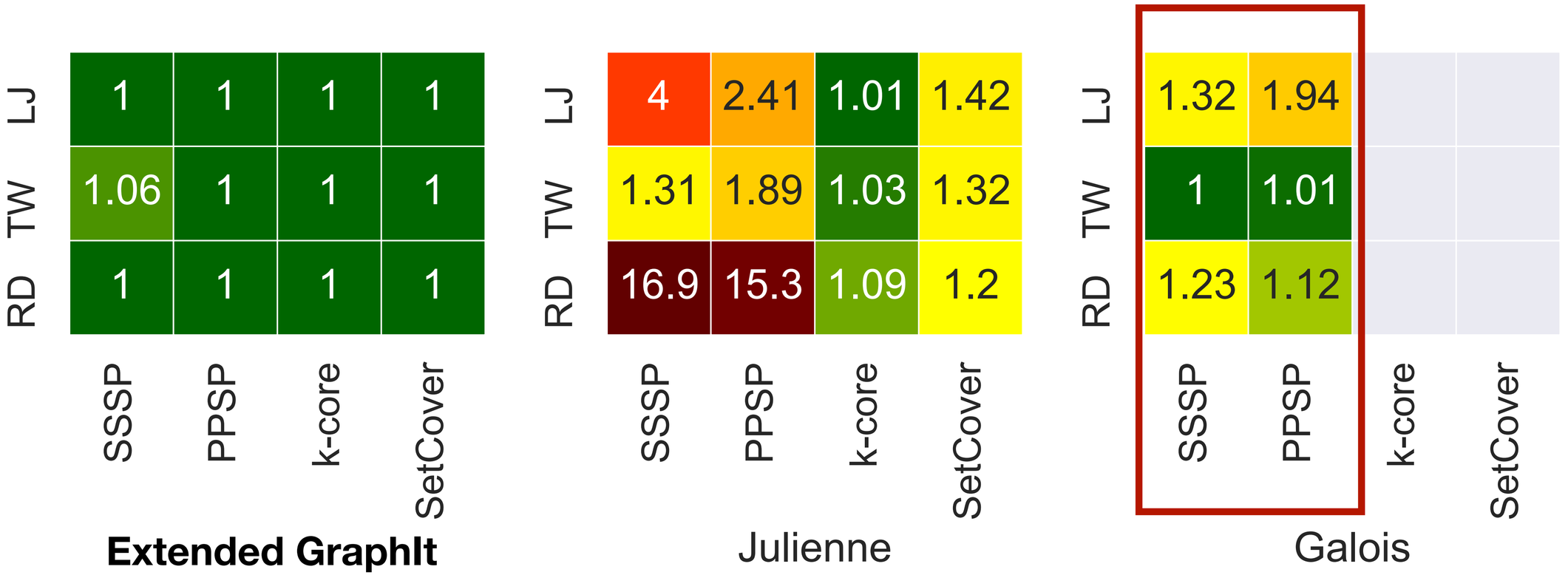


Comparisons with Ordered Frameworks

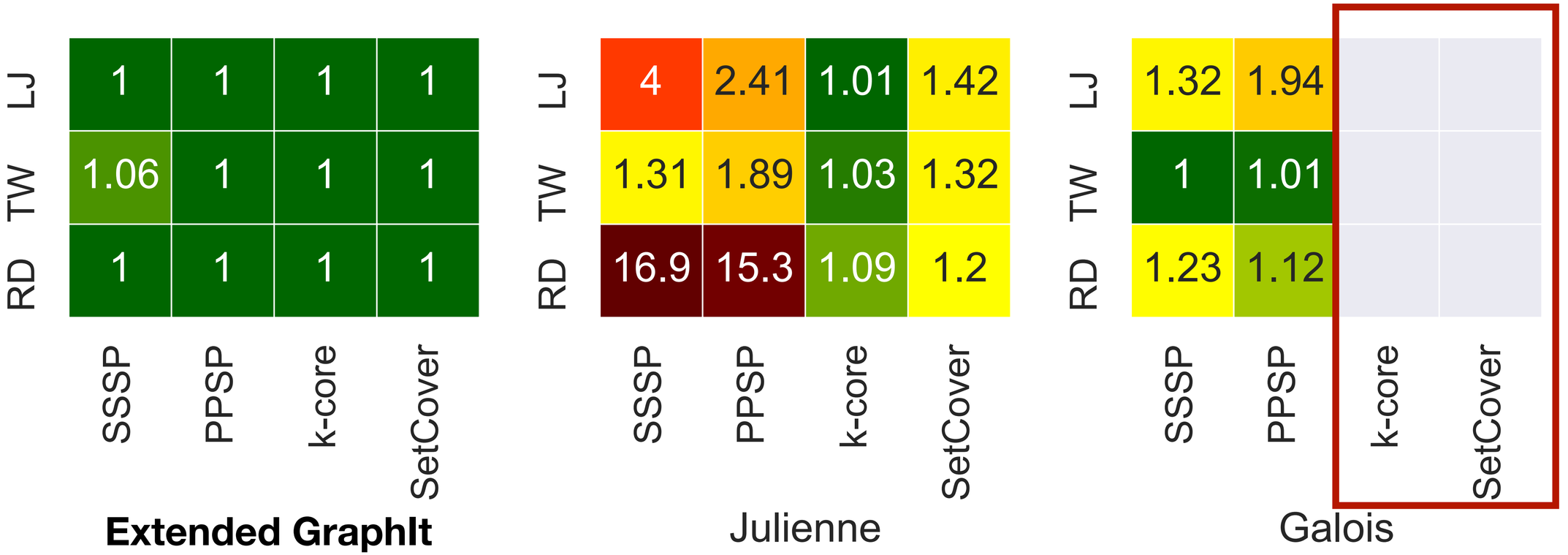


No Support for Eager Bucket Update

Comparisons with Ordered Frameworks

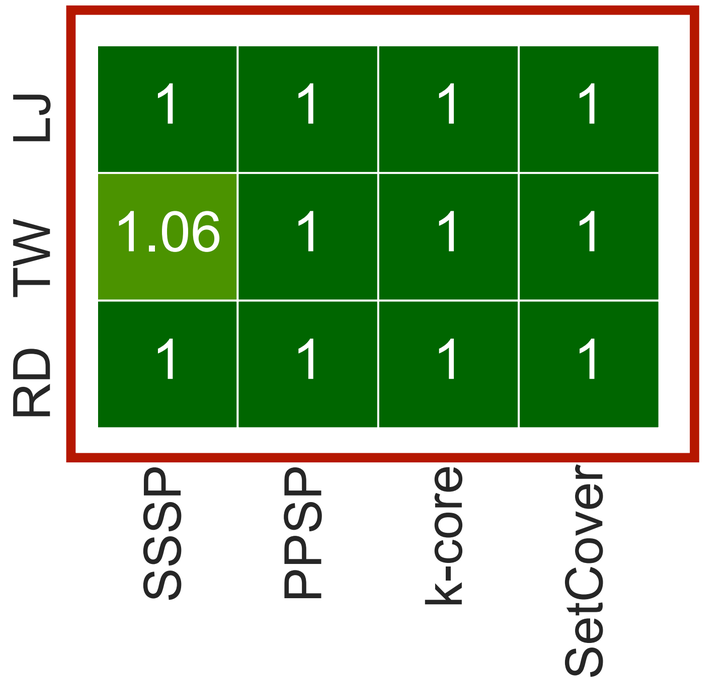


Comparisons with Ordered Frameworks

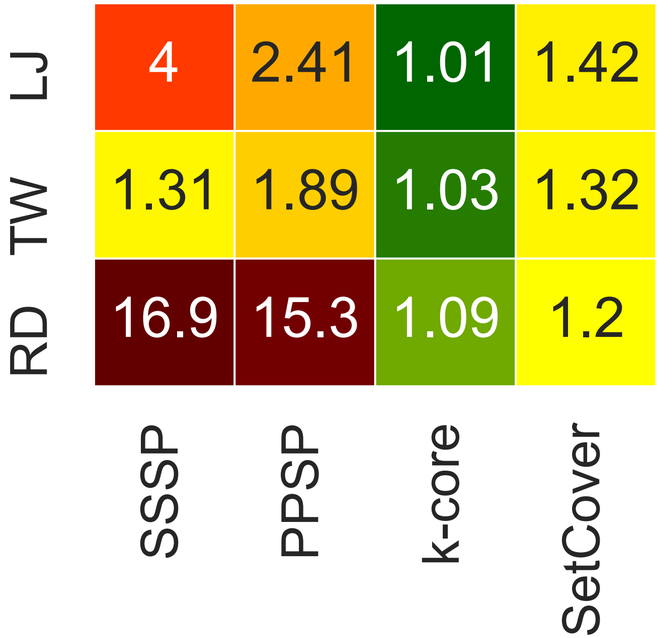


No Support for Strict Priority Ordering Needed for Correctness

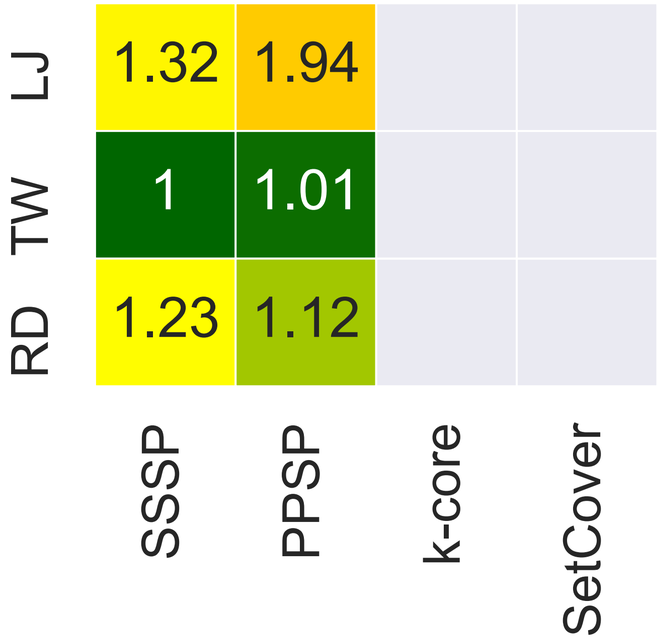
Comparisons with Ordered Frameworks



Extended GraphIt



Julienne



Galois

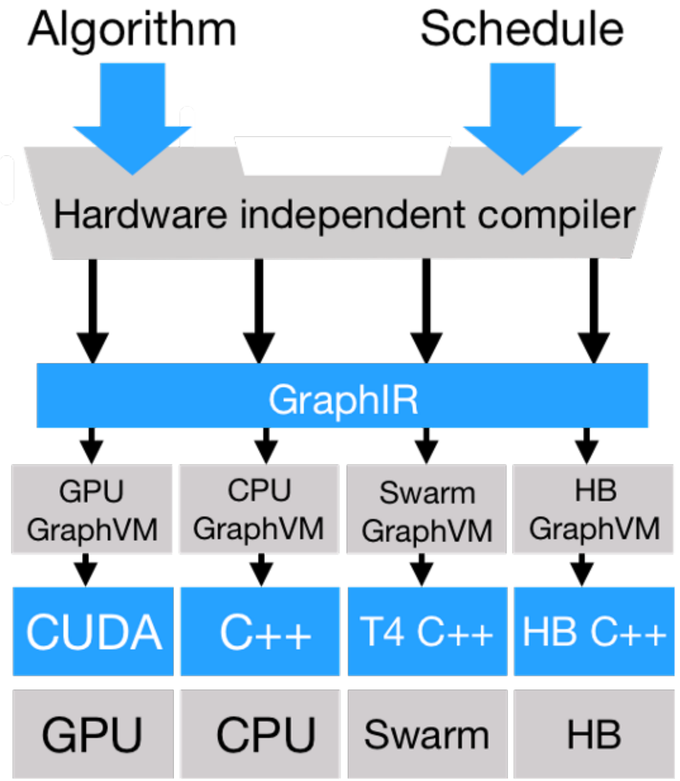
Achieves consistent high-performance across algorithms and graphs

Achieving Portability

GraphIR consists of hardware-independent compiler passes

Extensible to new hardware backends (**GraphVMs**)

GraphVMs have their own scheduling languages inherited from a base scheduling language



Achieving Portability

Our GraphIR allows significant code reuse inside the implementations of different backends

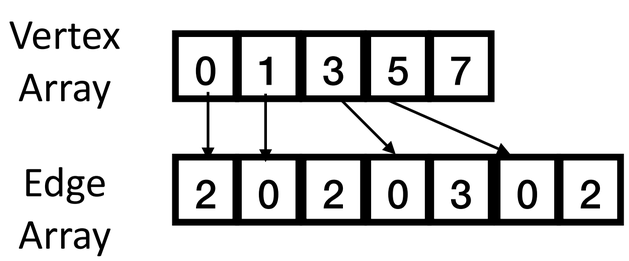
	Module	Base version	CPU	GPU	Swarm	HammerBlade
Frontend	Algorithm parser & AST definitions	**10,900**	0	0	0	0
	Schedule language functions	**136**	306	385	524	89
Hardware-Independent Compiler	Frontier Reuse Analysis	**125**	Not used	0	0	0
	Property analysis/Atomic insertion	**536**	0	0	Not used	0
