# Exploiting Locality in Graph Applications

Presenter: Yunming Zhang
(joint work with many others )

# Outline

- Overview
- Milk (PACT16, Vladimir Kiriansky et all)
- Cagra (BigData17, Yunming Zhang et all)
- GraphIt (OOPSLA18, Yunming Zhang et all)
- Conclusion

# Outline

- Overview
- Milk
- Cagra
- GraphIt
- Conclusion

# Outline

- Overview
- Milk
- Cagra
- GraphIt
- Conclusion

# Sparse Data Processing

- Sparse Data (Graphs, Sparse Matrices, Tensors) are Everywhere
- Difficult to Write High-Performance Implementations
  - Hard to Parallelize (Load Balance, Synchronizations …)
  - Hard to Exploit Locality (Blocking, CacheLine utilization …)

# Locality by the Numbers

```
local  L1 CACHE hit,                                    ~4 cycles (   2.1 -  1.2 ns )
local  L2 CACHE hit,                                   ~10 cycles (   5.3 -  3.0 ns )
local  L3 CACHE hit, line unshared                     ~40 cycles (  21.4 - 12.0 ns )
local  L3 CACHE hit, shared line in another core       ~65 cycles (  34.8 - 19.5 ns )
local  L3 CACHE hit, modified in another core          ~75 cycles (  40.2 - 22.5 ns )

remote L3 CACHE (Ref: Fig.1 [Pg. 5])          ~100-300 cycles ( 160.7 - 30.0 ns )

local  DRAM                                                          ~60 ns
remote DRAM                                                         ~100 ns
```

# Locality by the Numbers

```
local  L1 CACHE hit,                                    ~4 cycles (   2.1 -  1.2 ns )
local  L2 CACHE hit,                                   ~10 cycles (   5.3 -  3.0 ns )
local  L3 CACHE hit, line unshared                     ~40 cycles (  21.4 - 12.0 ns )
local  L3 CACHE hit, shared line in another core       ~65 cycles (  34.8 - 19.5 ns )
local  L3 CACHE hit, modified in another core          ~75 cycles (  40.2 - 22.5 ns )

remote L3 CACHE (Ref: Fig.1 [Pg. 5])          ~100-300 cycles ( 160.7 - 30.0 ns )

local  DRAM                                                          ~60 ns
remote DRAM                                                         ~100 ns
```

**Going to DRAM is usually 2.5-4x slower than L3 Cache**

# Today's Talks

- Milk (Exploit Locality through Runtime Memory Accesses Reordering)

- Cagra (Exploit Locality through Preprocessing)

- GraphIt (Explore the Tradeoff Space between Locality, Parallelism, and Work-Efficiency)

# Outline

- Overview
- Milk
- Cagra
- GraphIt
- Conclusion

# Optimizing Indirect Memory References with `milk`

*Vladimir Kiriansky,* Yunming Zhang, Saman Amarasinghe

MIT


**PACT** ’16

September 13, 2016, Haifa, Israel

# Indirect Accesses

```
for(int i=0; i<N; i++)
  count[d[i]]++;
```

# Indirect Accesses with OpenMP

```
01 #pragma omp parallel for
02 for(int i=0; i<N; i++)
03     #pragma omp atomic
04     count[d[i]]++;
```

# Indirect Accesses with OpenMP

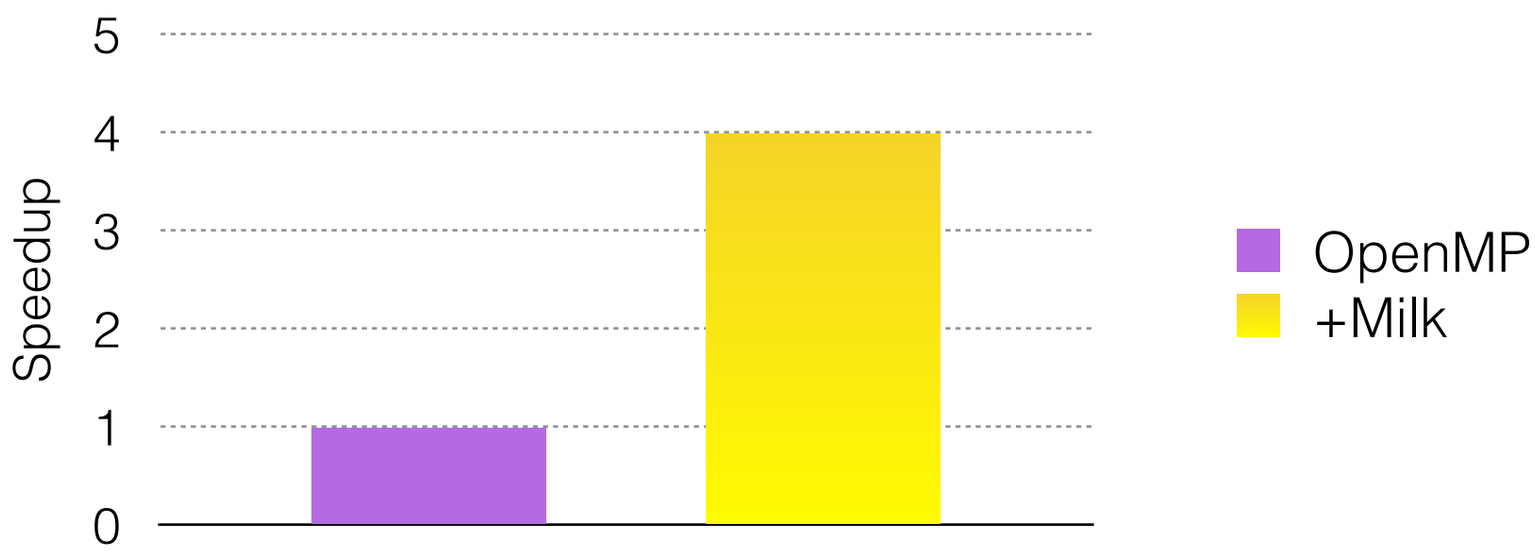
```
01 #pragma omp parallel for
02 for(int i=0; i<N; i++)
03     #pragma omp atomic
04     count[d[i]]++;
```


uniform [0..100M)

8 threads, 8MB L3

# Indirect Accesses with **milk**

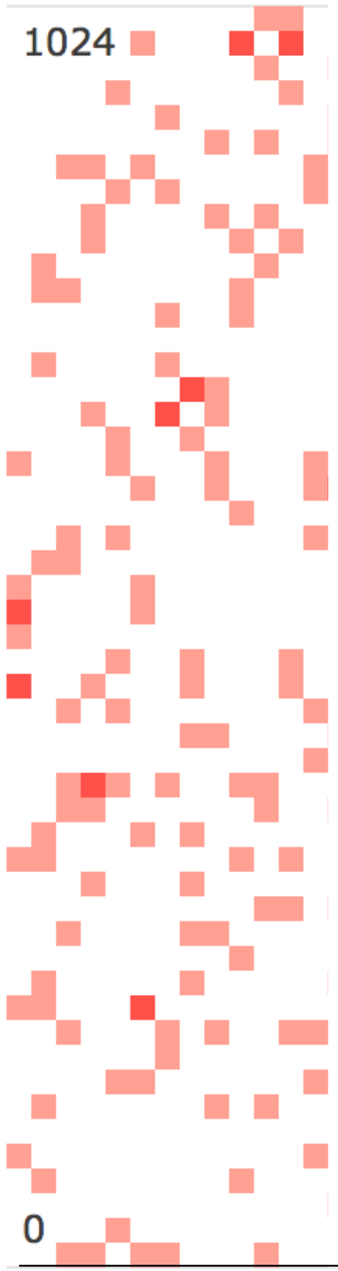
```
01 #pragma omp parallel for milk
02 for(int i=0; i<N; i++)
03     #pragma omp atomic   if(!milk)
04     count[d[i]]++;
```




uniform [0..100M)

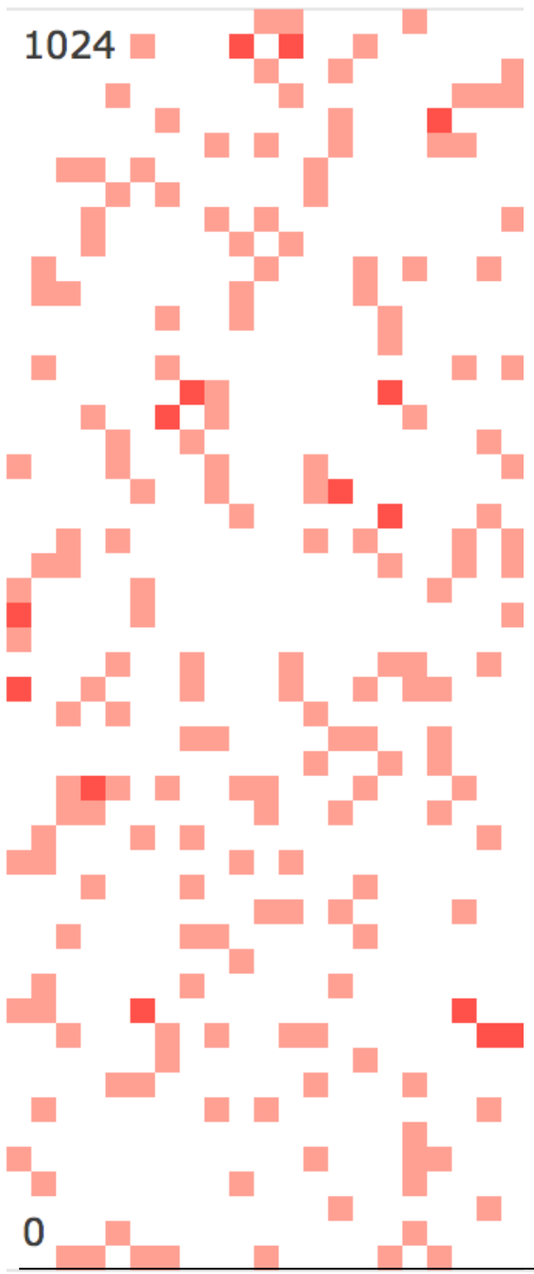
8 threads, 8MB L3

# No Locality?

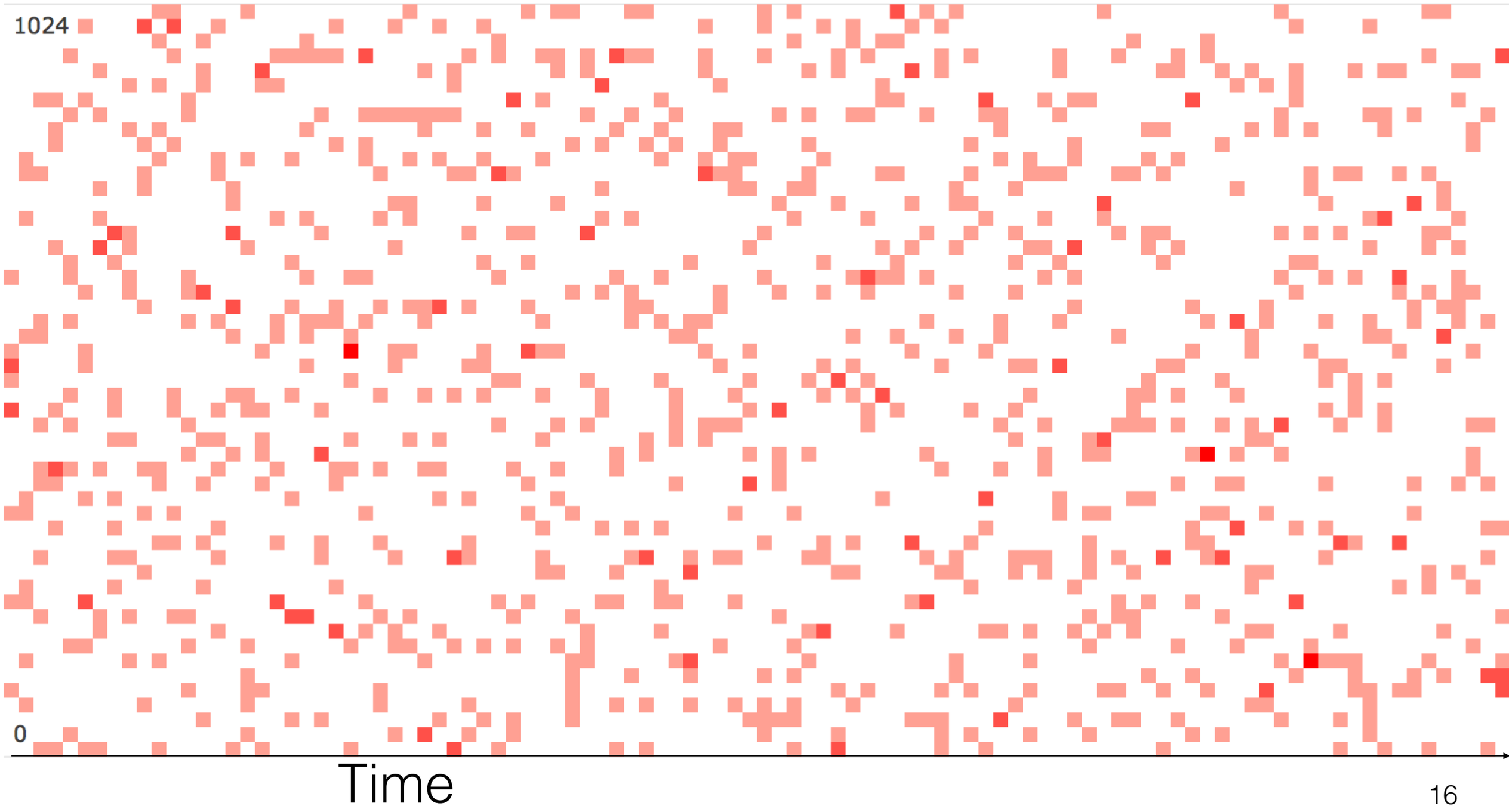



- Cache miss
- TLB miss
- DRAM row miss
- No prefetching

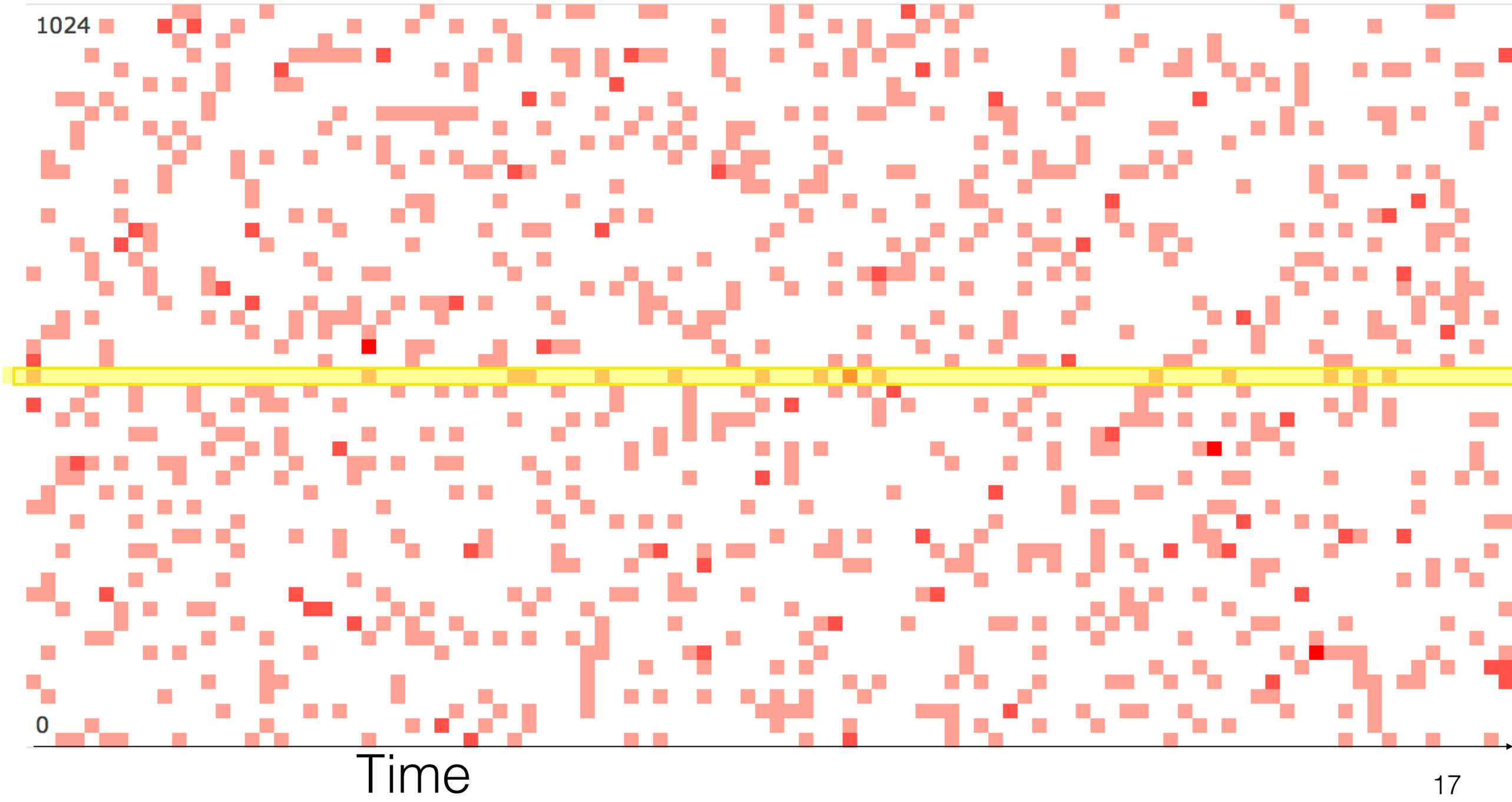
# No Locality?



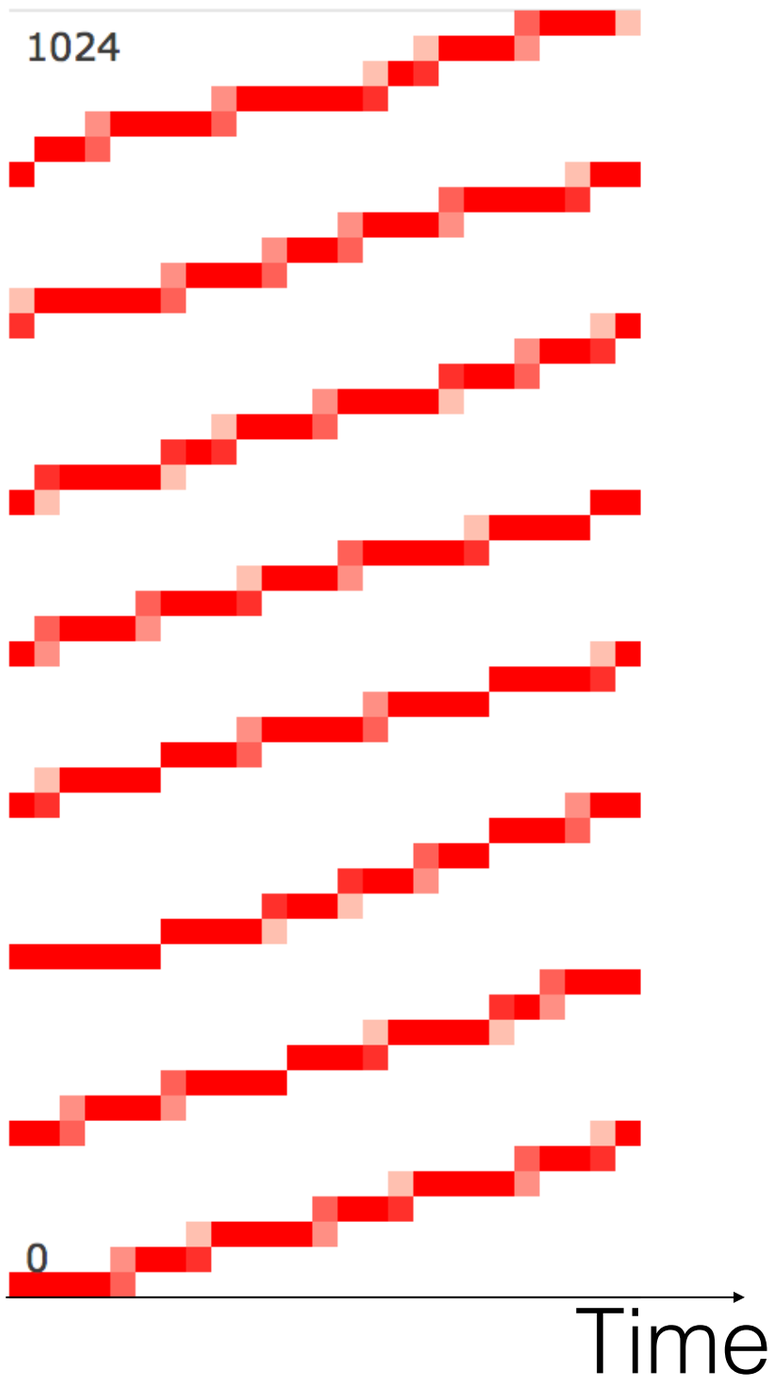
# No Locality?



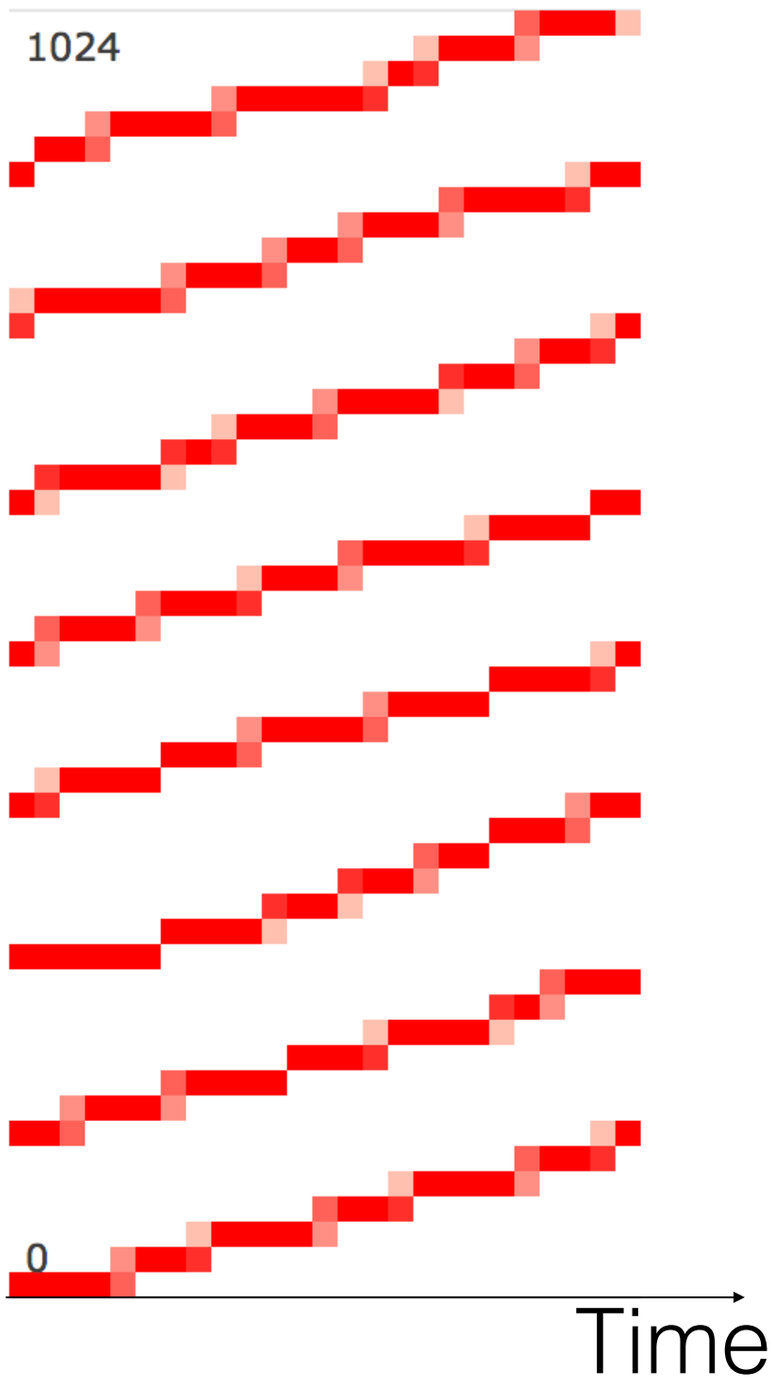
# No Locality?



# Milk Clustering

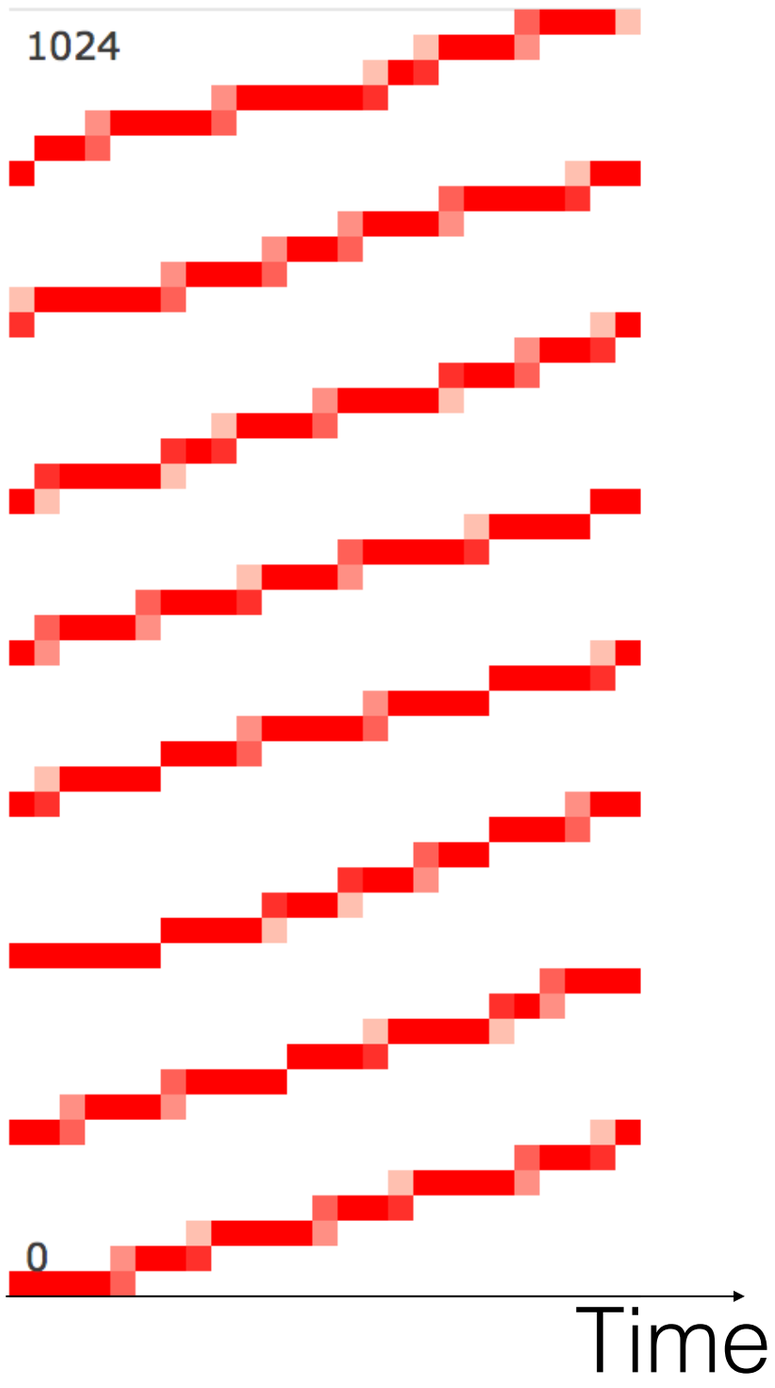


# Milk Clustering




- Cache hit
- TLB hit
- DRAM row hit
- Effective prefetching

# Milk Clustering




- Cache hit
- TLB hit
- DRAM row hit
- Effective prefetching
- No need for atomics!

# Outline

- Milk programming model


- **`milk`** syntax

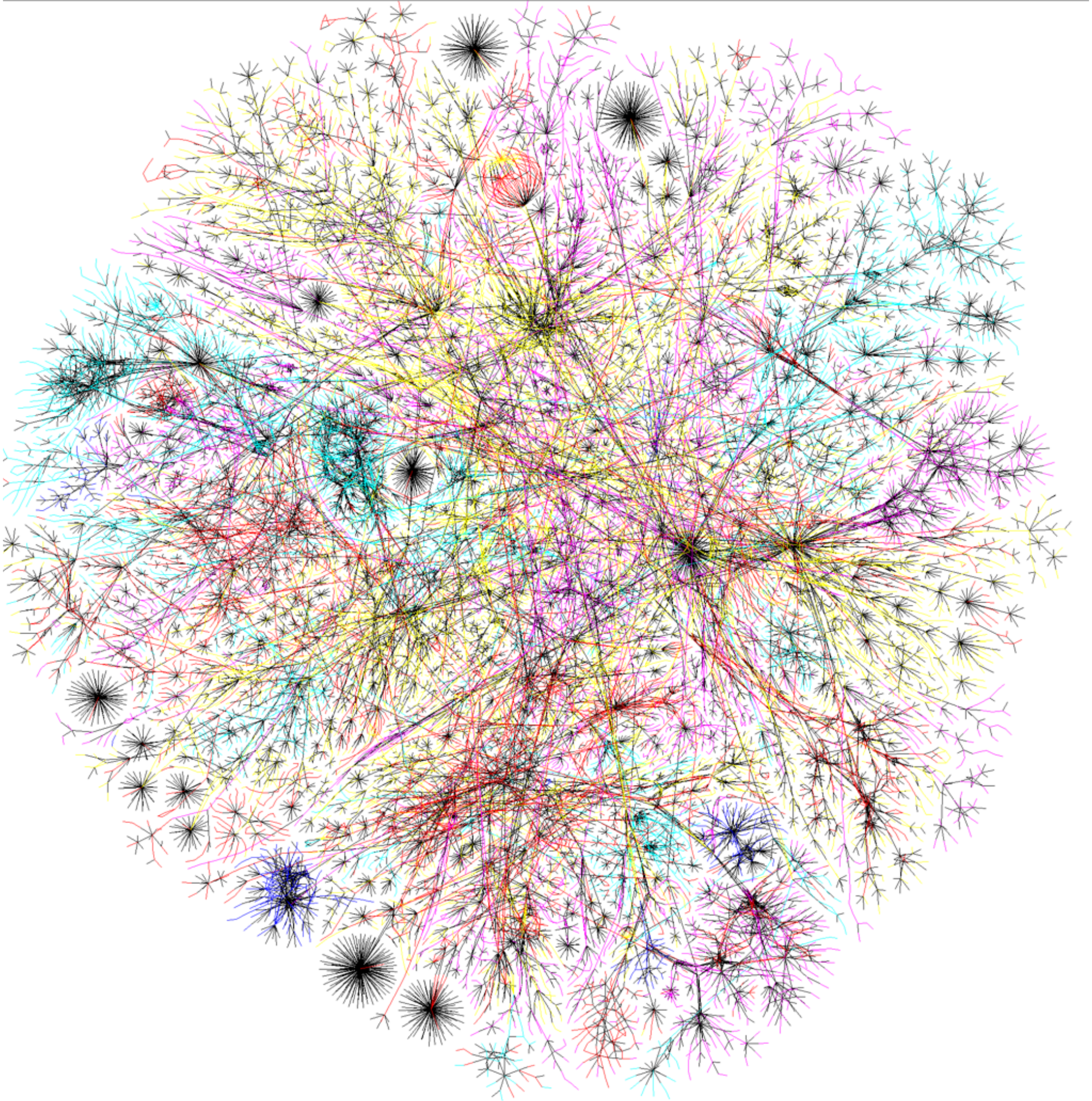
- MILK compiler and runtime

# Foundations

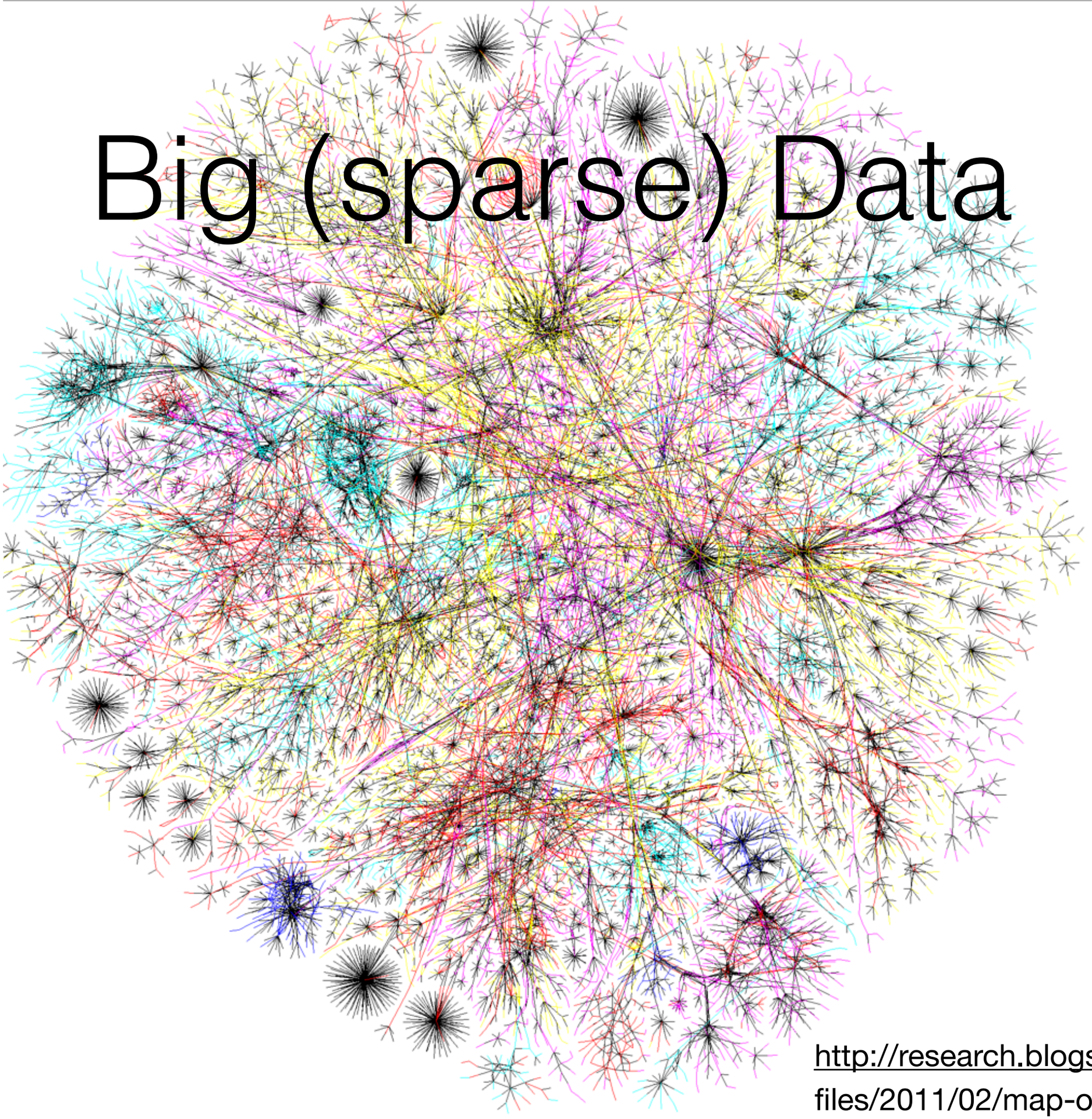
- Milk programming model — extending BSP

- **`milk`** syntax — OpenMP for C/C++

- **MILK** compiler and runtime — LLVM/Clang



# Big (sparse) Data



http://research.blogs.lincoln.ac.uk/files/2011/02/map-of-internet.png

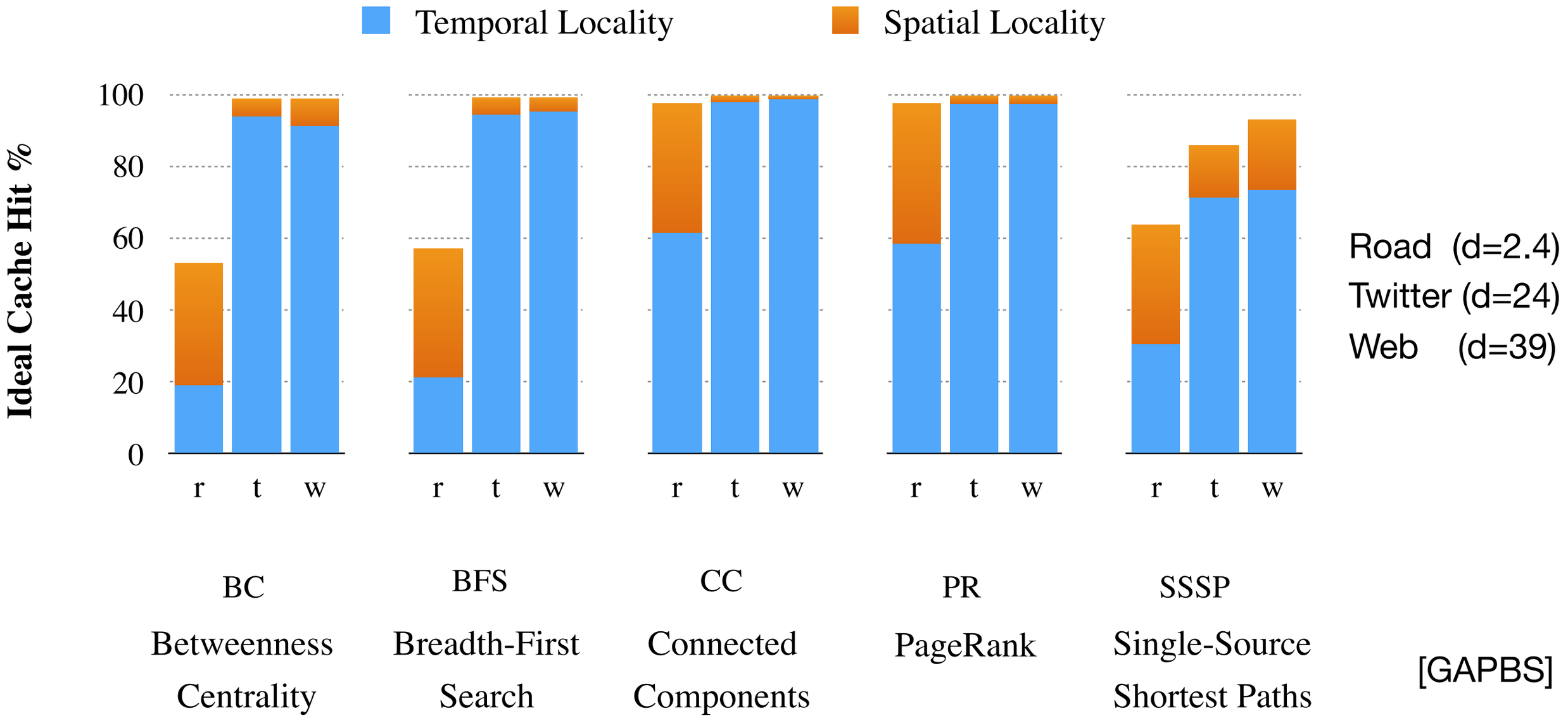
# Big (sparse) Data

- Terabyte Working Sets
  - AWS 2TB VM
- In-memory Databases, Key-value stores
- Machine Learning
- Graph Analytics

# Milk — BSP extension

- Bulk-synchronous parallel (BSP) *superstep*
  - updates visible after a barrier
- Virtual processors can access only
  - One random cache line from DRAM
  - Sequential streams
  - Cache-resident data

# Infinite Cache Locality in Graph Applications



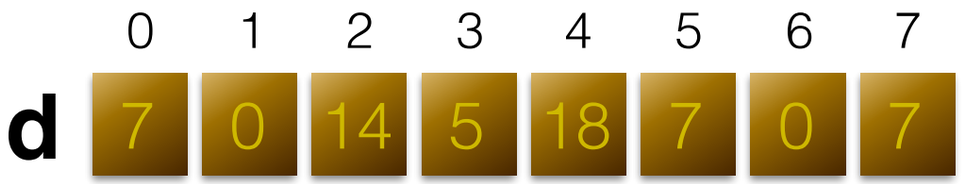

[GAPBS]

# Milk Execution Model

- Collection
- Distribution
- Delivery

# Collection

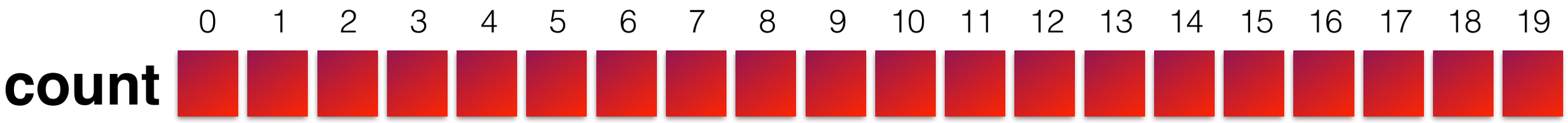
```
01 #pragma omp parallel for
02 for(int i=0; i<N; i++)
03     #pragma omp atomic
04     count[d[i]]+· += f(i);
```




# Collection

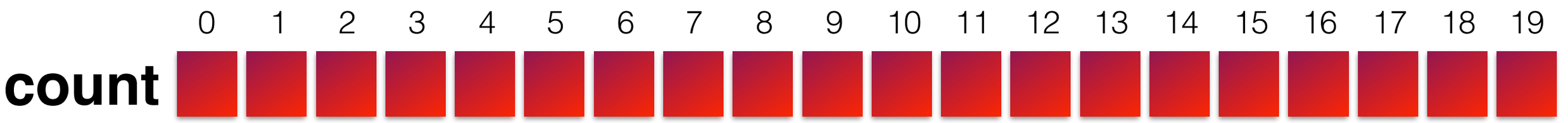
```
01 #pragma omp parallel for
02 for(int i=0; i<N; i++)
03     #pragma omp atomic
04     count[d[i]]+· += f(i);
```