	Ordered Processing Lowering	**386**	0	120	0	0
	Other Lowering Passes	**4,171**	0	0	0	0
GraphVM	Ordered Processing Specialization	**104**	0	Not used	0	Not used
	Kernel Fusion	276	Not used	0	Not used	Not used
	Code Generator	-	3,843	1,874	959	2,282
	Runtime Library	-	10,385	2,470	156	1,127

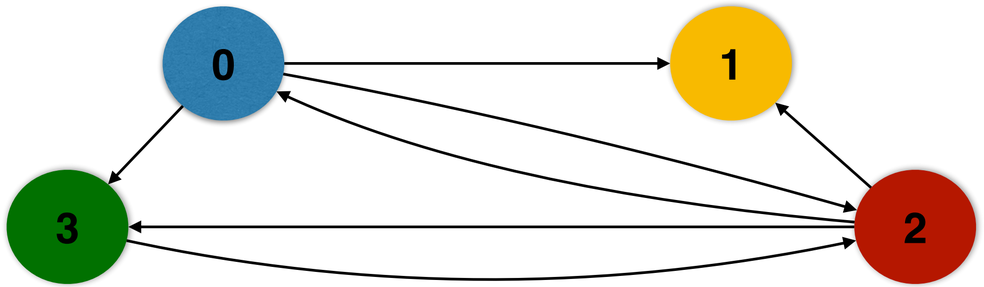
Cache Segmenting Optimization

PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Compressed Sparse Row (CSR)



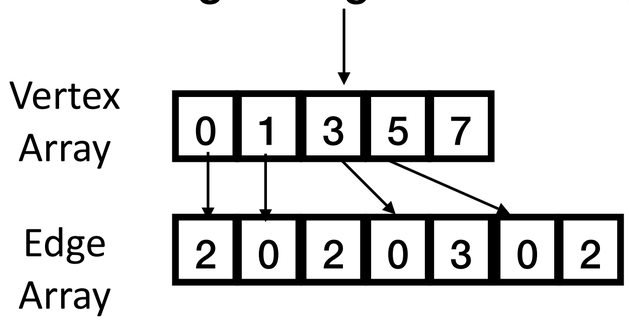


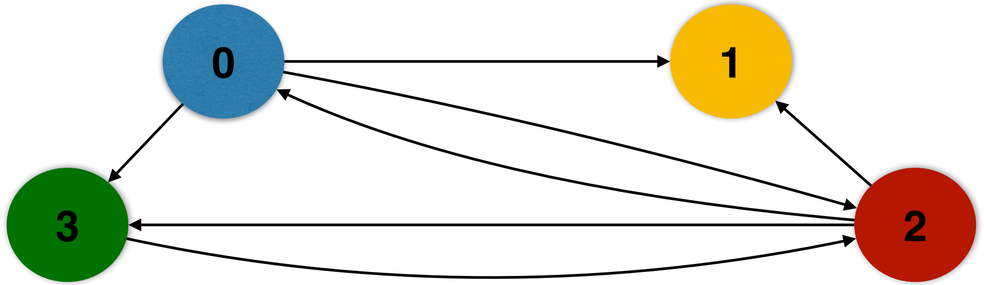
Vertex Array stores indices into the Edge Array. Edge Array stores neighbors' ID in the CSR

PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

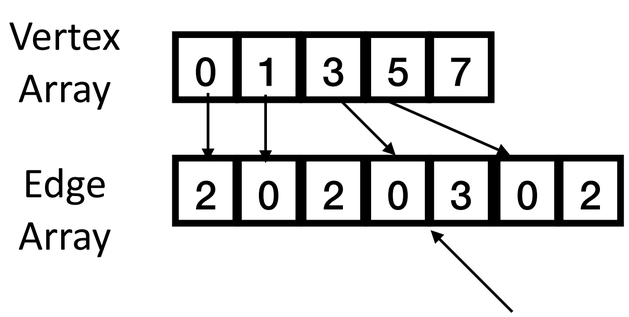
Sequential access on node when scanning through vertex array

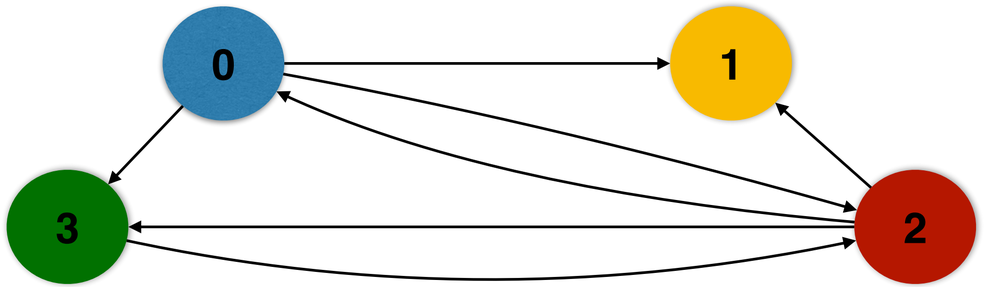




PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```





Irregular access on ngh's rank and outDegree data when scanning through the edge array

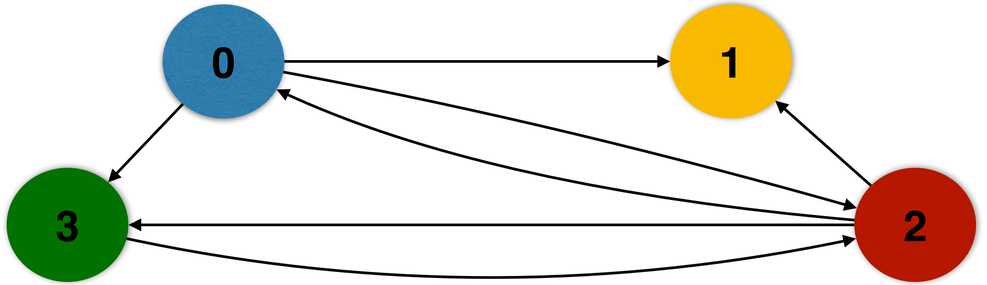
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

#hits: 0

#misses: 0



PageRank

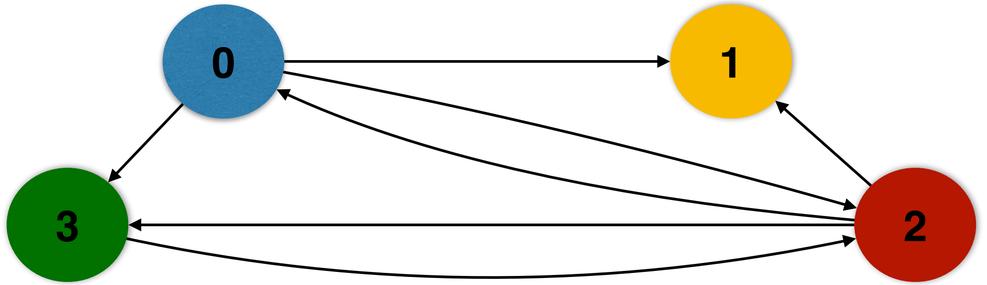
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Focus on the random memory accesses on ranks array

Cache

#hits: 0

#misses: 0



PageRank

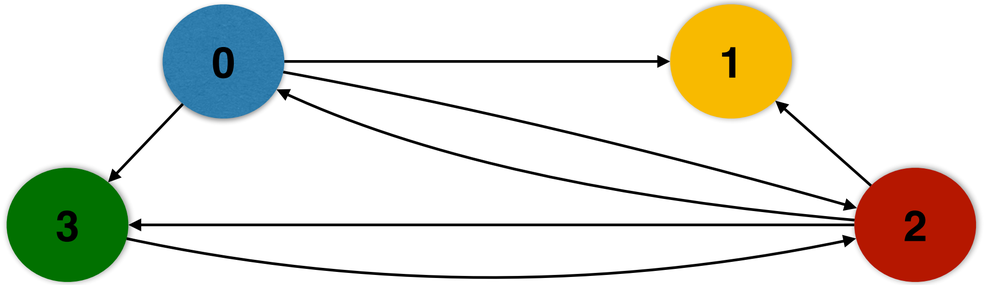
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Focus on the random memory accesses on ranks array

Cache ← **holds one cache line**

#hits: 0

#misses: 0



PageRank

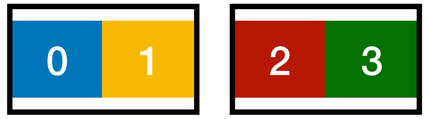
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Focus on the random memory accesses on ranks array

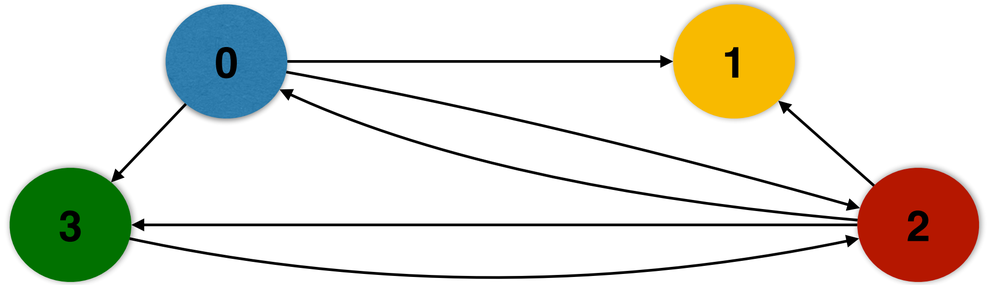
Cache ← **holds one cache line**

#hits: 0

#misses: 0



stored in two cache lines



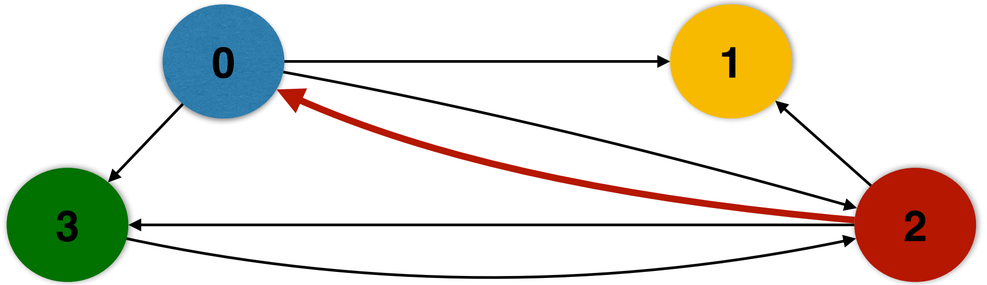
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

#hits: 0

#misses: 0



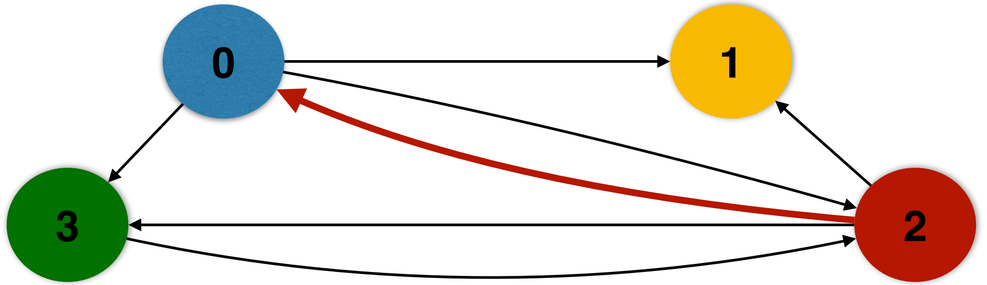
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

#hits: 0

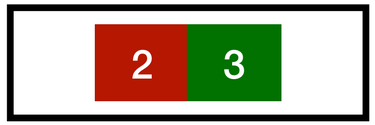
#misses: 0



PageRank

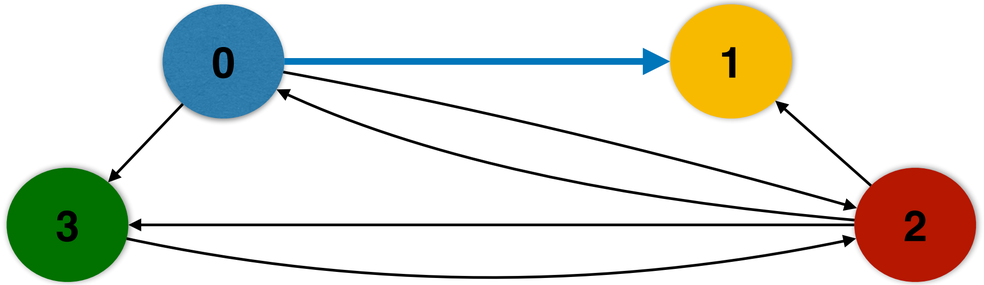
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache



#hits: 0

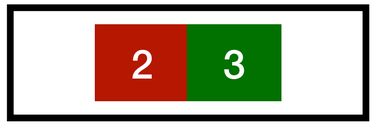
#misses: 1



PageRank

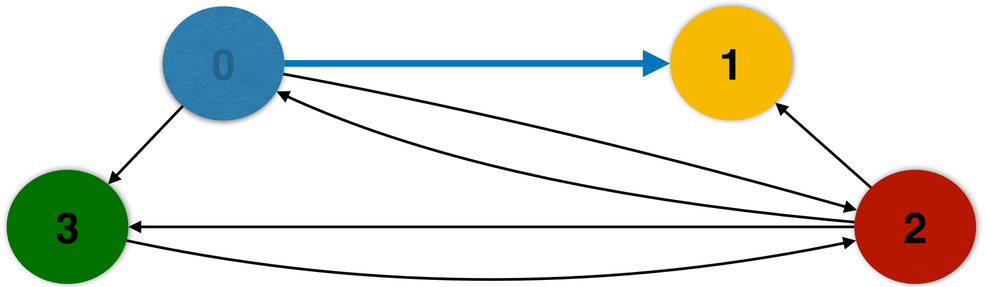
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache



#hits: 0

#misses: 2

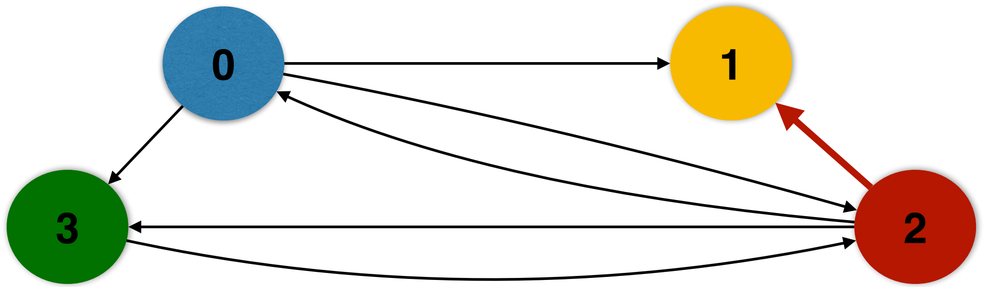


PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

#hits: 0
#misses: 2



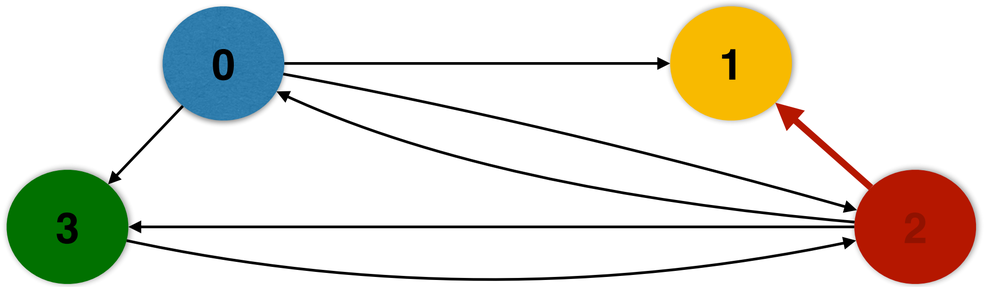
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

#hits: 0

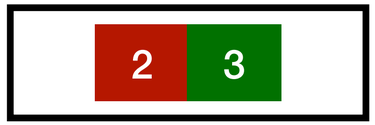
#misses: 2



PageRank

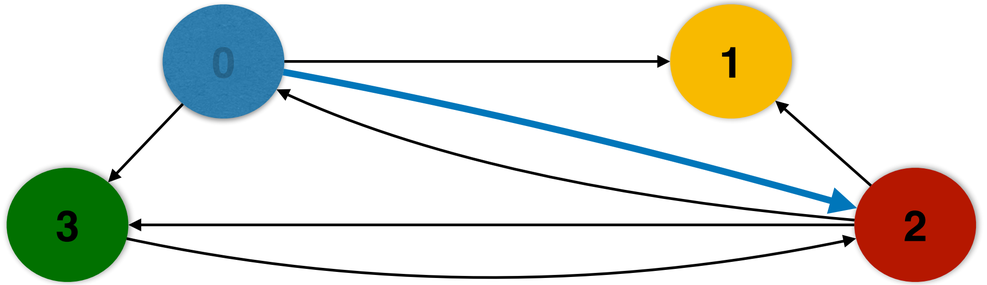
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache



#hits: 0

#misses: 8



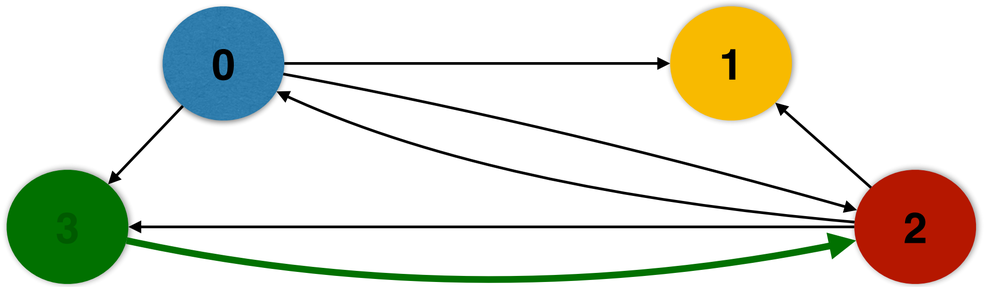
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

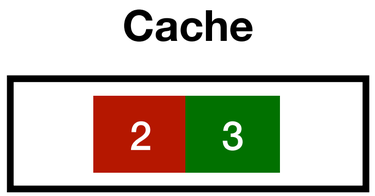
#hits: 0

#misses: 4

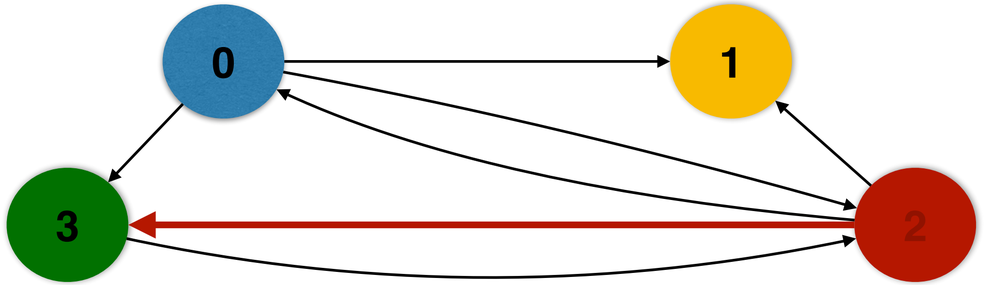


PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



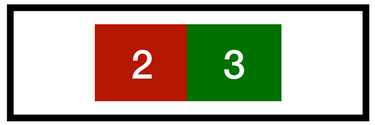
#hits: 0
#misses: 5



PageRank

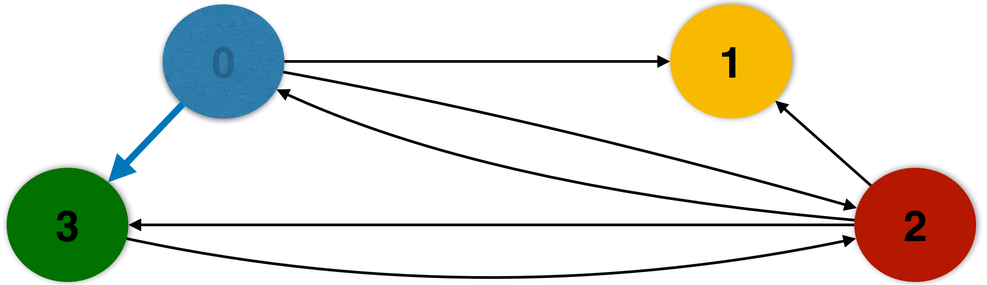
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache



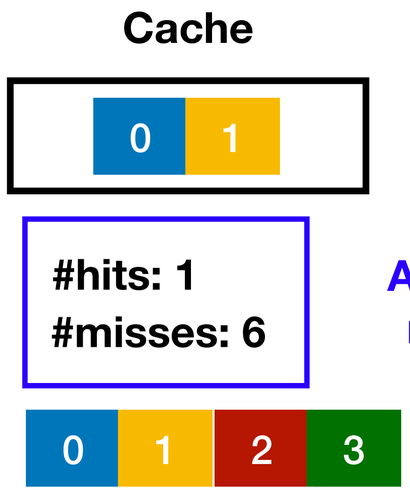
#hits: 1

#misses: 6

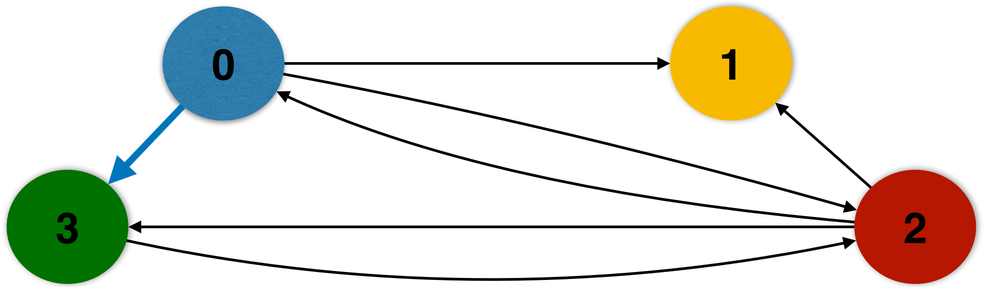


PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

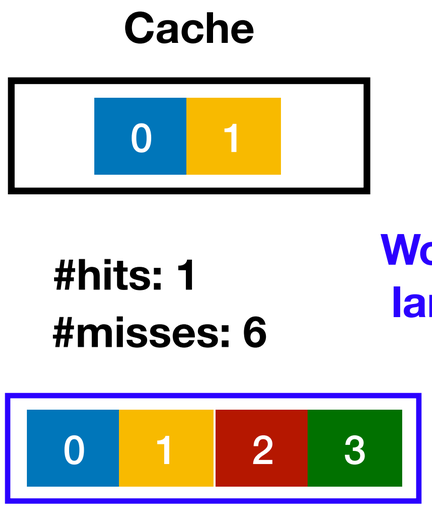


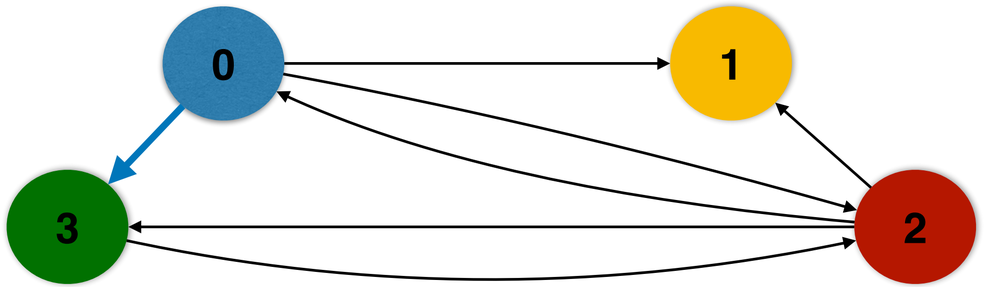
A very high miss rate



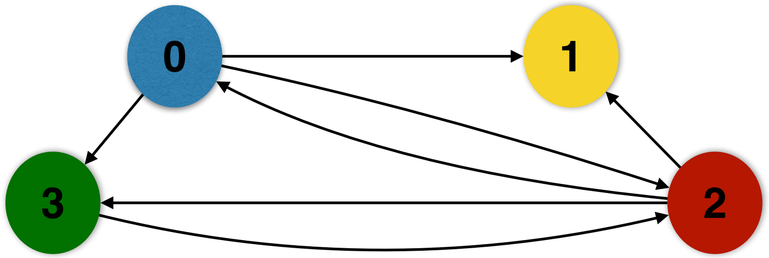
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