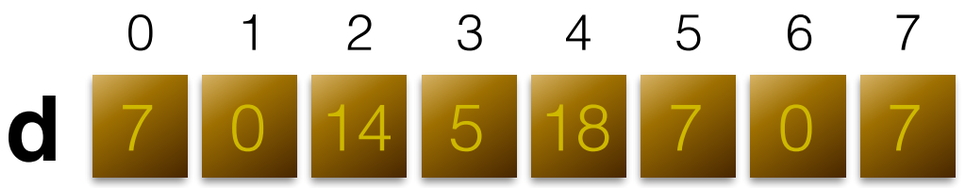
# Distribution

```
01 #pragma omp parallel for
02 for(int i=0; i<N; i++)
03     #pragma omp atomic
04     count[d[i]]+· += f(i);
```

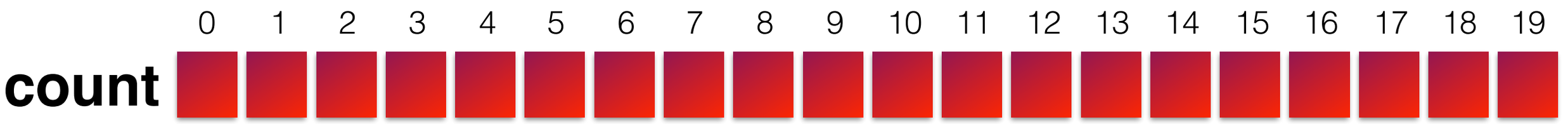



# Distribution

```
01 #pragma omp parallel for
02 for(int i=0; i<N; i++)
03     #pragma omp atomic
04     count[d[i]]+· += f(i);
```







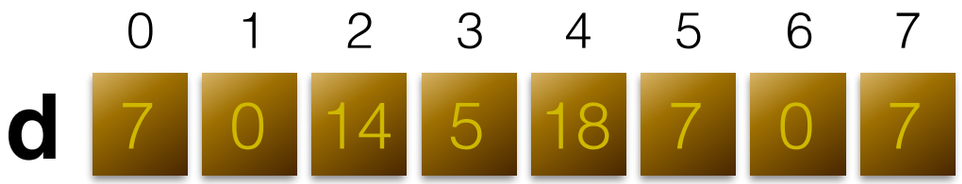
# Delivery

```
01 #pragma omp parallel for
02 for(int i=0; i<N; i++)
03     #pragma omp atomic
04     count[d[i]]+· += f(i);
```



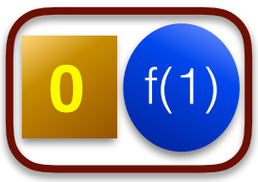
# Delivery

```
01 #pragma omp parallel for
02 for(int i=0; i<N; i++)
03     #pragma omp atomic
04     count[d[i]]+· += f(i);
```




# `milk` syntax

- **`milk`** clause in parallel loop
- **`milk`** directive per indirect access
  - `tag(i)` — address to group by
  - `pack(v)` — additional state

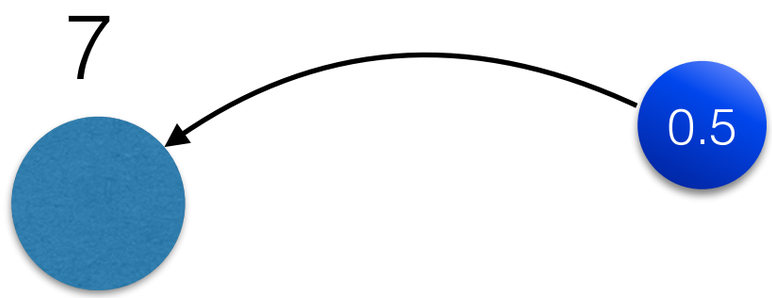


# `pack` Combiners

```
pack(v[:all])
pack(v:+|*|min|max|any)
```

# PageRank

```
vector<float> contrib, new_rank;

void PageRank_Push() {
  for (Node u=0; u < g.num_nodes(); u++) {
    float contribU = contrib[u];
    for (Node v : g.out_neigh(u))


      new_rank[v] += contribU;
  }
}
```



# PageRank with OpenMP

```
vector<float> contrib, new_rank;

void PageRank_Push() {
#pragma omp parallel for
  for (Node u=0; u < g.num_nodes(); u++) {
    float contribU = contrib[u];
    for (Node v : g.out_neigh(u))

#pragma omp atomic
      new_rank[v] += contribU;
  }
}
```

# PageRank with **milk**

```
vector<float> contrib, new_rank;

void PageRank_Push() {
#pragma omp parallel for milk
  for (Node u=0; u < g.num_nodes(); u++) {
    float contribU = contrib[u];
    for (Node v : g.out_neigh(u))

#pragma omp atomic if(!milk)
      new_rank[v] += contribU;
  }
}
```

# PageRank with **milk**

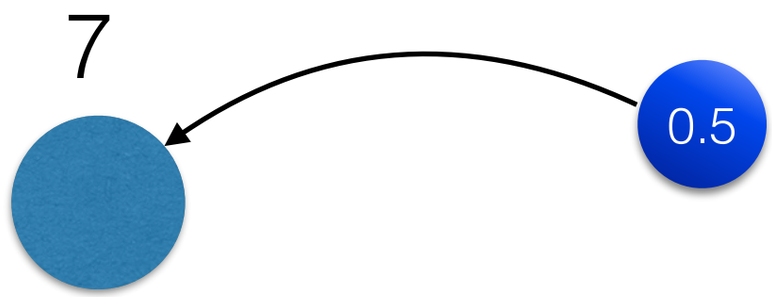
```
vector<float> contrib, new_rank;

void PageRank_Push() {
#pragma omp parallel for milk
  for (Node u=0; u < g.num_nodes(); u++) {
    float contribU = contrib[u];
    for (Node v : g.out_neigh(u))
#pragma milk pack(contribU : +) tag(v)
#pragma omp atomic if(!milk)
      new_rank[v] += contribU;
  }
}
```

# **MILK** compiler and runtime

- Collection — loop transformation
- Distribution — runtime library
- Delivery — continuation

# PageRank with **milk**

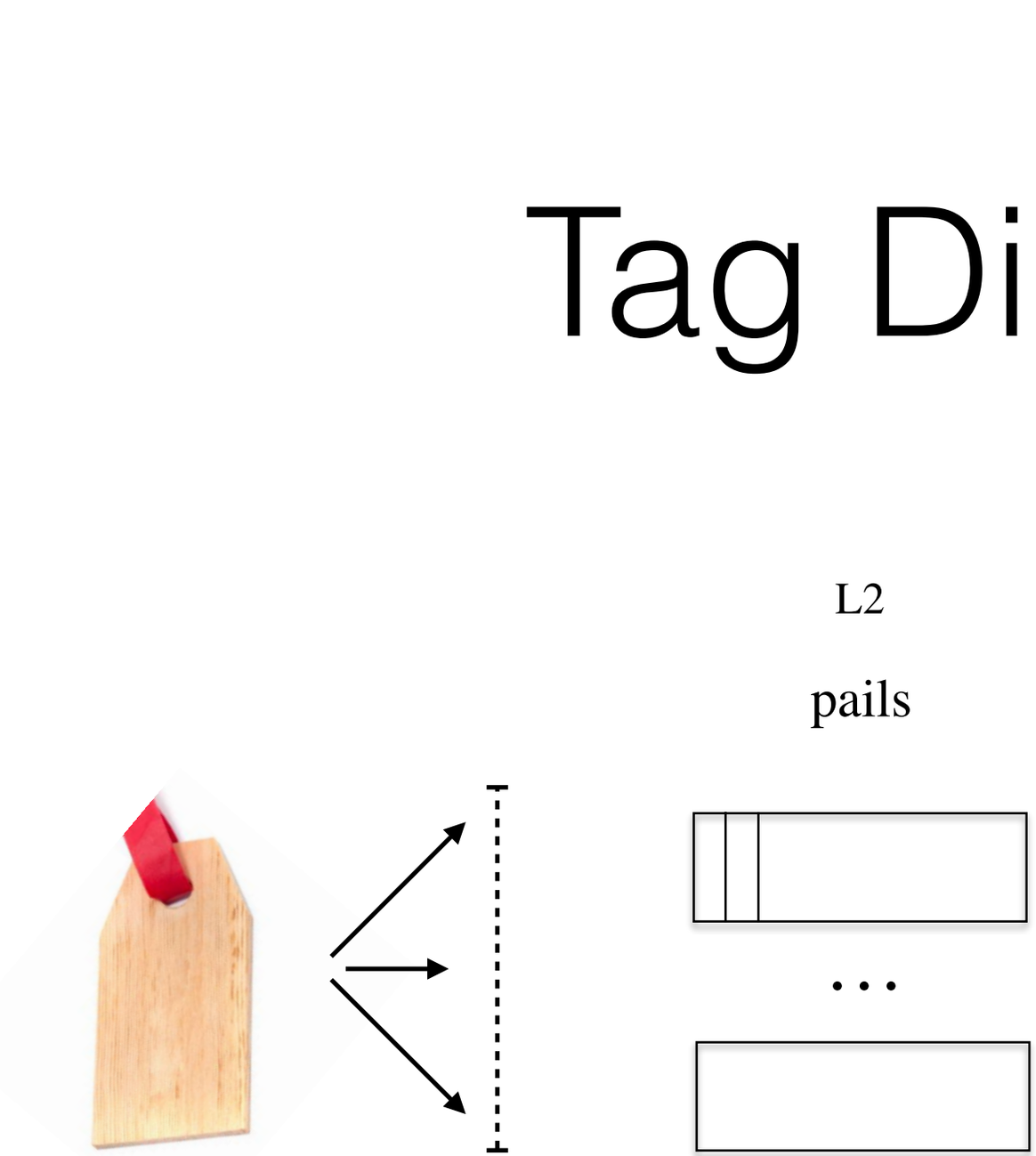
```
vector<float> contrib, new_rank;

void PageRank_Push() {
#pragma omp parallel for milk
  for (Node u=0; u < g.num_nodes(); u++) {
    float contribU = contrib[u];
    for (Node v : g.out_neigh(u))
#pragma milk pack(contribU : +) tag(v)
#pragma omp atomic if(!milk)
      new_rank[v] += contribU;
  }
}
```



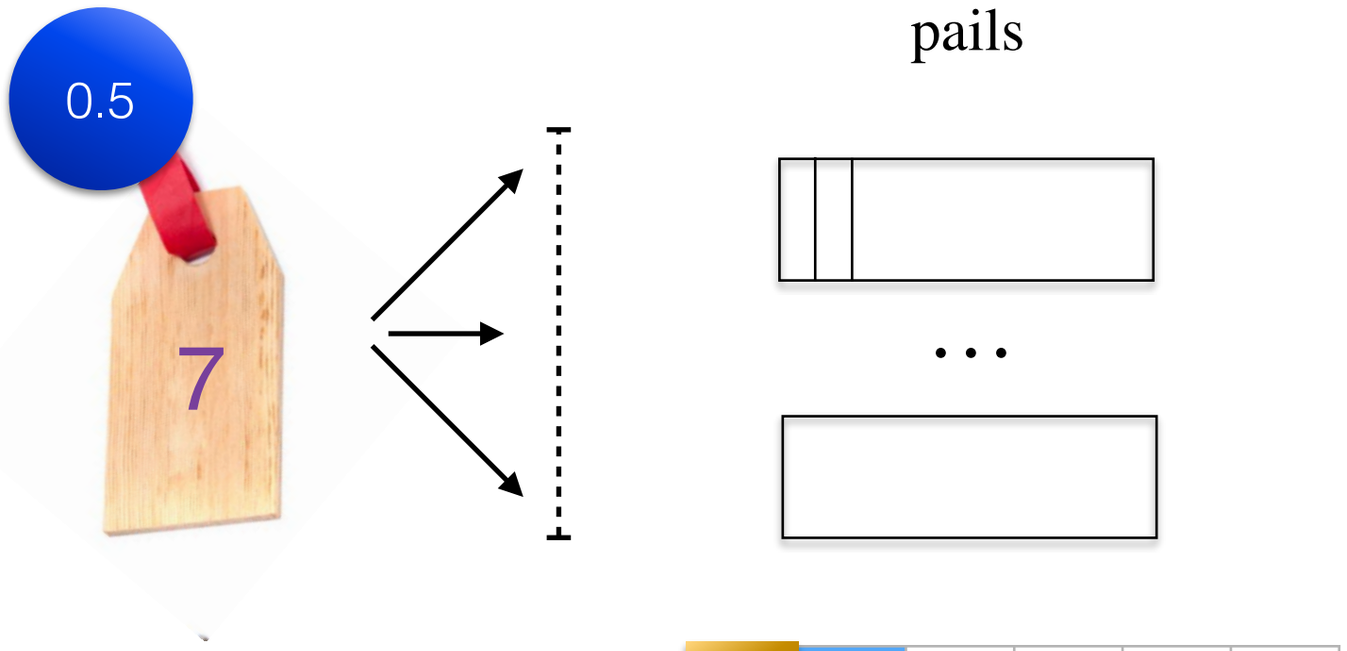
# PageRank: Collection

```
vector<float> contrib, new_rank;

void PageRank_Push() {
#pragma omp parallel for milk
  for (Node u=0; u < g.num_nodes(); u++) {
    float contribU = contrib[u];
    for (Node v : g.out_neigh(u))
#pragma milk pack(contribU : +) tag(v)


  }
}
```

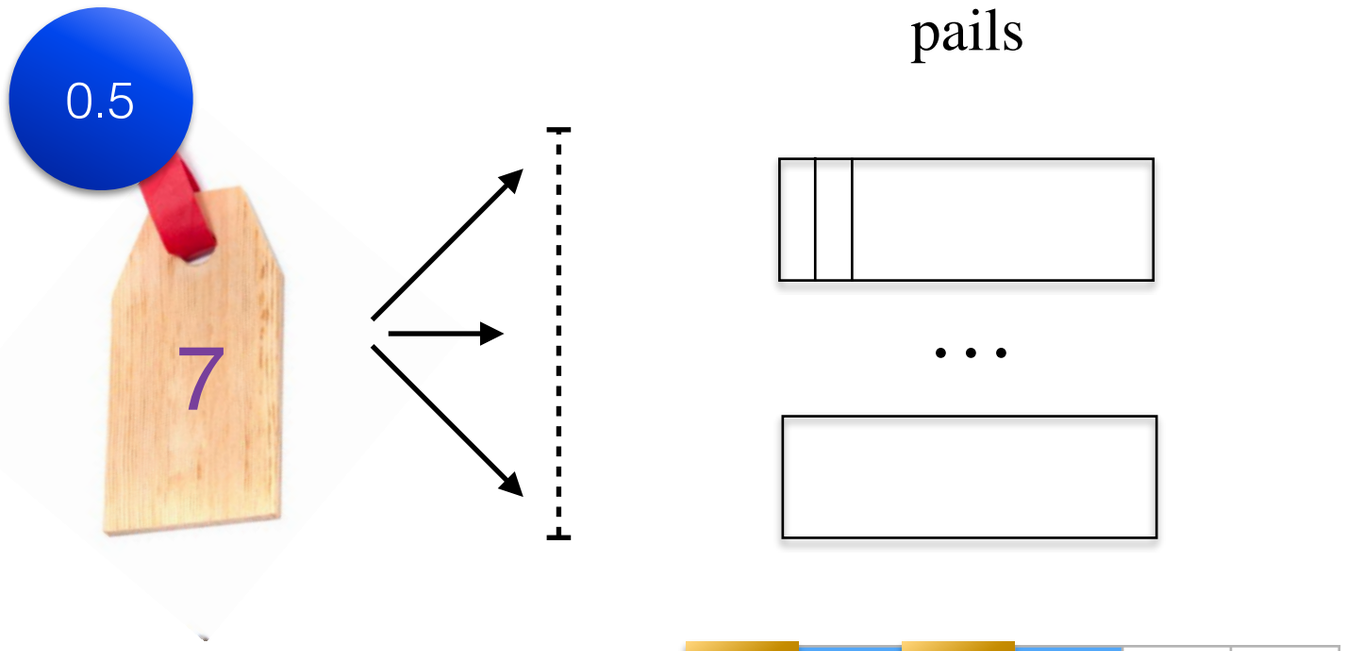
# Tag Distribution



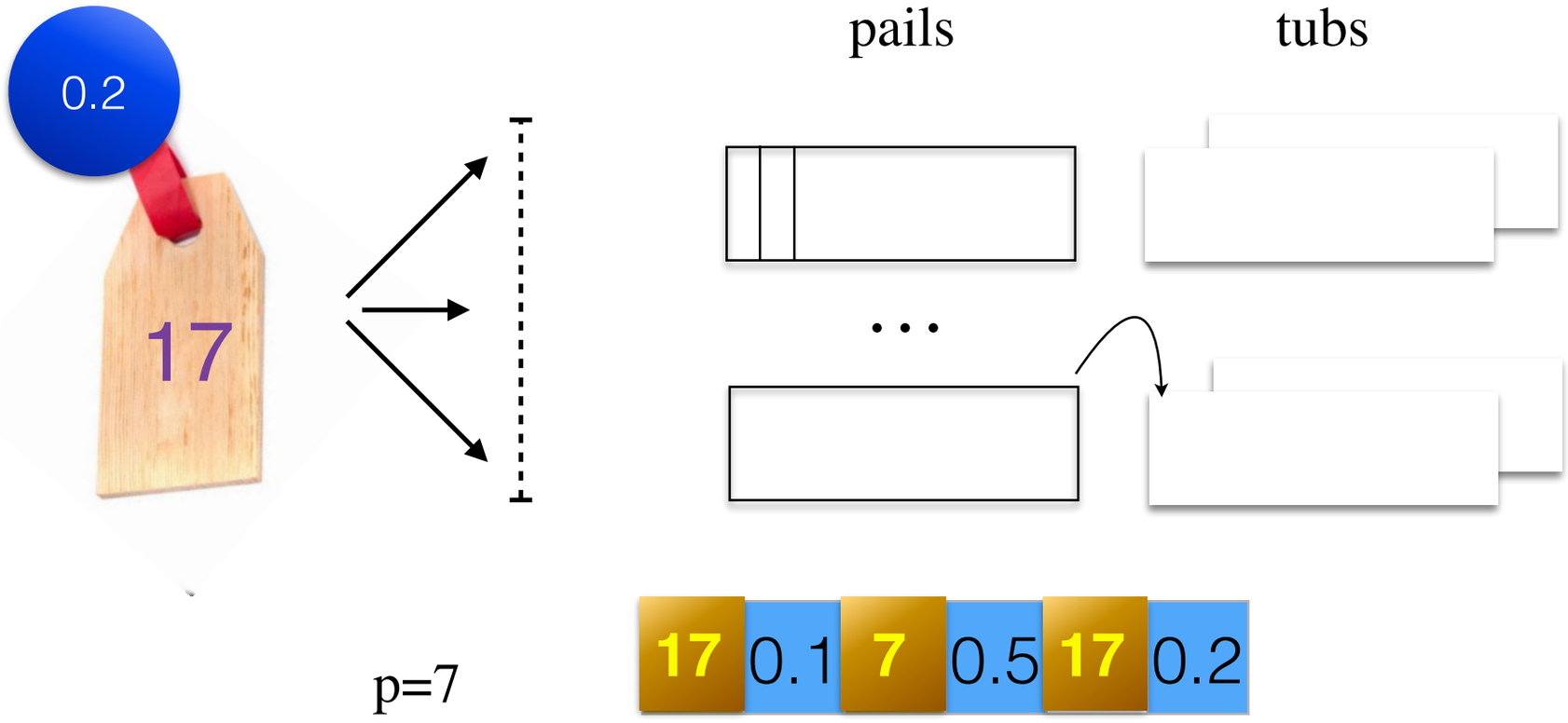
# Tag Distribution



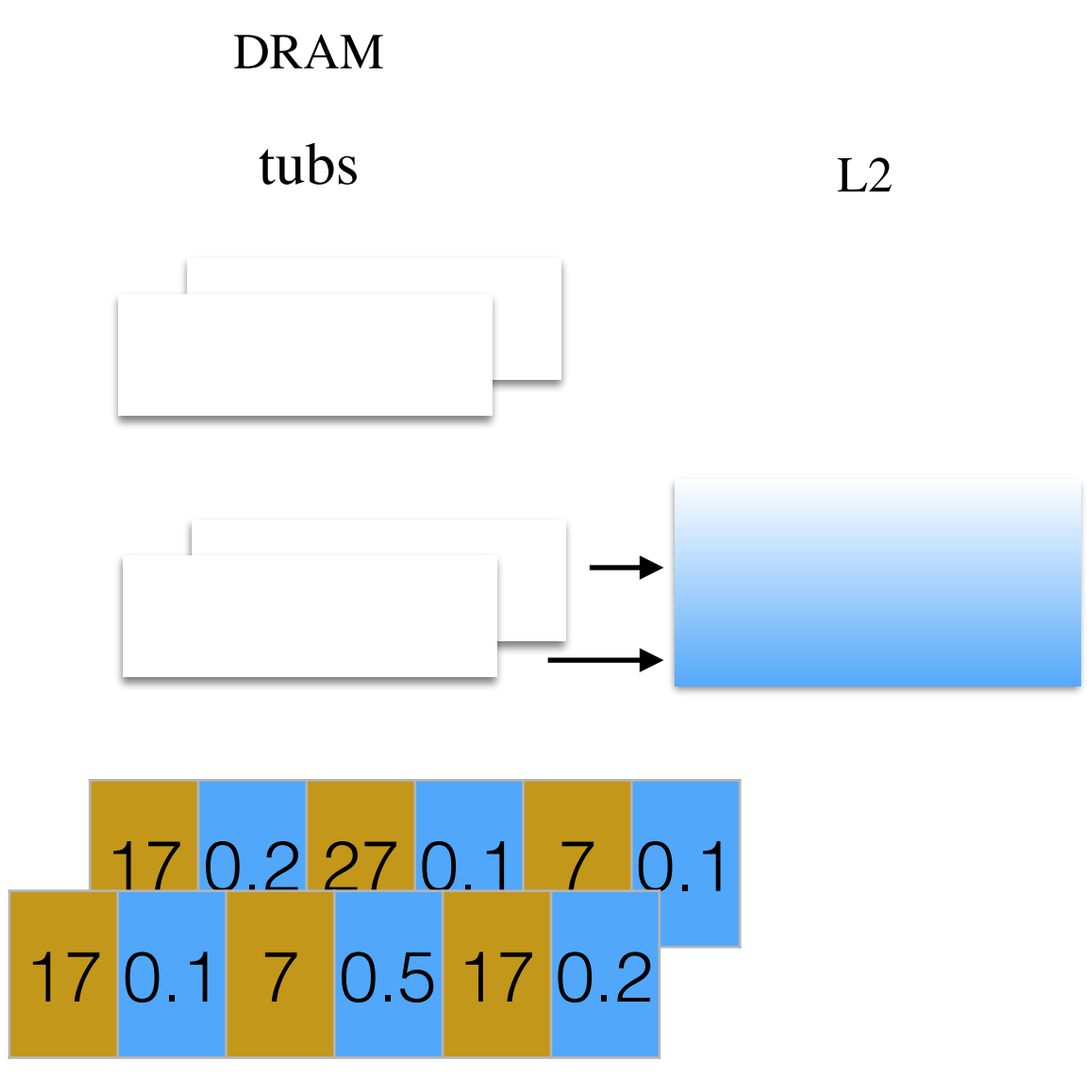
# Tag Distribution



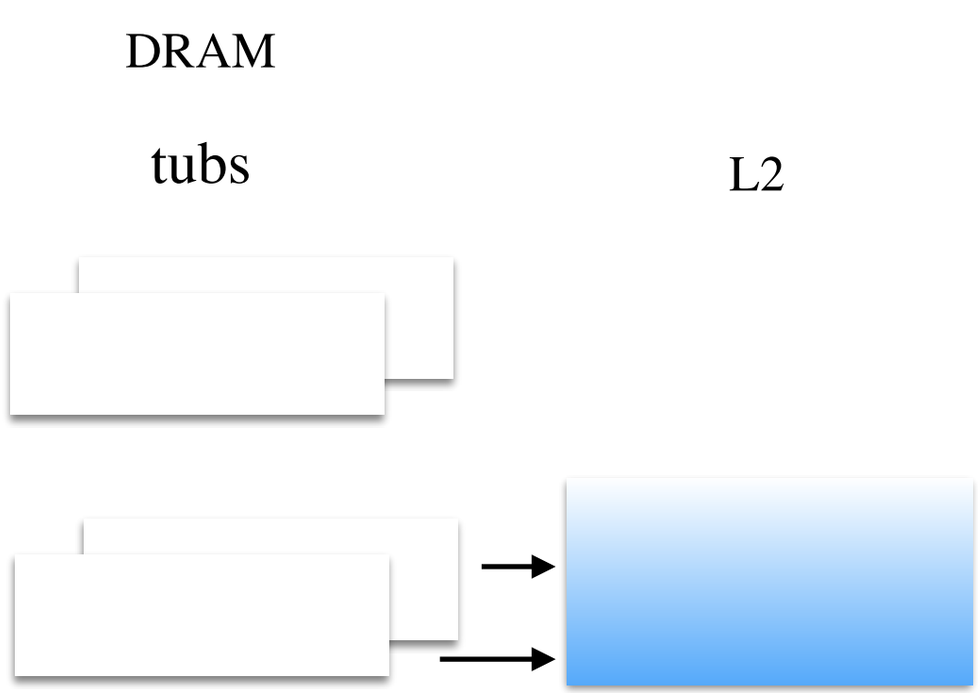
# Distribution: Pail Overflow



# Milk Delivery



# Milk Delivery

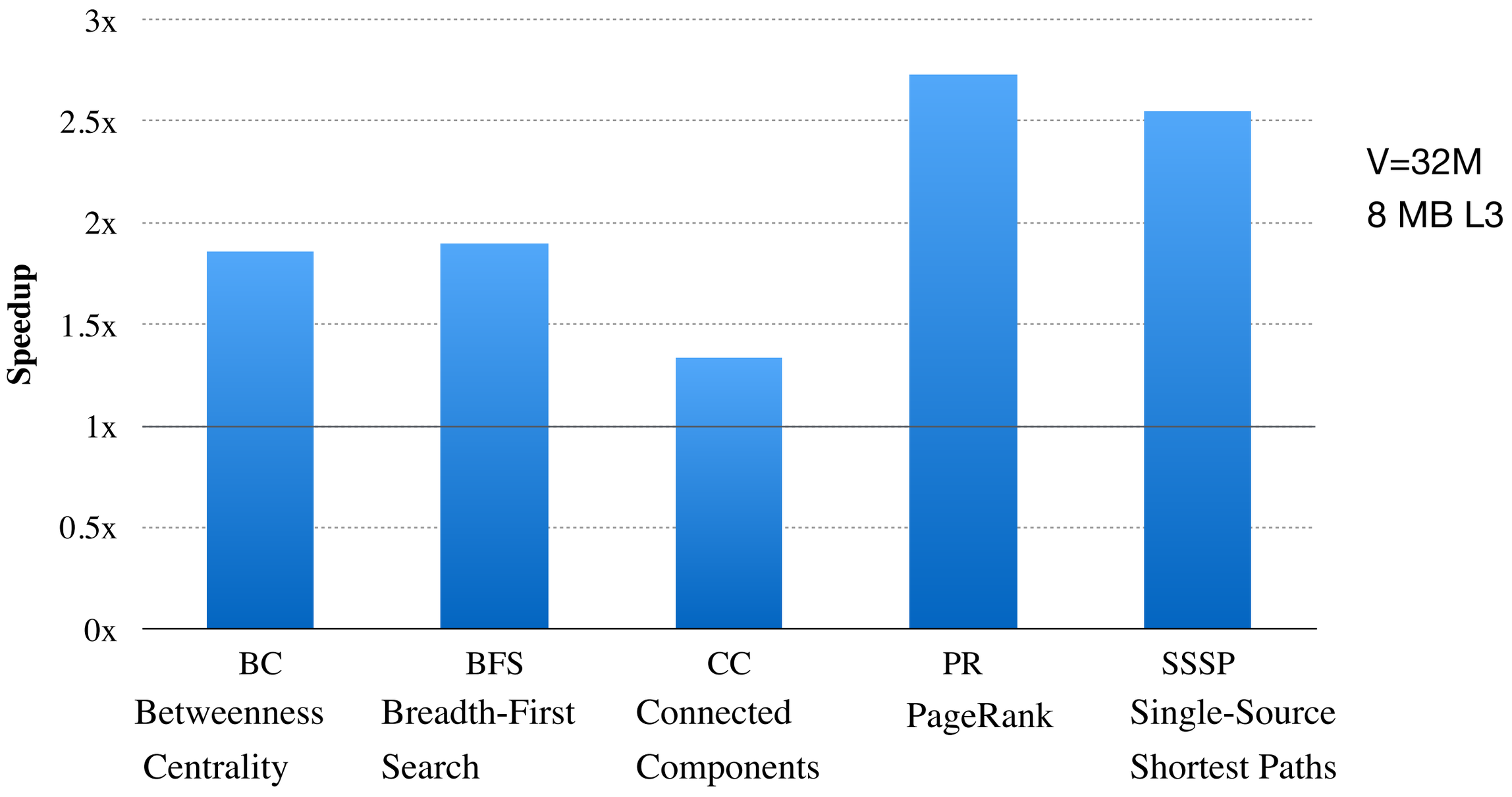



```
#pragma milk pack(contribU : +) tag(v)
#pragma omp atomic if(!milk)
      new_rank[v] += contribU;
```

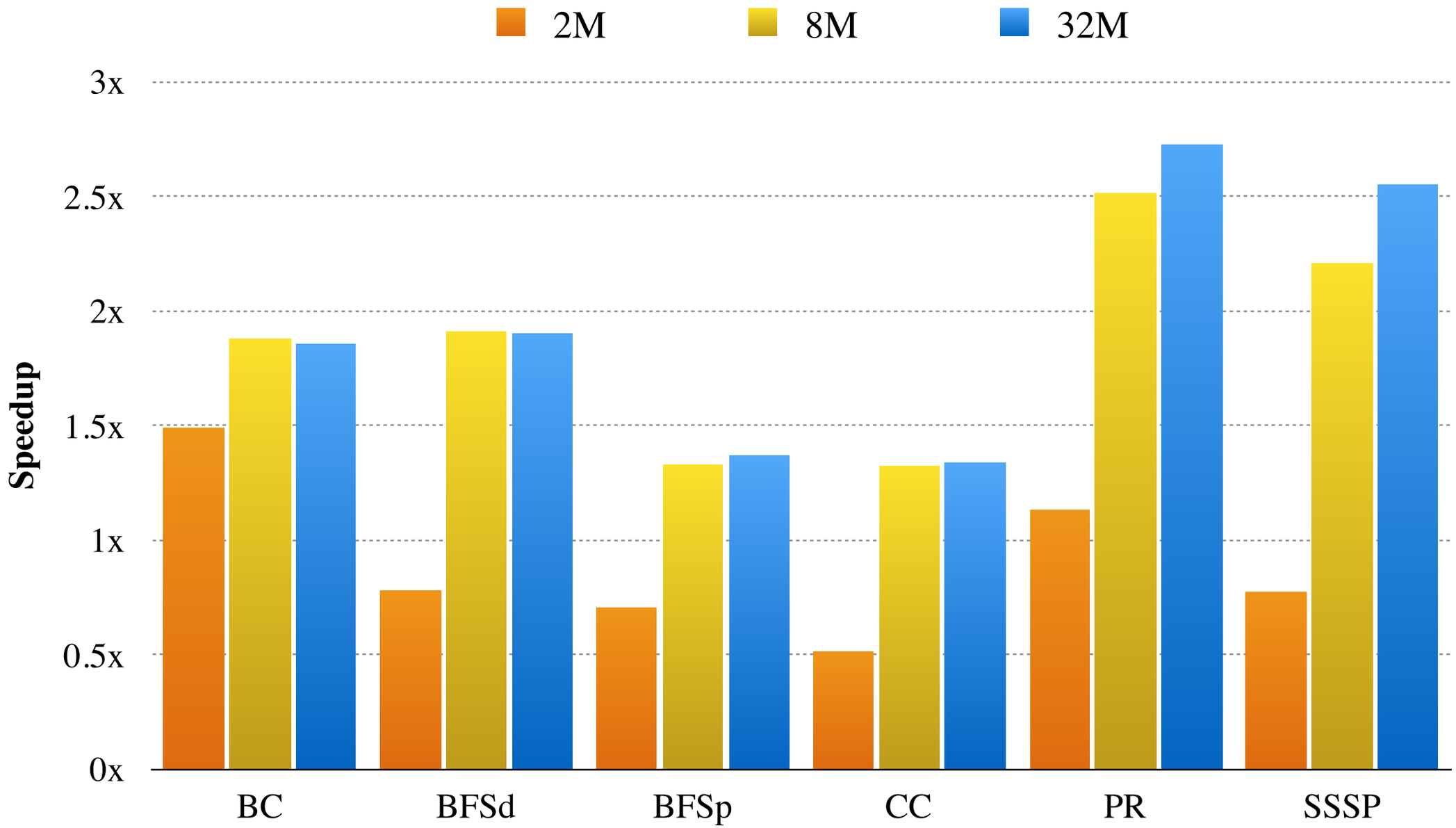
# Related Work

- Database JOIN optimizations
- [Shatdal94] cache partitioning
- [Manegold02, Kim09, Albutiu12, Balkesen15]
  TLB, SIMD, NUMA,
  non-temporal writes, software write buffers

# Overall Speedup with **`milk`**

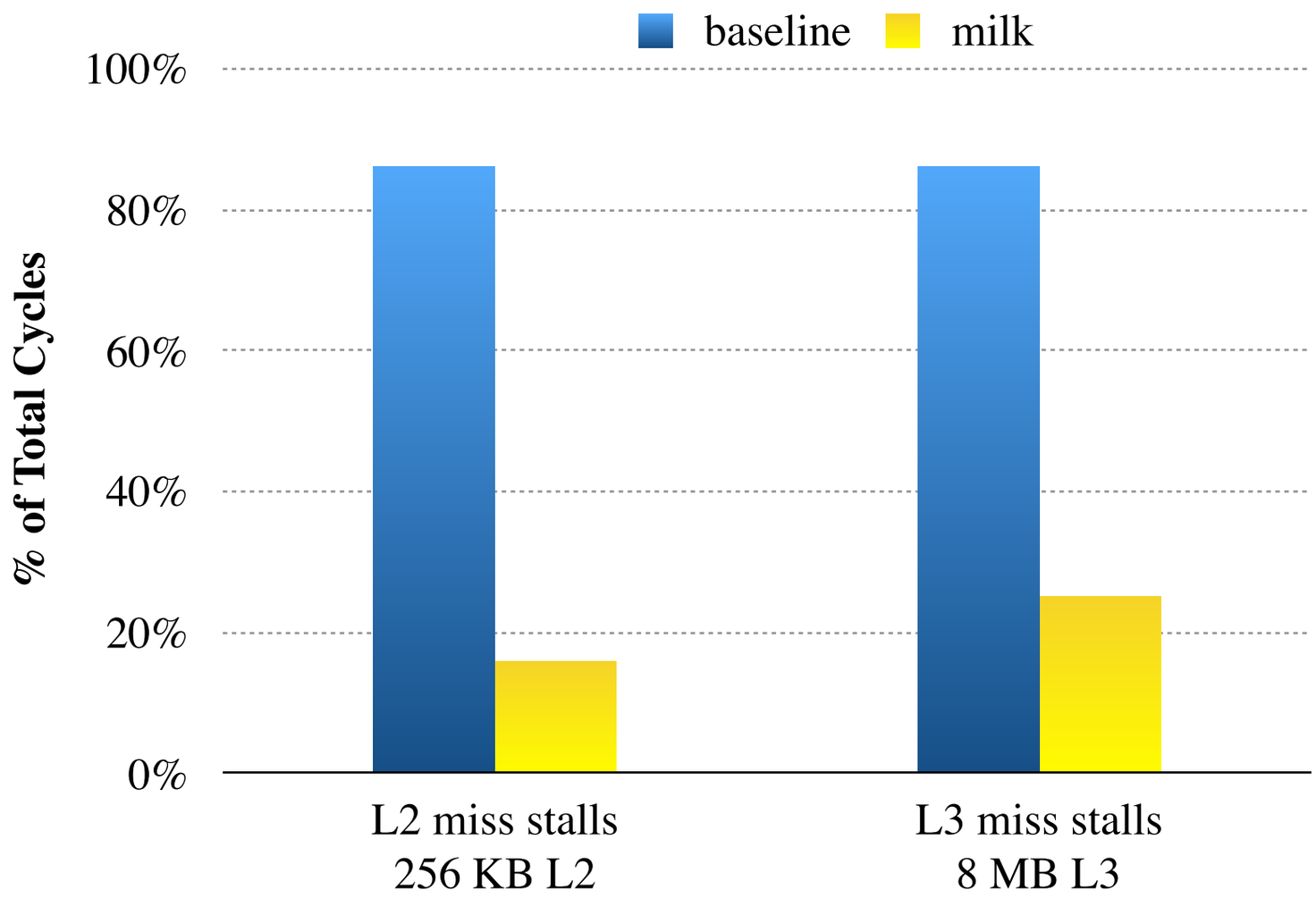


# Overall Speedup with **`milk`**



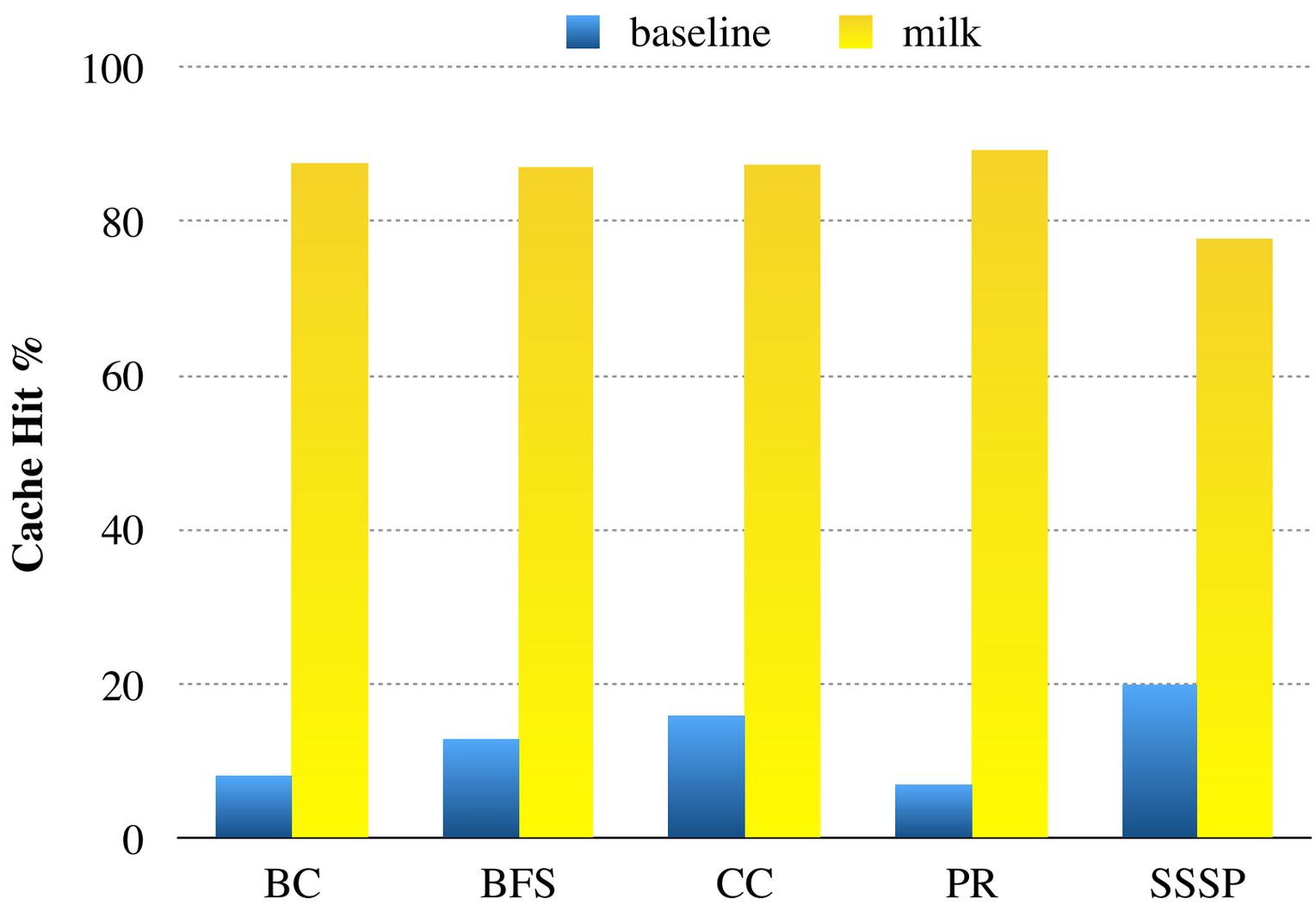

8 MB L3

# Stall Cycle Reduction



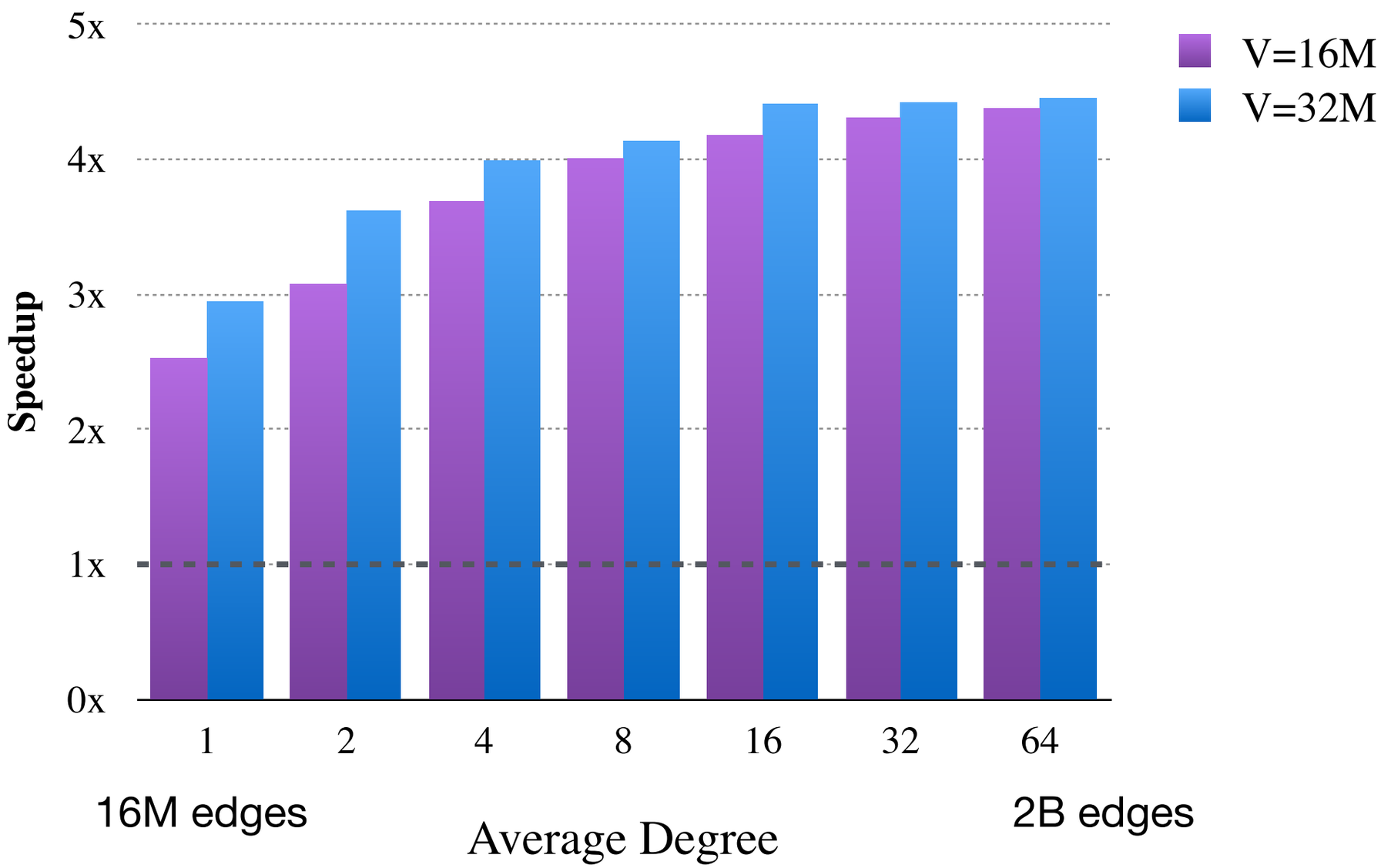

PageRank,
V=32M, d=16 (uniform)

# Indirect Access Cache Hit%




V=32M
8 MB L3
256KB L2

# Higher Degree
# → Higher Locality



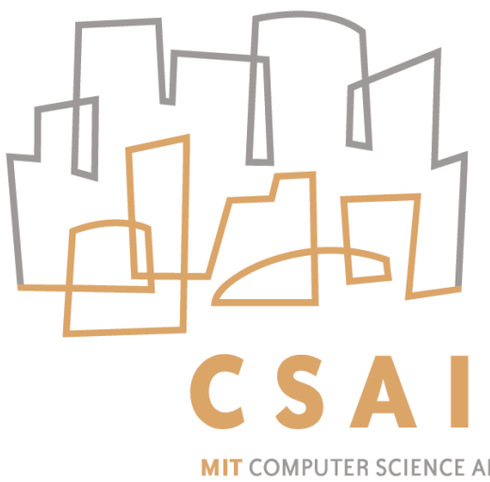
# Outline

- Overview
- Milk
- Cagra
- GraphIt
- Conclusion

# Making Caches Work for Graph Analytics

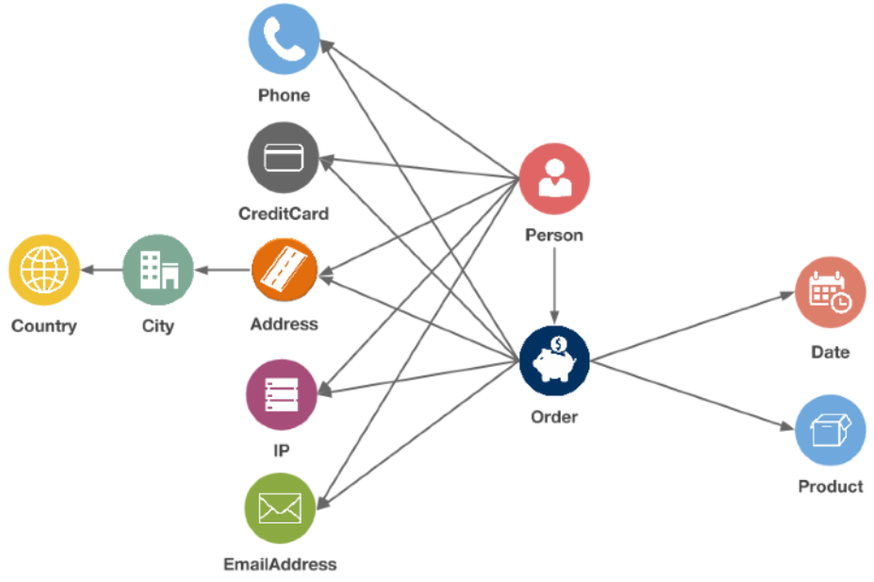
**Yunming Zhang**, Vladimir Kiriansky, Charith Mendis, Matei Zaharia*, Saman Amarasinghe

**MIT CSAIL and *Stanford InfoLab**

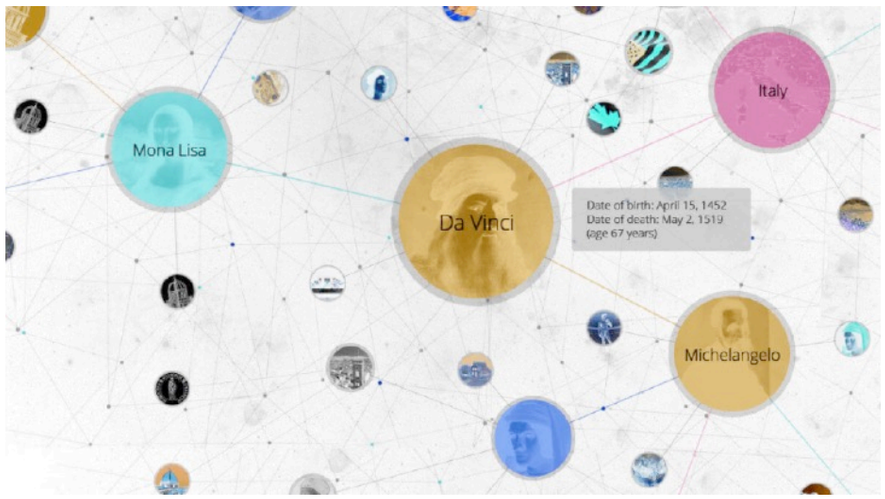



# Large Graphs

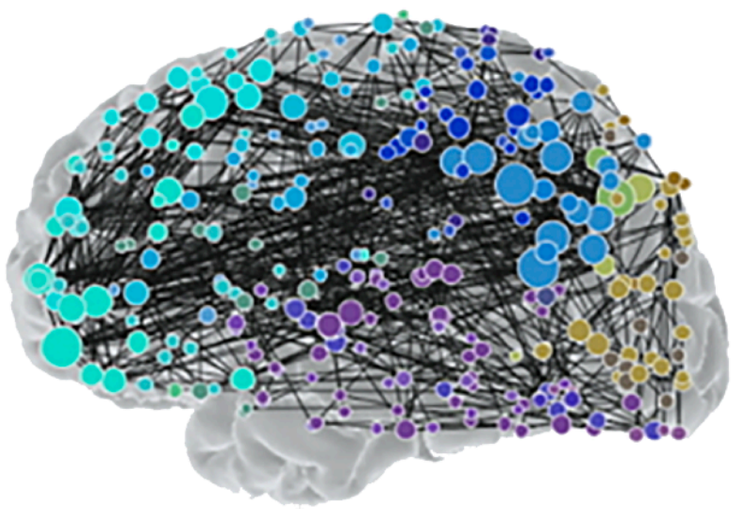
## Social Graphs




## Transaction Graph




## Knowledge Graphs

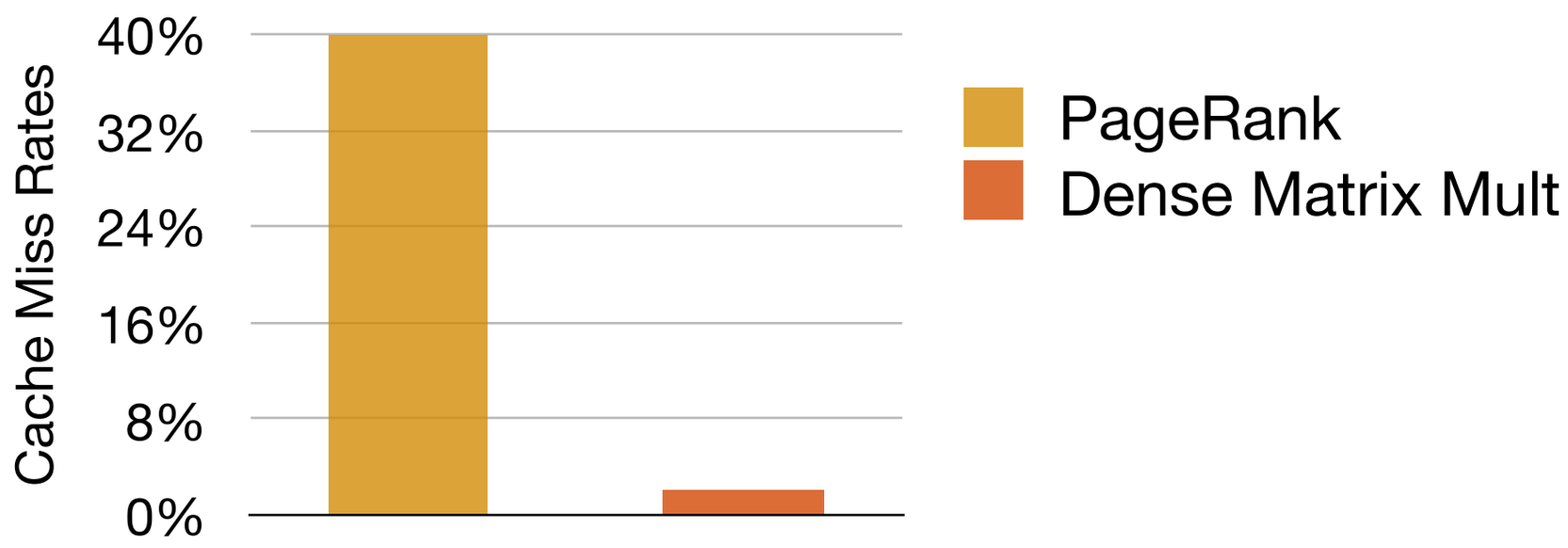


## Biological Graphs

**1.** http://www.facebookfever.com/introducing-facebook-new-graph-api-explorer-features/ 2. https://linkurio.us/wp-content/uploads/2015/06/Retail-fraud-graph-data-model.png
3. http://searchengineland.com/laymans-visual-guide-googles-knowledge-graph-search-api-241935 **4.** http://www.pnas.org/content/109/15/5549/F4.expansion.html

# Background

- Achieving good performance for graph applications is not easy
  - Graph applications have a lot of irregular memory accesses
  - PageRank incurs a last level cache (LLC) miss rate of 40% on Twitter graph
- Other important applications, such as Dense Matrix Multiplication, have LLC miss rates are often be < 2%



# Outline

- Motivation
- Related Works
- Frequency based Vertex Reordering
- Cache-aware Segmenting
- Evaluation

# 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
```

# 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
```

# 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
```

# 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

# PageRank

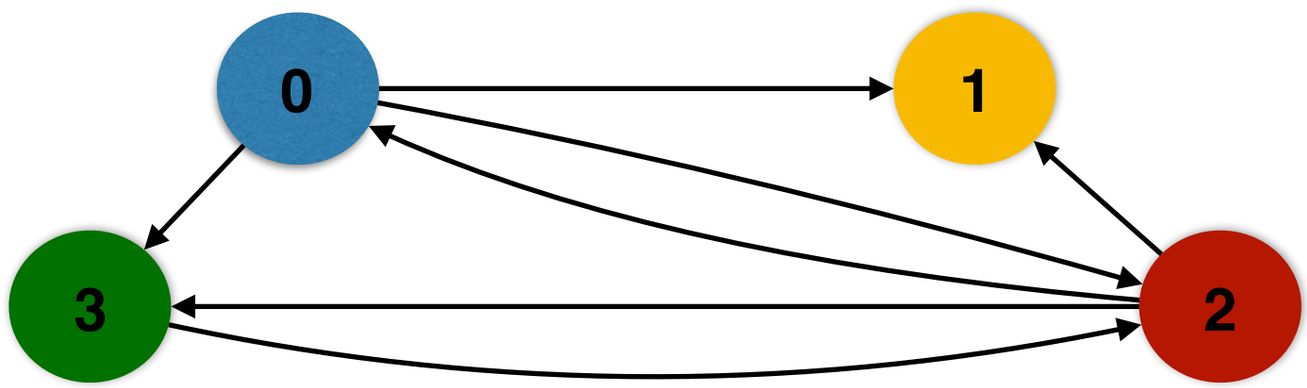
```
while …
      for node : graph.vertices
          for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
          newRanks[node] = baseScore +
damping*newRanks[node];
swap ranks and newRanks
```

# 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
```

# PageRank

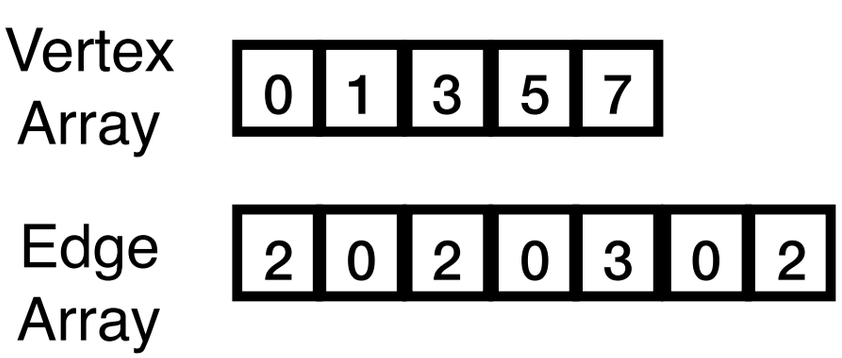
```
while …
      for node : graph.vertices
          for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
          newRanks[node] = baseScore +
damping*newRanks[node];
swap ranks and newRanks
```

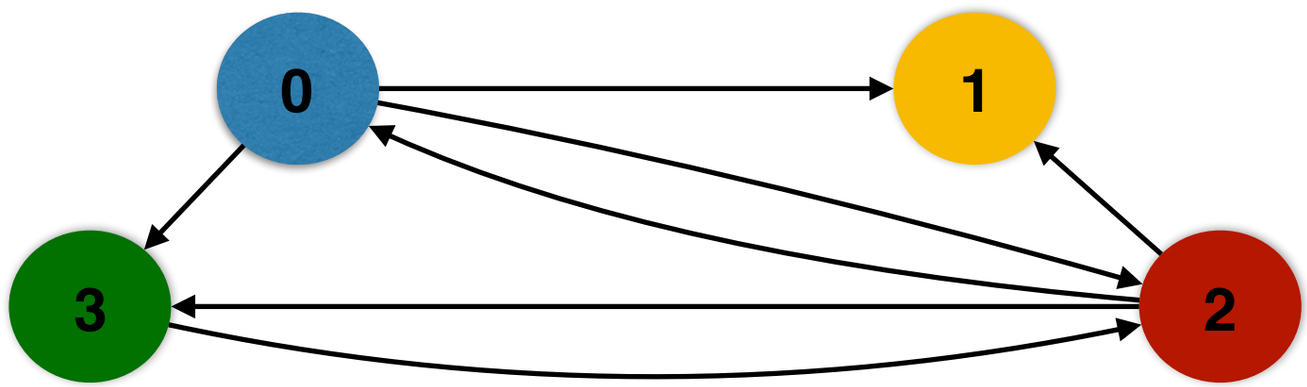


# PageRank

```
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)**

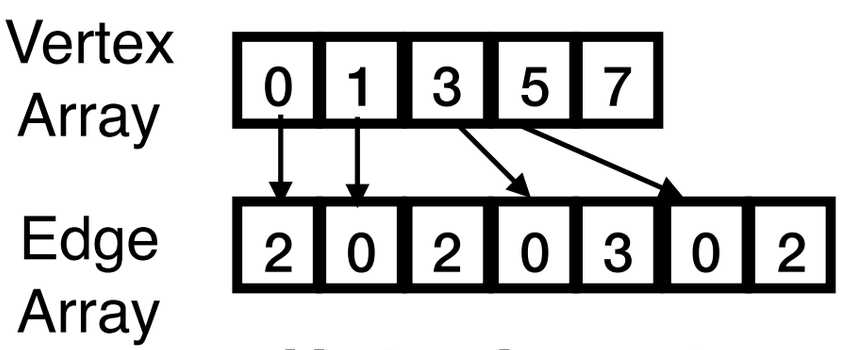


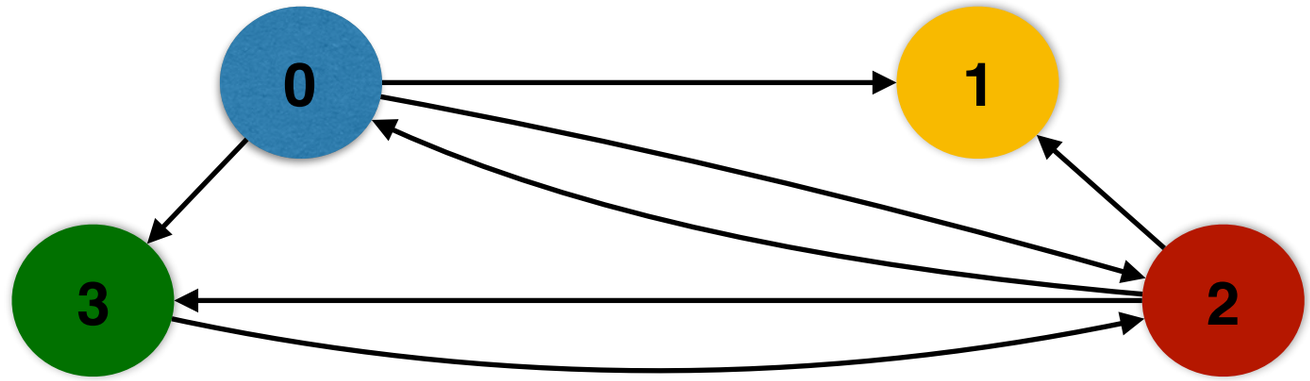
# PageRank

```
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)**



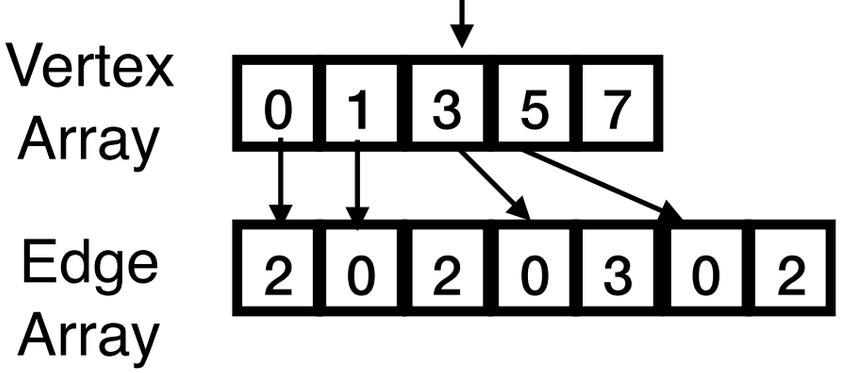

**Vertex Array stores indices into the Edge Array. Edge Array stores neighbors' ID in the CSR**

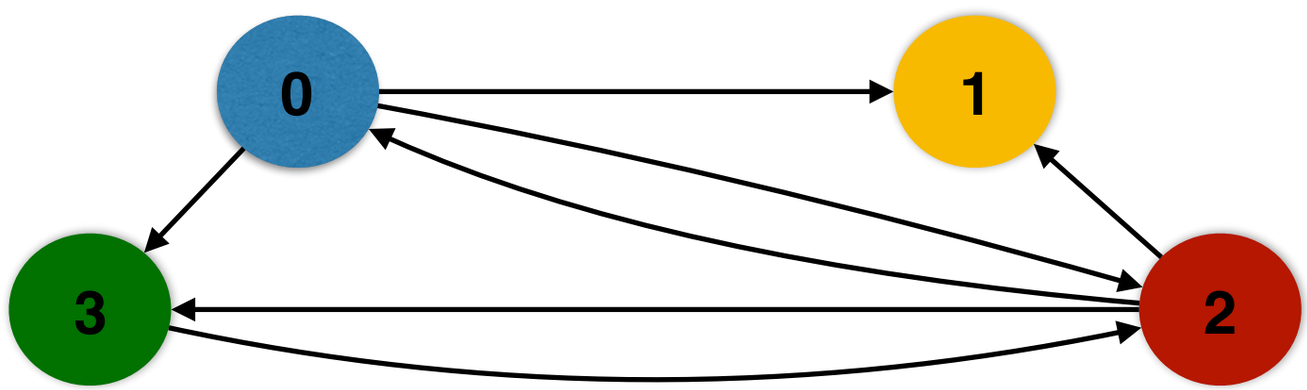


# PageRank

```
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**

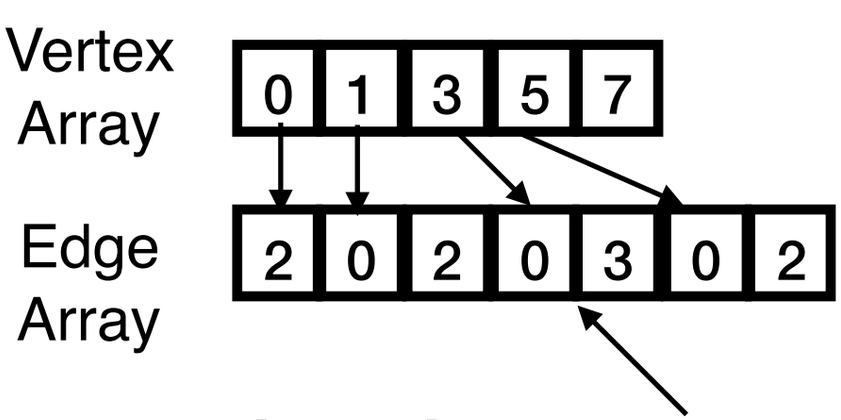


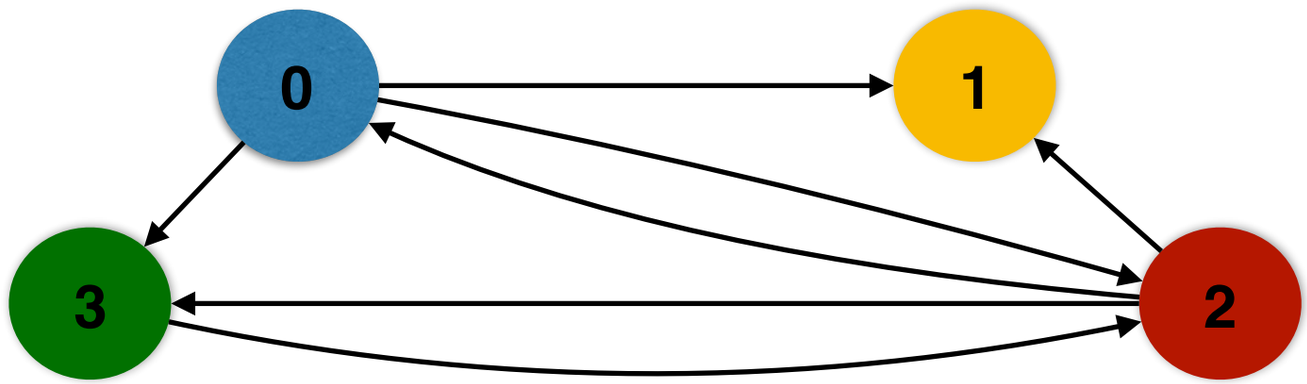



# PageRank

```
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



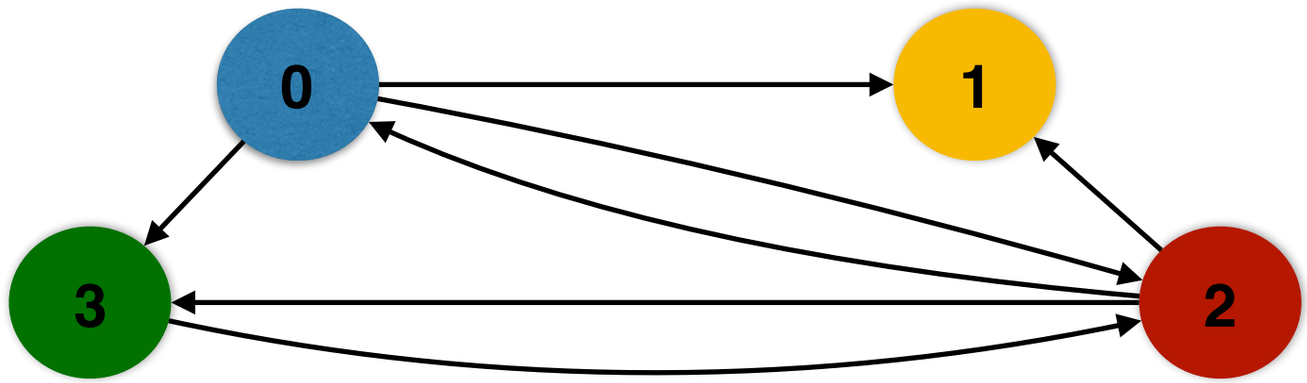

**Irregular access on ngh's rank and outDegree data when scanning through the edge array**

# 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**

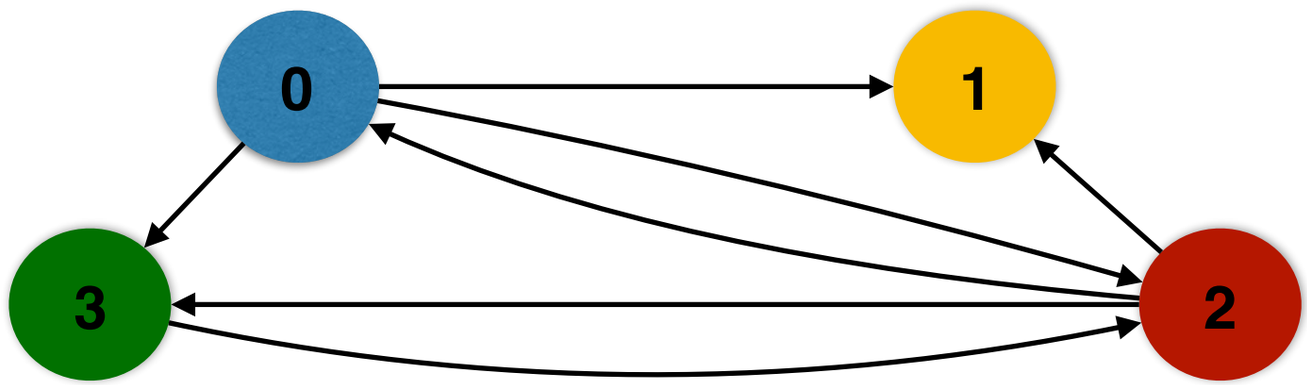
# PageRank

```
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**

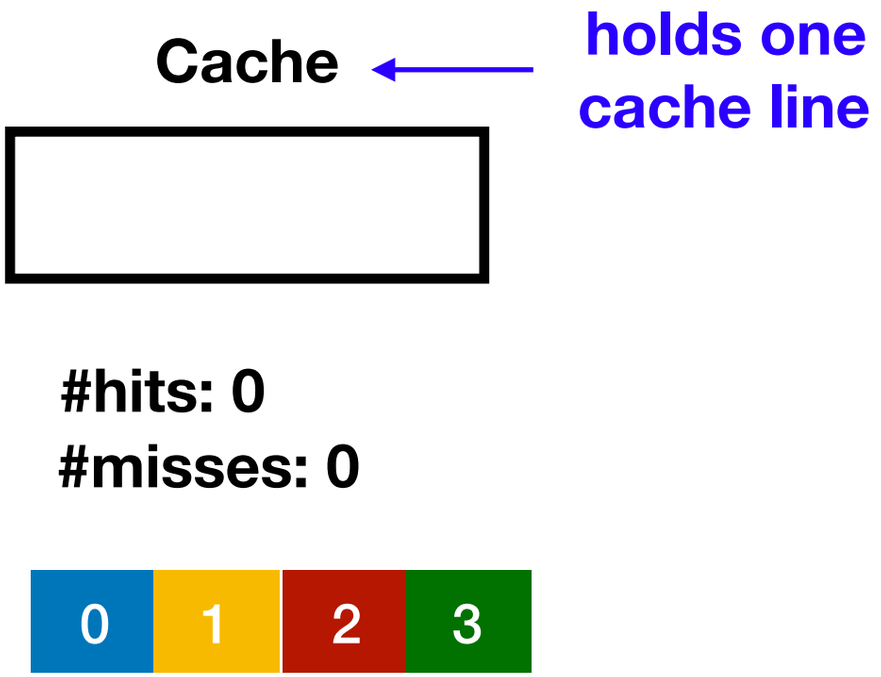
**#hits: 0**
**#misses: 0**

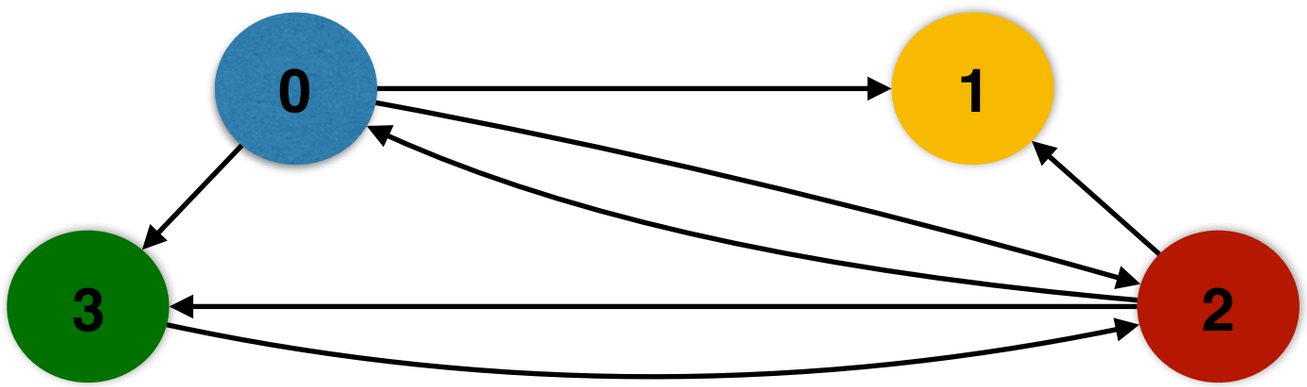



# PageRank

```
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**

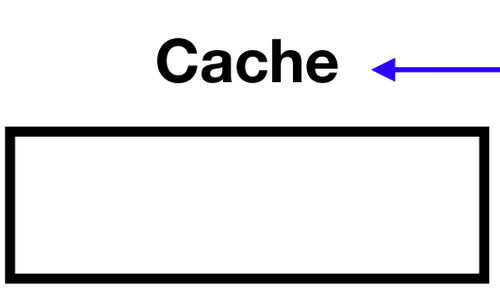


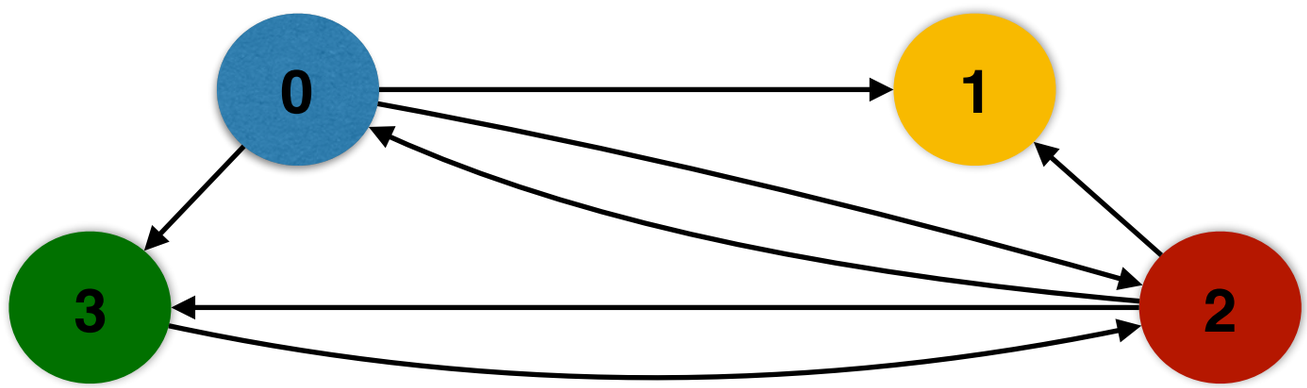



# PageRank

```
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**



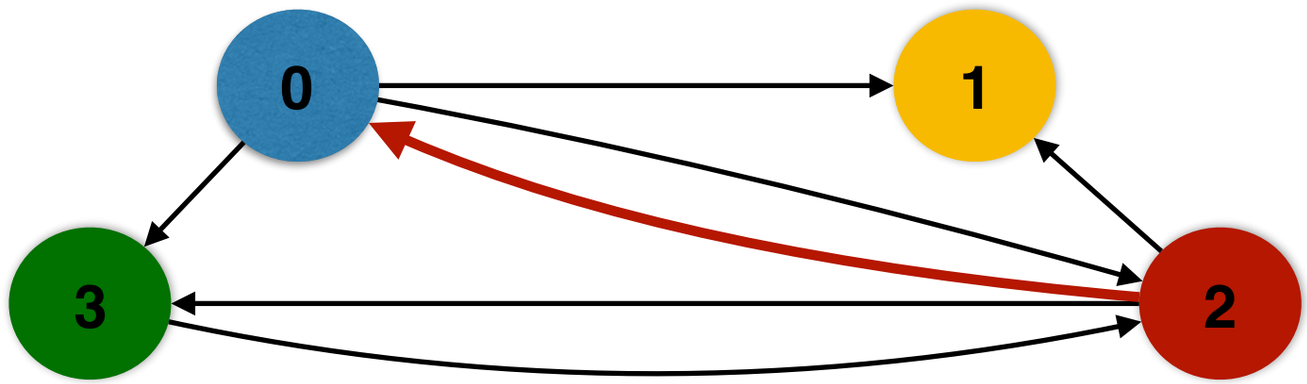



# PageRank

```
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**

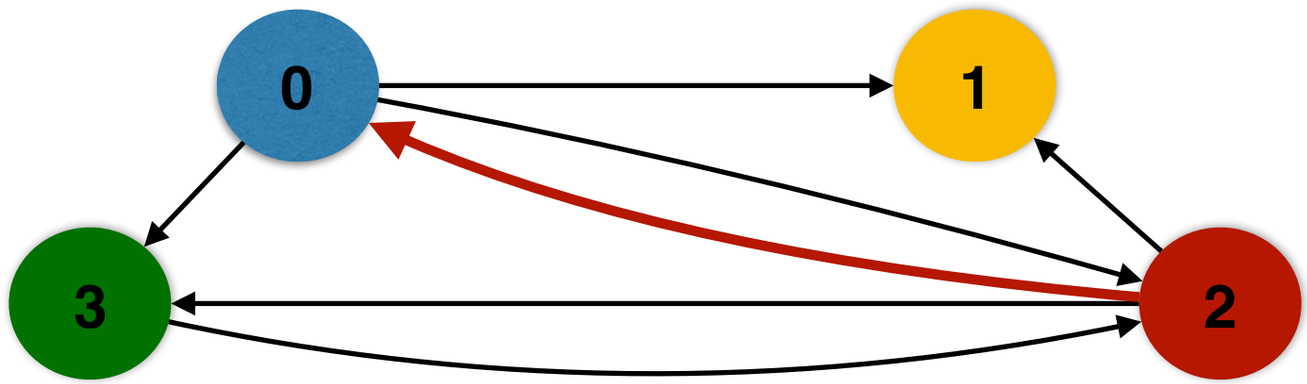



# PageRank

```
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**

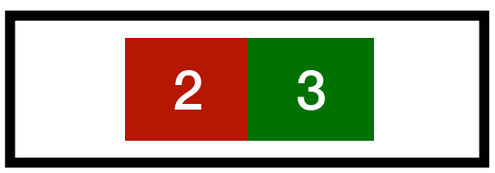



# PageRank

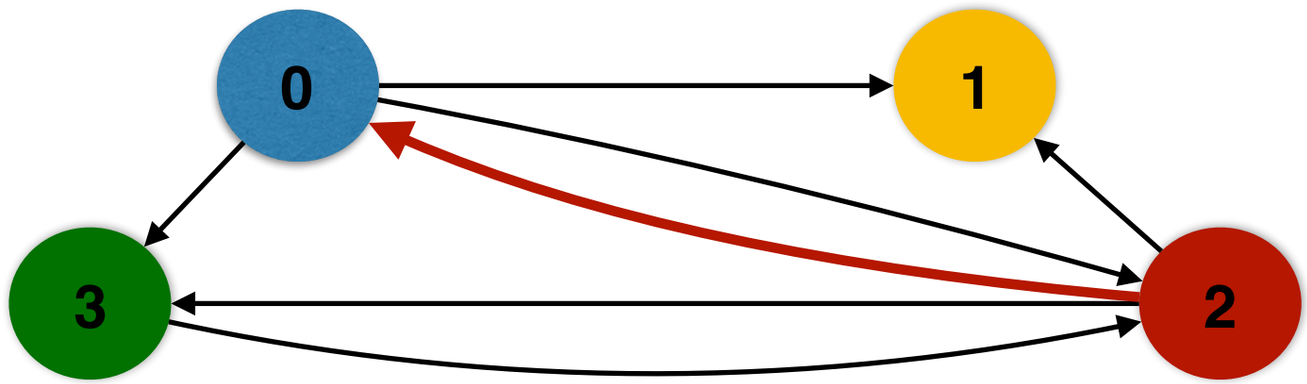
```
while …
      for node : graph.vertices
          for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
          newRanks[node] = baseScore +
damping*newRanks[node];
```

swap ranks and newRanks

**Cache**



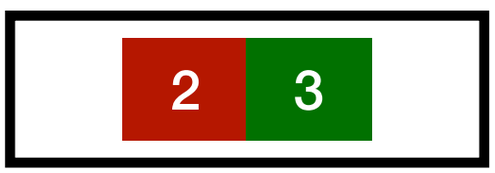

**#hits: 0**
**#misses: 1**

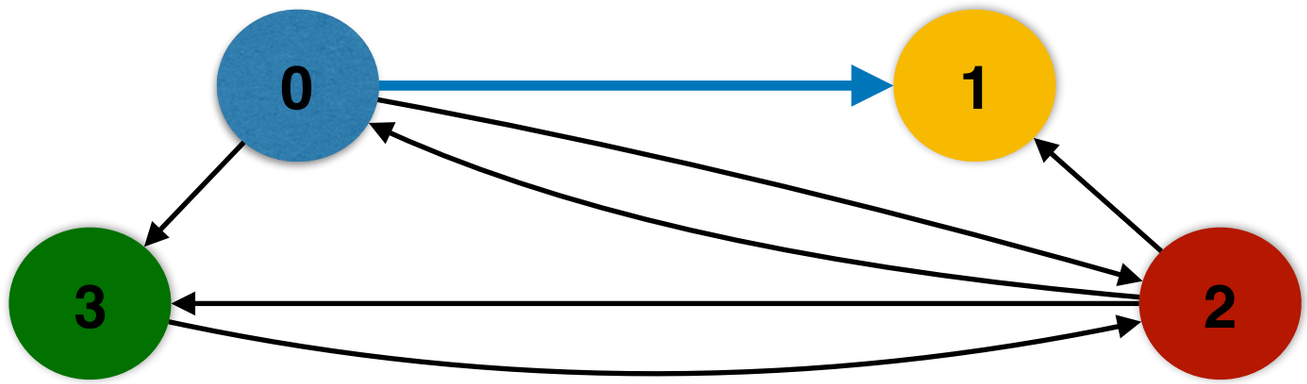
# PageRank

```
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**



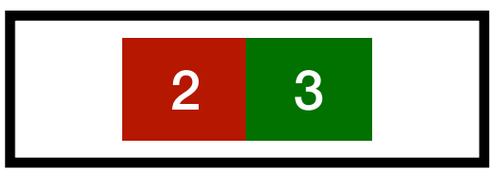

**#hits: 0**
**#misses: 1**

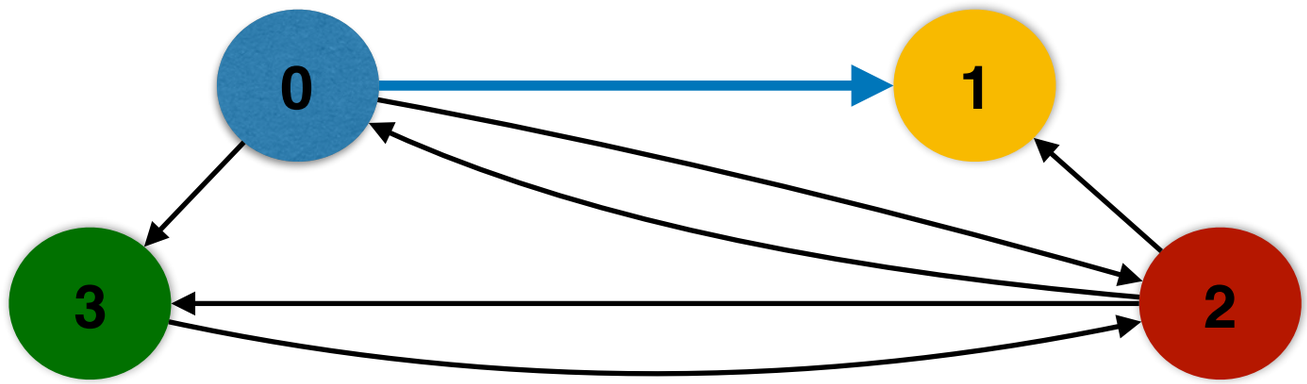
# PageRank

```
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: 1**

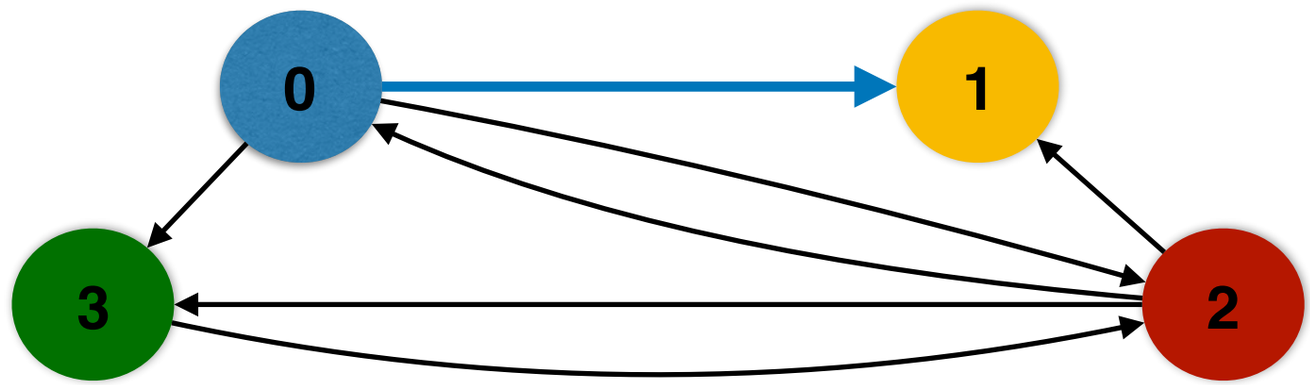



# PageRank