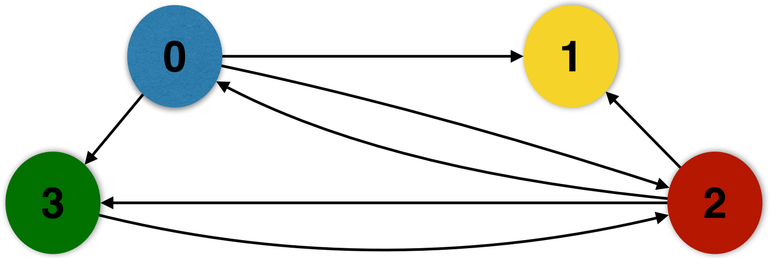




Graph Partitioning

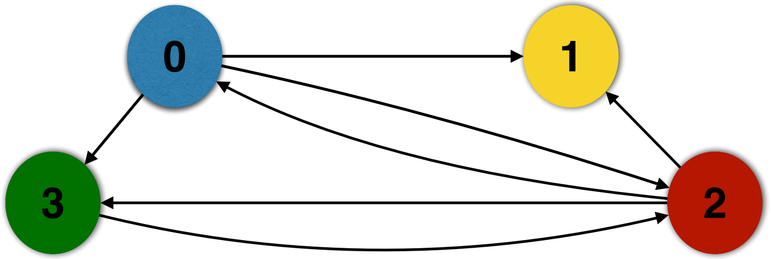


Graph Partitioning



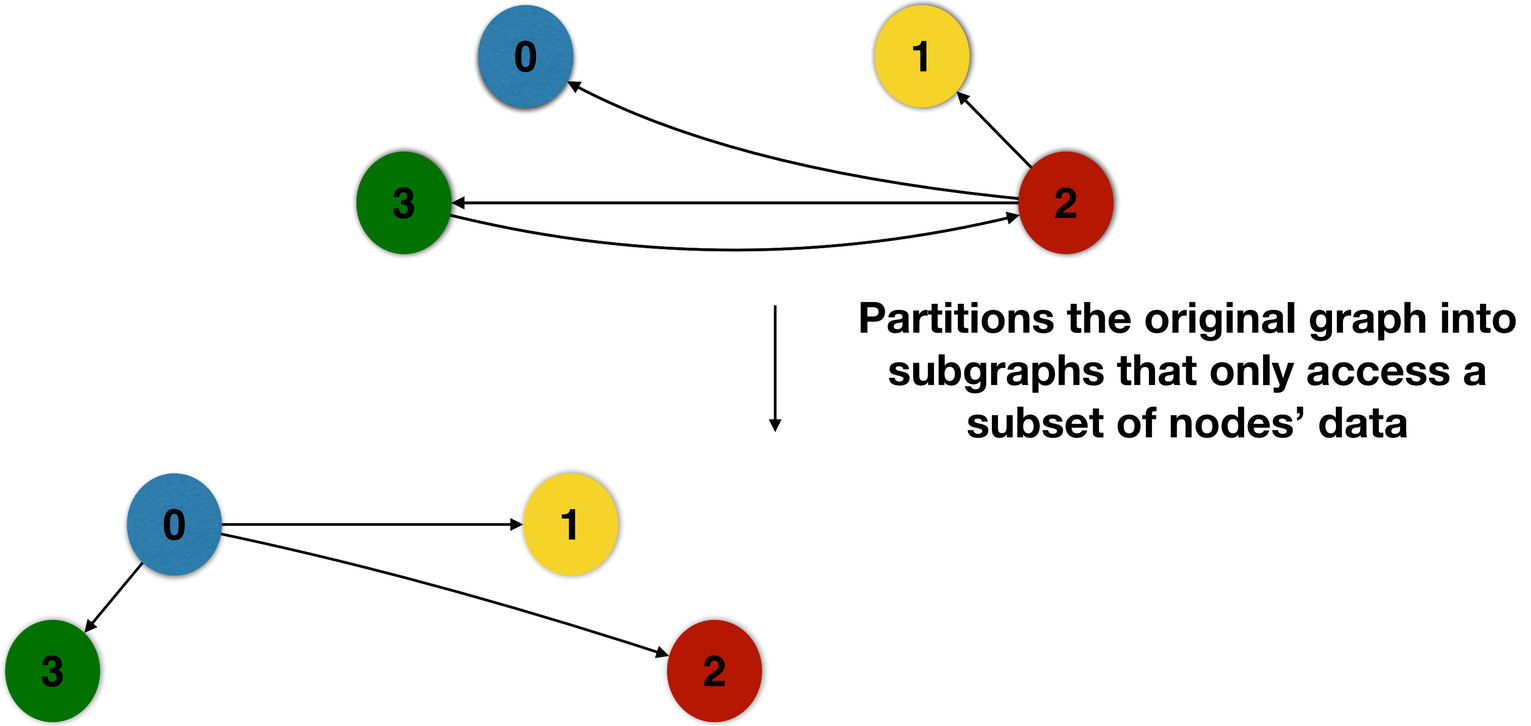
Partitions the original graph into subgraphs that only access a subset of nodes' data

Graph Partitioning

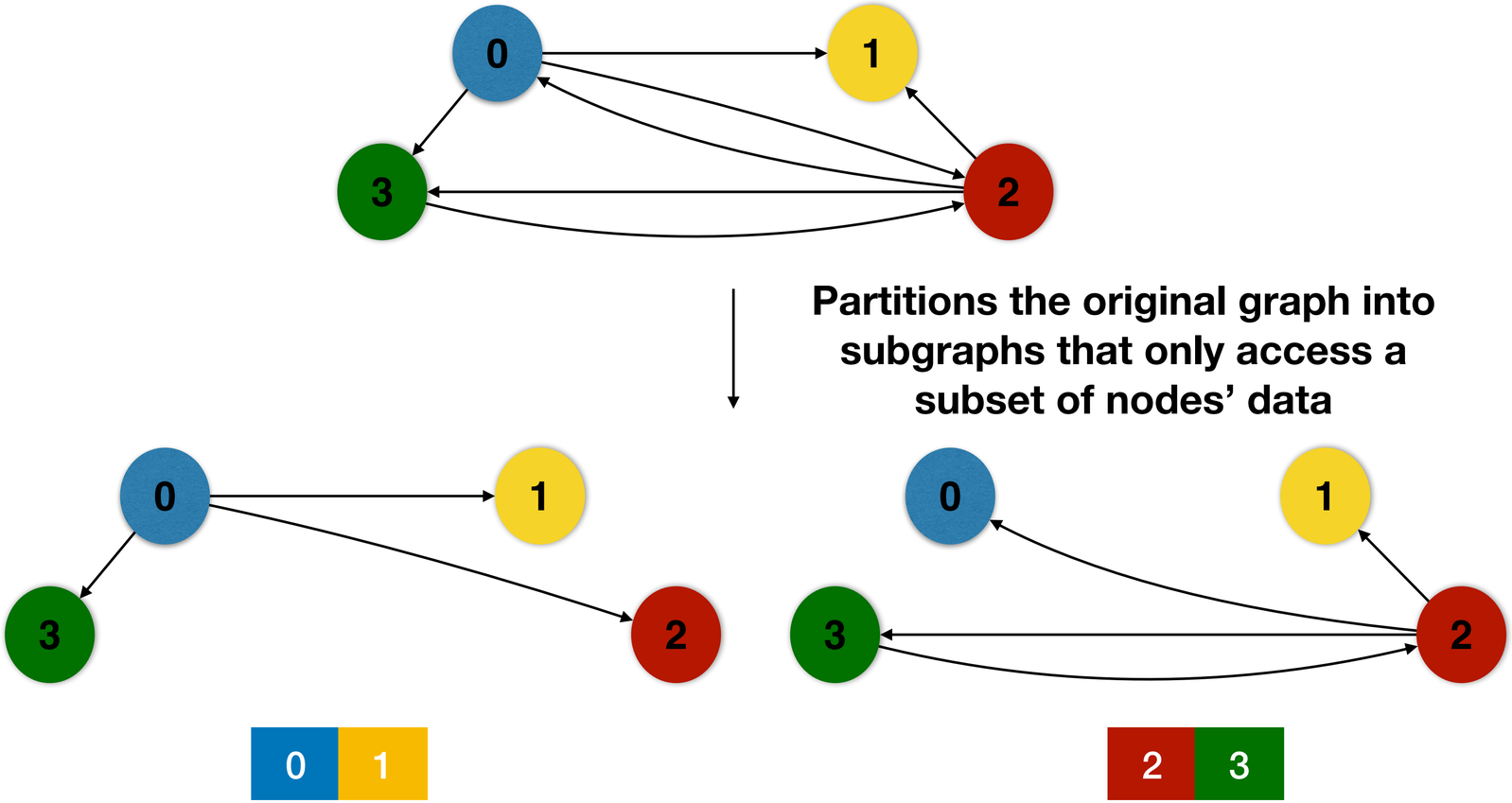


Partitions the original graph into subgraphs that only access a subset of nodes' data

Graph Partitioning



Graph Partitioning

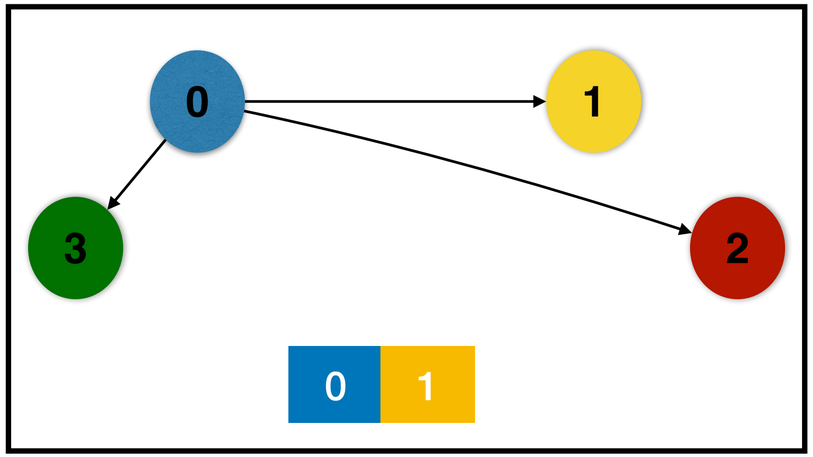


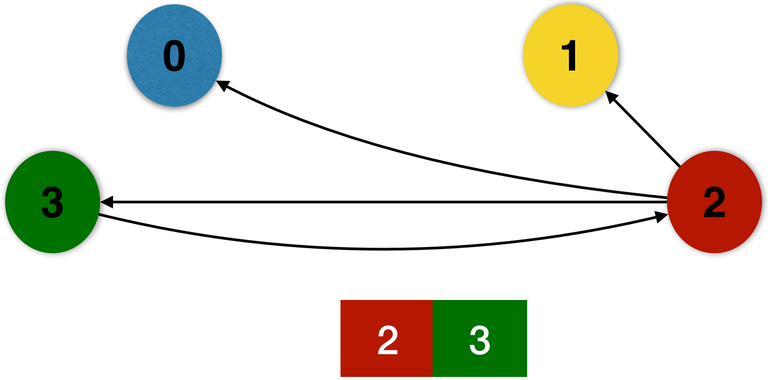
Graph Processing

Cache

#hits: 0

#misses: 0



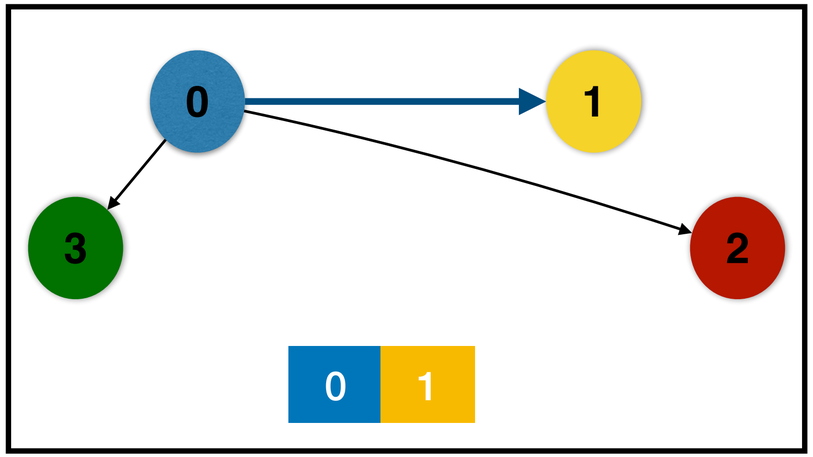


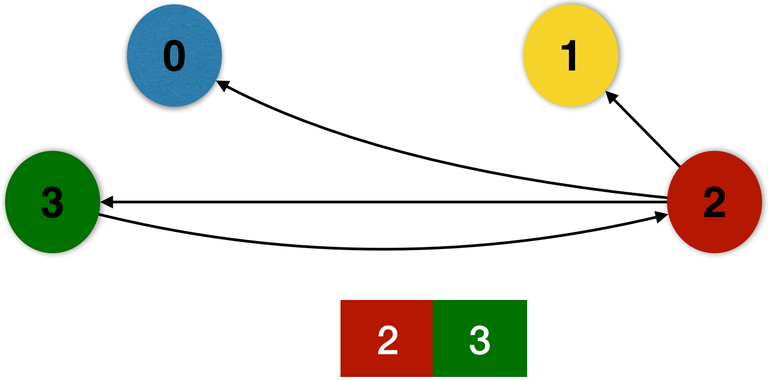
Graph Processing

Cache

#hits: 0

#misses: 0

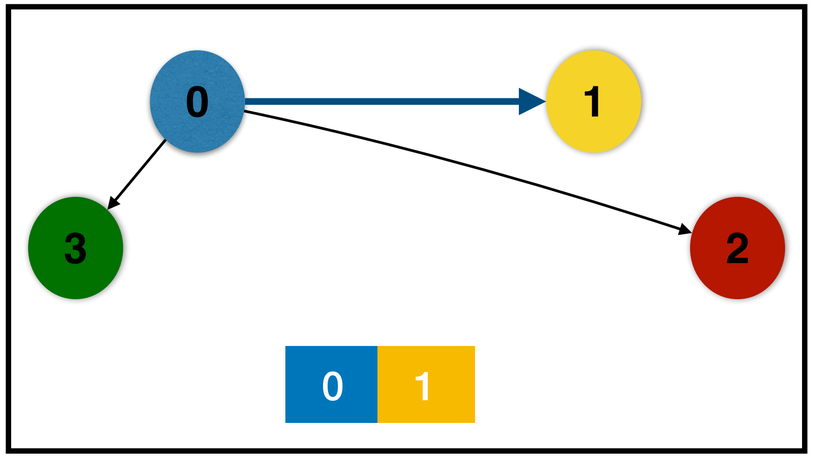


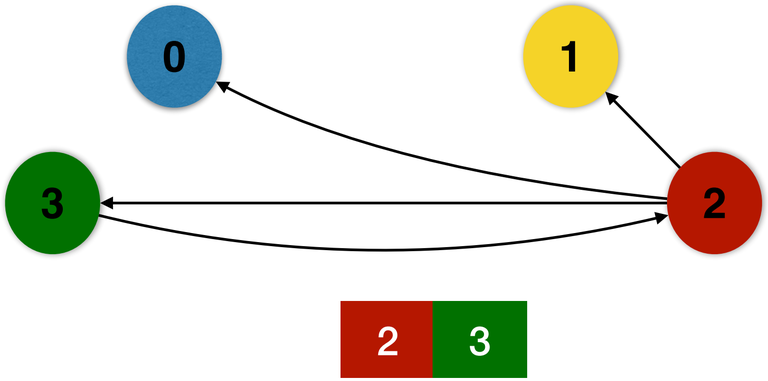