```
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**

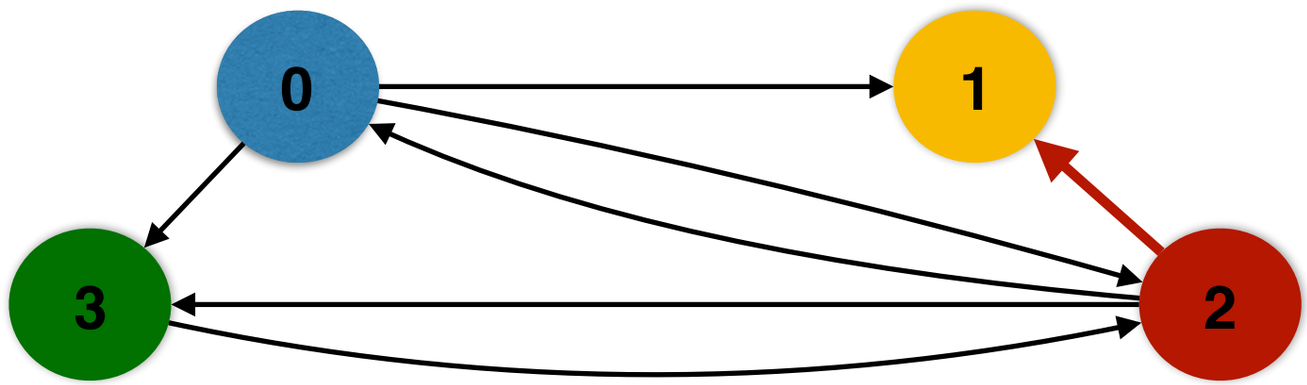
# PageRank

```
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**

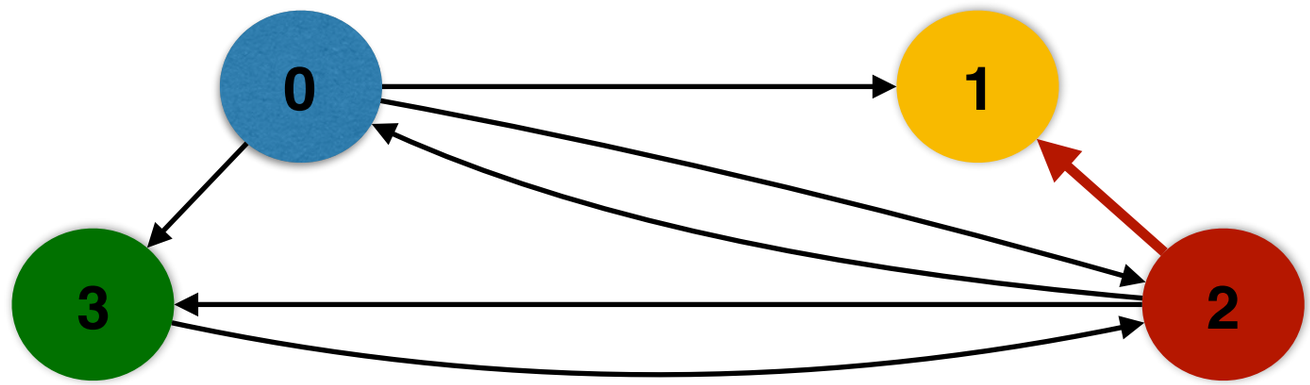
# PageRank

```
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**

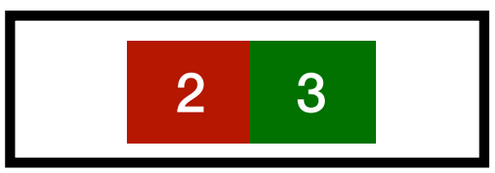



# PageRank

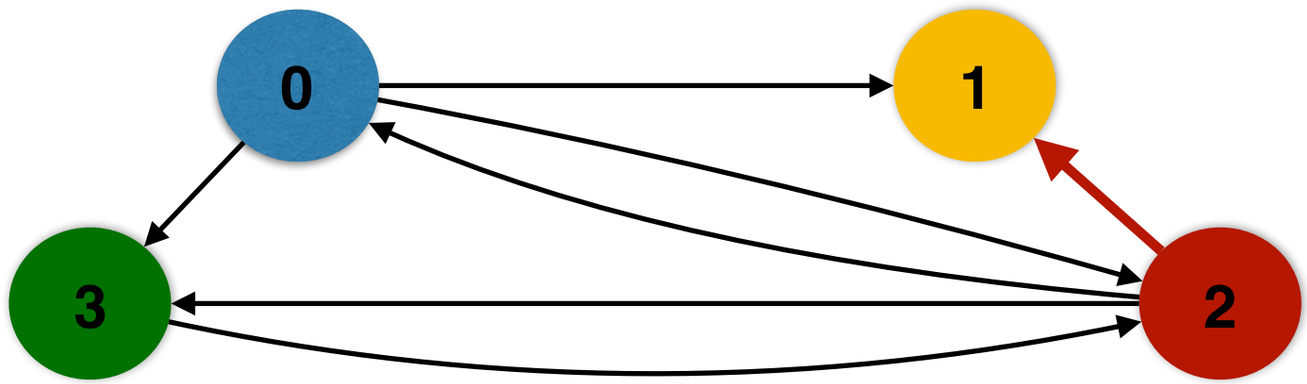
```
while …
      for node : graph.vertices
          for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
          newRanks[node] = baseScore +
damping*newRanks[node];
```

swap ranks and newRanks

**Cache**



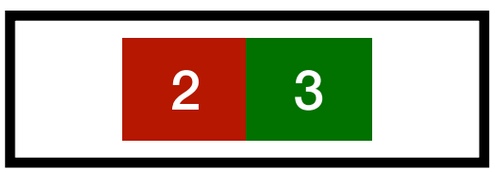

**#hits: 0**
**#misses: 3**

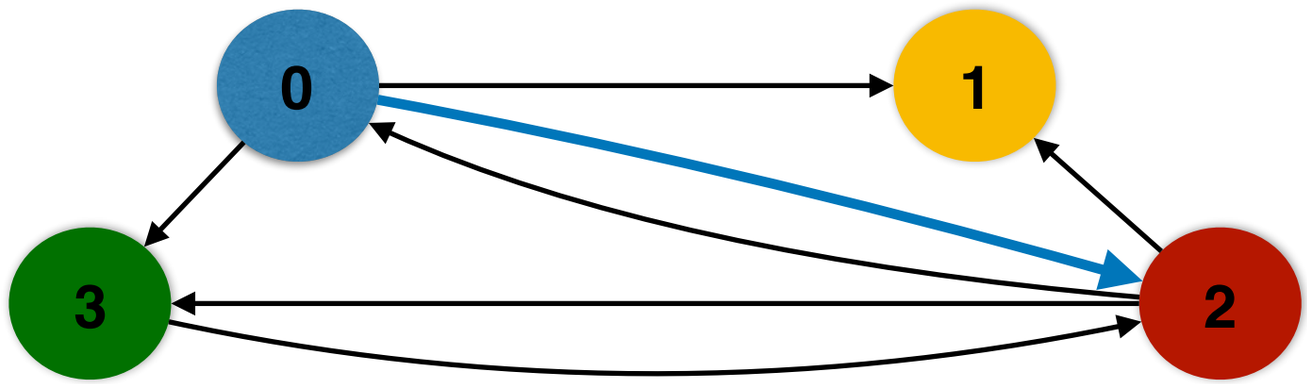
# PageRank

```
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: 3**

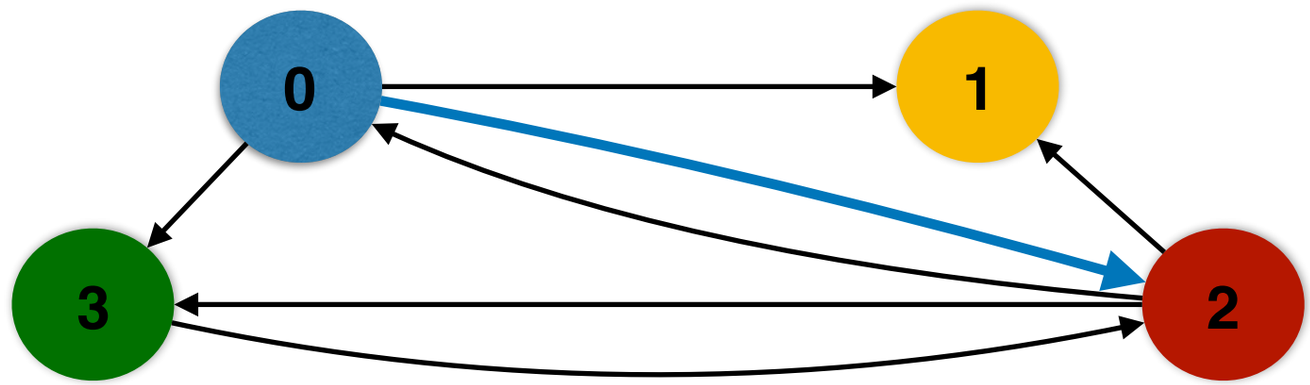



# PageRank

```
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**

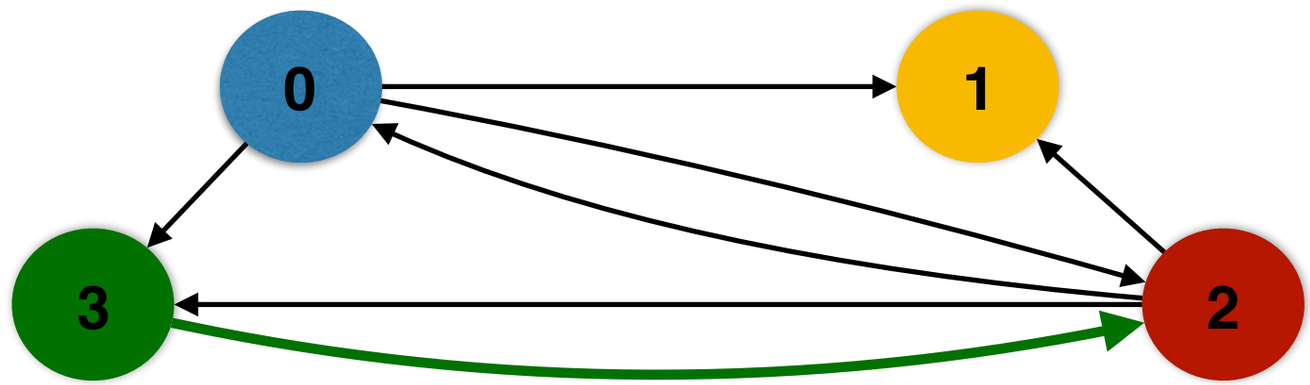
# PageRank

```
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

**Cache**


**#hits: 0**
**#misses: 4**

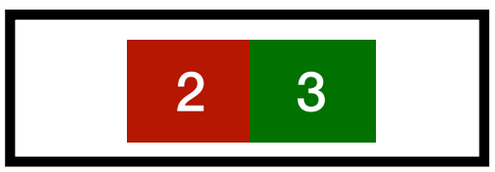



# PageRank

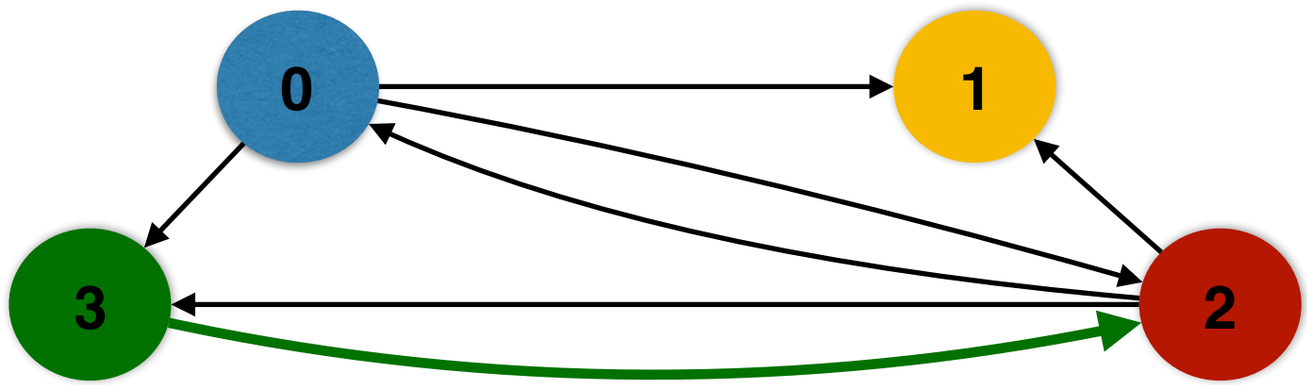
```
while …
      for node : graph.vertices
          for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
          newRanks[node] = baseScore +
damping*newRanks[node];
```

swap ranks and newRanks

**Cache**




**#hits: 0**
**#misses: 5**

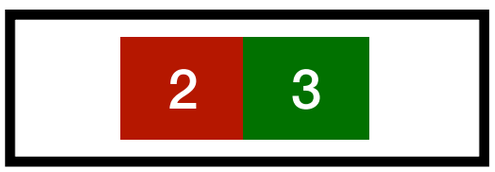



# PageRank

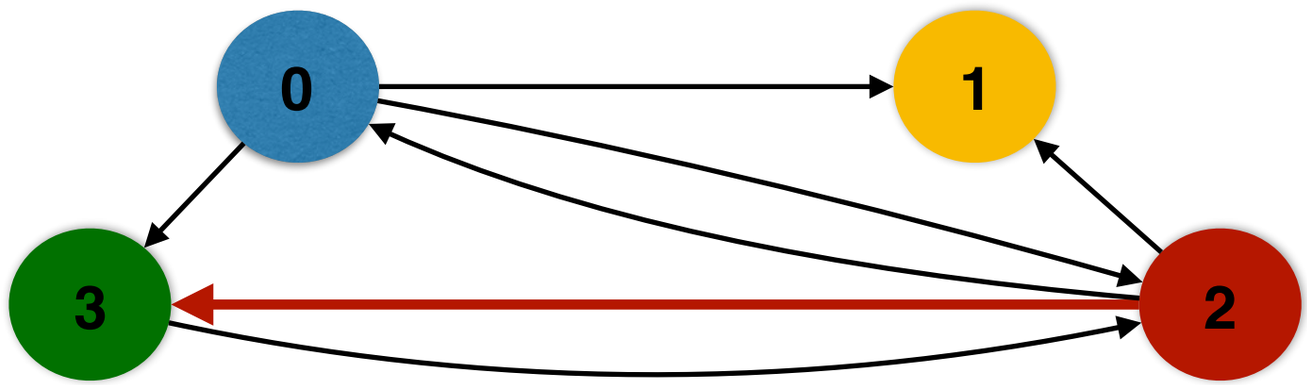
```
while …
      for node : graph.vertices
          for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
          newRanks[node] = baseScore +
damping*newRanks[node];
```

swap ranks and newRanks

**Cache**




**#hits: 0**
**#misses: 5**

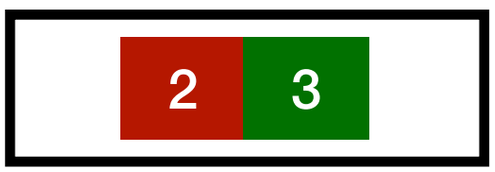



# PageRank

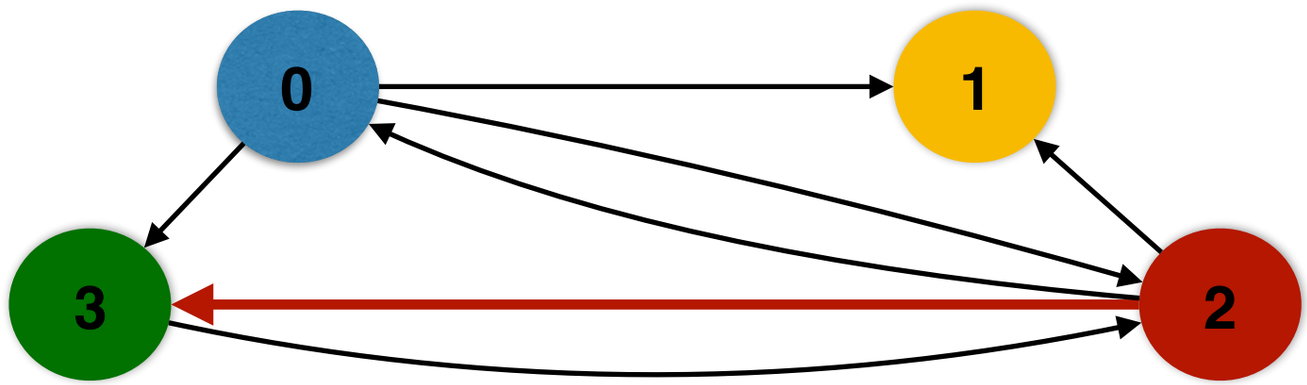
```
while …
      for node : graph.vertices
          for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
          newRanks[node] = baseScore +
damping*newRanks[node];
```

swap ranks and newRanks

**Cache**




**#hits: 1**
**#misses: 5**

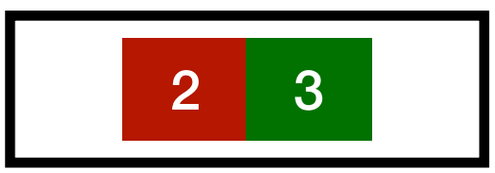



# PageRank

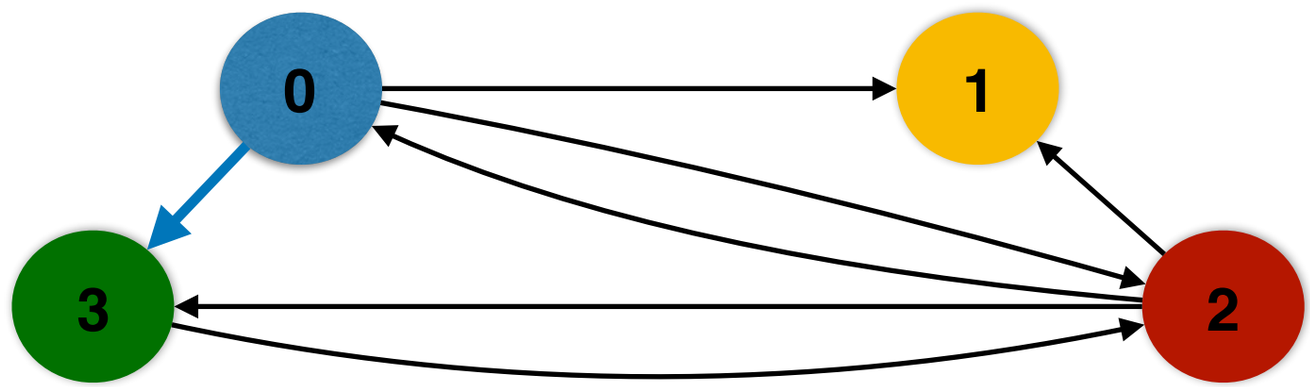
```
while …
      for node : graph.vertices
          for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
          newRanks[node] = baseScore +
damping*newRanks[node];
```

swap ranks and newRanks

**Cache**




**#hits: 1**
**#misses: 5**

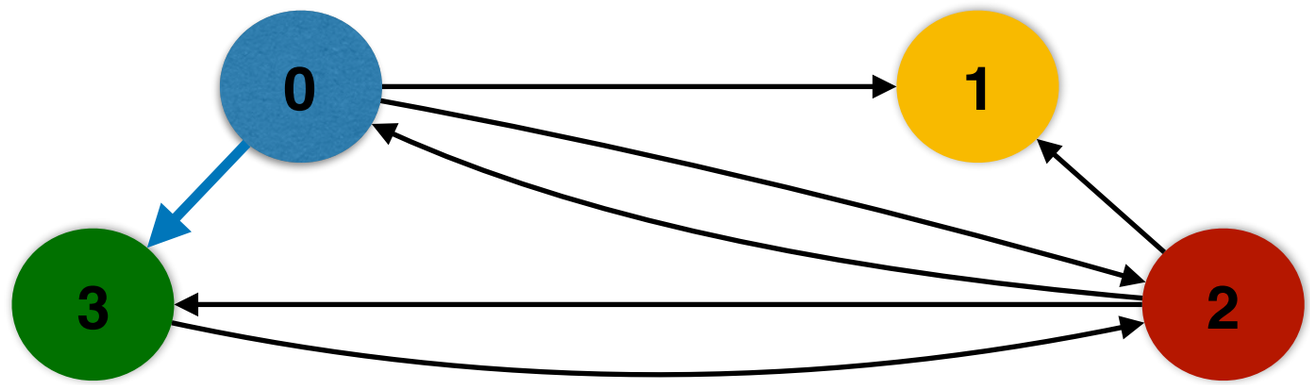
# PageRank

```
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**

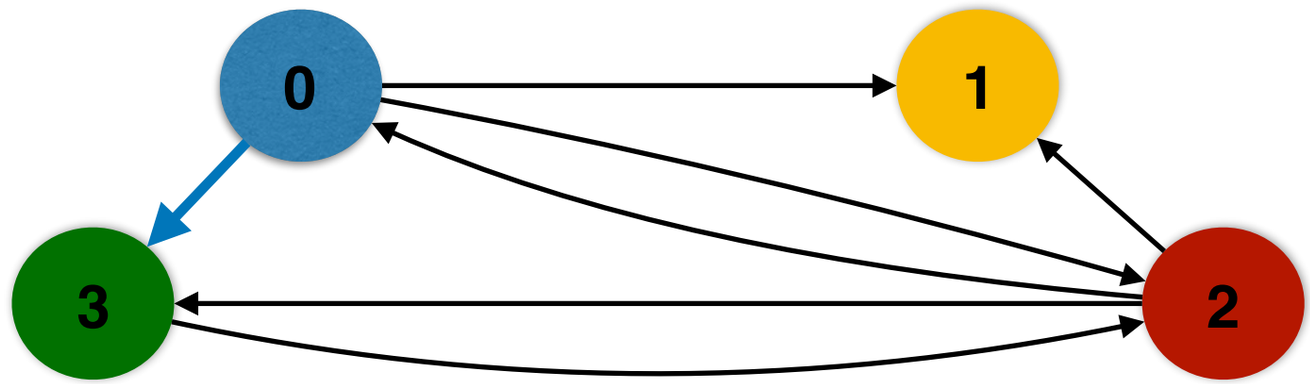
# PageRank

```
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
```

**Cache**


**#hits: 1**
**#misses: 6**

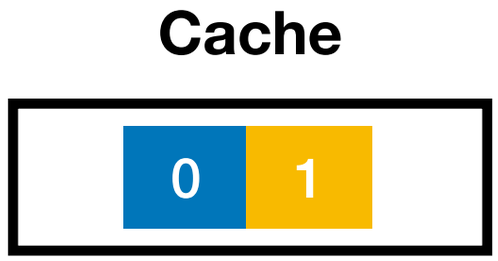
**A very high miss rate**

# PageRank

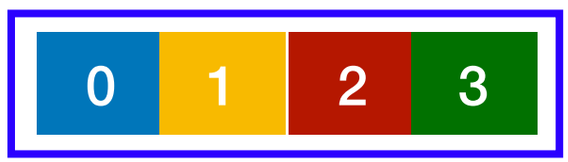
```
while …
      for node : graph.vertices
          for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
          newRanks[node] = baseScore +
damping*newRanks[node];
```

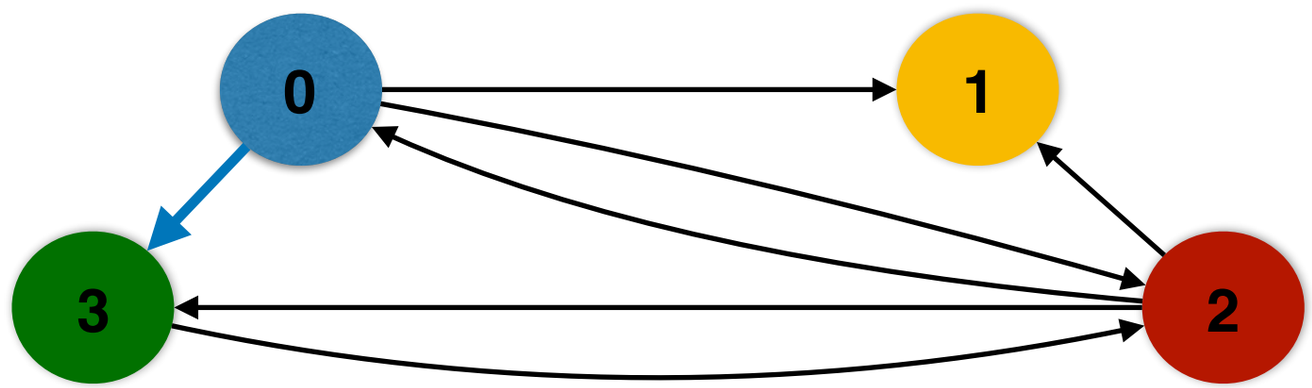
swap ranks and newRanks




**#hits: 1**
**#misses: 6**

**Working set larger than cache**



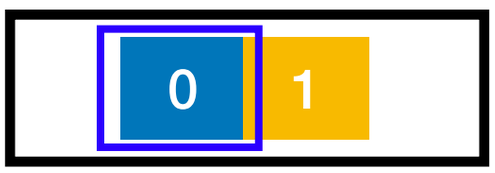
# PageRank

```
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];
```

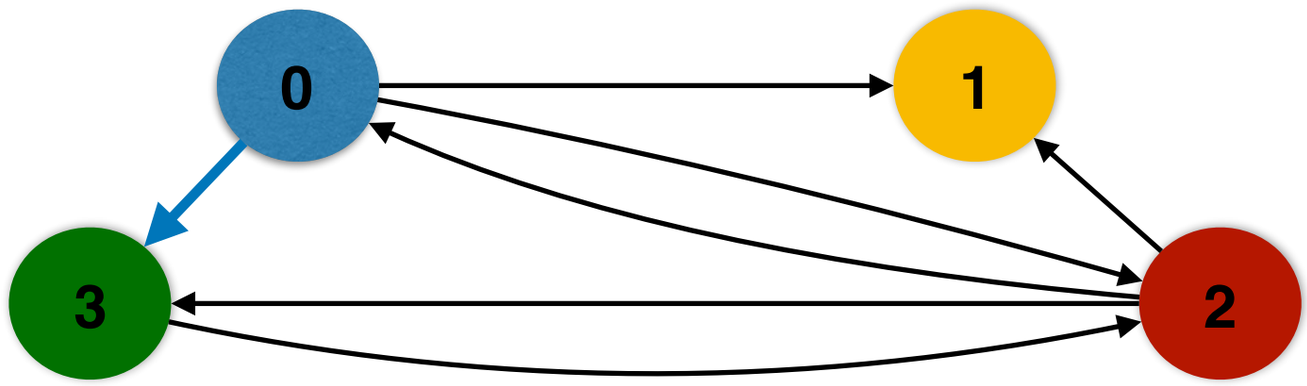
swap ranks and newRanks

**Often only use part of the cache line**

**Cache**




**#hits: 1**
**#misses: 6**

# Performance Bottleneck

- Working set much larger than cache size
- Access pattern is irregular
- Often uses part of the cache line
- Hard to benefit from hardware prefetching
- TLB miss, DRAM row miss (hundreds of cycles )

# Performance Bottleneck

**Real-world graphs often have working set 10-200x larger than cache size**

- Working set much larger than cache size
- Access pattern is irregular
- Often uses part of the cache line
- Hard to benefit from hardware prefetching
- TLB miss, DRAM row miss (hundreds of cycles )

# Performance Bottleneck

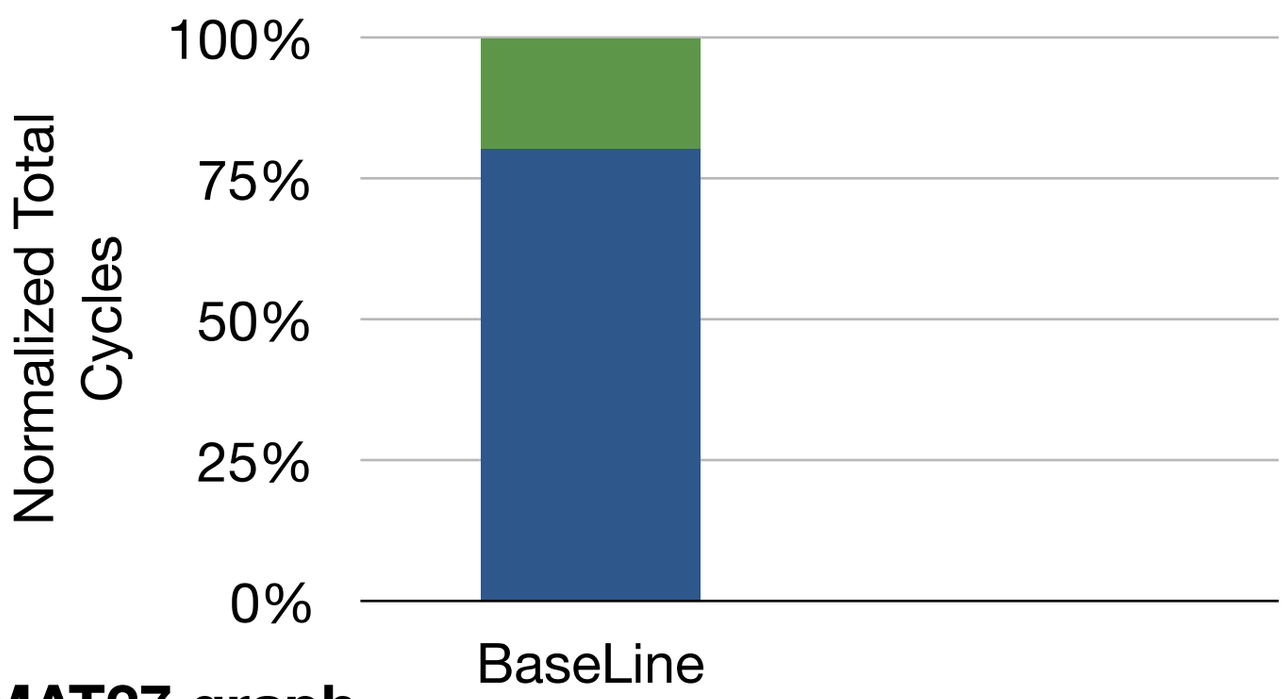
- Working set much larger than cache size
- Access pattern is irregular
- Often uses part of the cache line
- Hard to benefit from hardware prefetching
- TLB miss, DRAM row miss (hundreds of cycles )

# Performance Bottleneck

- Working set much larger than cache size
- Access pattern is irregular **Often only use 1/16 - 1/8 of a cache line in modern hardware**
- Often uses part of the cache line
- Hard to benefit from hardware prefetching
- TLB miss, DRAM row miss (hundreds of cycles )

# PageRank

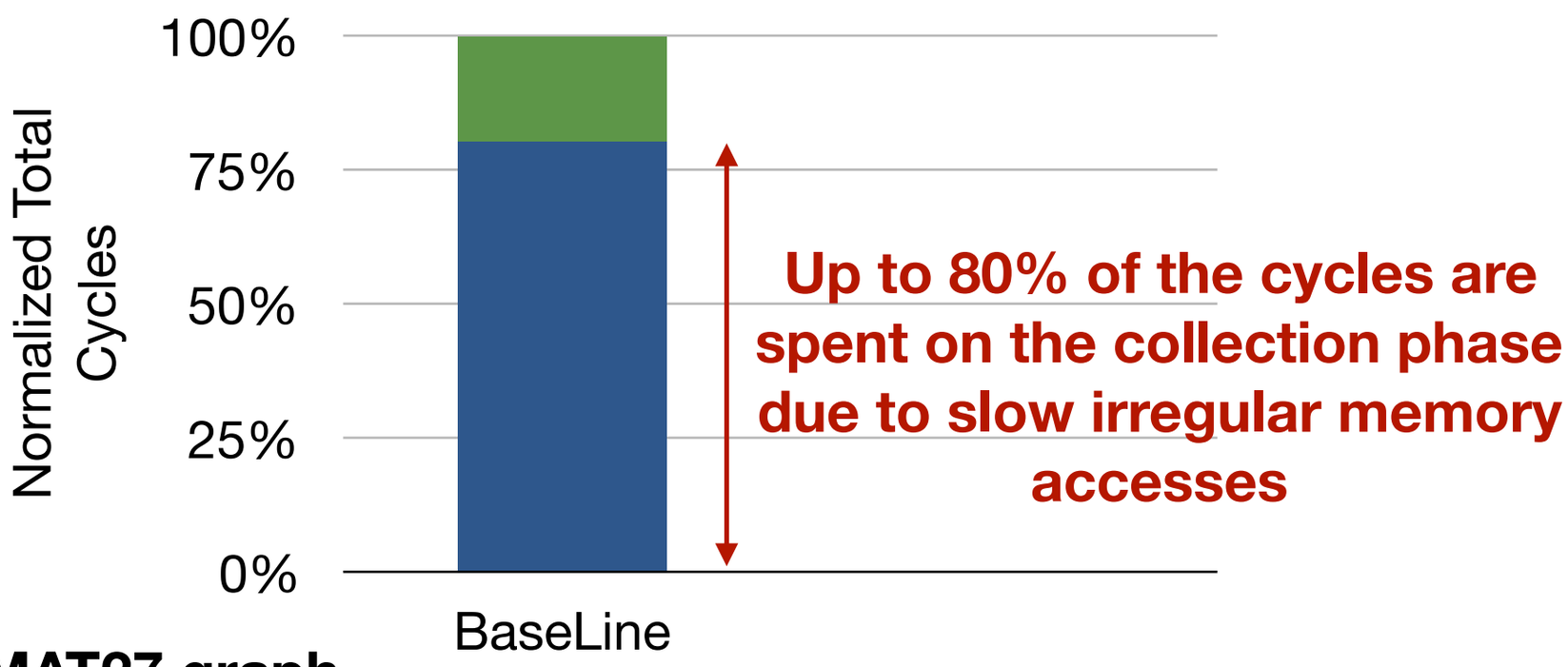
```
while …
      for node : graph.vertices
          for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
          newRanks[node] = baseScore +
damping*newRanks[node];
swap ranks and newRanks
```




**on RMAT27 graph**

# 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
```

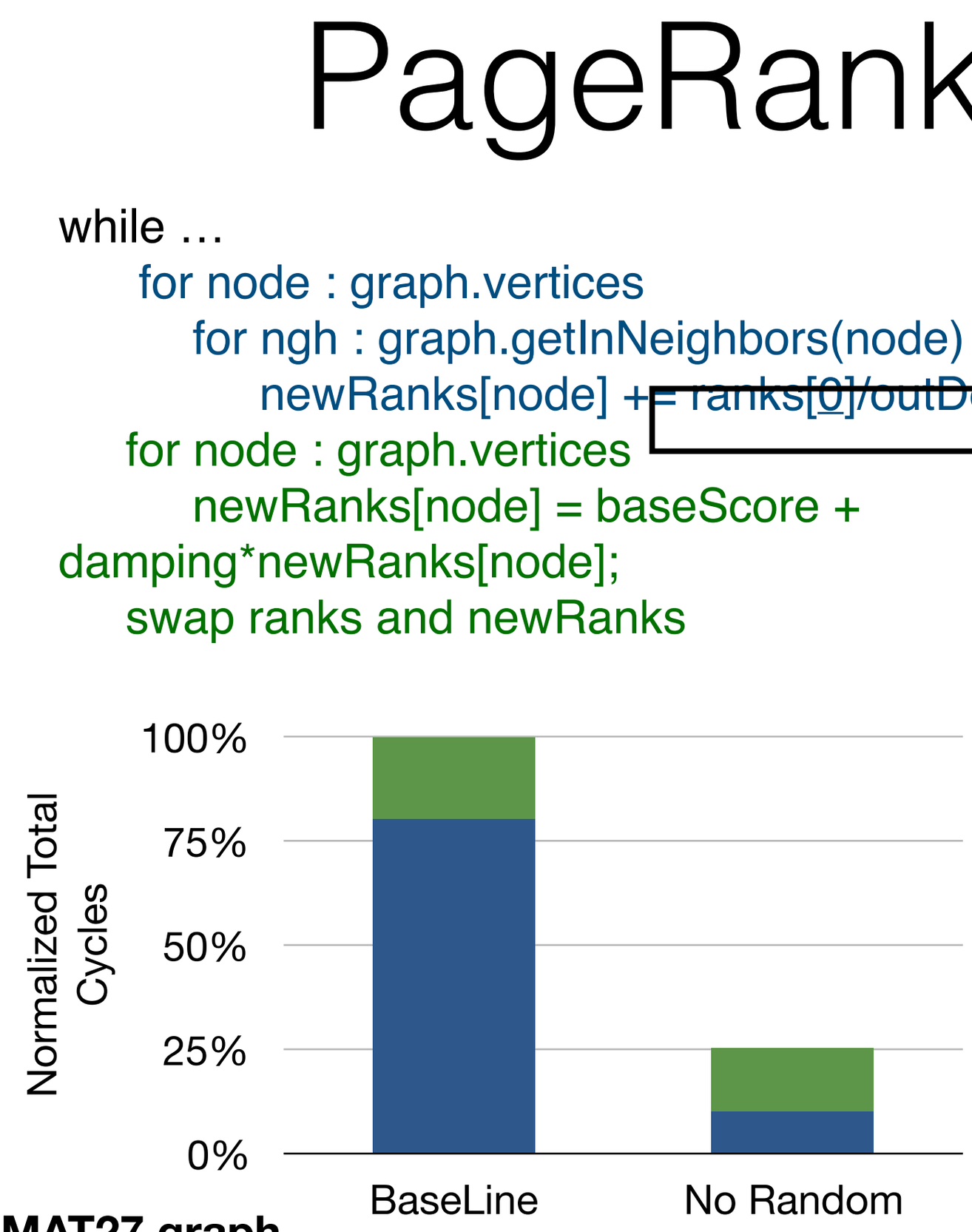



**on RMAT27 graph**

# PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[0]/outDegree[0];
    for node : graph.vertices
        newRanks[node] = baseScore +
damping*newRanks[node];
    swap ranks and newRanks
```

**Removing Random Accesses (Incorrect)**

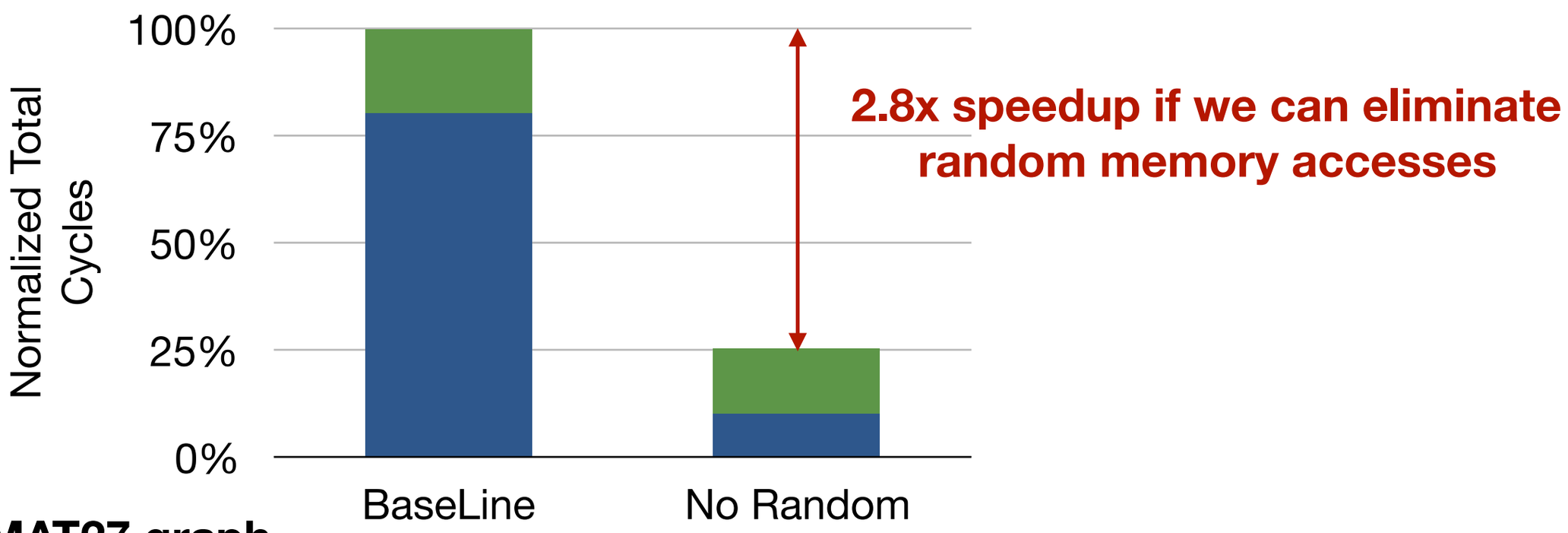



**on RMAT27 graph**

# PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[0]/outDegree[0];
    for node : graph.vertices
        newRanks[node] = baseScore +
damping*newRanks[node];
    swap ranks and newRanks
```

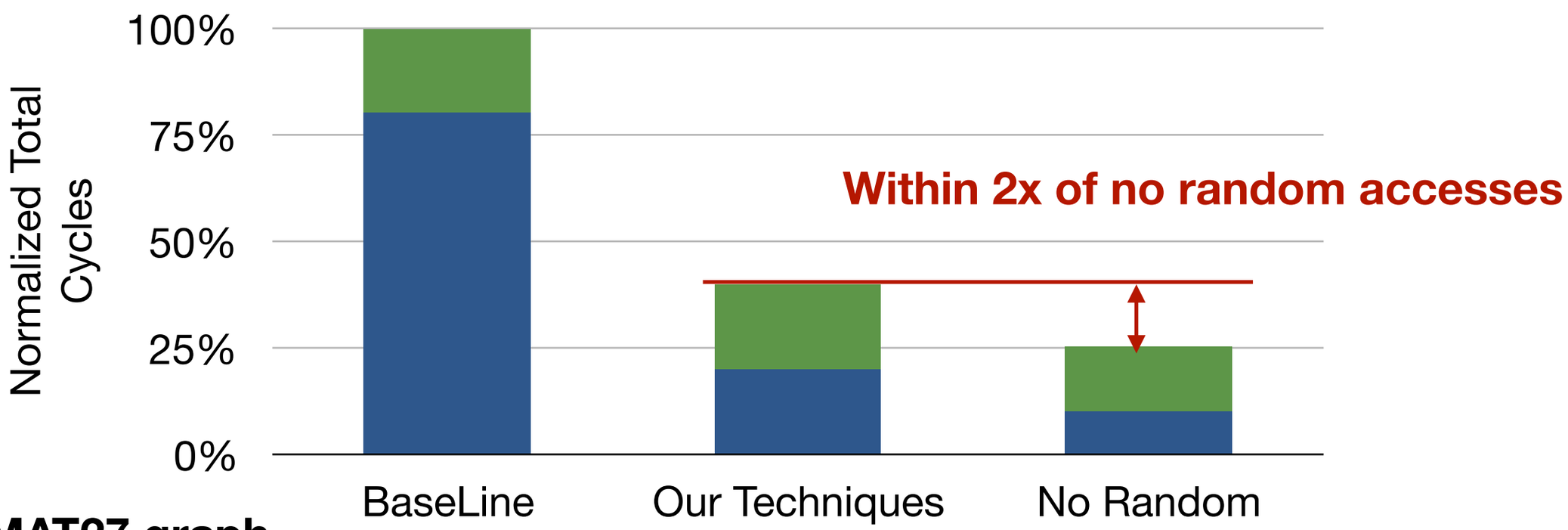
**Removing Random Accesses (Incorrect)**




**on RMAT27 graph**

# PageRank

```
while …
      for node : graph.vertices
          for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[0]/outDegree[0];
     for node : graph.vertices
          newRanks[node] = baseScore +
damping*newRanks[node];
swap ranks and newRanks
```




**on RMAT27 graph**

# Outline

- Motivation
- Related Works
- Frequency based Vertex Reordering
- Cache-aware Segmenting
- Evaluation

# Related Work

- Distributed Graph Systems
  - Shared memory efficiency is a key component of distributed graph processing systems (PowerGraph, GraphLab, Pregel..)
- Shared-memory Graph Systems
  - Frameworks (Ligra, Galois, GraphMat ..) did not focus on cache optimizations
  - Milk [PACT16], Propagation Blocking[IPDPS17]
- Out-of-core Systems (GraphChi, FlashGraph, BigSparse …)

# Outline

- Motivation
- Related Works
- Frequency based Vertex Reordering
- Cache-aware Segmenting
- Evaluation

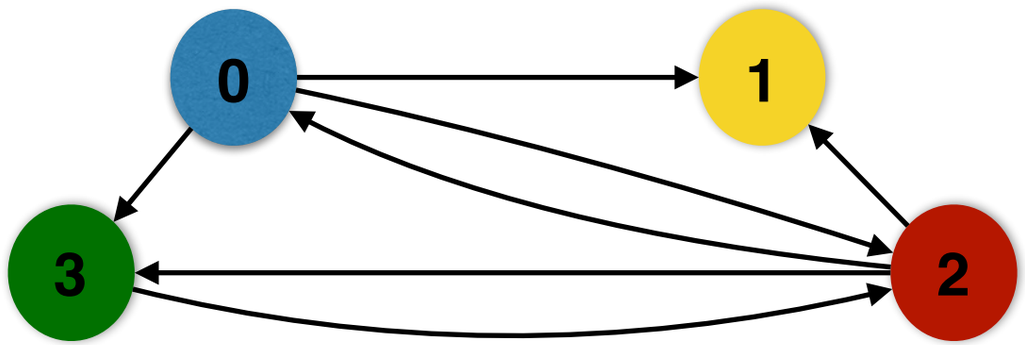
# Frequency based Vertex Reordering

- Key Observations
  - Cache lines are underutilized
  - Certain vertices are much more likely to be accessed than other vertices

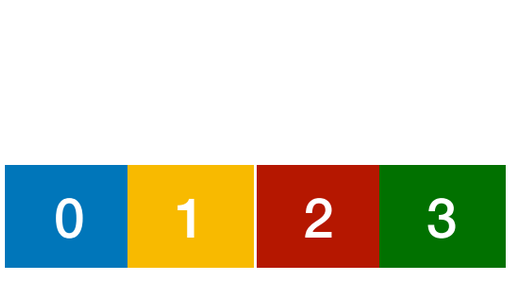
# Frequency based Vertex Reordering

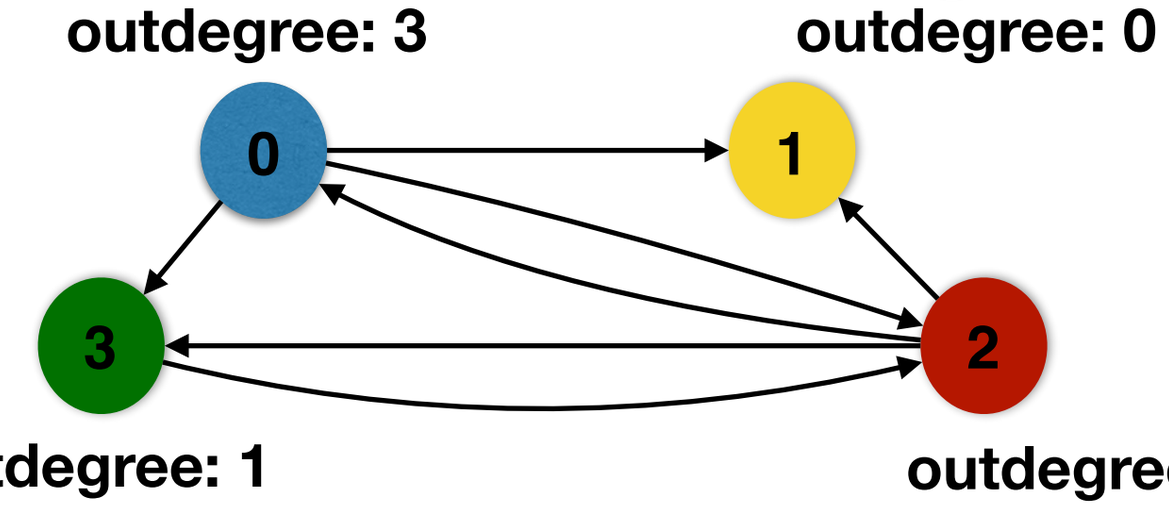
- Key Observations
  - Cache lines are underutilized
  - Certain vertices are much more likely to be accessed than other vertices
- Design
  - Group together the frequently accessed nodes
  - Keep the ordering of average degree nodes

# Frequency based Vertex Reordering

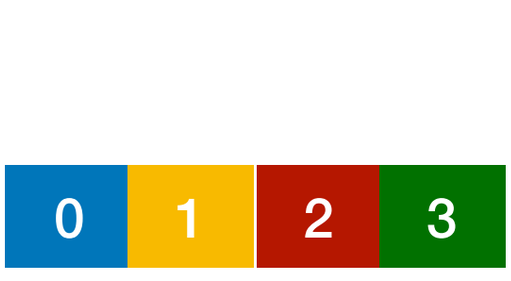



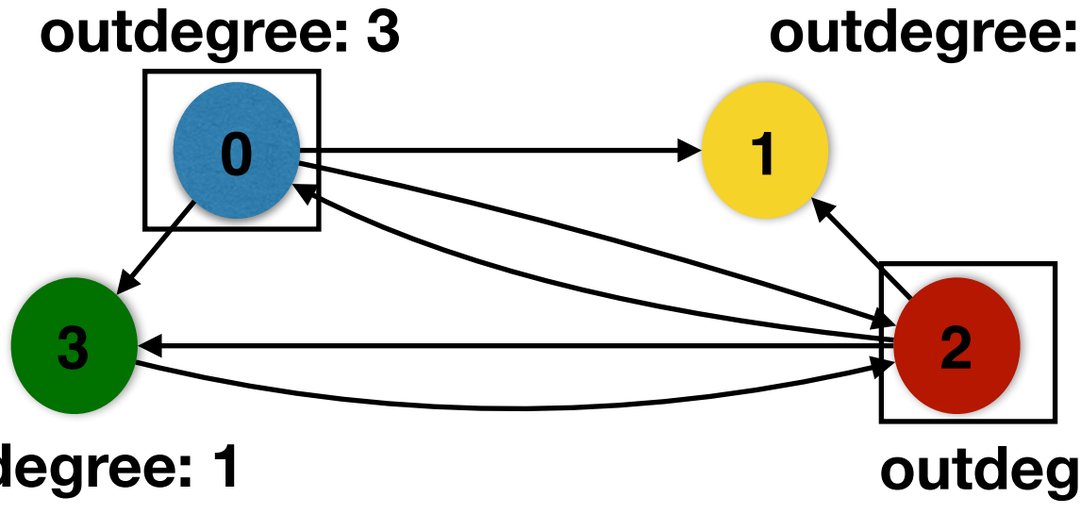
# Frequency based Vertex Reordering



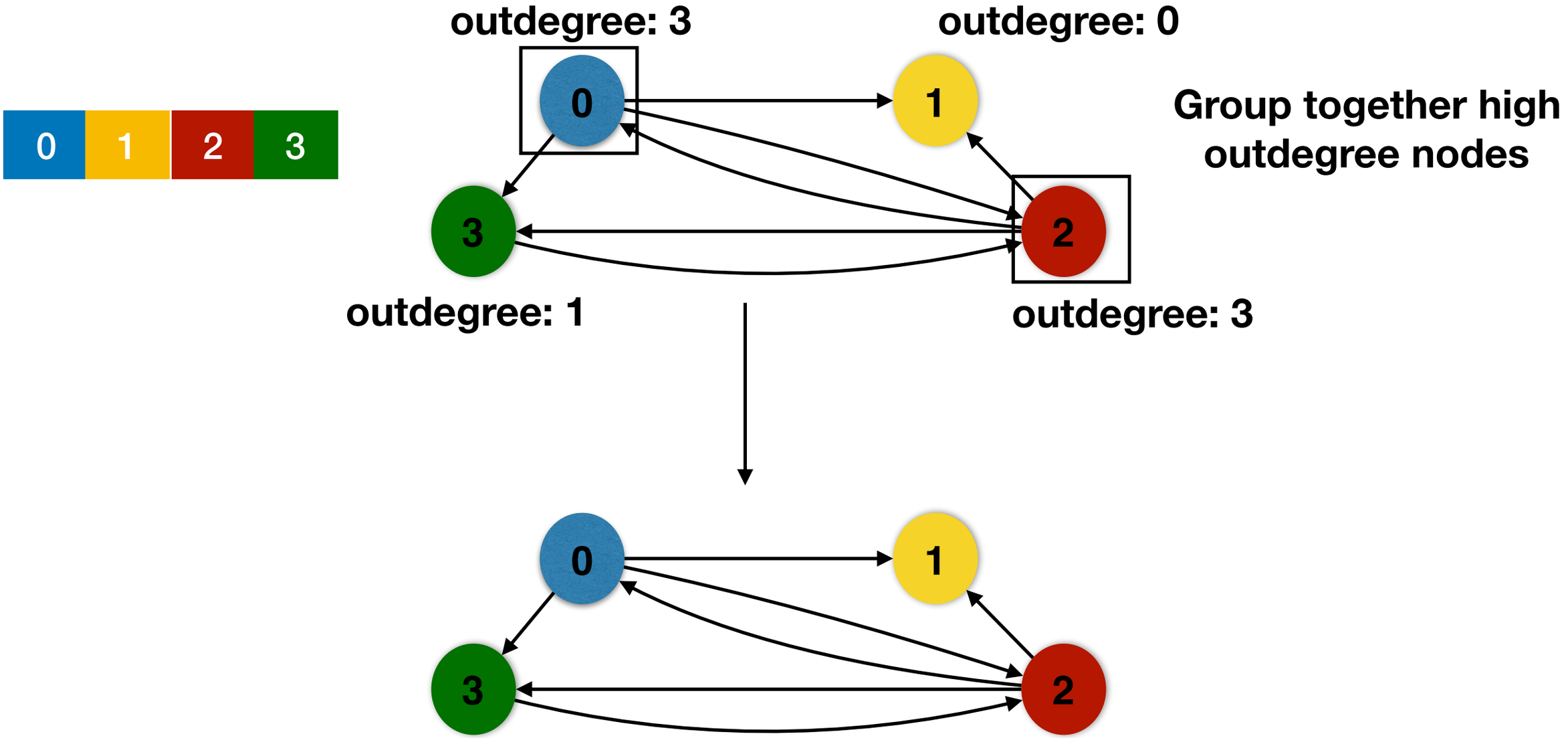
# Frequency based Vertex Reordering



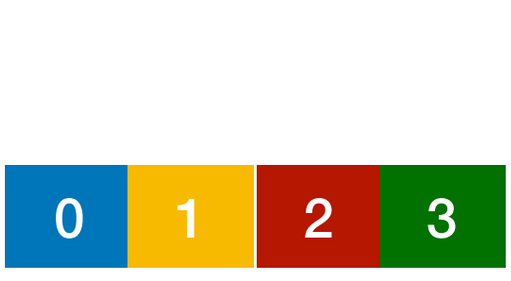



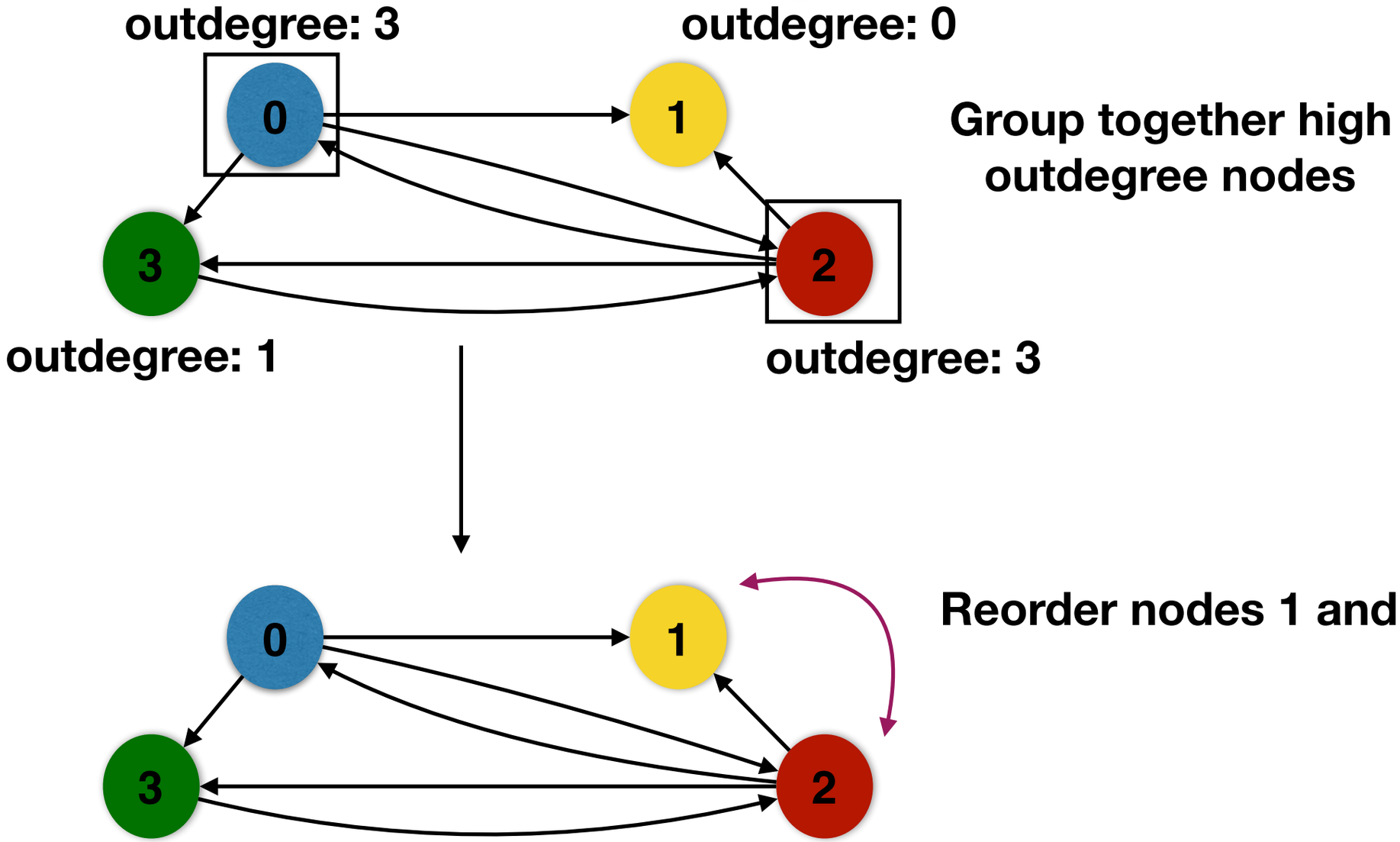

**Group together high outdegree nodes**

# Frequency based Vertex Reordering



# Frequency based Vertex Reordering

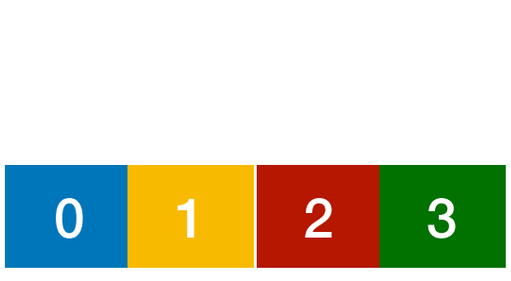


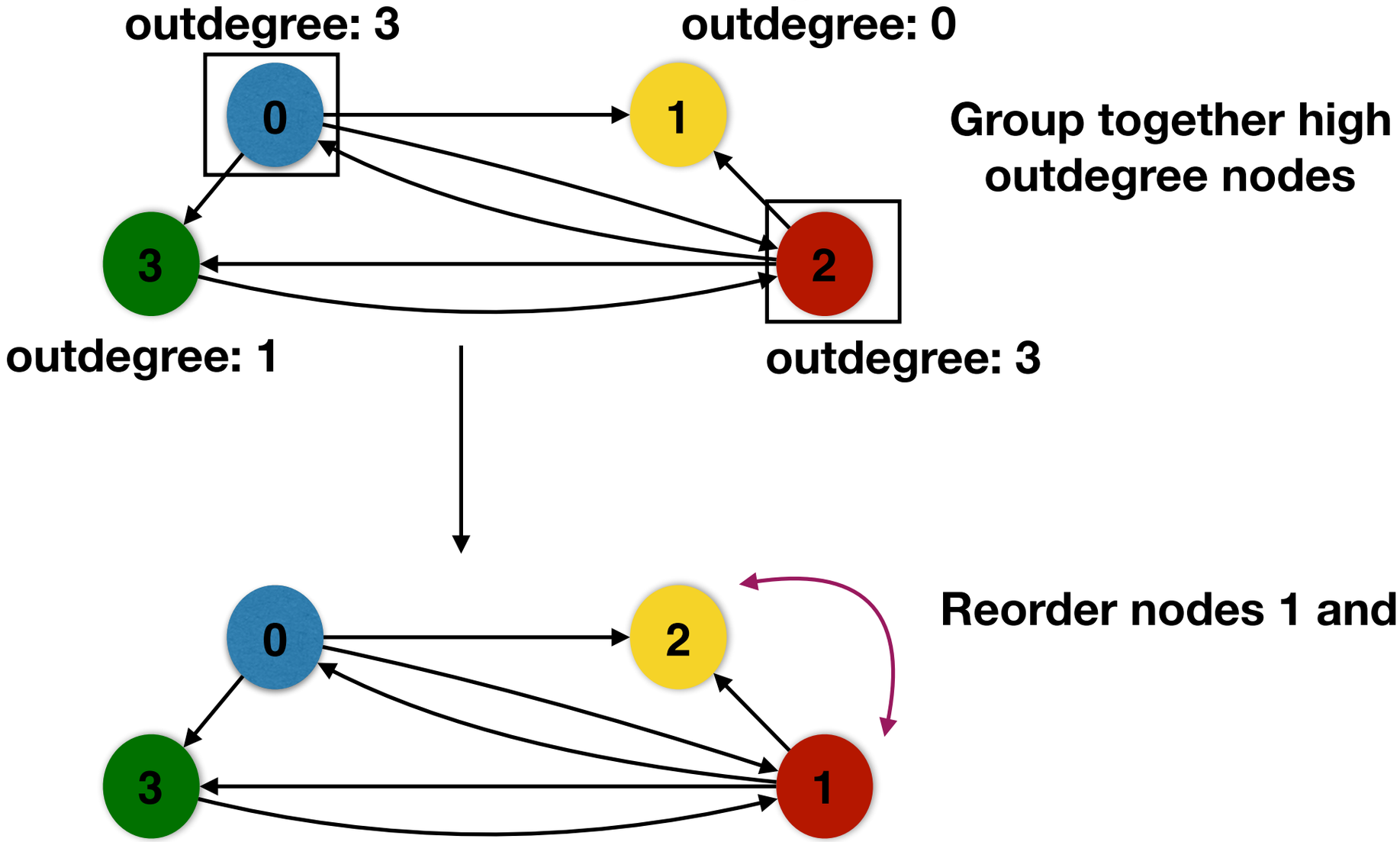




**Group together high outdegree nodes**

**Reorder nodes 1 and 2**

# Frequency based Vertex Reordering

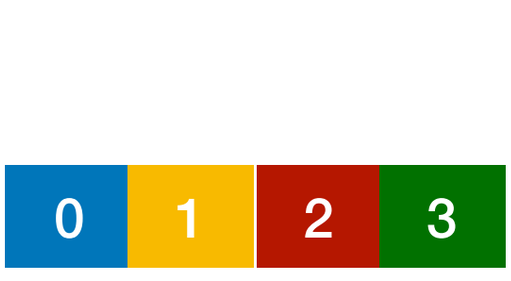


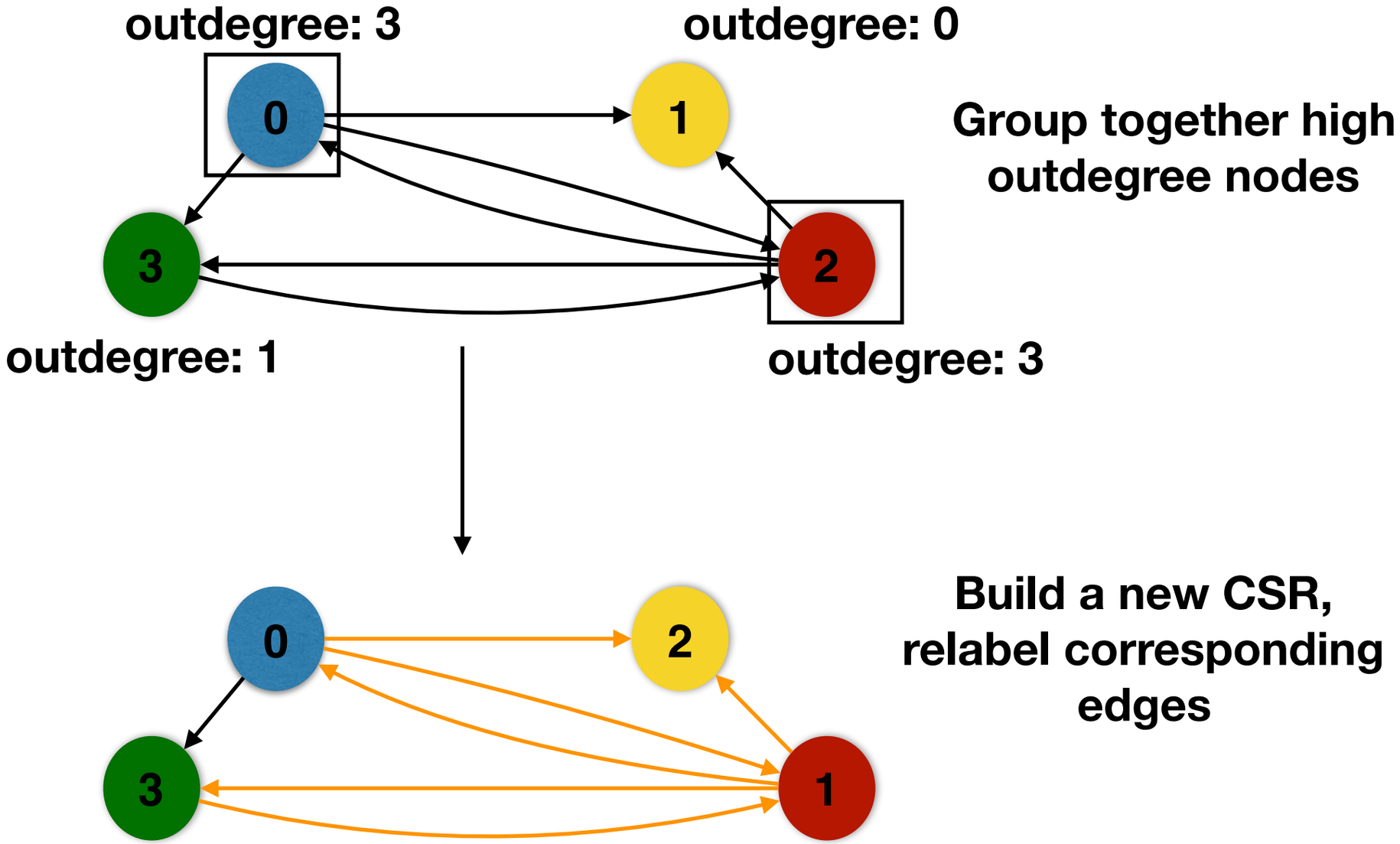




**Group together high outdegree nodes**

**Reorder nodes 1 and 2**

# Frequency based Vertex Reordering

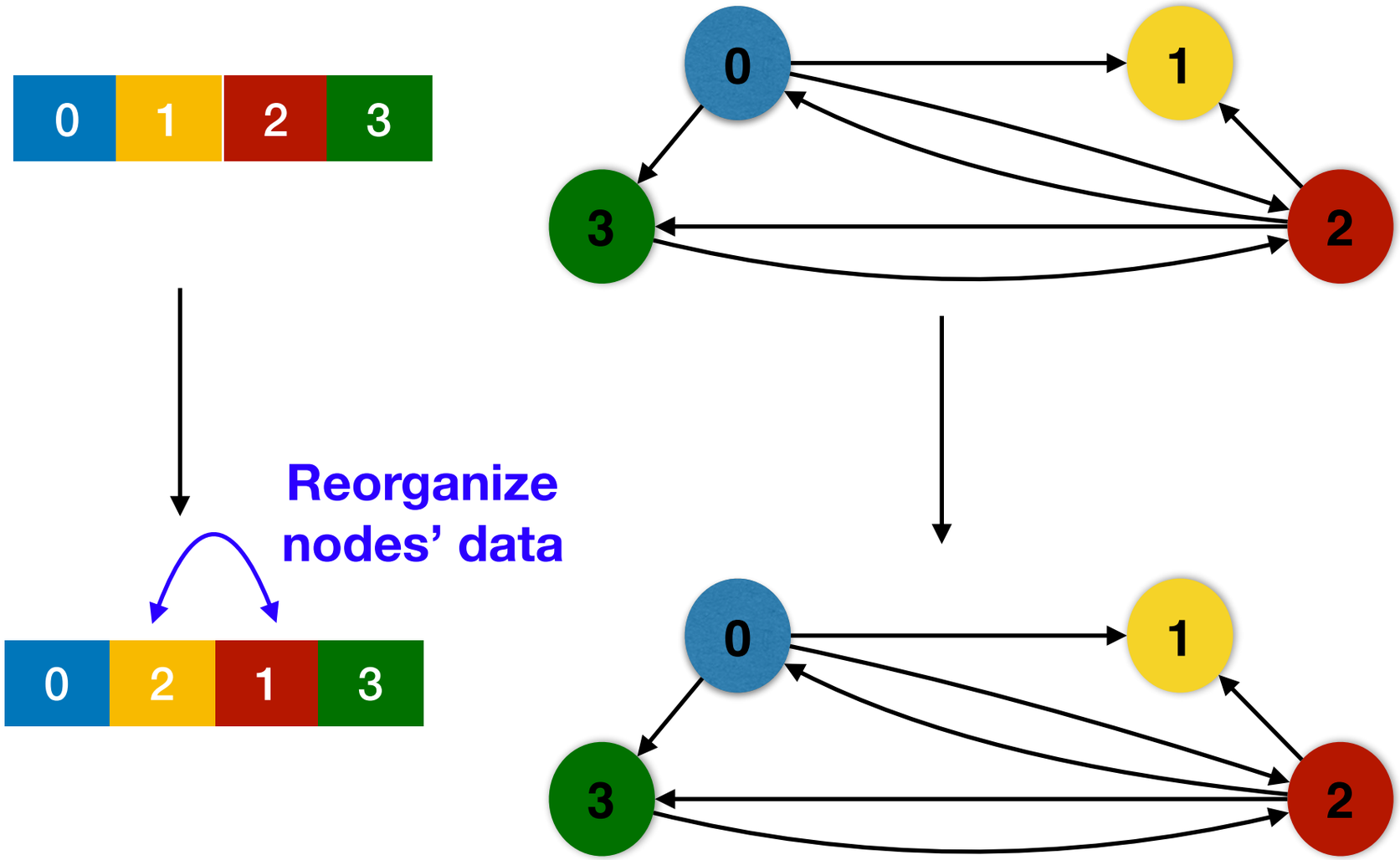


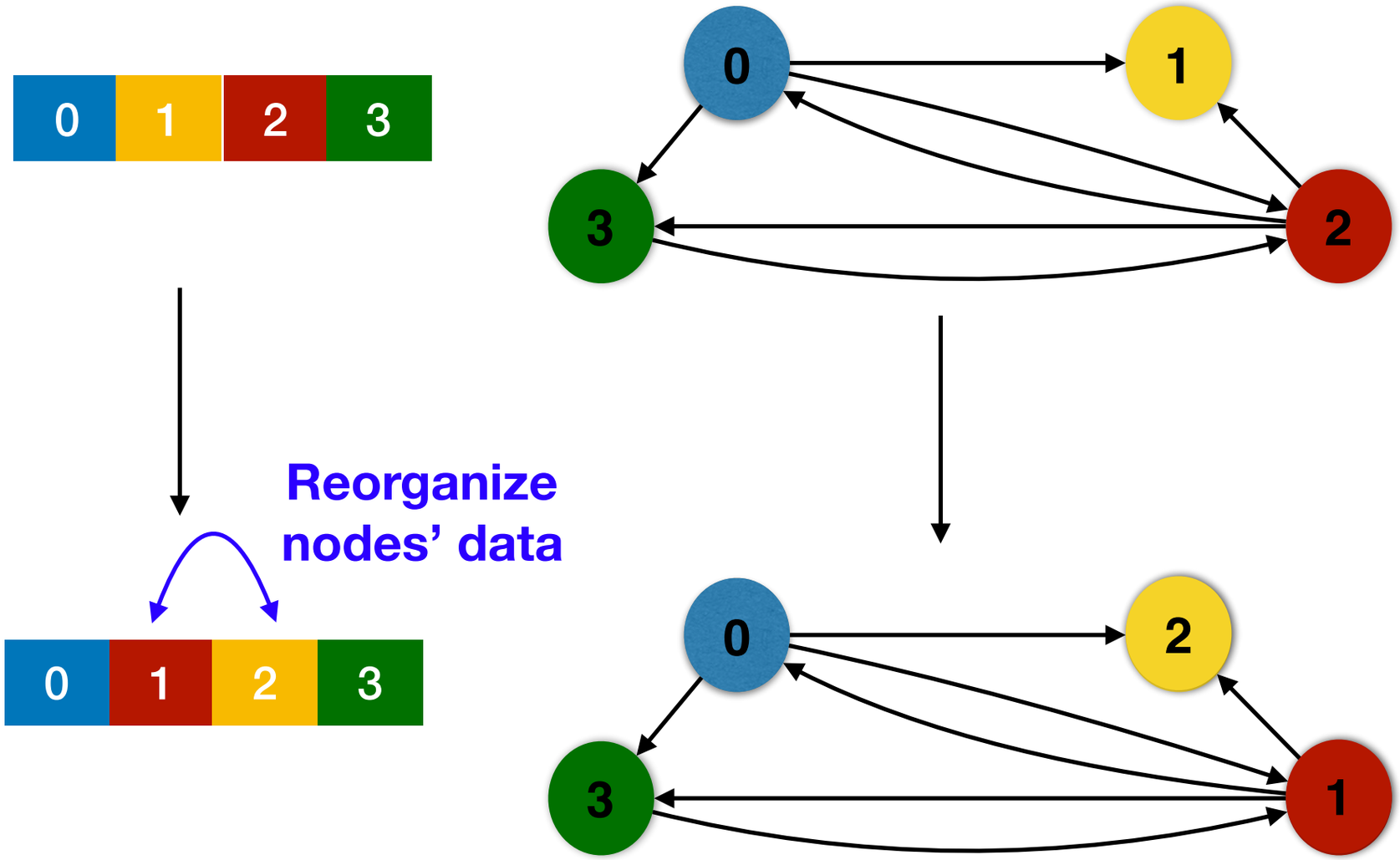

**Group together high outdegree nodes**

**Build a new CSR, relabel corresponding edges**

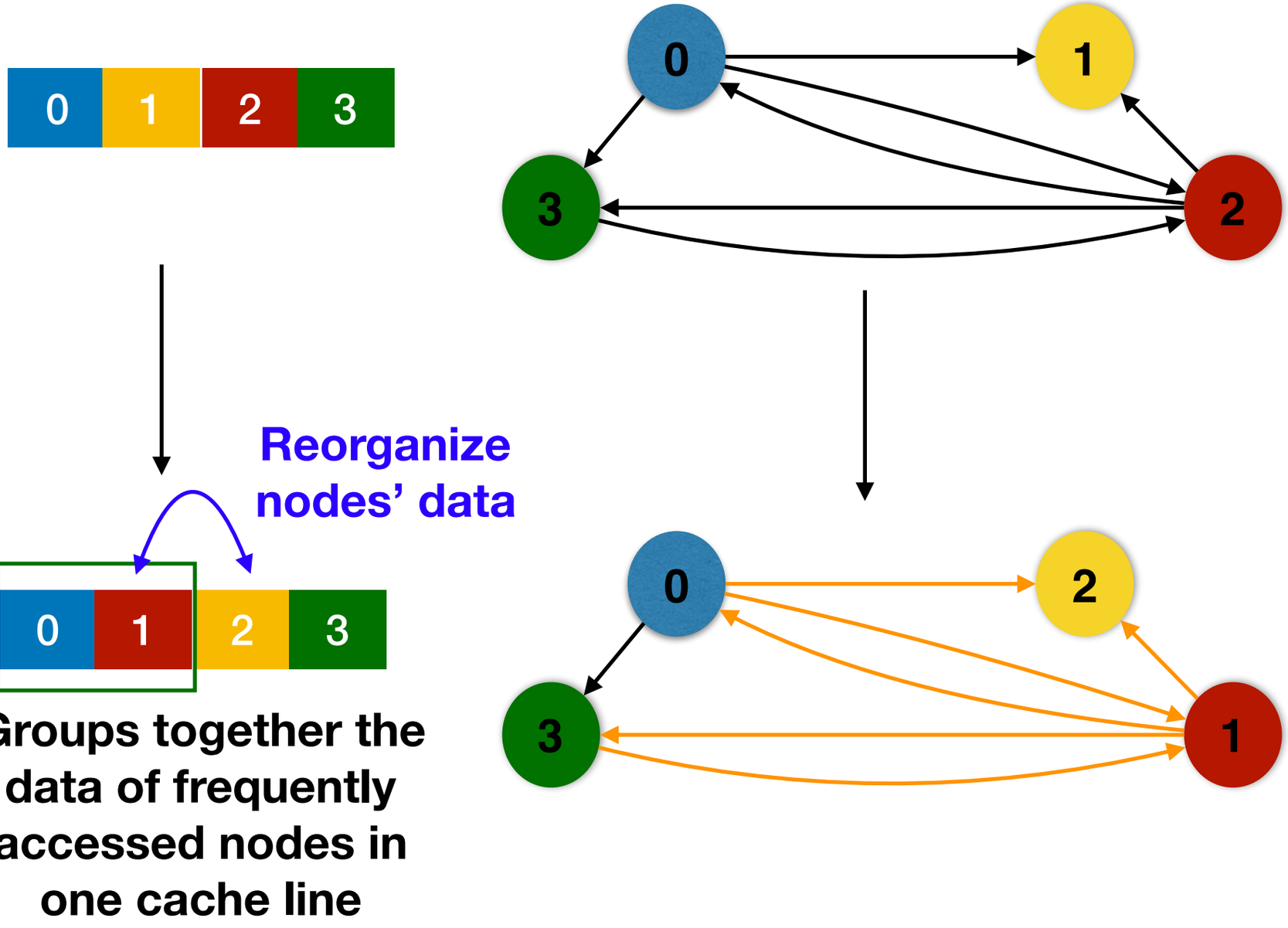
# Frequency based Vertex Reordering



# Frequency based Vertex Reordering



# Frequency based Vertex Reordering

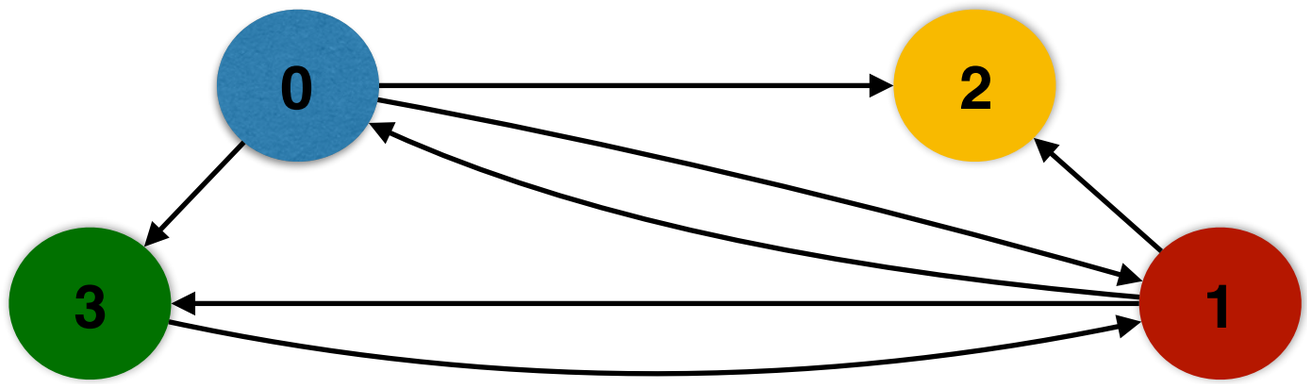


# PageRank

```
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

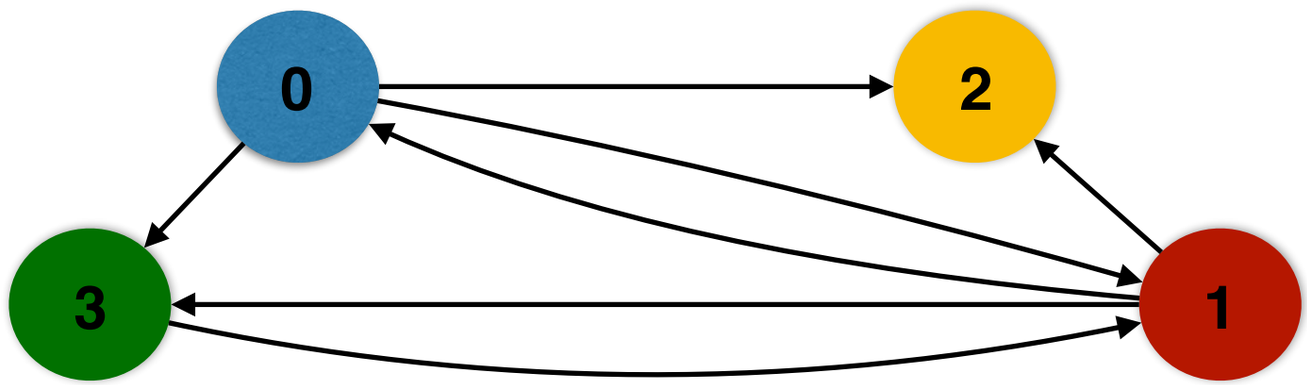
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore +
damping*newRanks[node];
swap ranks and newRanks
```

**Focus on the random memory accesses on ranks array**

**Cache**

**#hits: 0**

**#misses: 0**

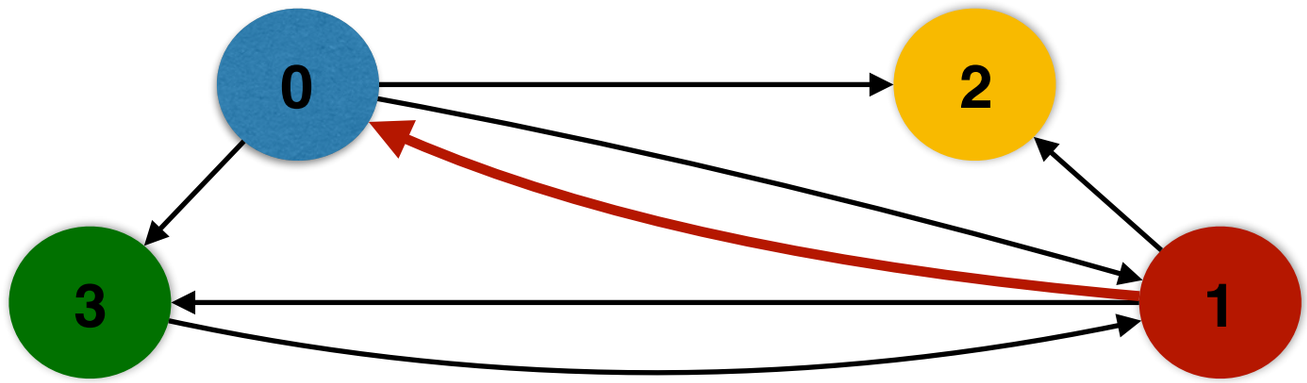
# PageRank