Graph Processing

Cache

#hits: 0
#misses: 1

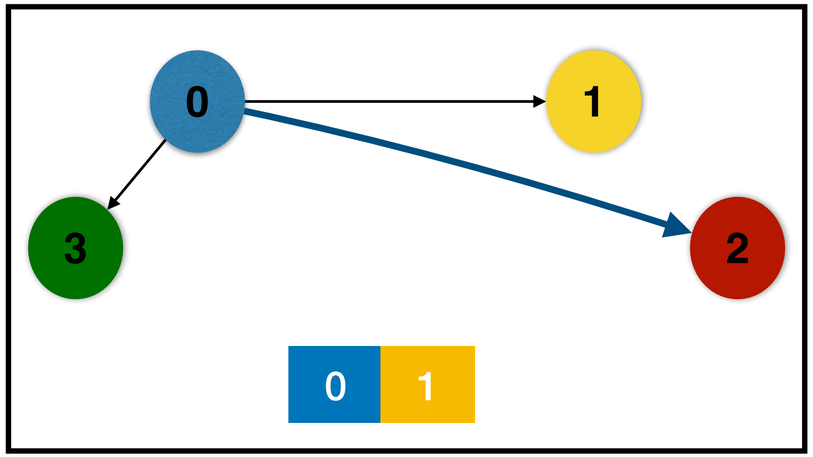


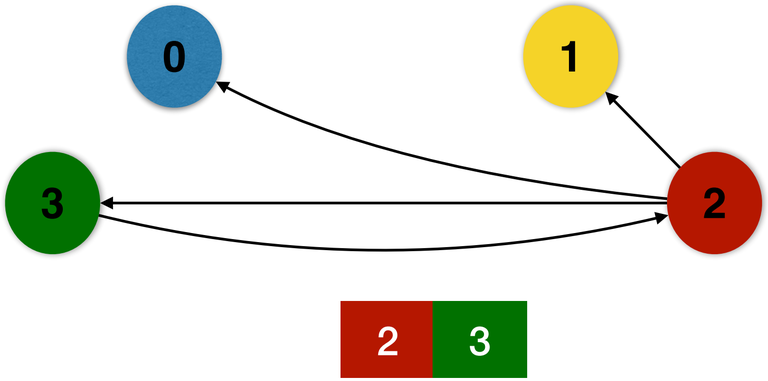


Graph Processing

Cache

#hits: 1
#misses: 1

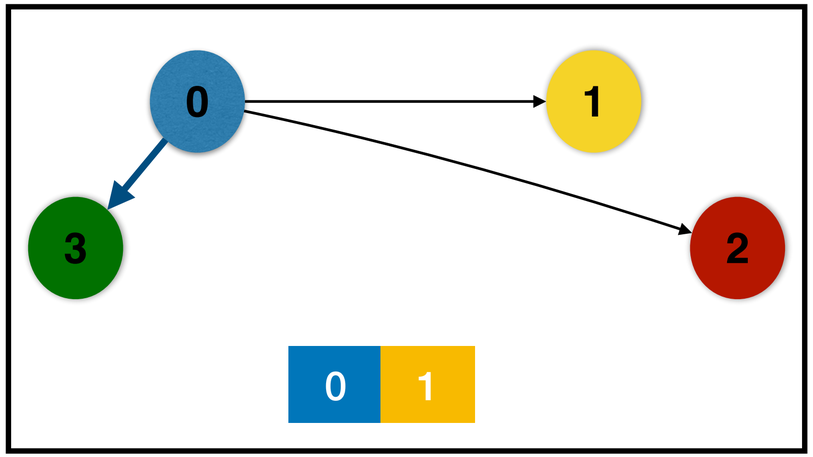


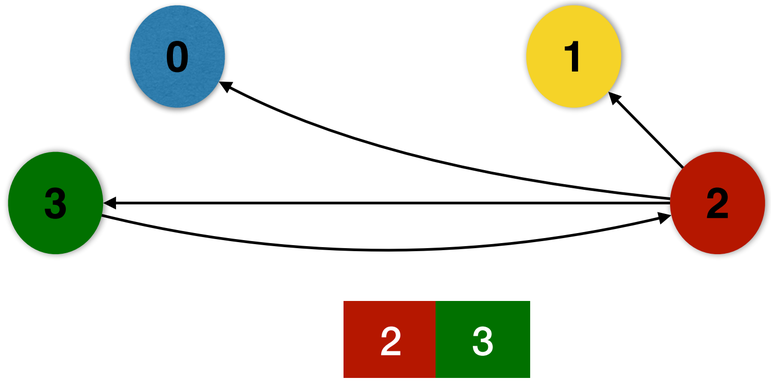


Graph Processing

Cache

#hits: 2
#misses: 1

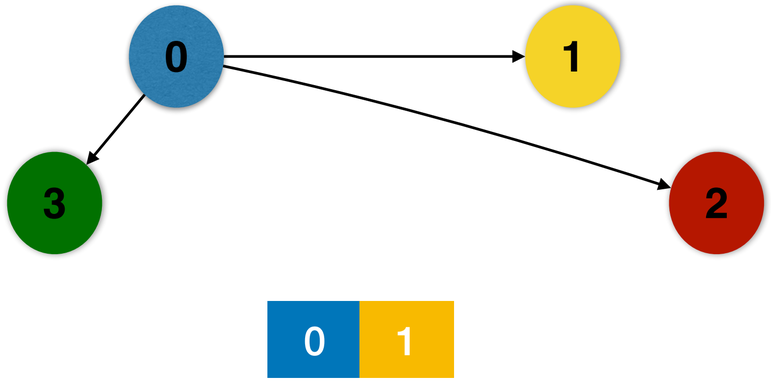


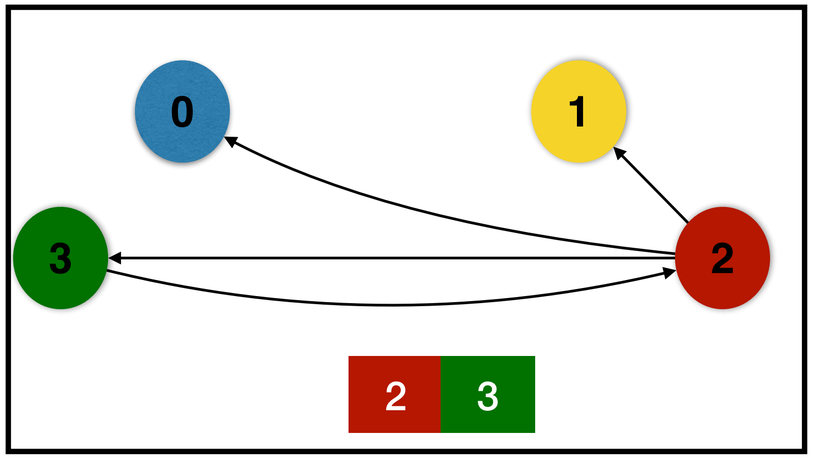