```
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

**Cache**

**#hits: 0**
**#misses: 0**

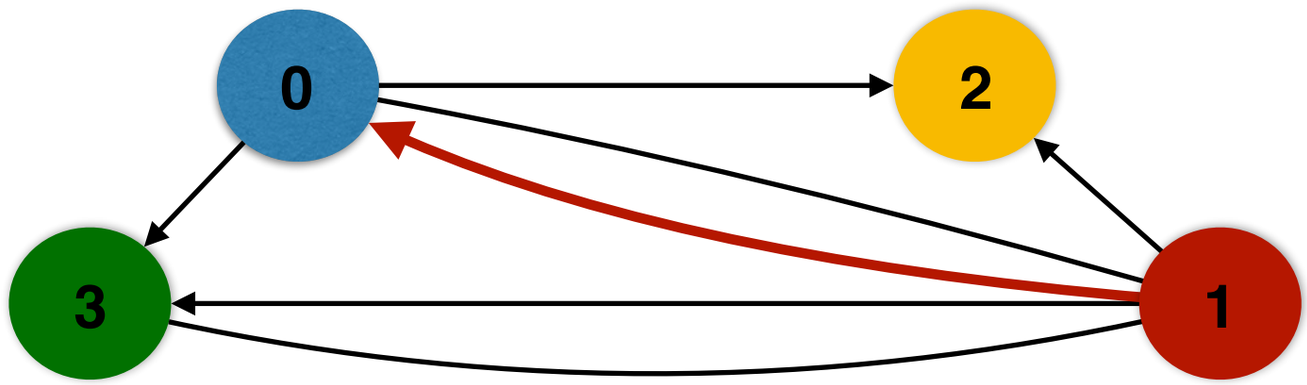



# PageRank

```
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**

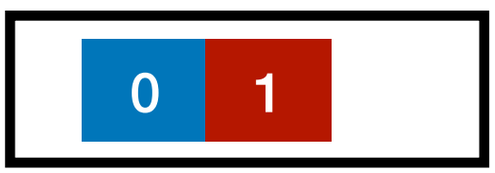
**#hits: 0**
**#misses: 0**

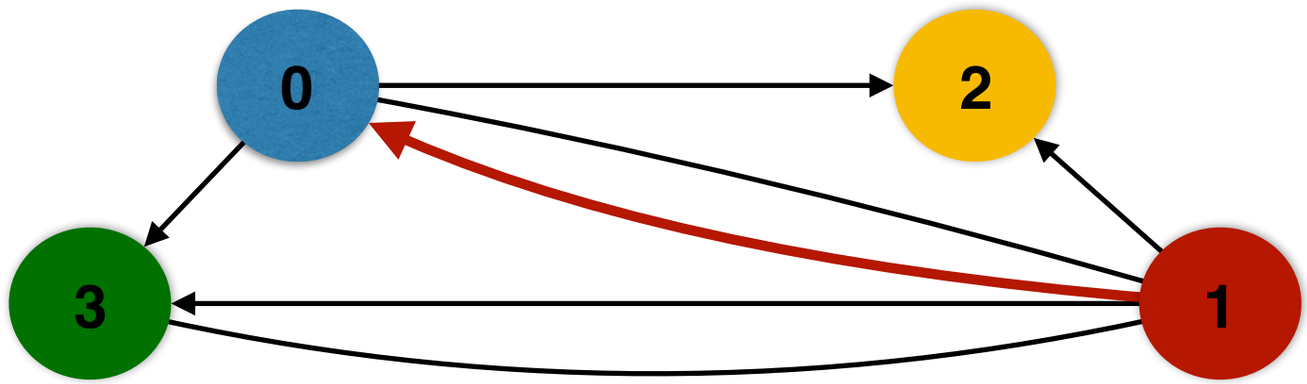
# PageRank

```
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**



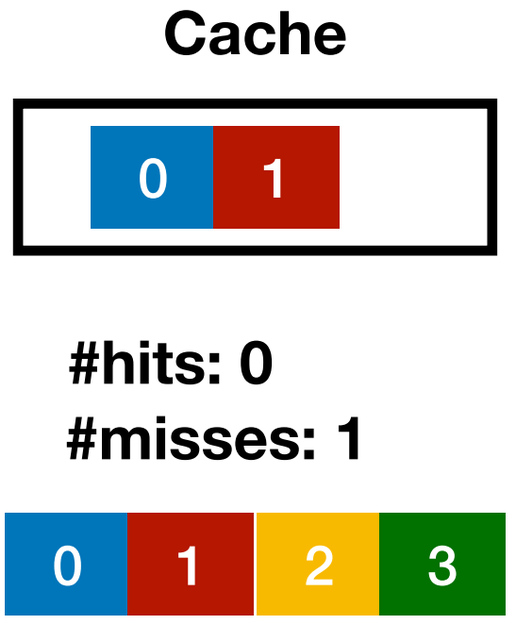

**#hits: 0**
**#misses: 1**

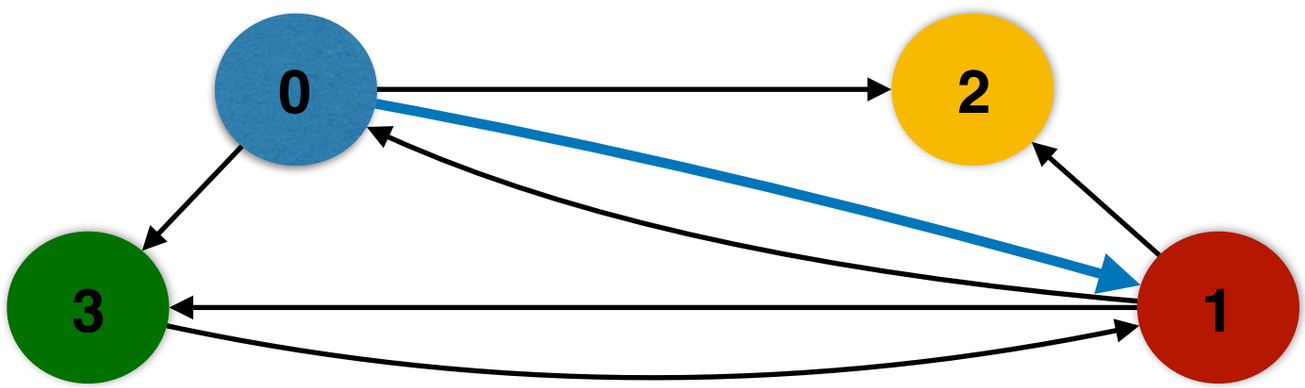



# PageRank

```
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

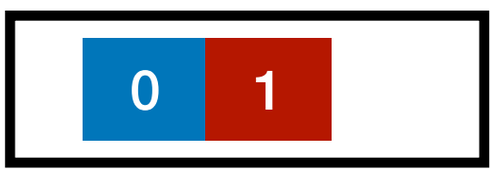


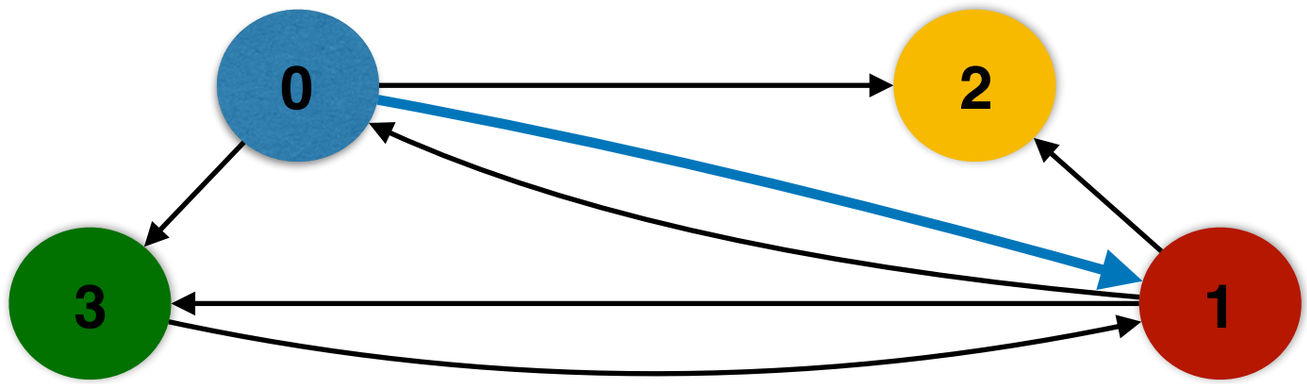
# PageRank

```
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**



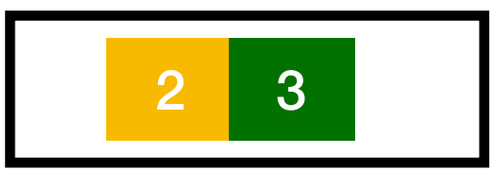

**#hits: 1**
**#misses: 1**

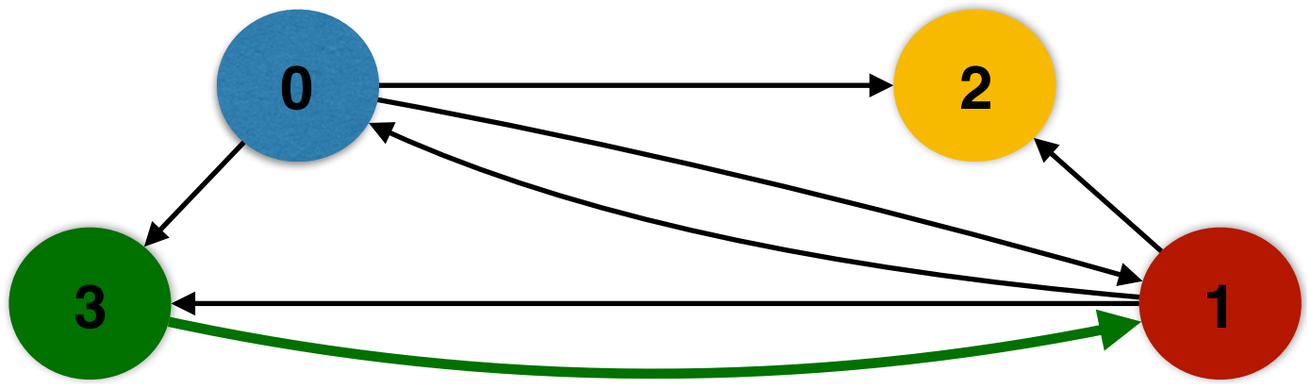
# PageRank

```
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**



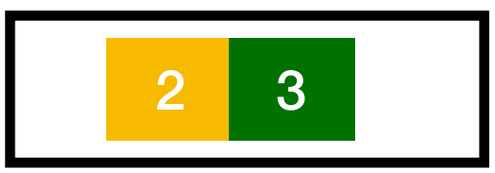

**#hits: 1**
**#misses: 2**

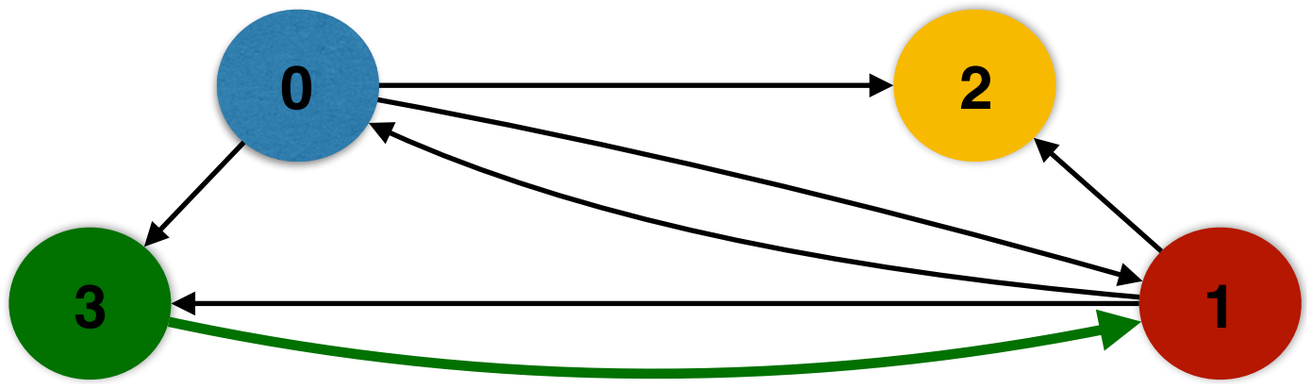
# PageRank

```
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: 1**
**#misses: 2**

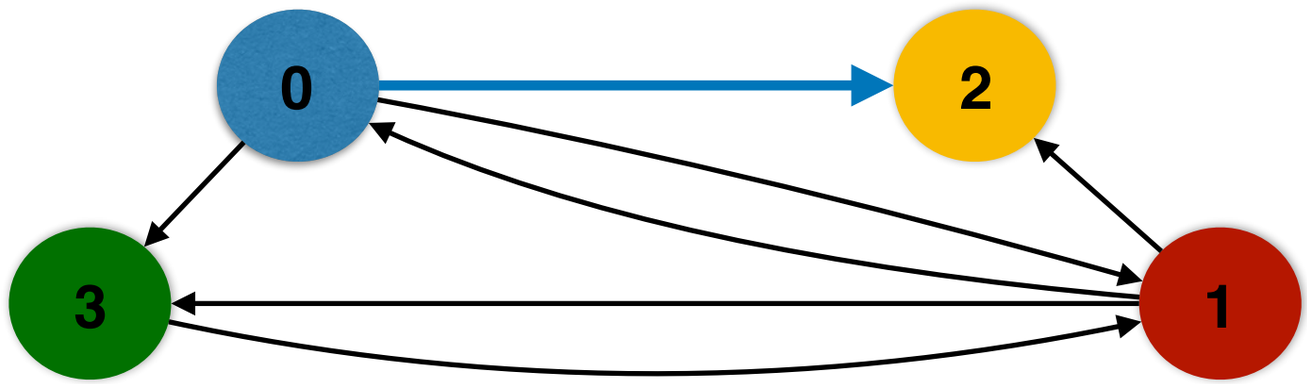



# PageRank

```
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
```

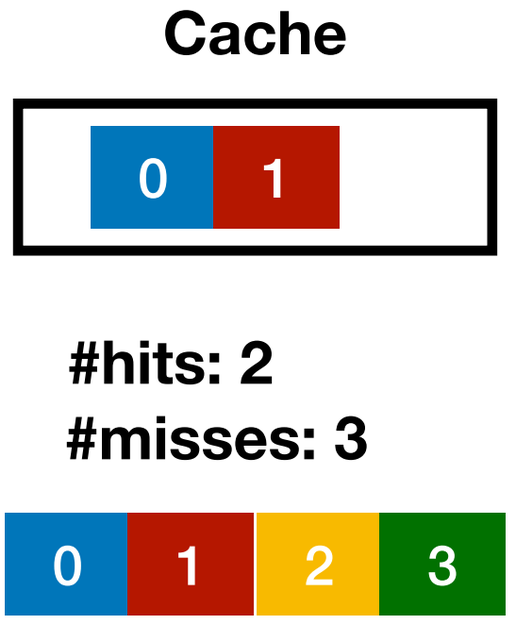
**Cache**

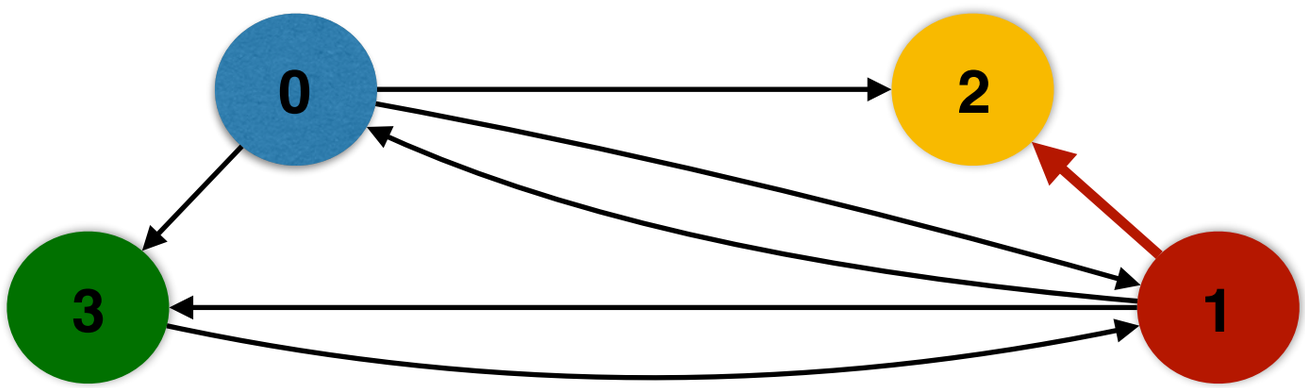

**#hits: 1**
**#misses: 3**

# 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
```

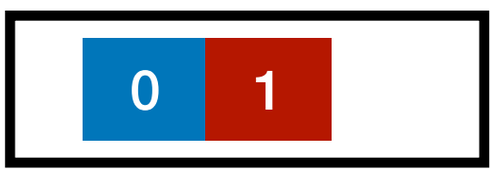


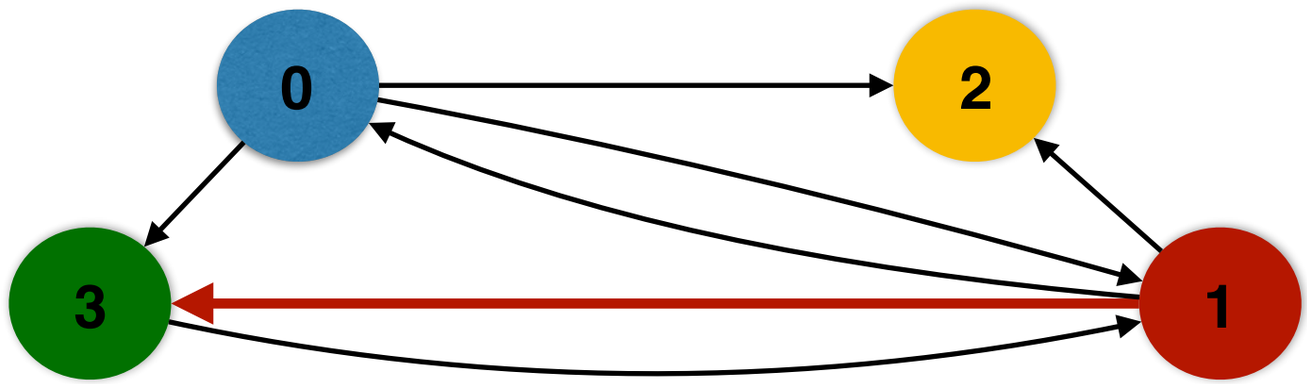
# PageRank

```
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**



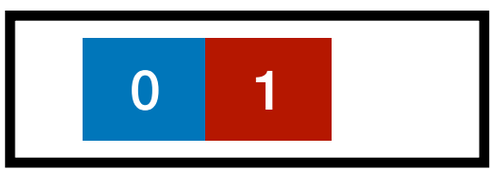

**#hits: 3**
**#misses: 3**

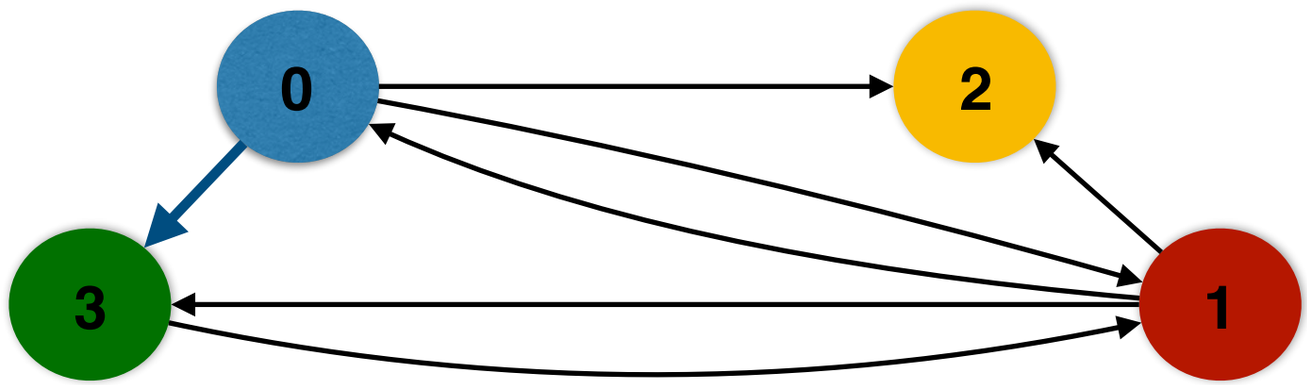
# PageRank

```
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**



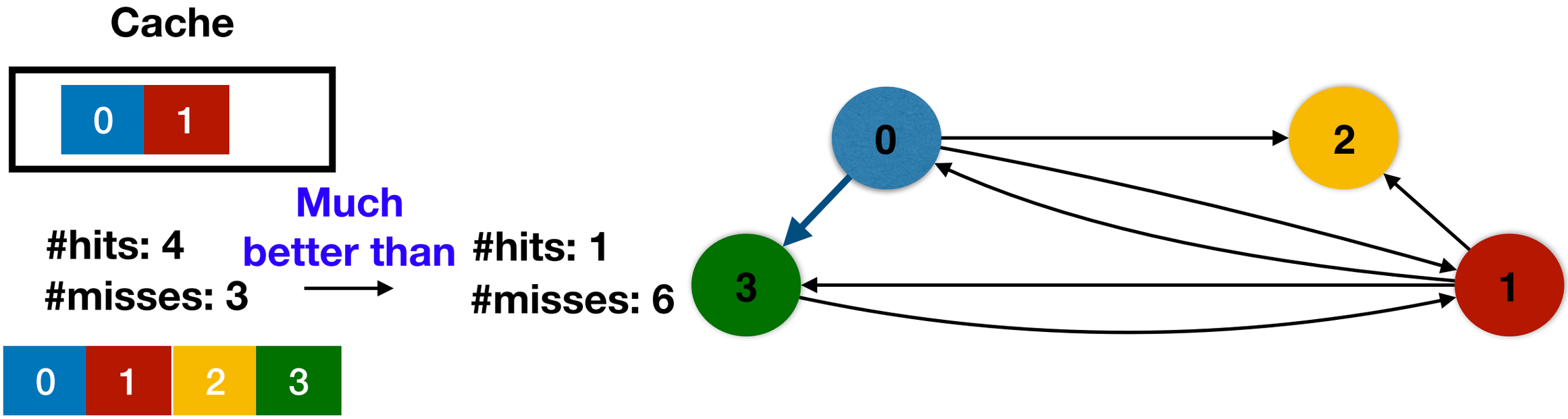

**#hits: 4**
**#misses: 3**

# 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];
```

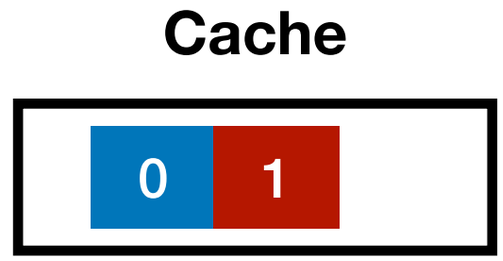
swap ranks and newRanks



# 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];
```

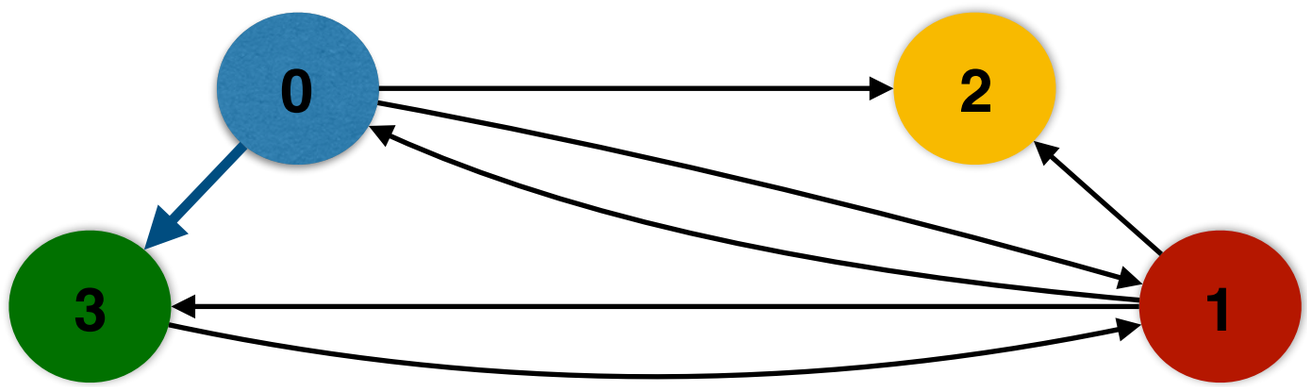
swap ranks and newRanks




**#hits: 4**
**#misses: 3**


**#hits: 1**
**#misses: 6**

**Better cache line utilization**



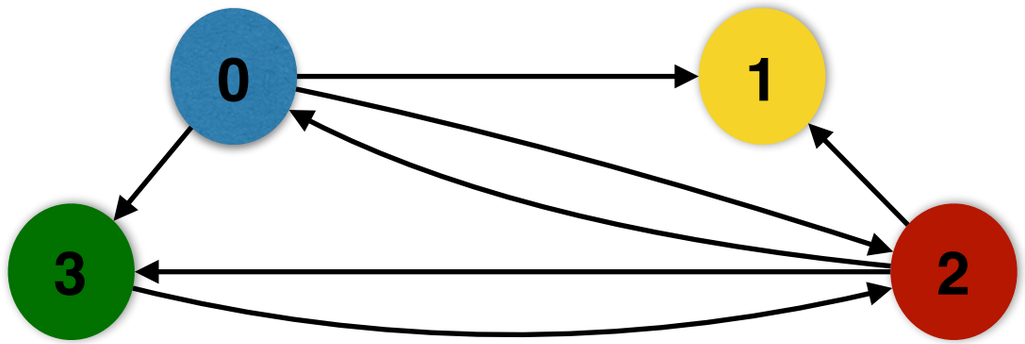
# Outline

- Motivation
- Related Works
- Frequency based Vertex Reordering
- Cache-aware Segmenting
- Evaluation

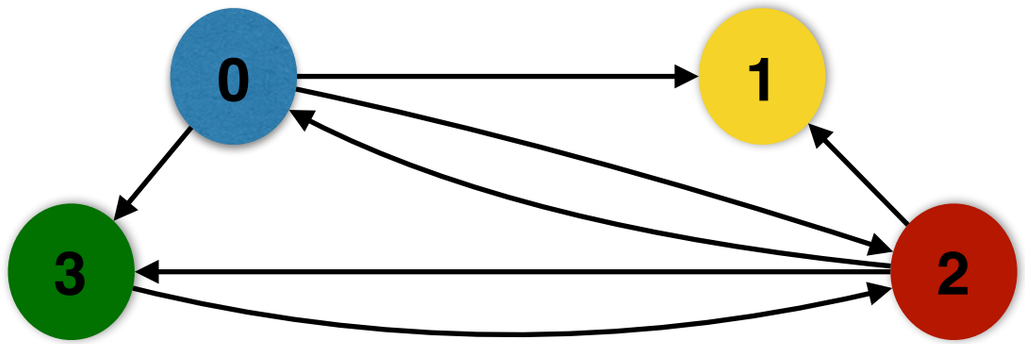
# Cache-aware Segmenting

- Design
  - Partition the graph into subgraphs where the random access are limited to LLC
  - Process each partition sequentially and accumulate rank contributions for each partition
  - Merge the rank contributions from all subgraphs

# Graph Partitioning

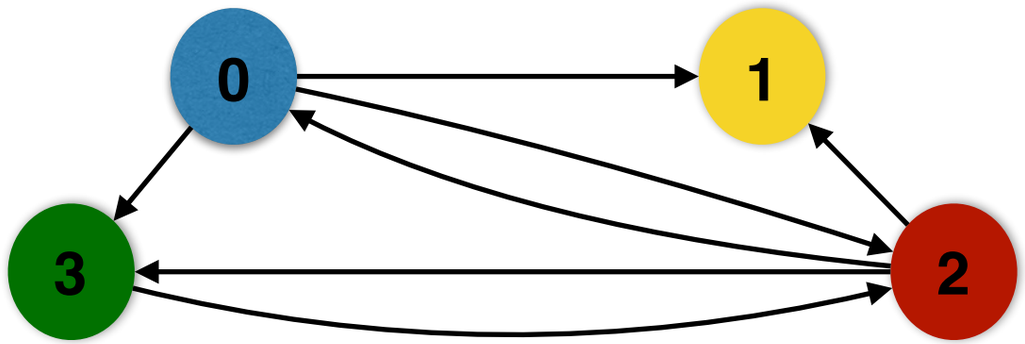


# Graph Partitioning



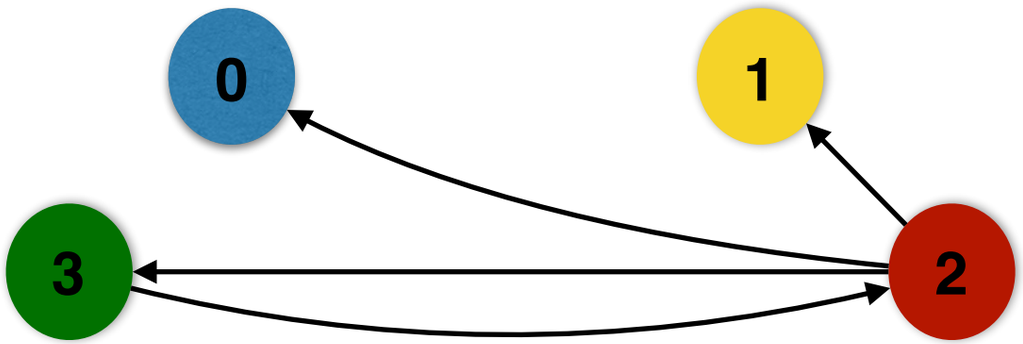

**Partitions the original graph into subgraphs that only access a subset of nodes' data**

# Graph Partitioning

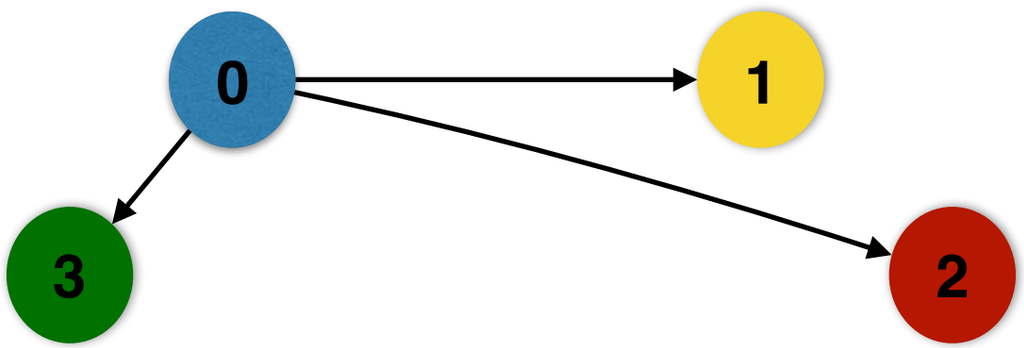



**Partitions the original graph into subgraphs that only access a subset of nodes' data**

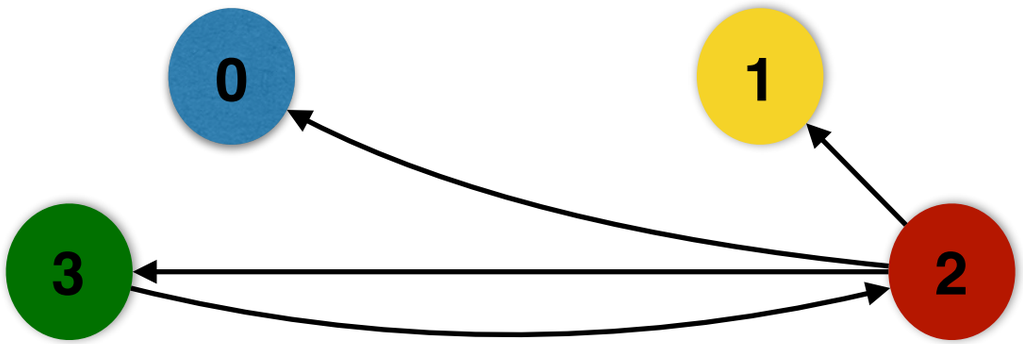
# Graph Partitioning



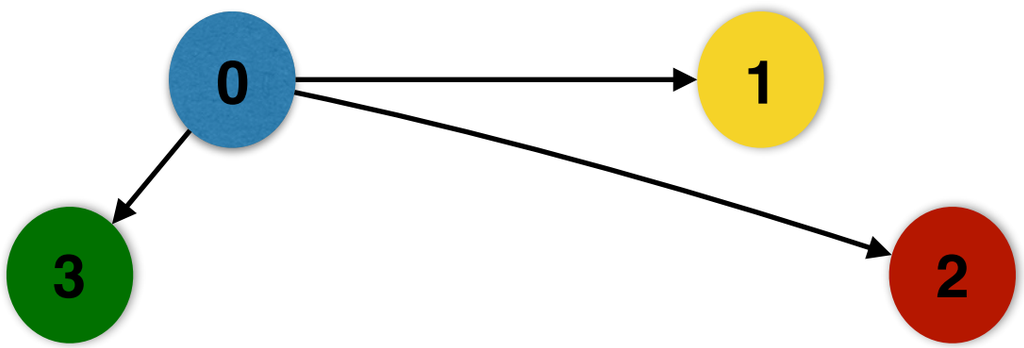

**Partitions the original graph into subgraphs that only access a subset of nodes' data**



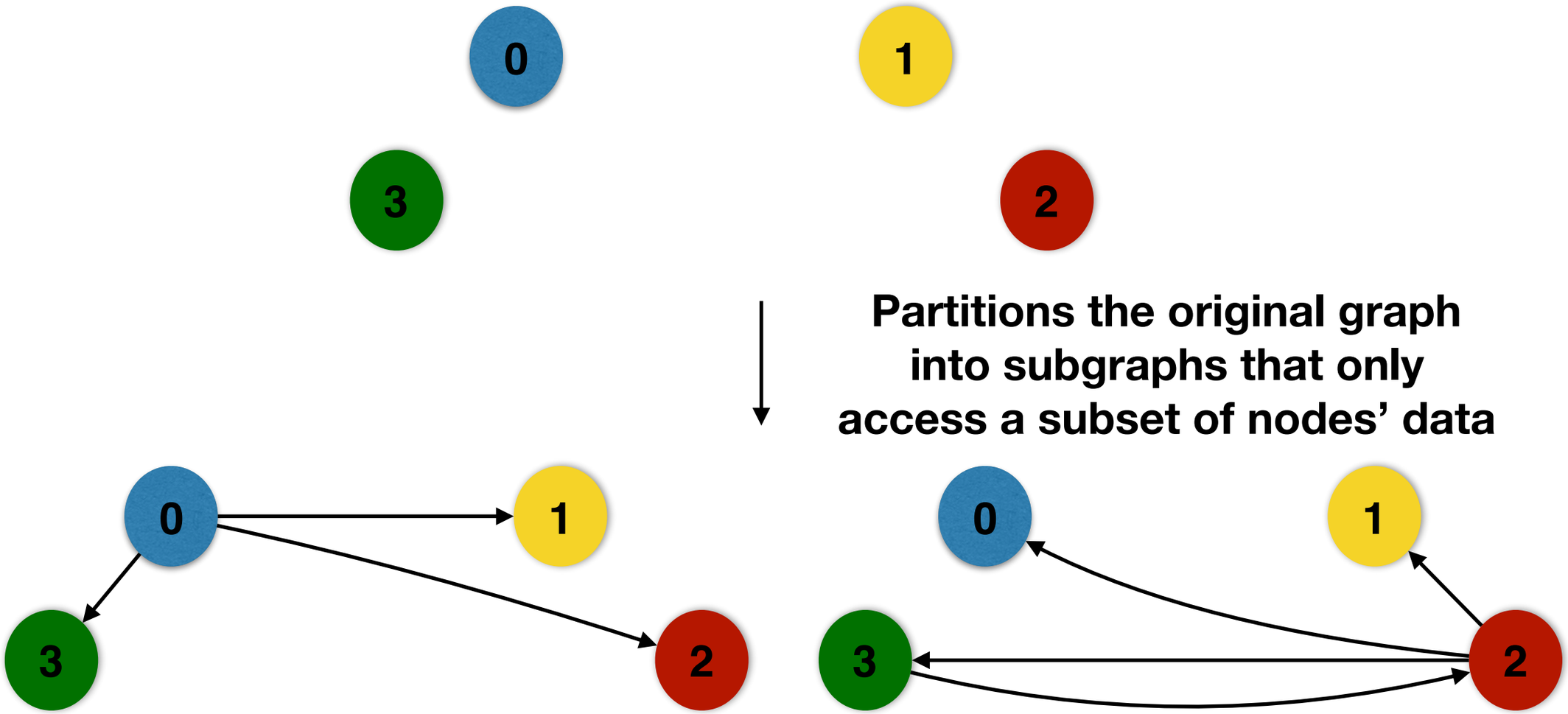
# Graph Partitioning



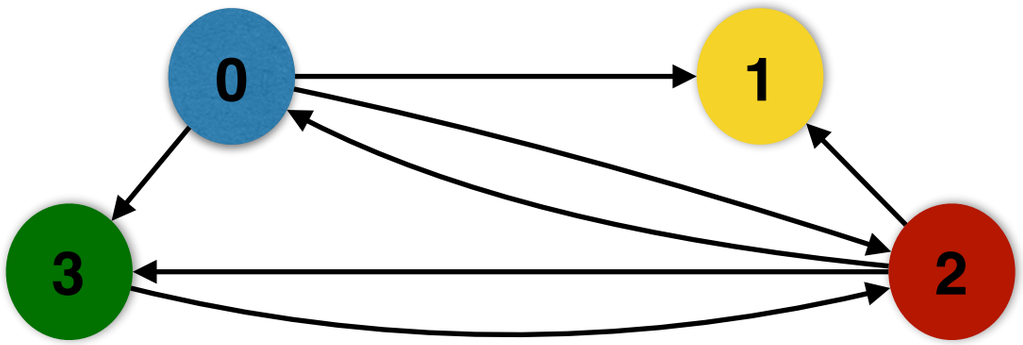

**Partitions the original graph into subgraphs that only access a subset of nodes' data**



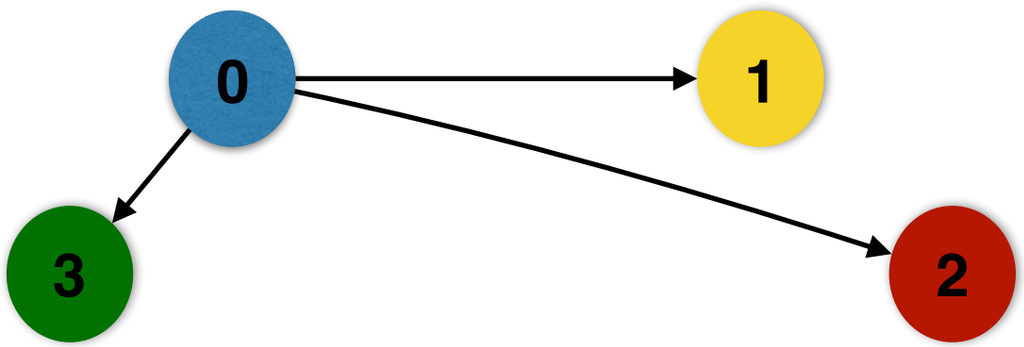
# Graph Partitioning



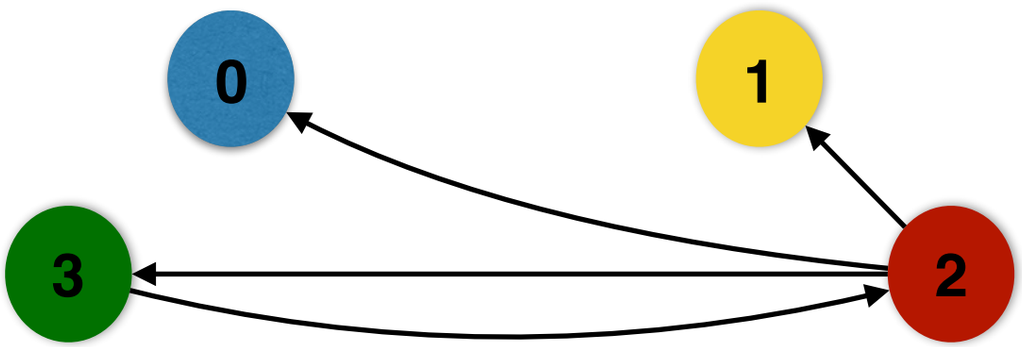
# Graph Partitioning




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



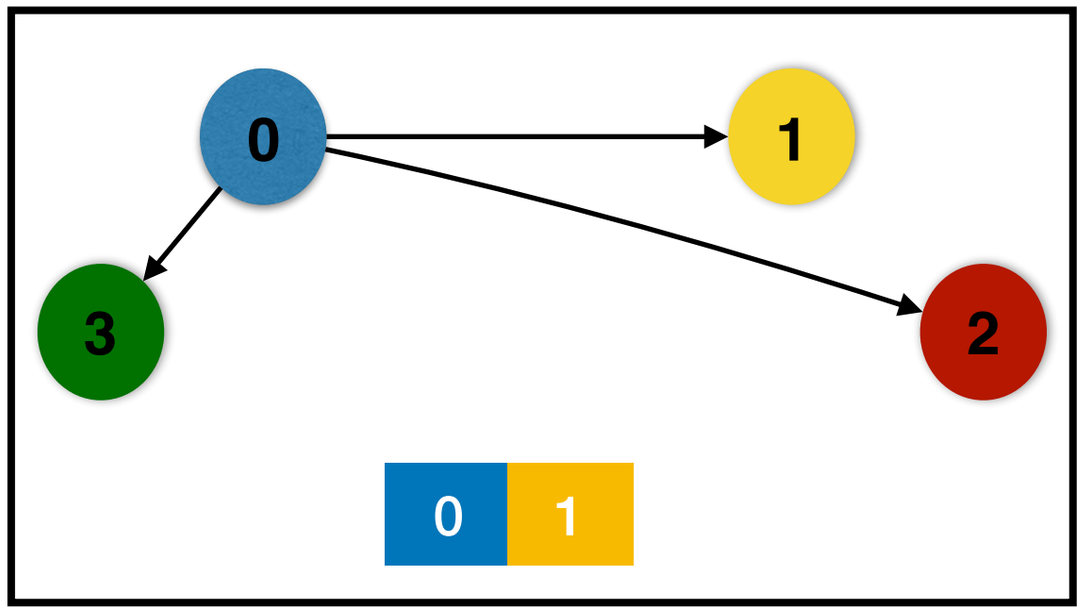


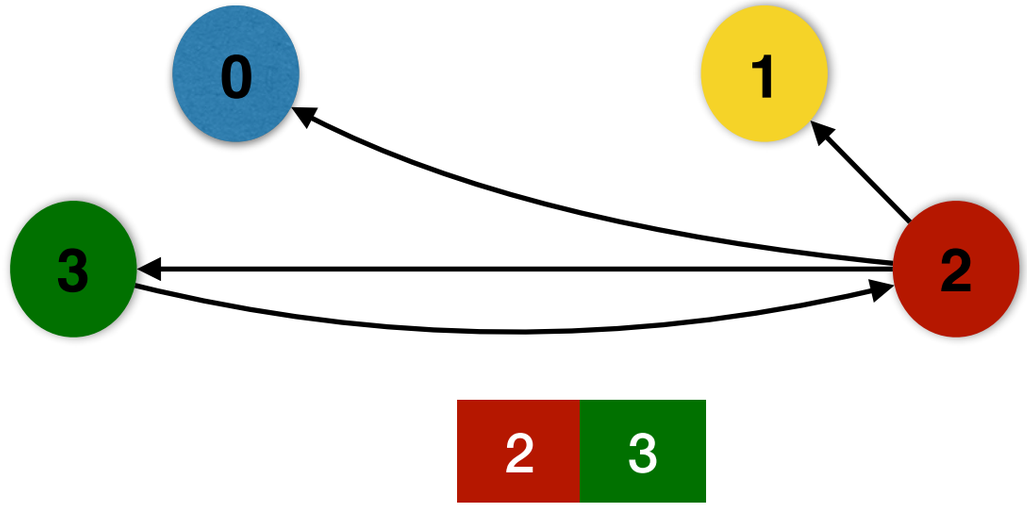



# Graph Processing

**Cache**

**#hits: 0**
**#misses: 0**

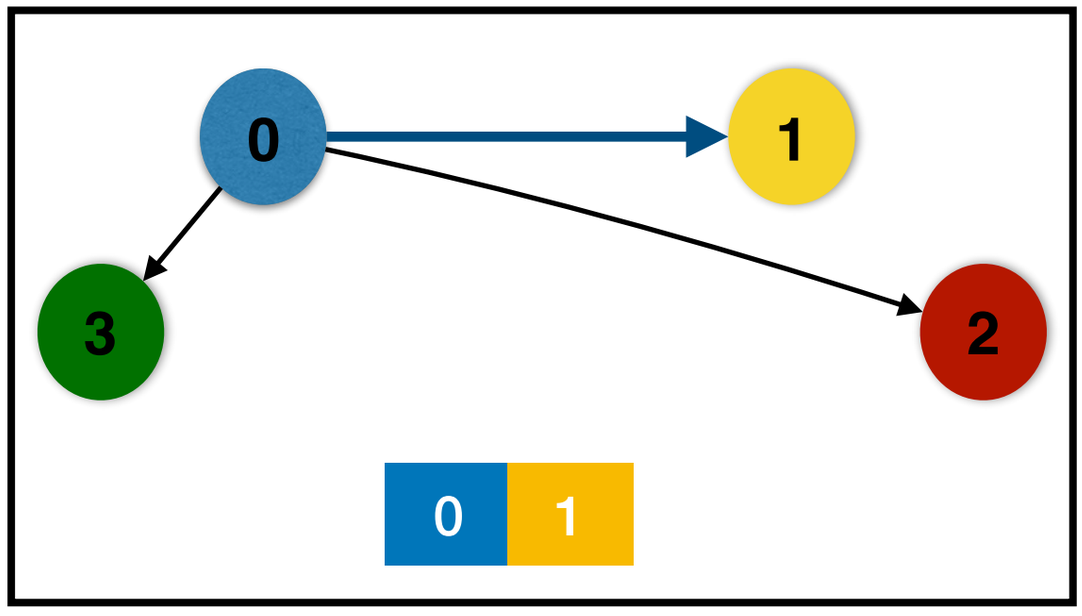


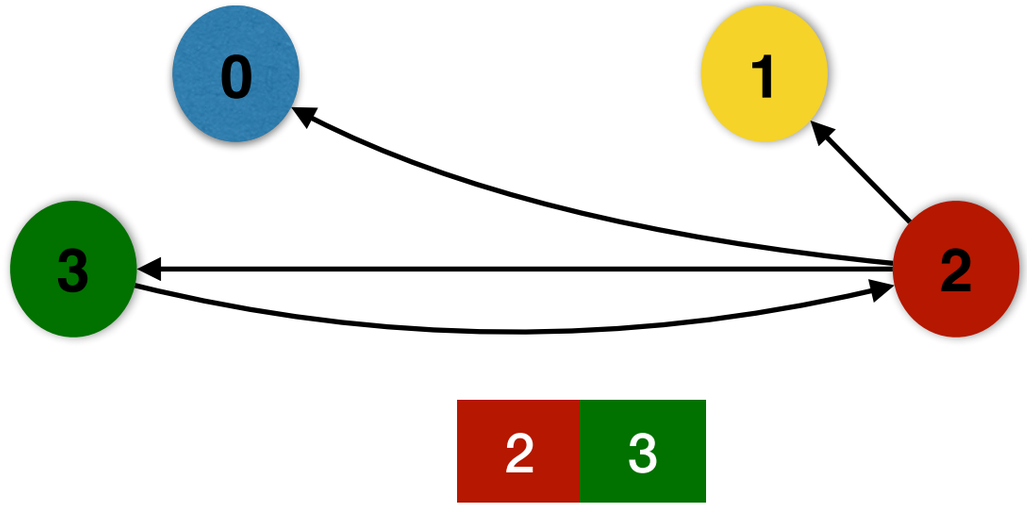



# Graph Processing

**Cache**

**#hits: 0**
**#misses: 0**

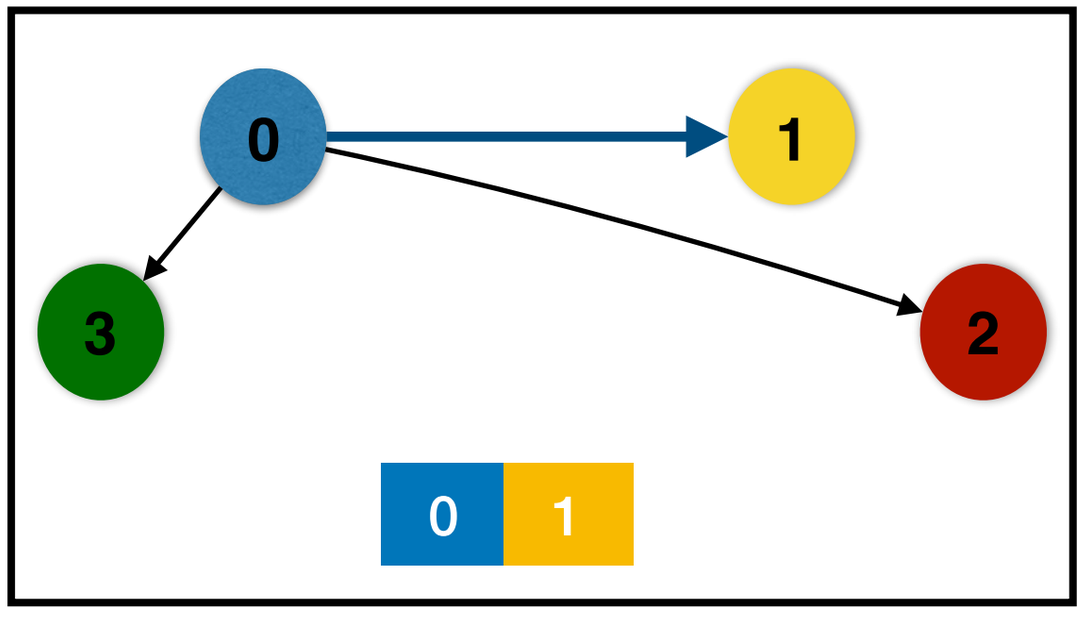


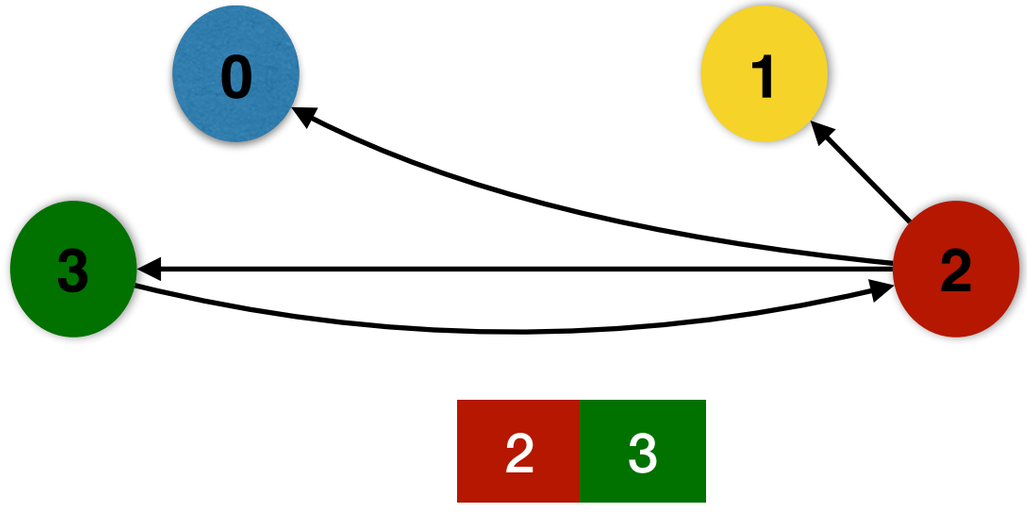



# Graph Processing

**Cache**

**#hits: 0**
**#misses: 1**

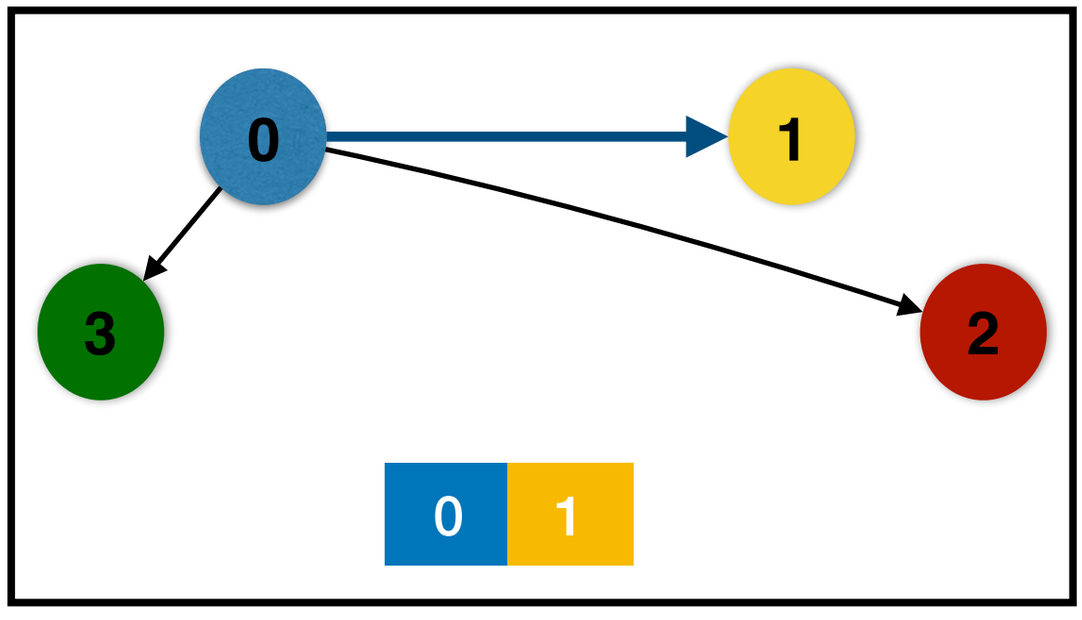


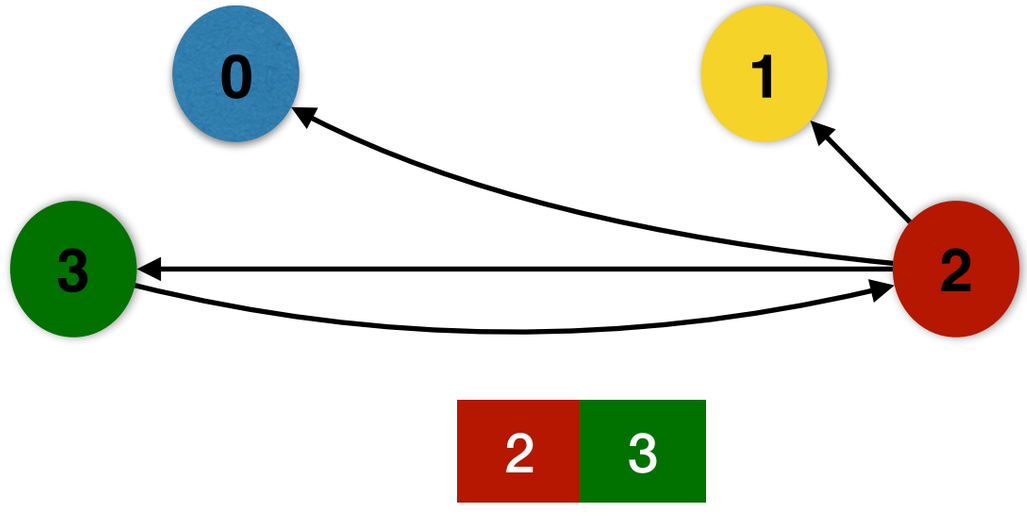



# Graph Processing

**Cache**


**#hits: 0**
**#misses: 1**

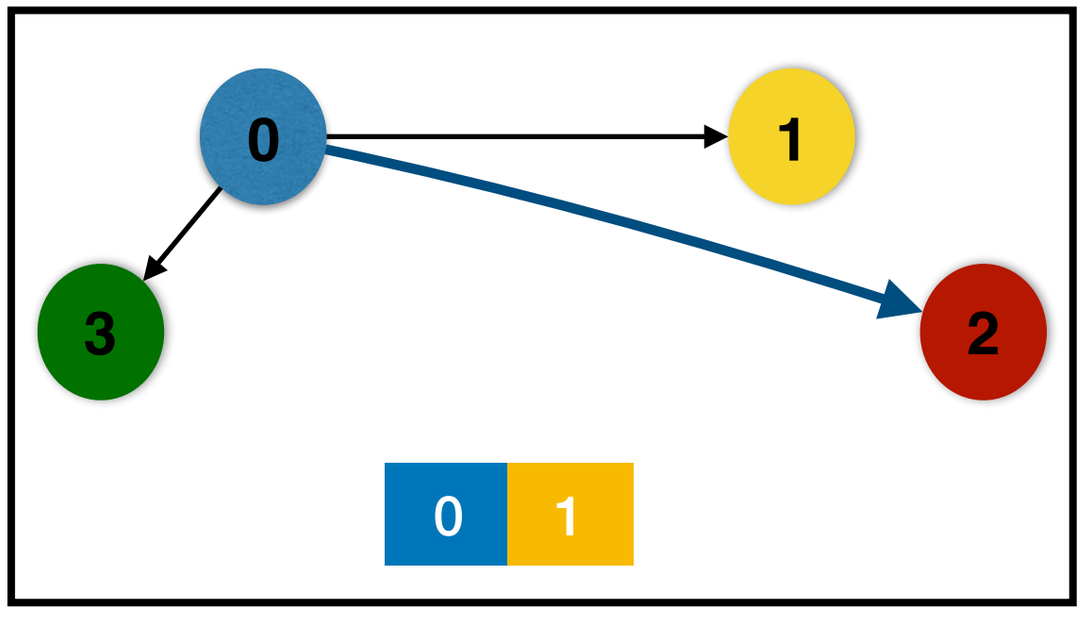


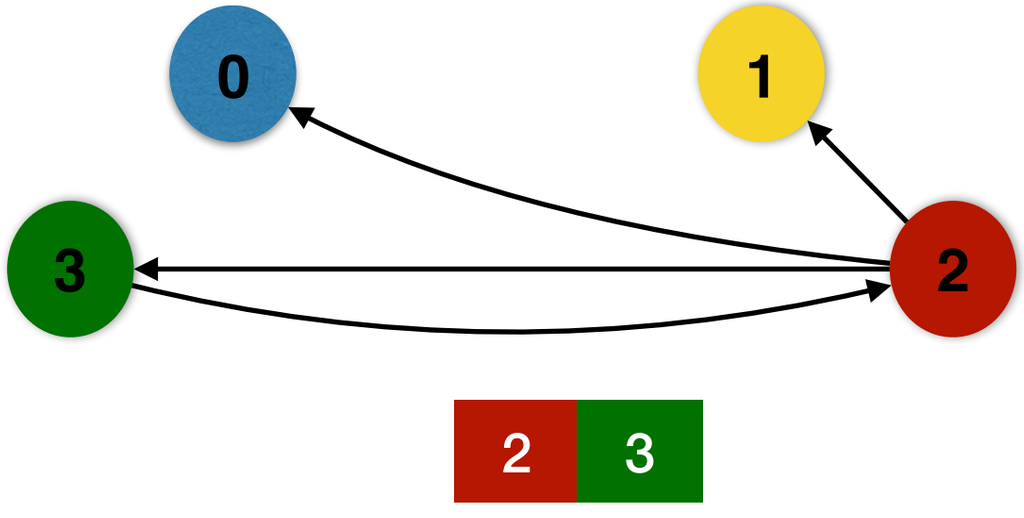



# Graph Processing

**Cache**


**#hits: 1**
**#misses: 1**

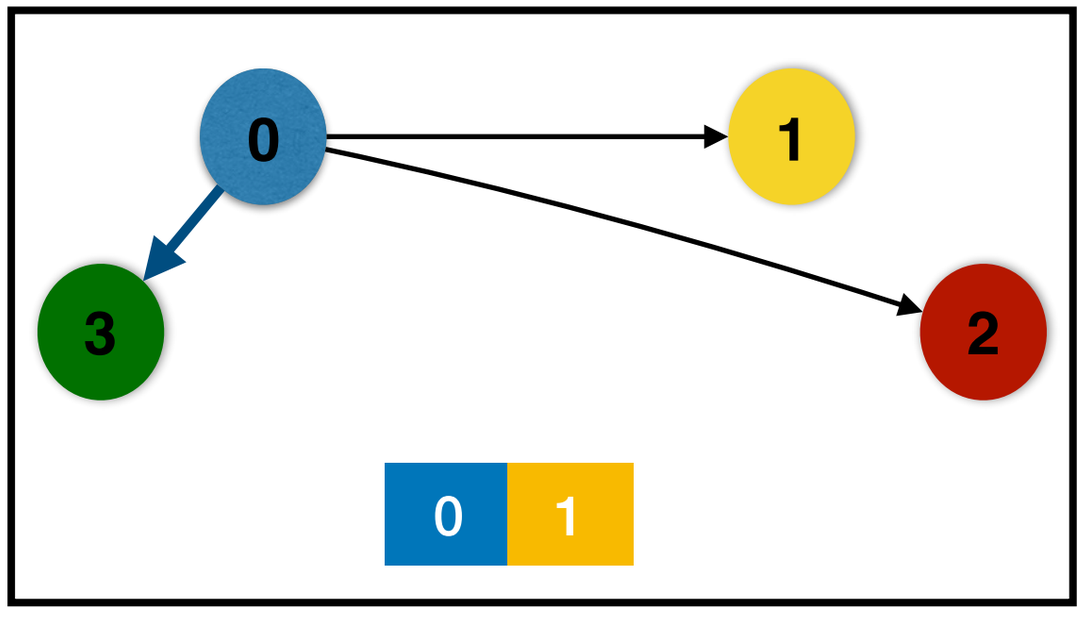