Graph Processing

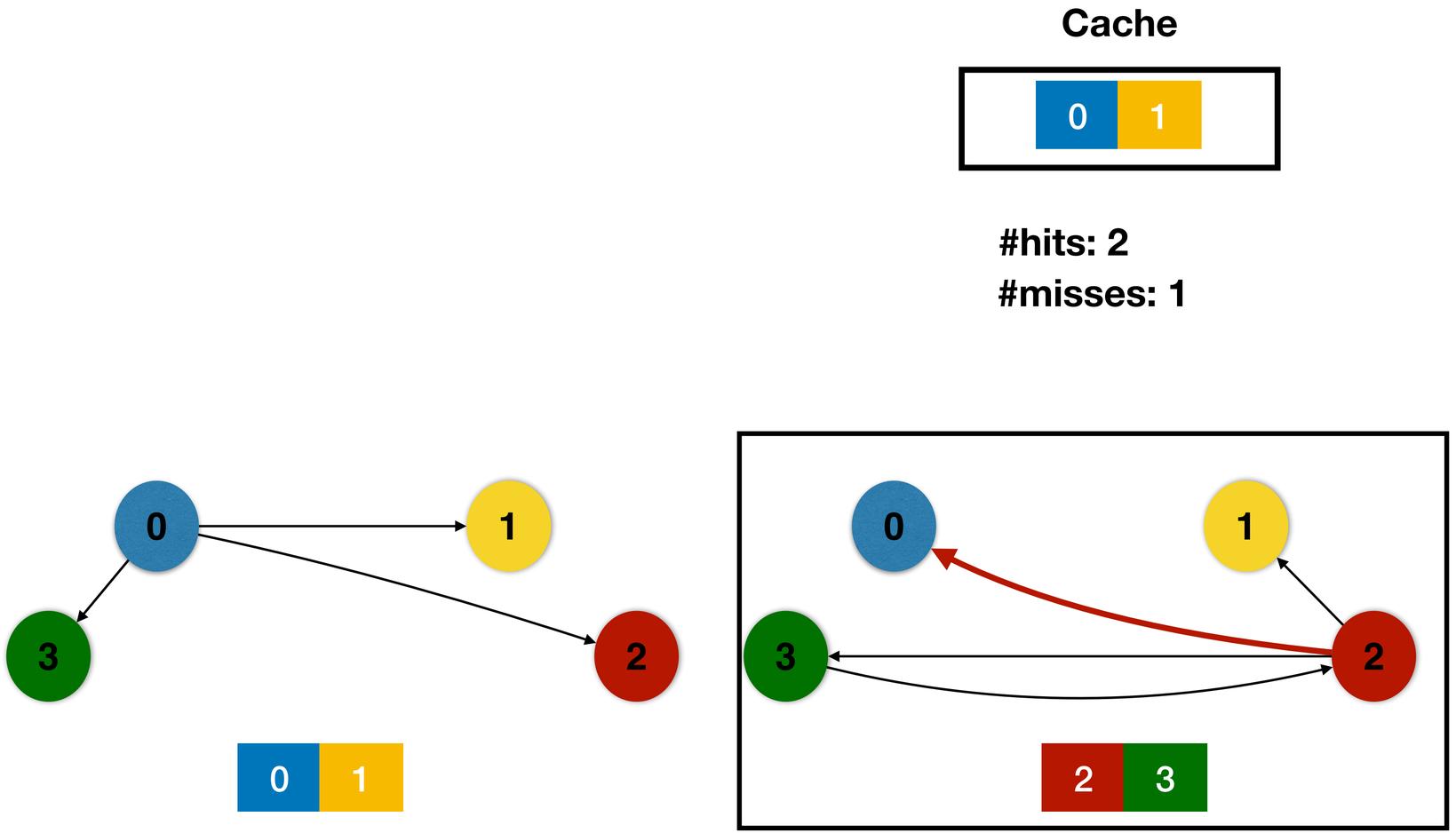
Cache

#hits: 2
#misses: 1

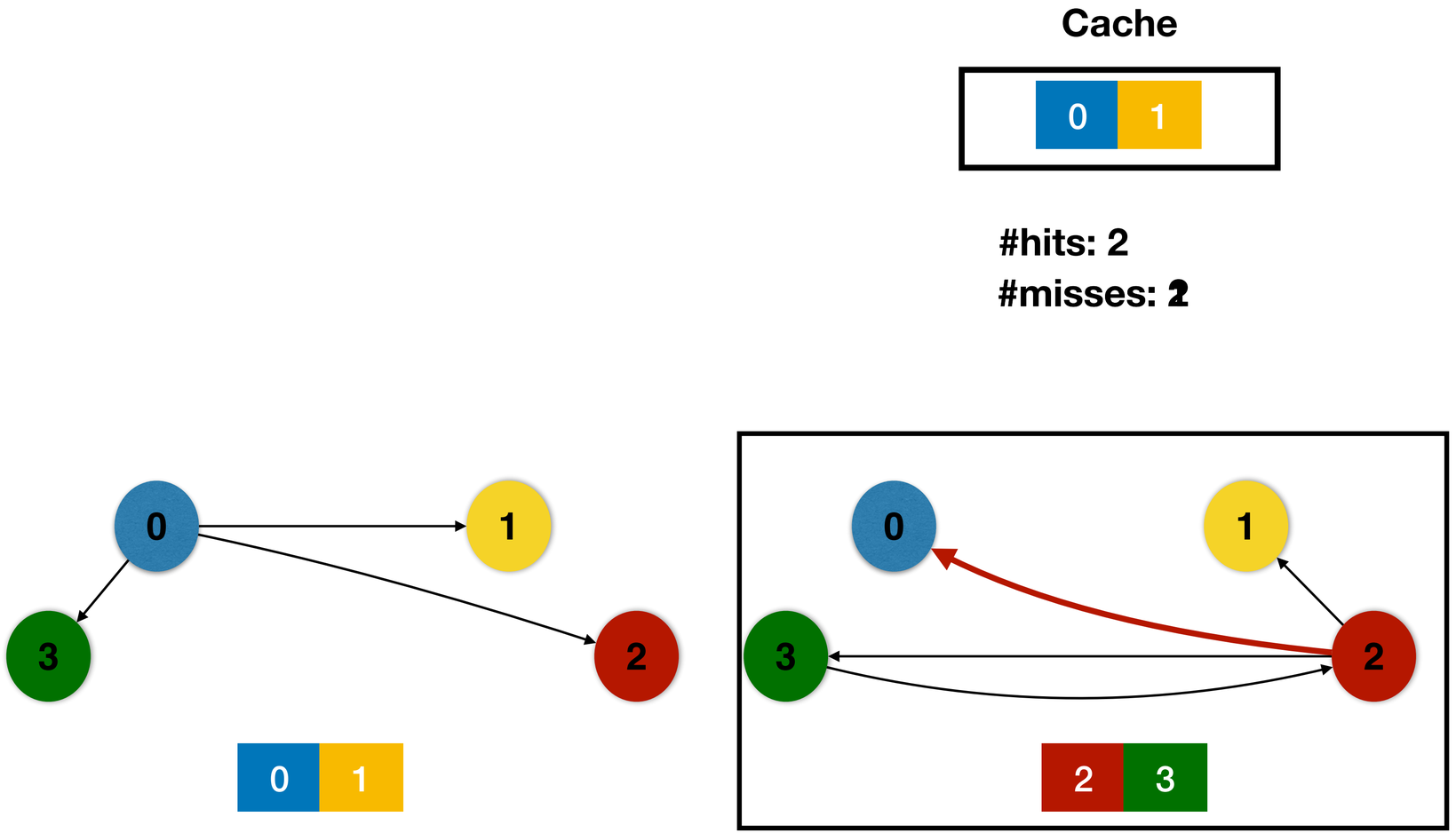




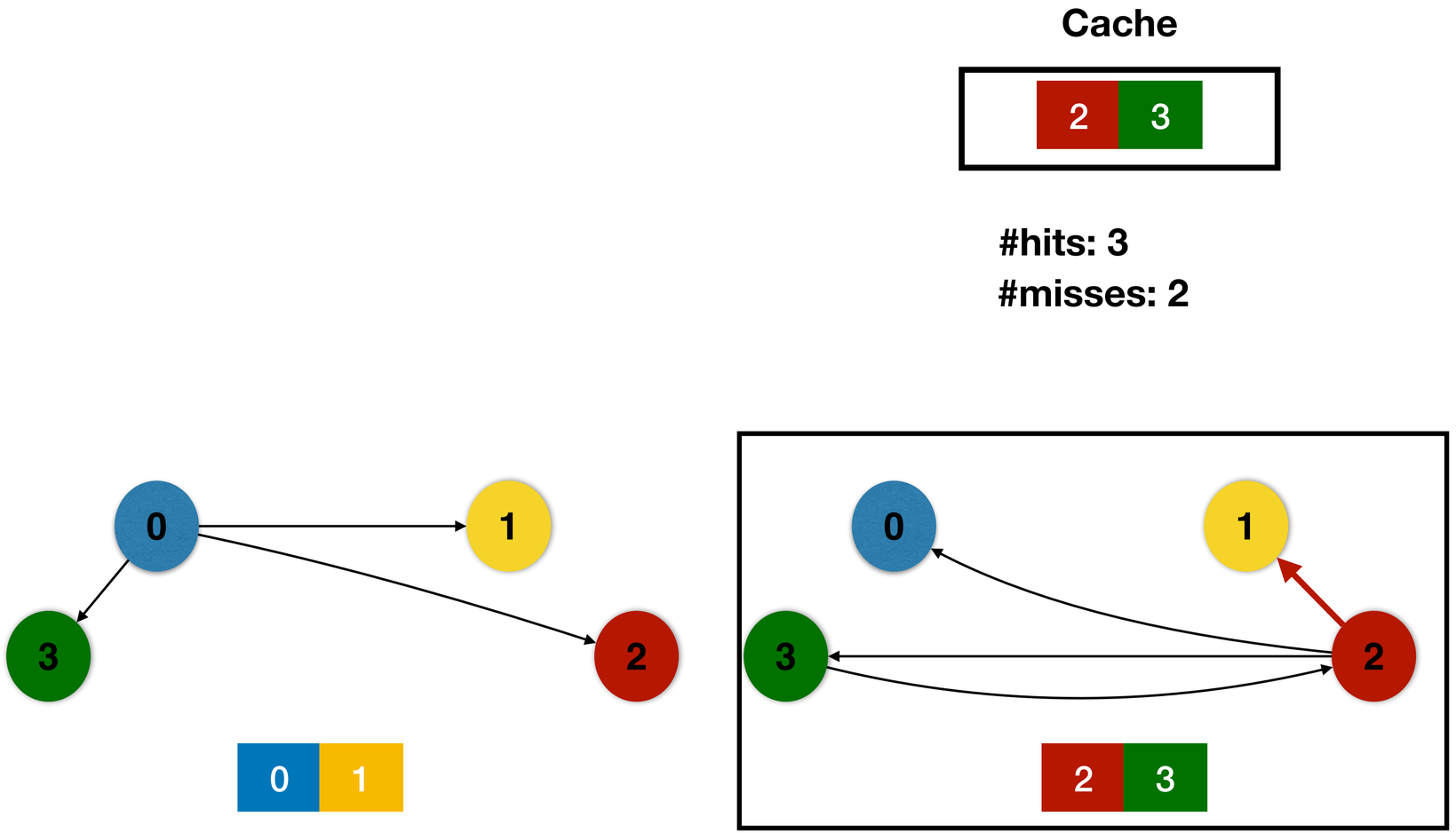
Graph Processing



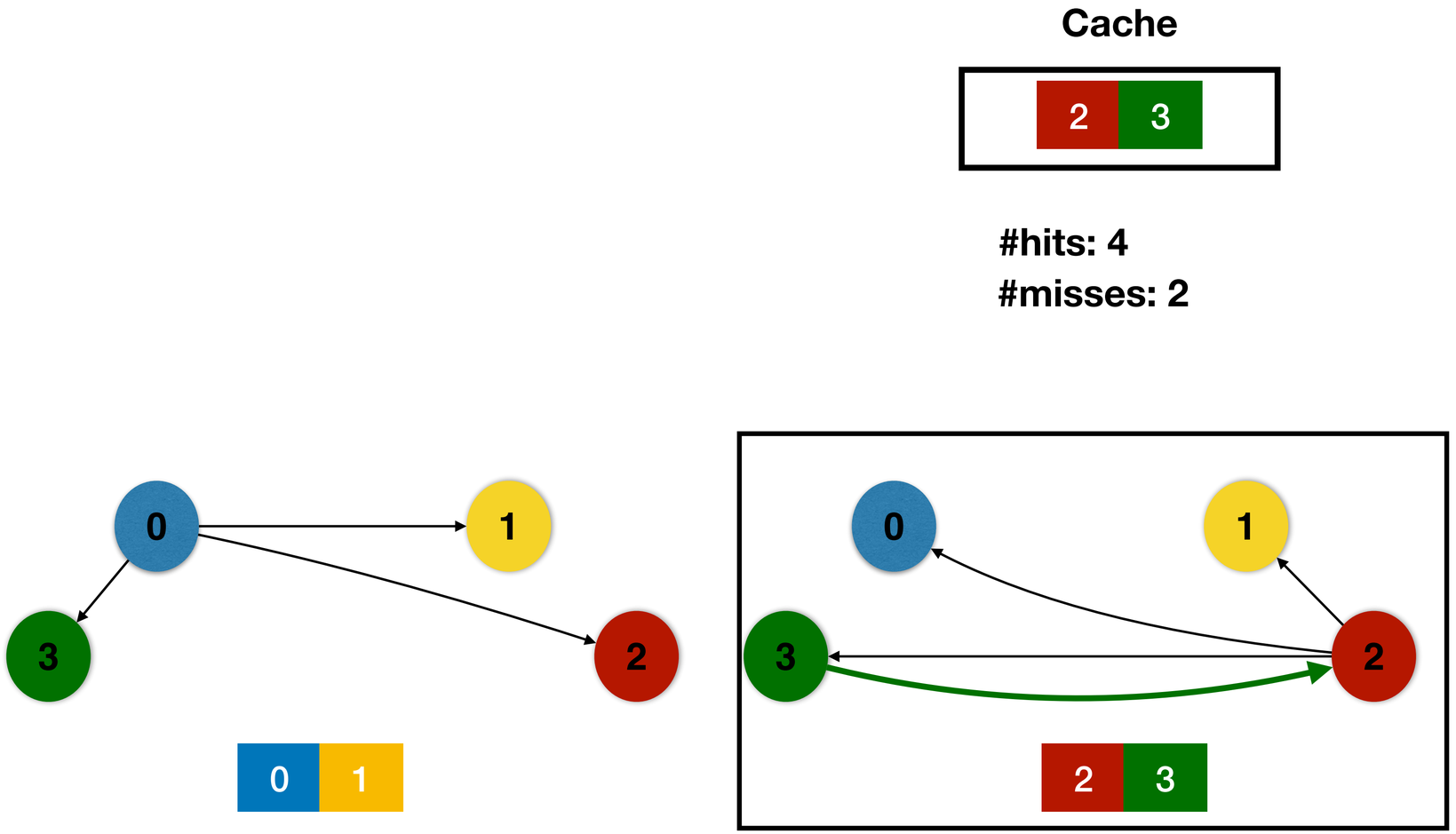
Graph Processing



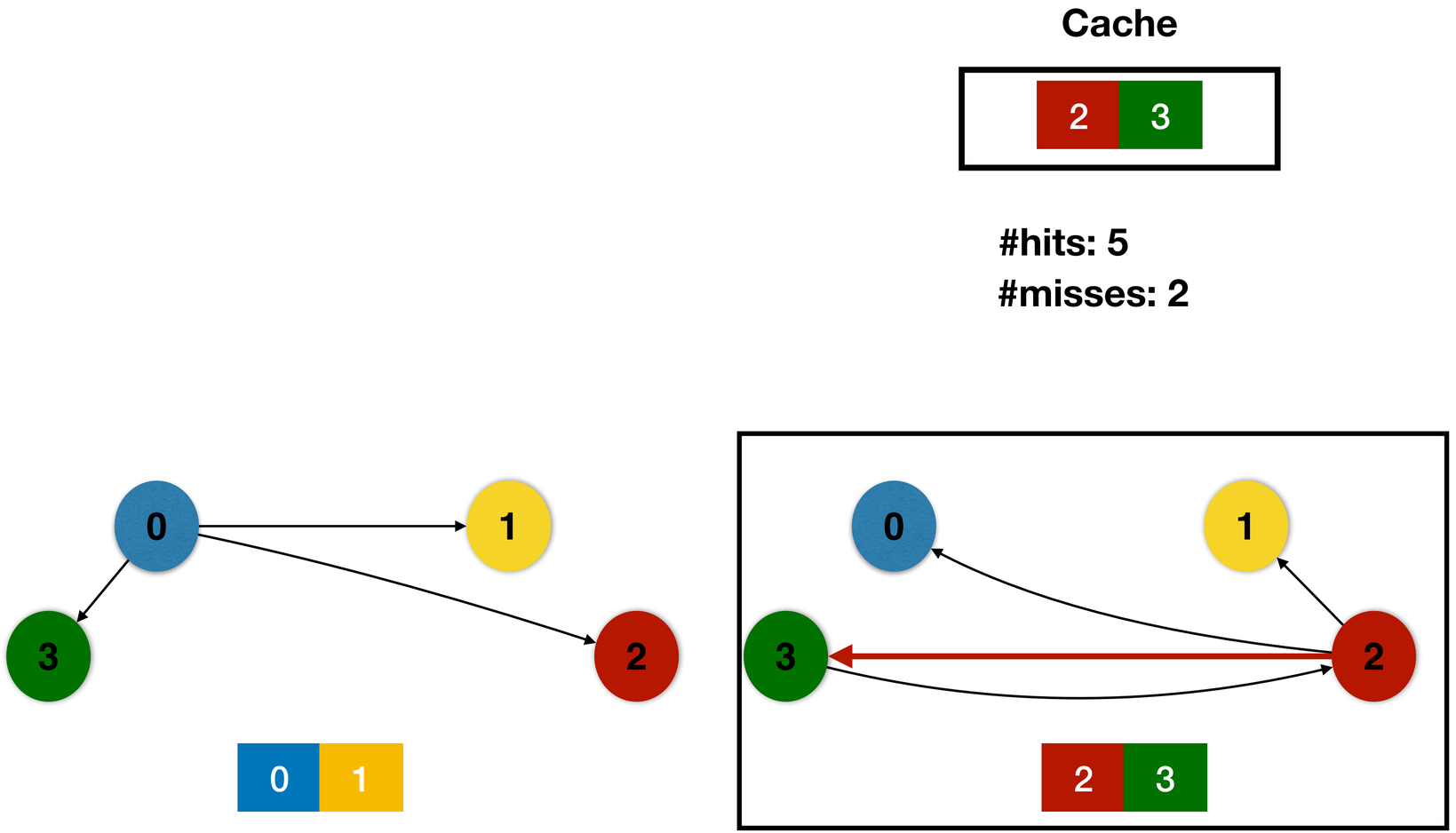
Graph Processing



Graph Processing



Graph Processing



Graph Processing