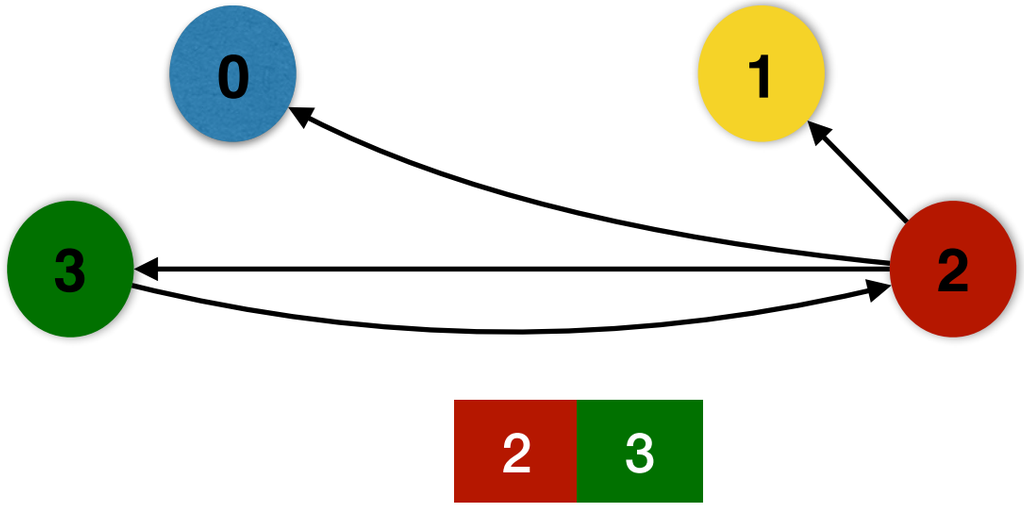



# Graph Processing

**Cache**


**#hits: 2**
**#misses: 1**

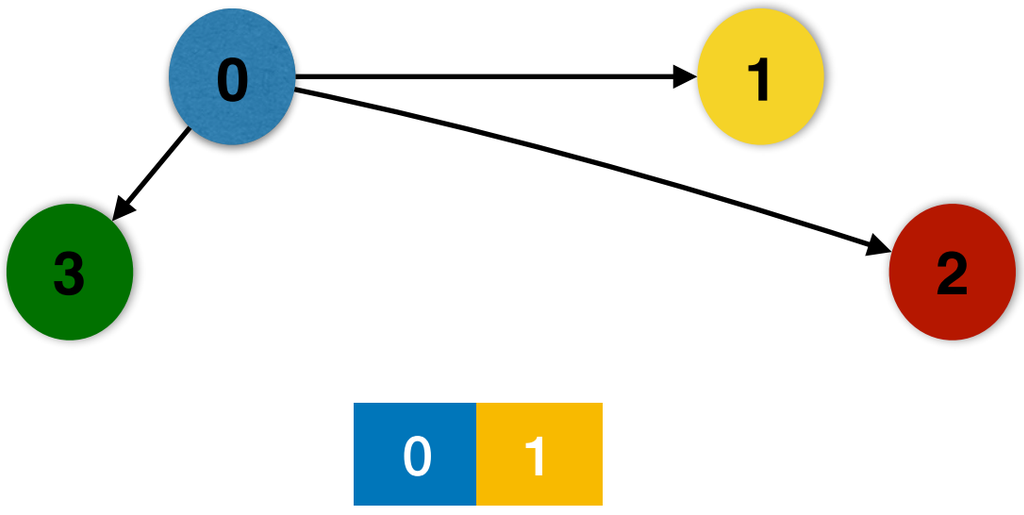


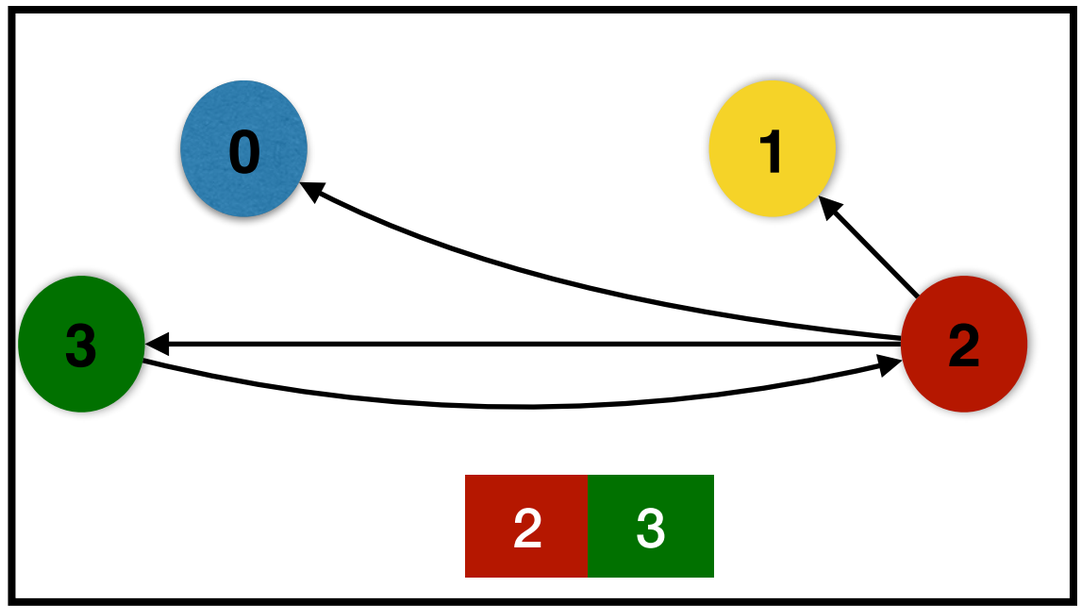



# Graph Processing

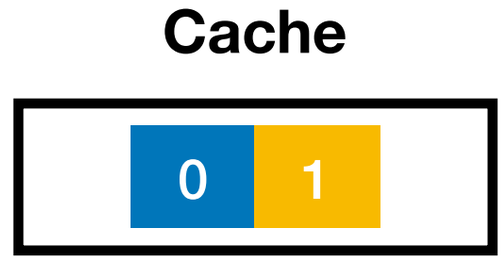
**Cache**


**#hits: 2**
**#misses: 1**



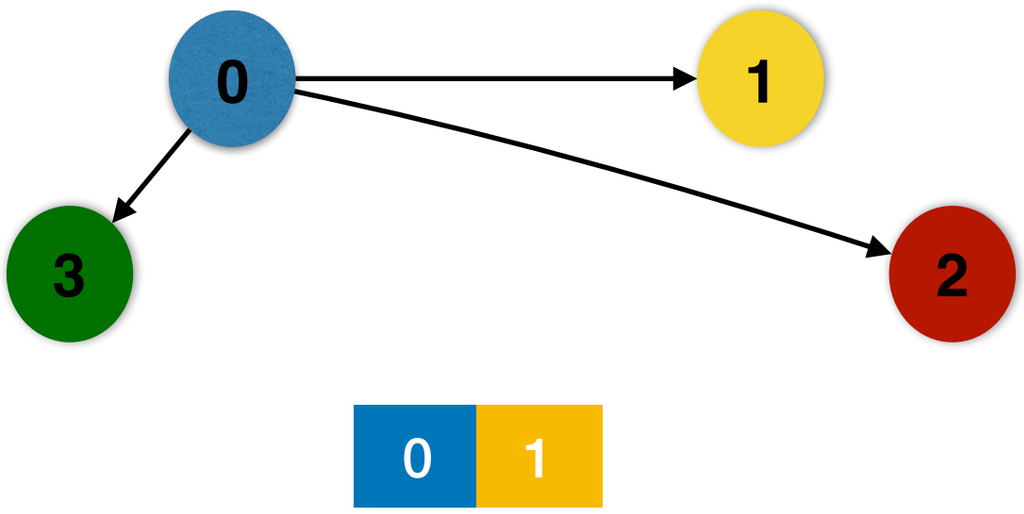



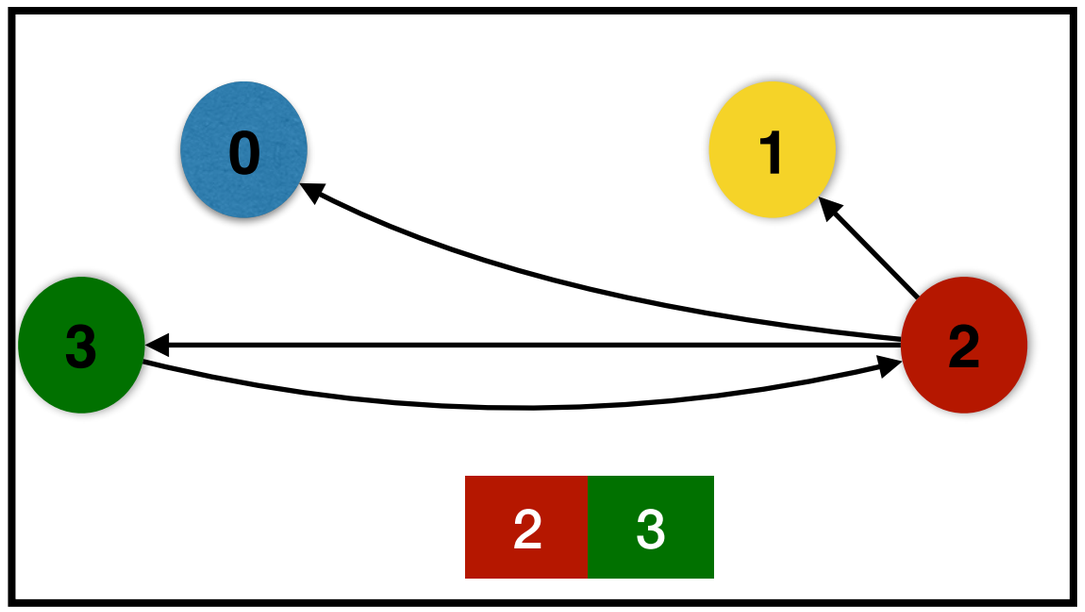
# Graph Processing



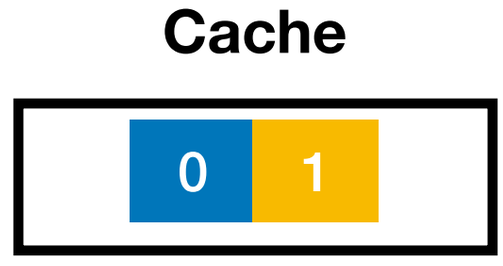

**#hits: 2**

**#misses: 1**



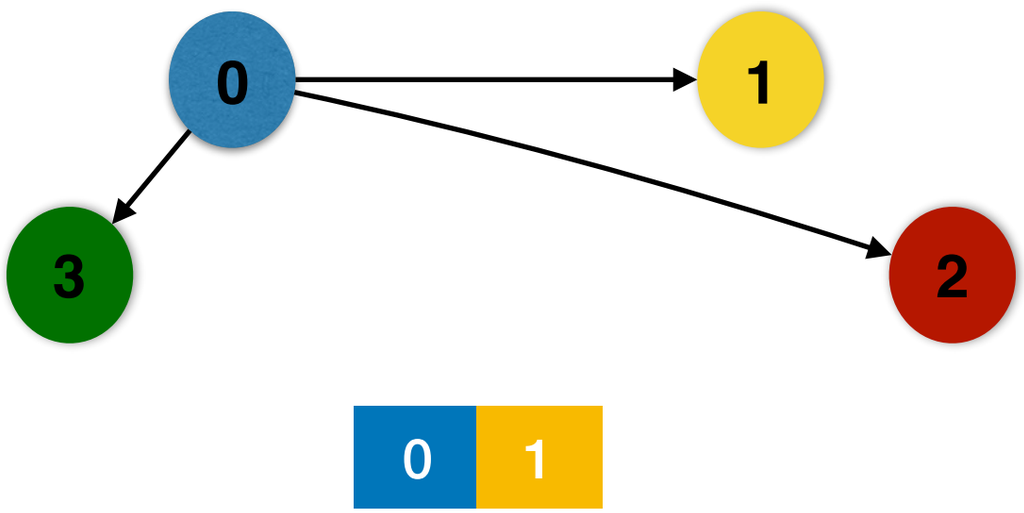



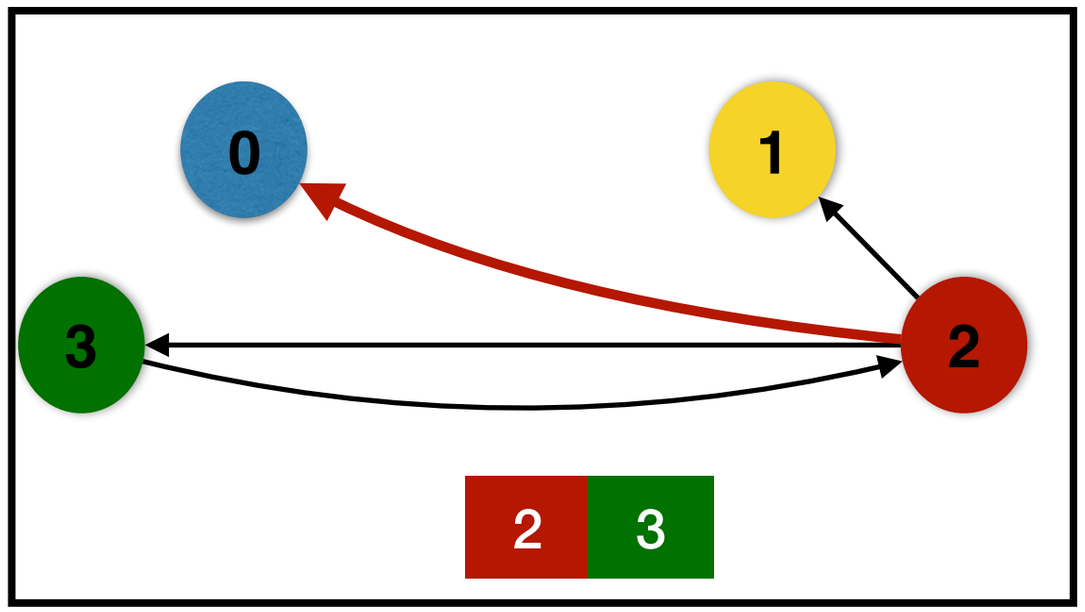
# Graph Processing



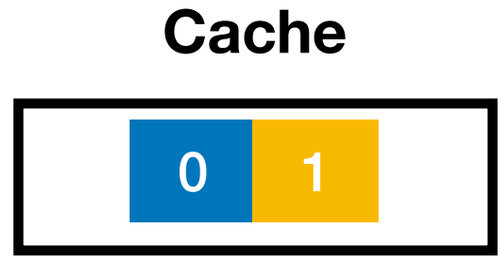

**#hits: 2**

**#misses: 1**



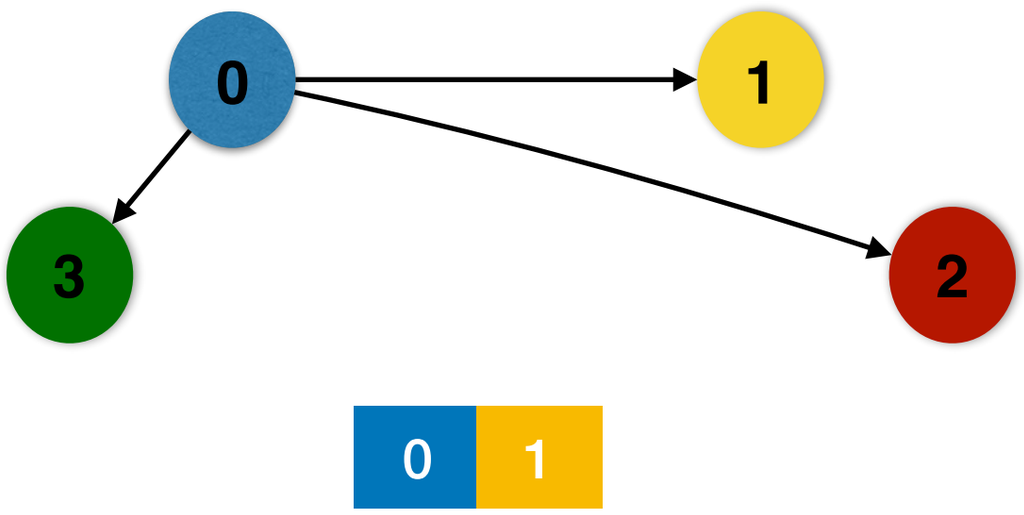



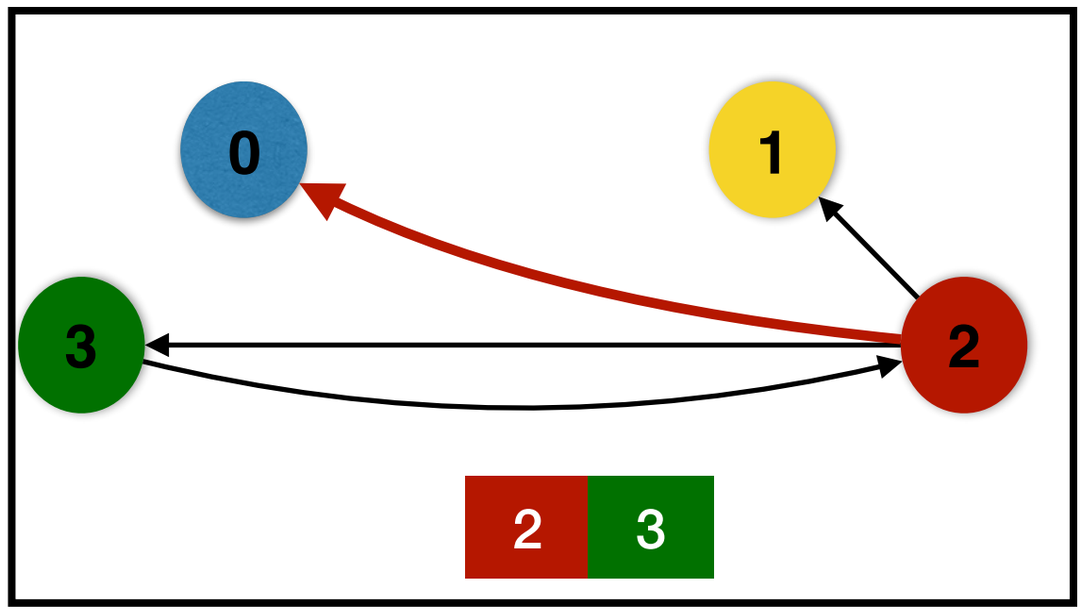
# Graph Processing



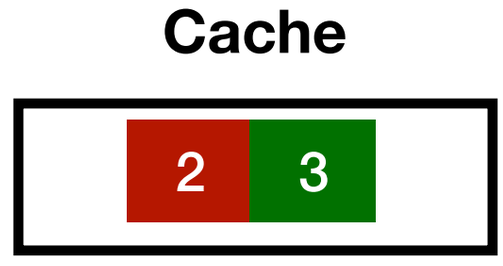

**#hits: 2**

**#misses: 1**



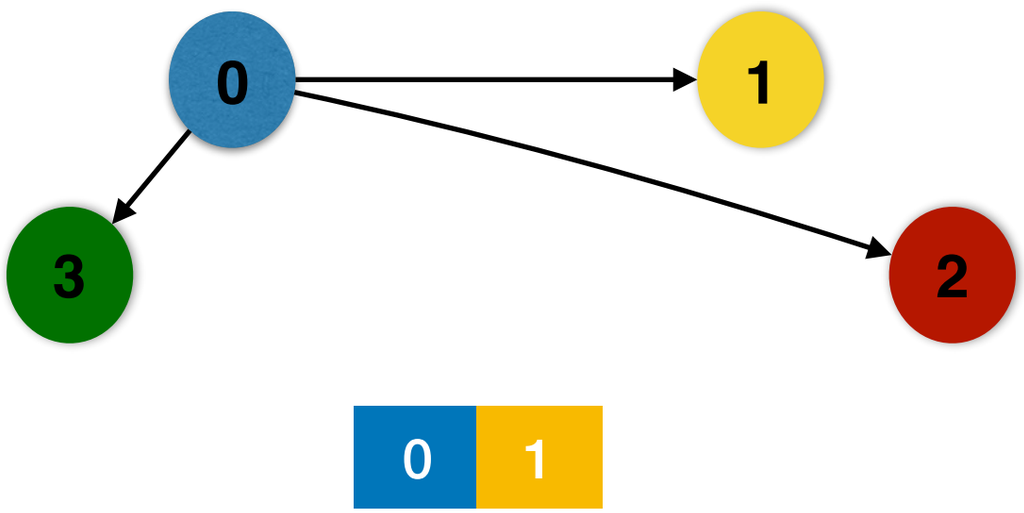



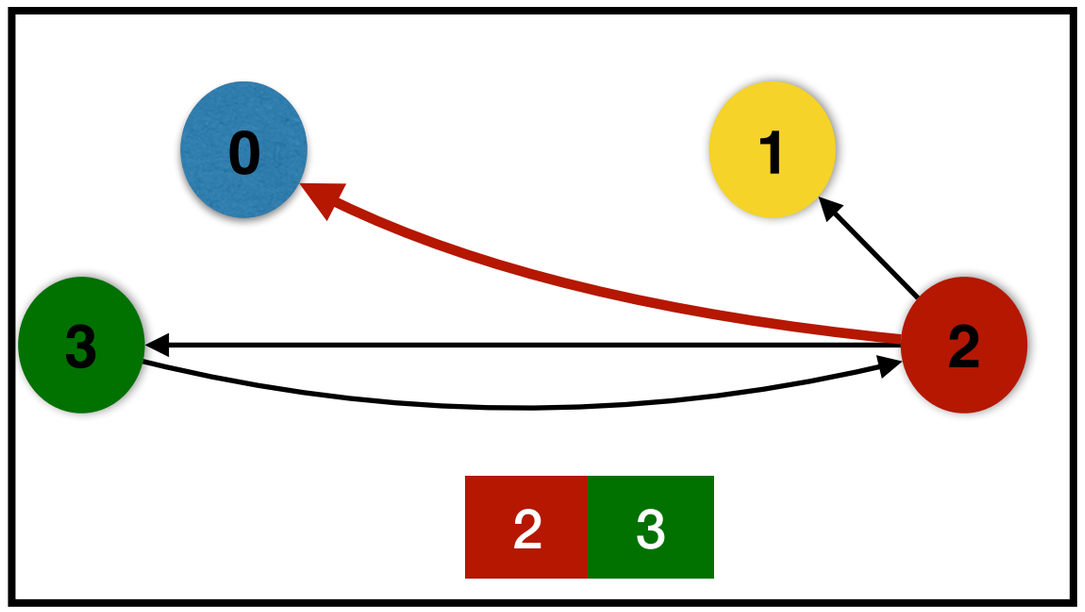
# Graph Processing



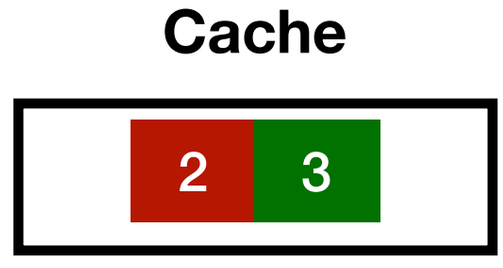

**#hits: 2**

**#misses: 2**

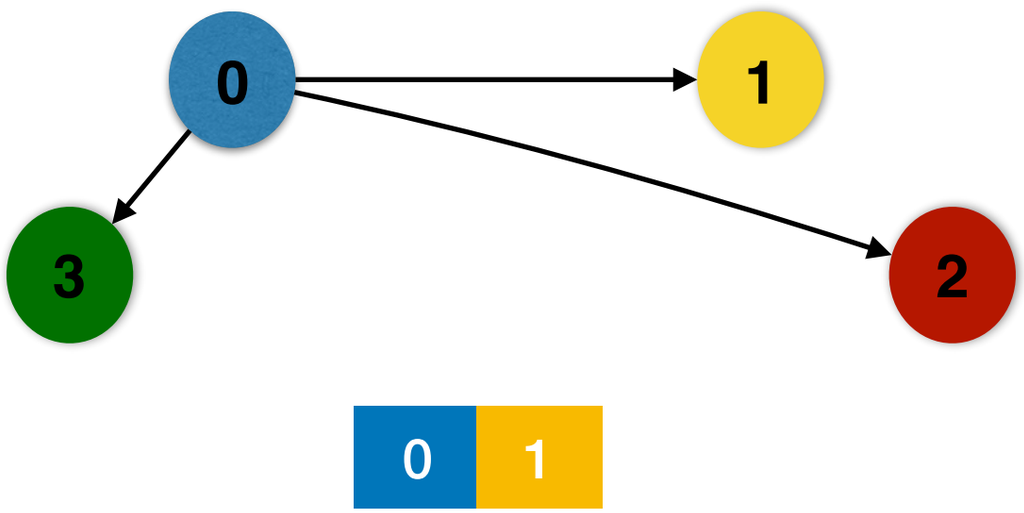


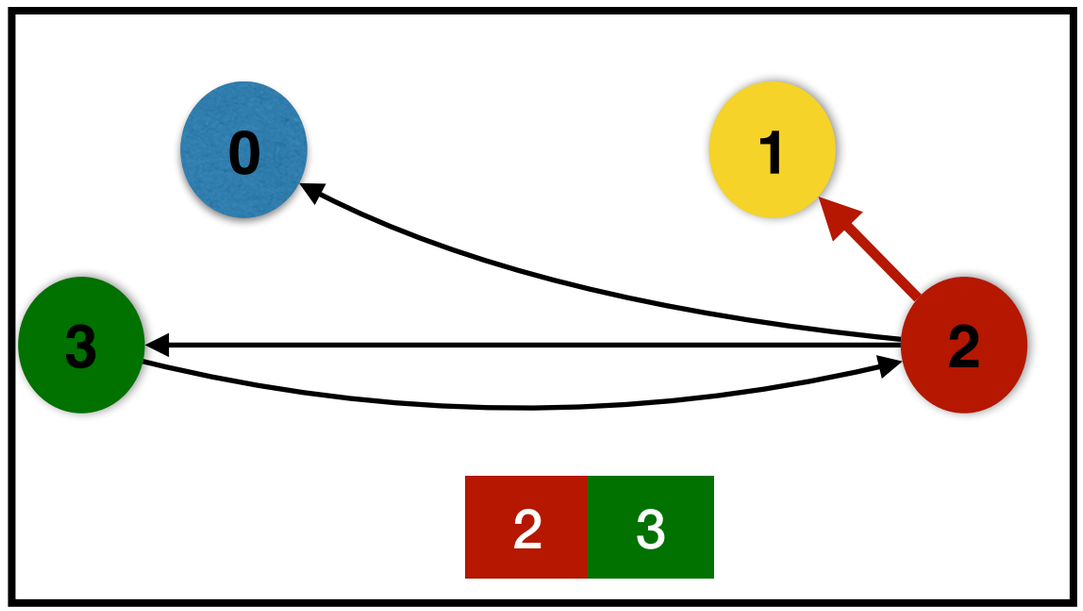



# Graph Processing

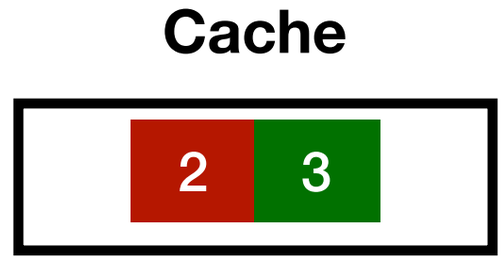



**#hits: 3**
**#misses: 2**

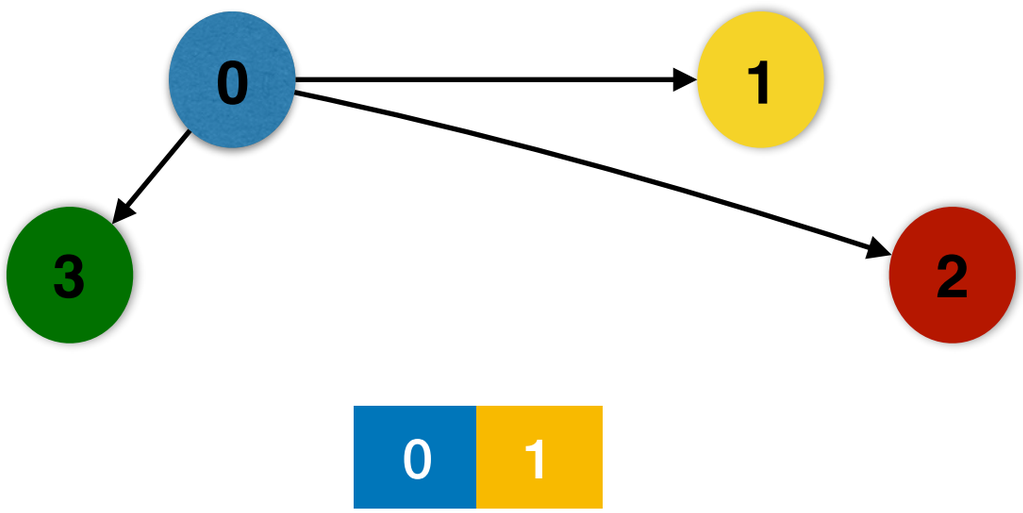


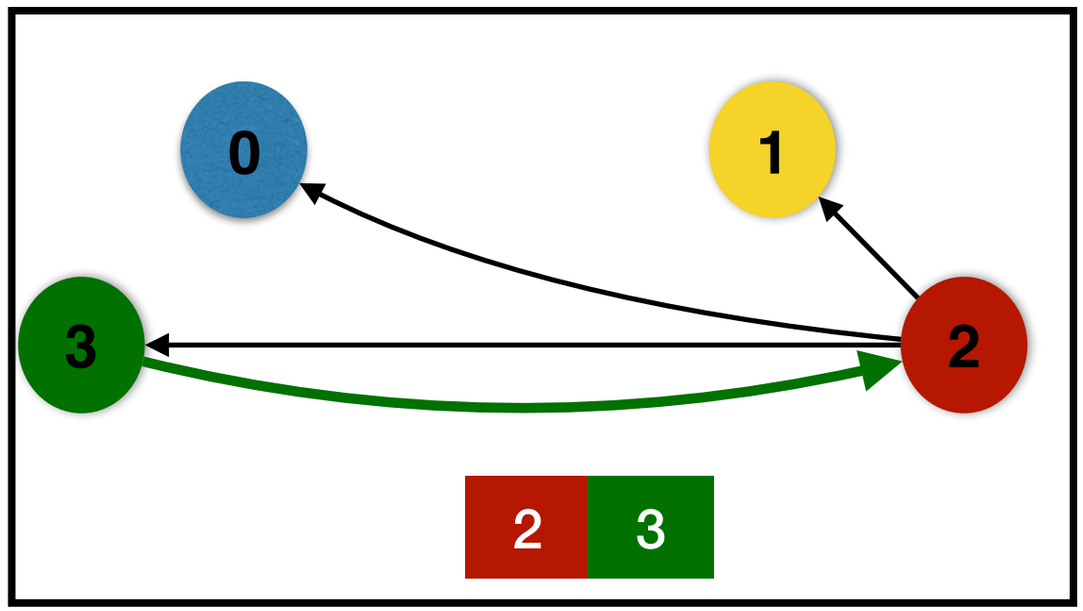



# Graph Processing

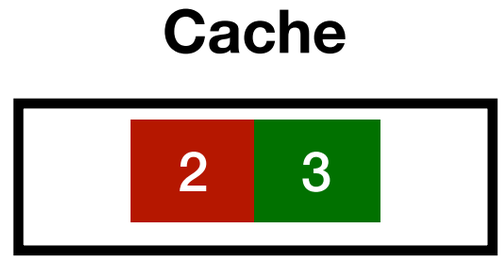



**#hits: 4**
**#misses: 2**

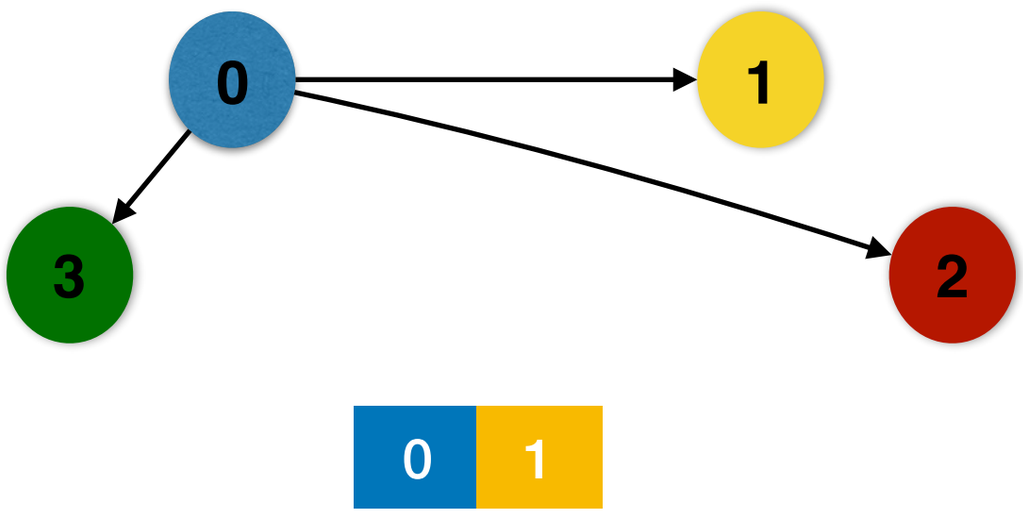


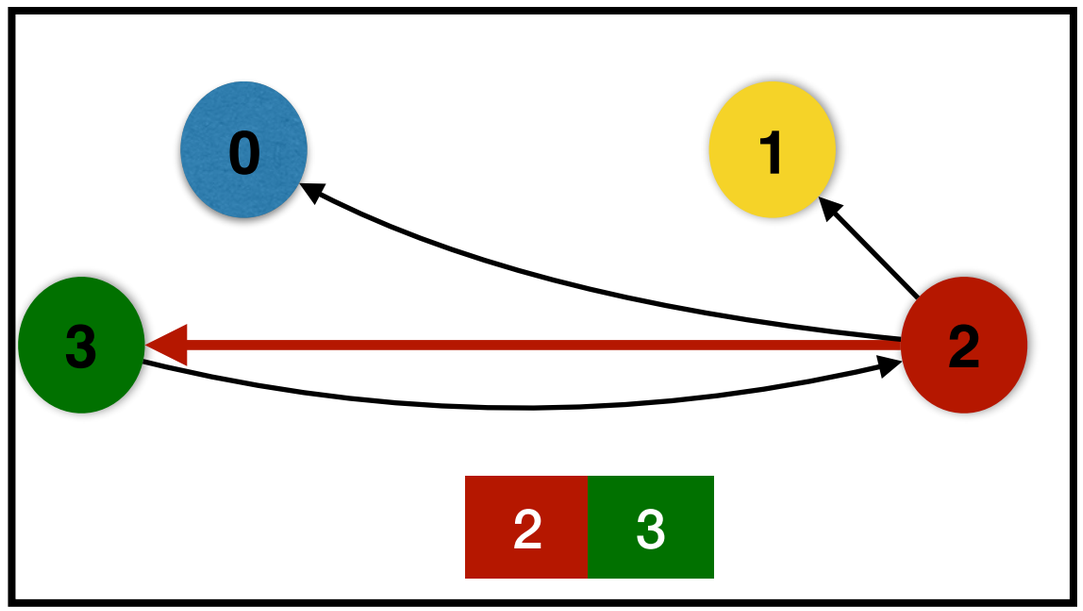



# Graph Processing



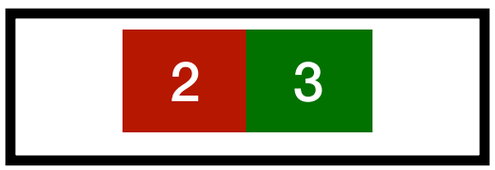

**#hits: 5**
**#misses: 2**



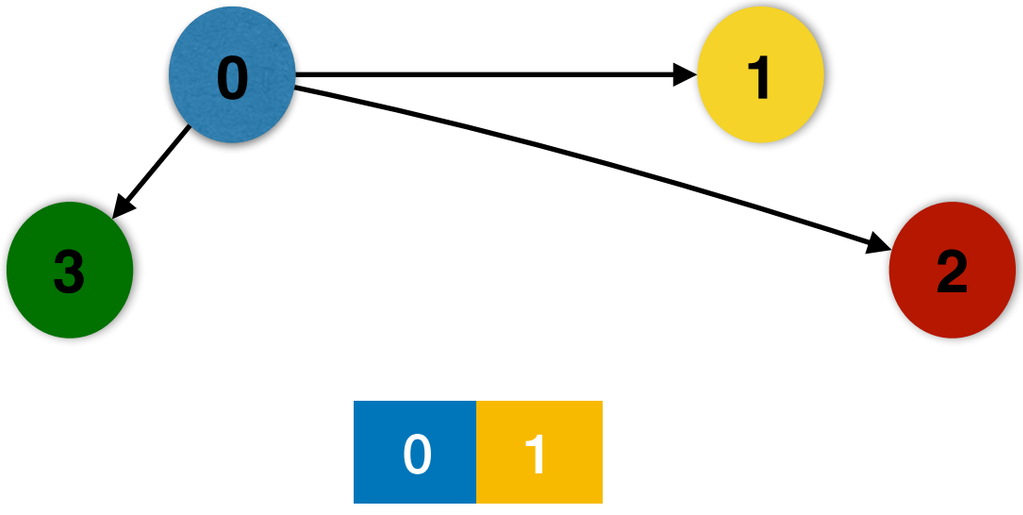
# Graph Processing

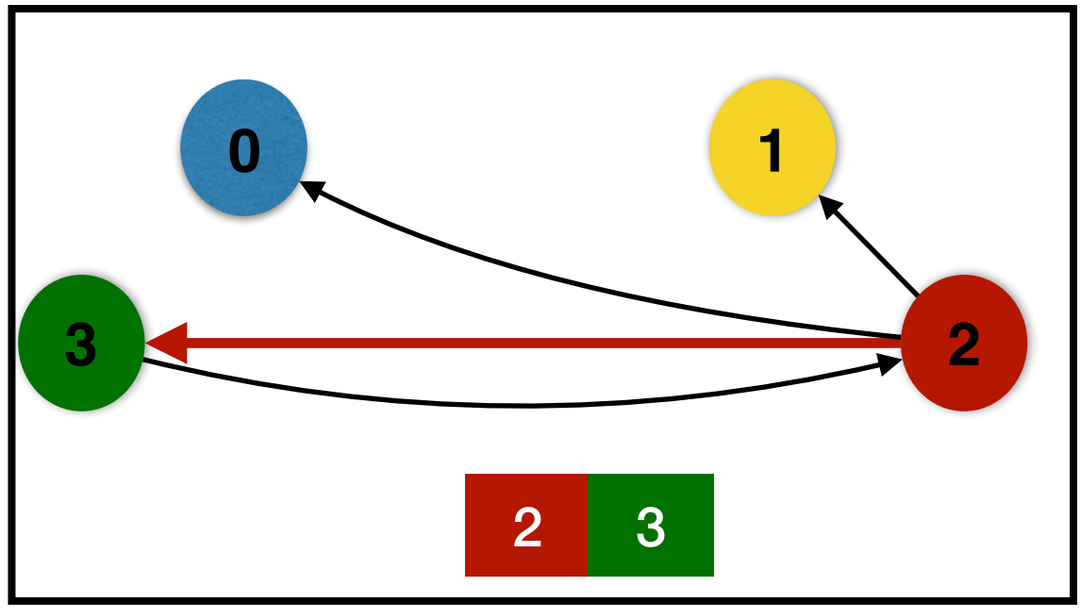
**Cache**




**Only have 2 misses**

**#hits: 5**
**#misses: 2**

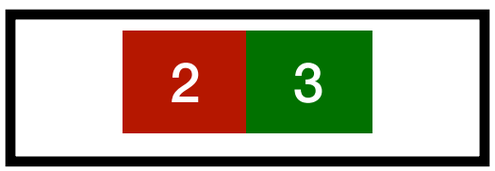


# Graph Processing

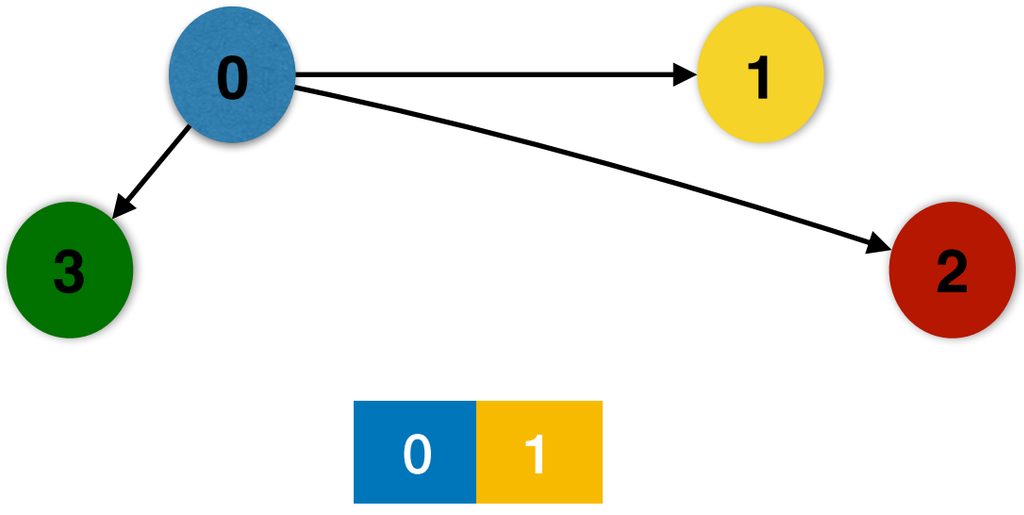
**Cache**

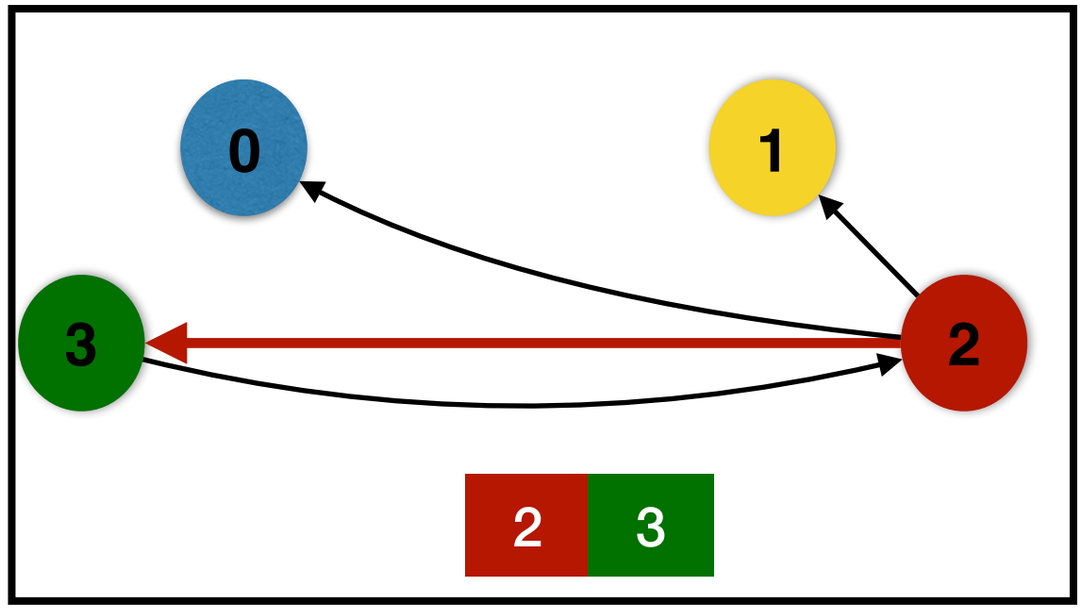
**Better than Frequency based Reordering**



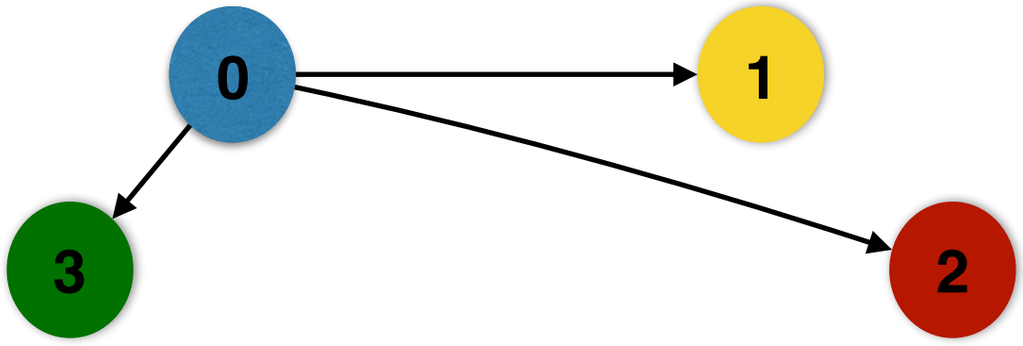

**#hits: 4**
**#misses: 3**

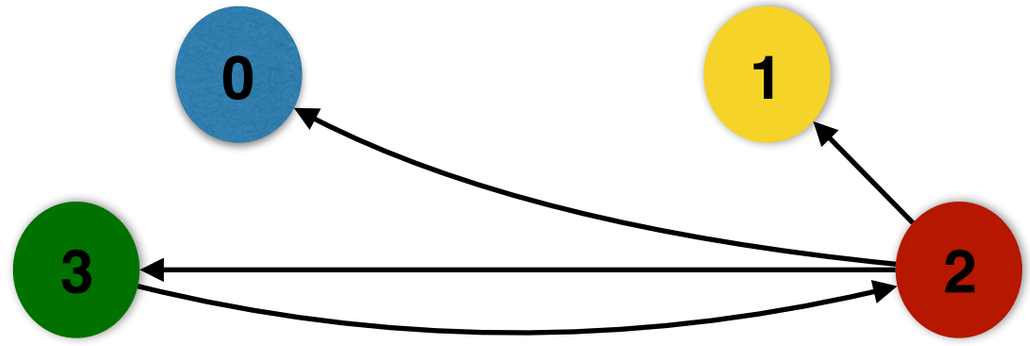
**#hits: 5**
**#misses: 2**



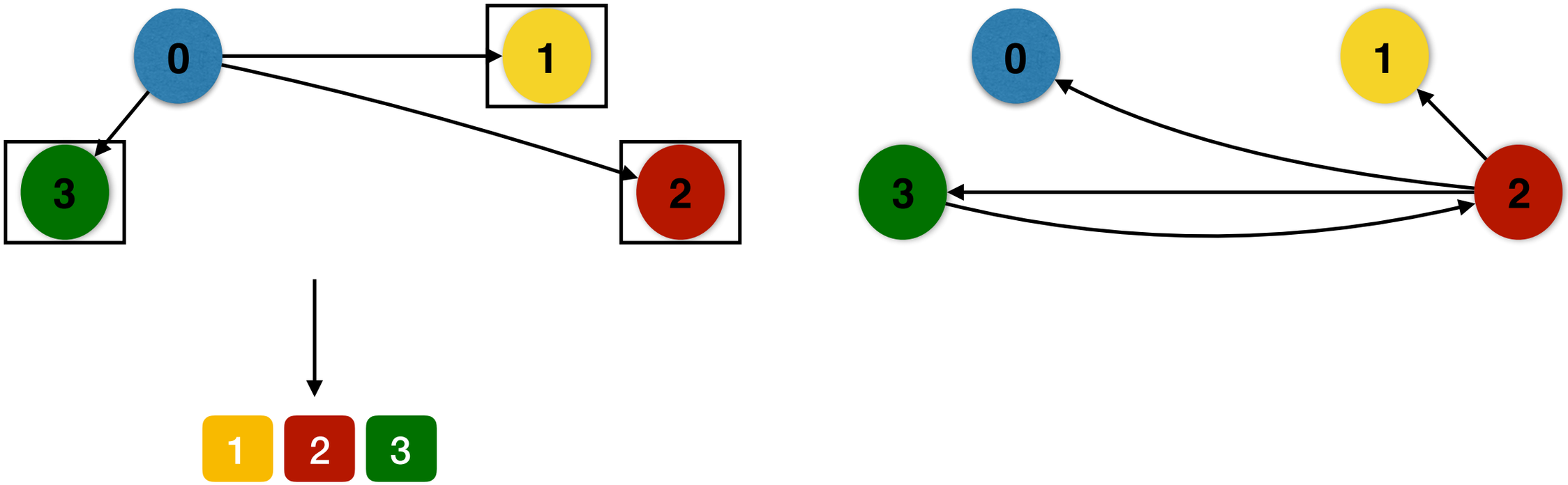



# Cache-aware Merge

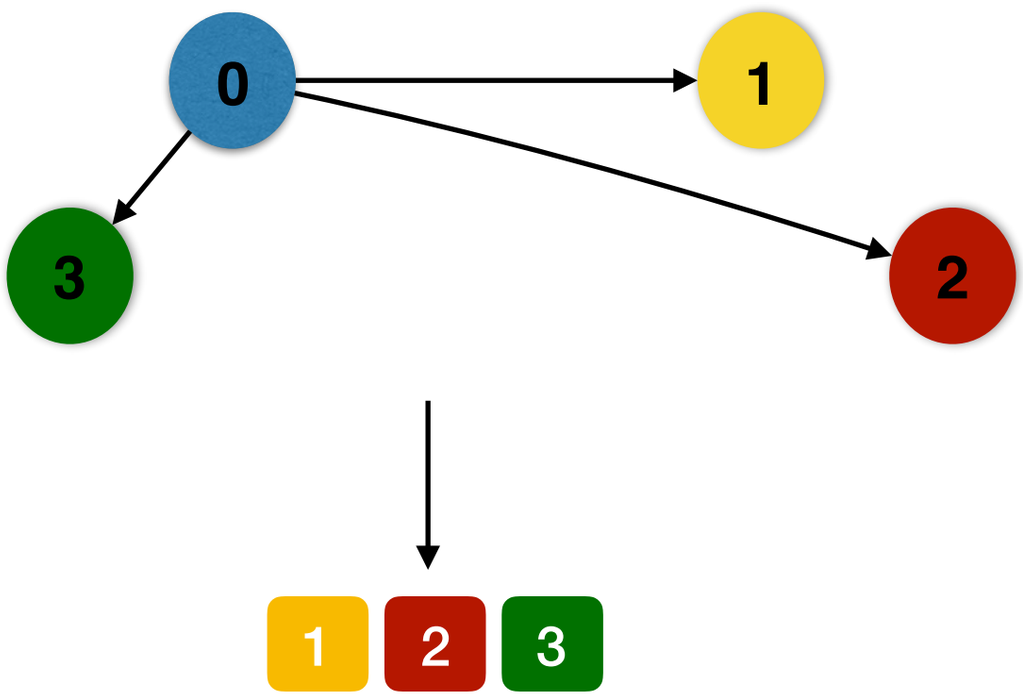


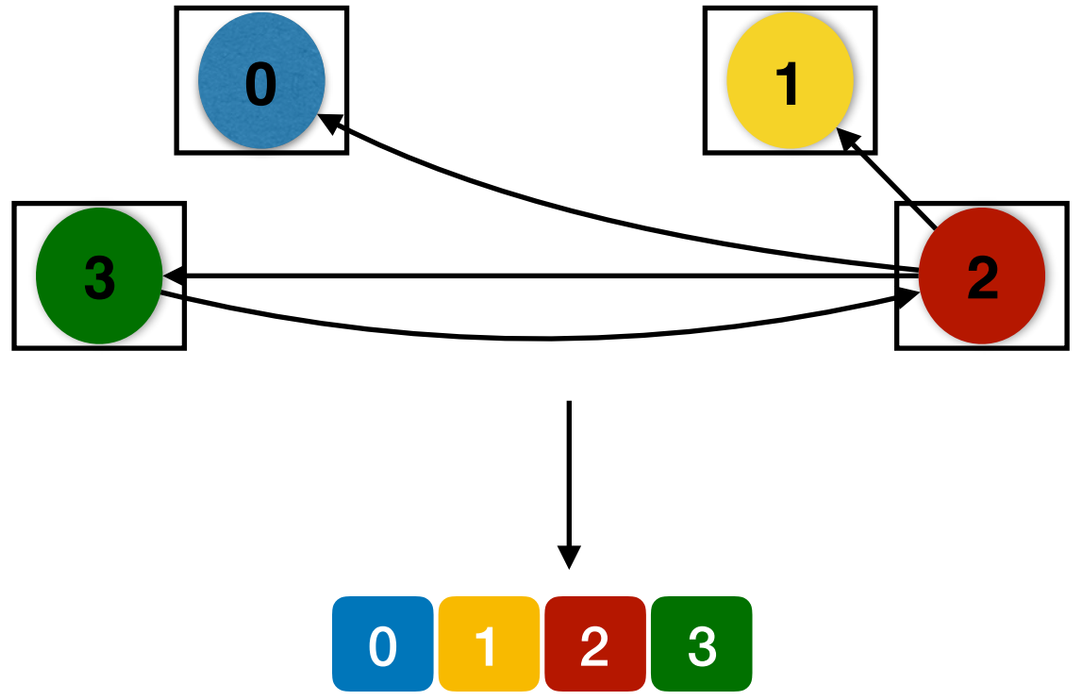



# Cache-aware Merge

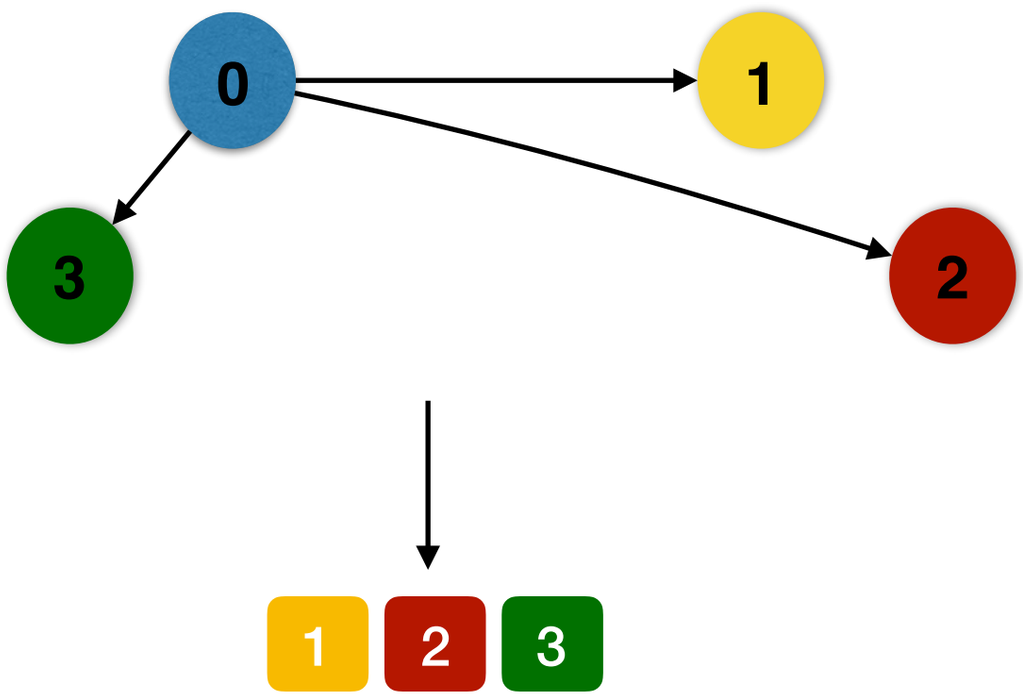


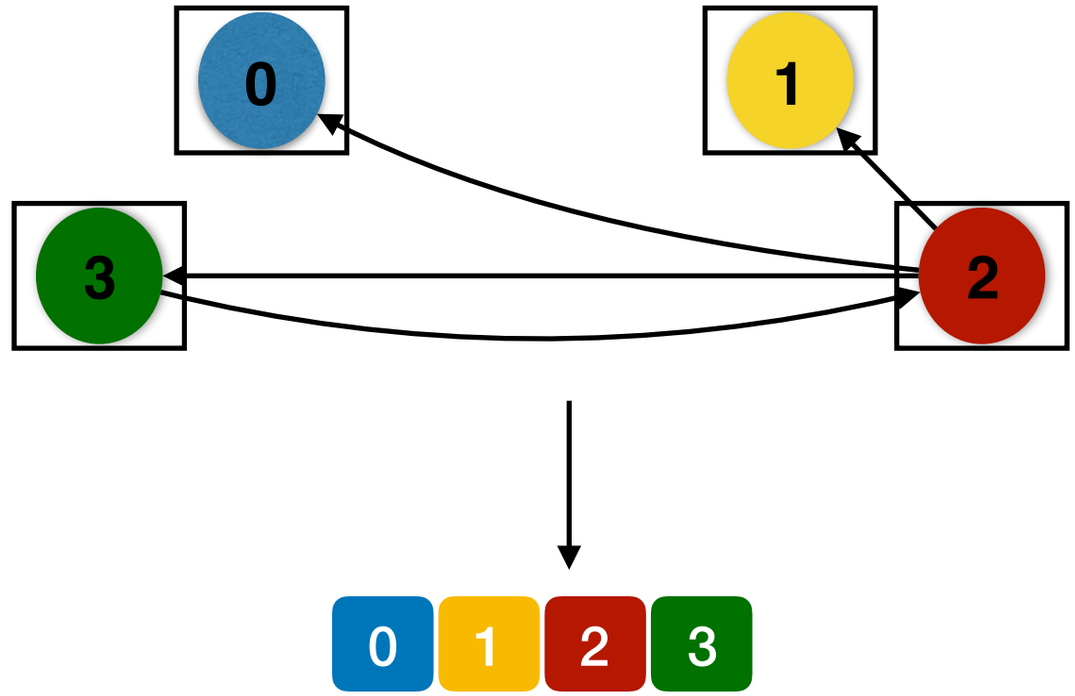
# Cache-aware Merge



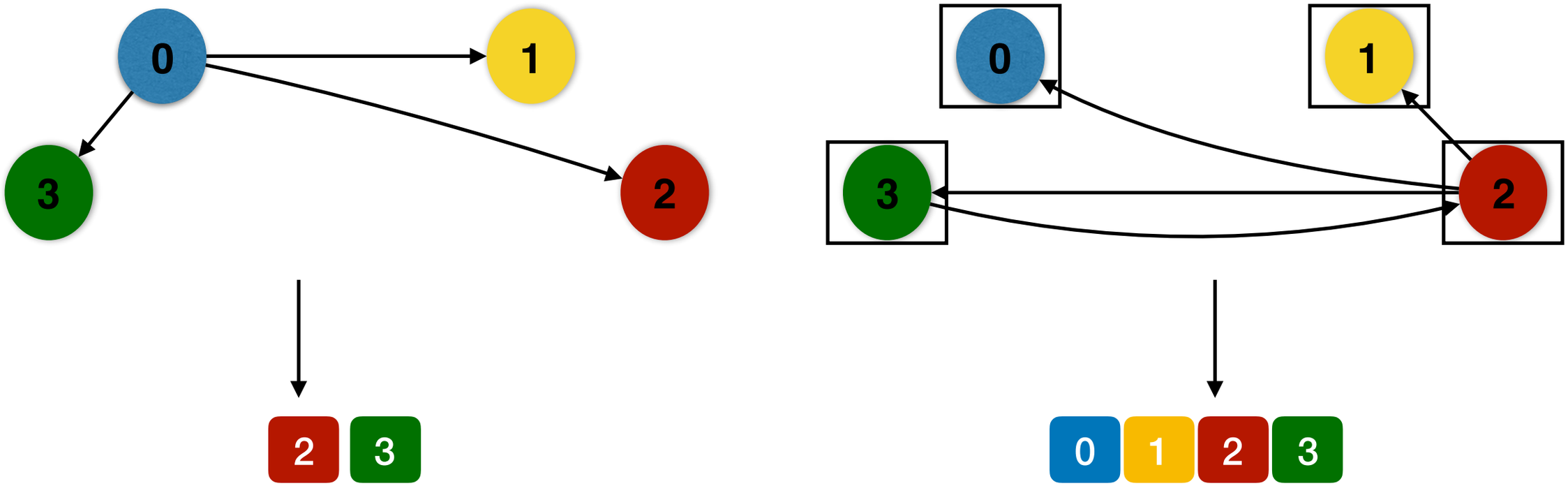



# Cache-aware Merge

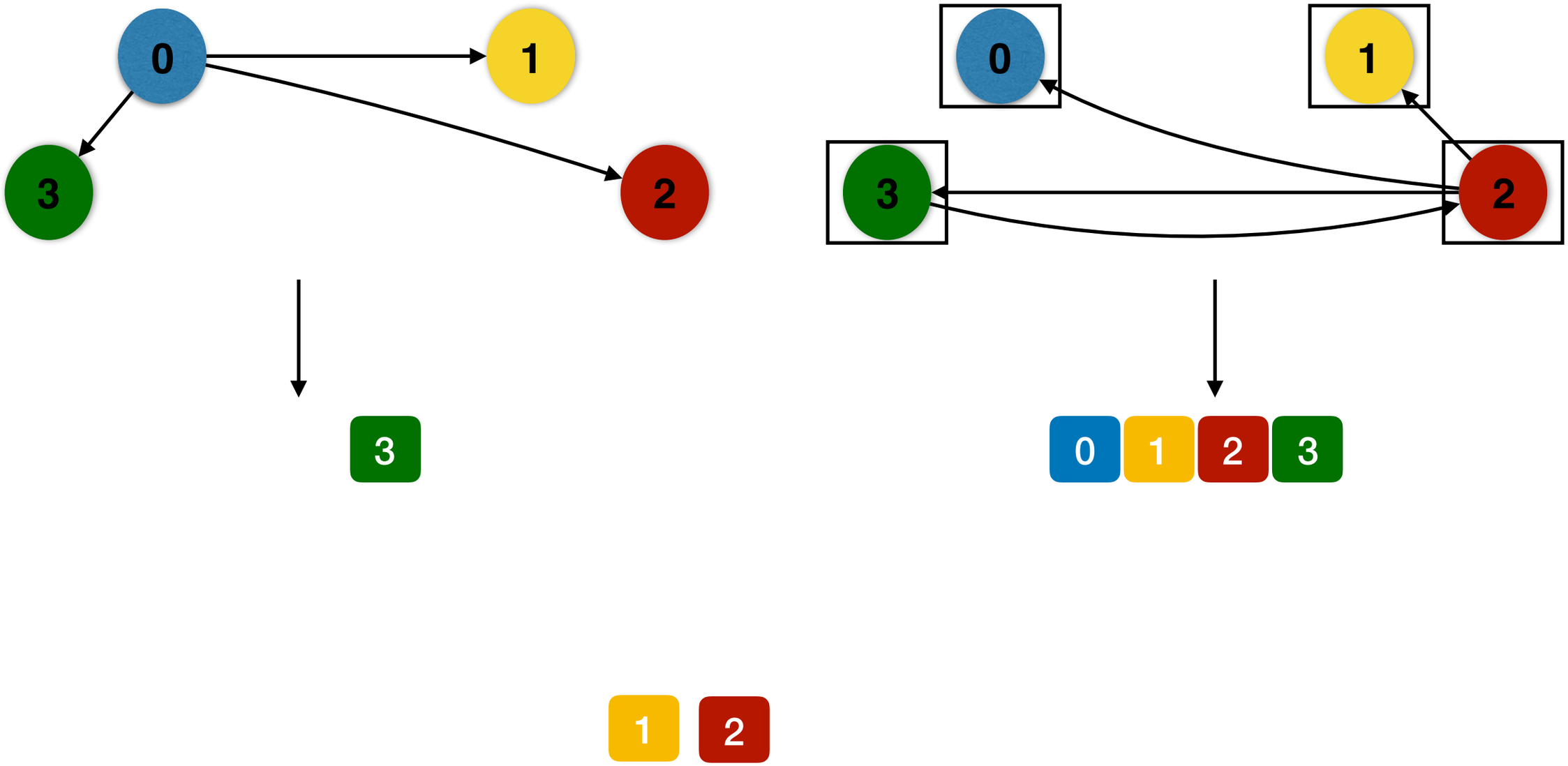


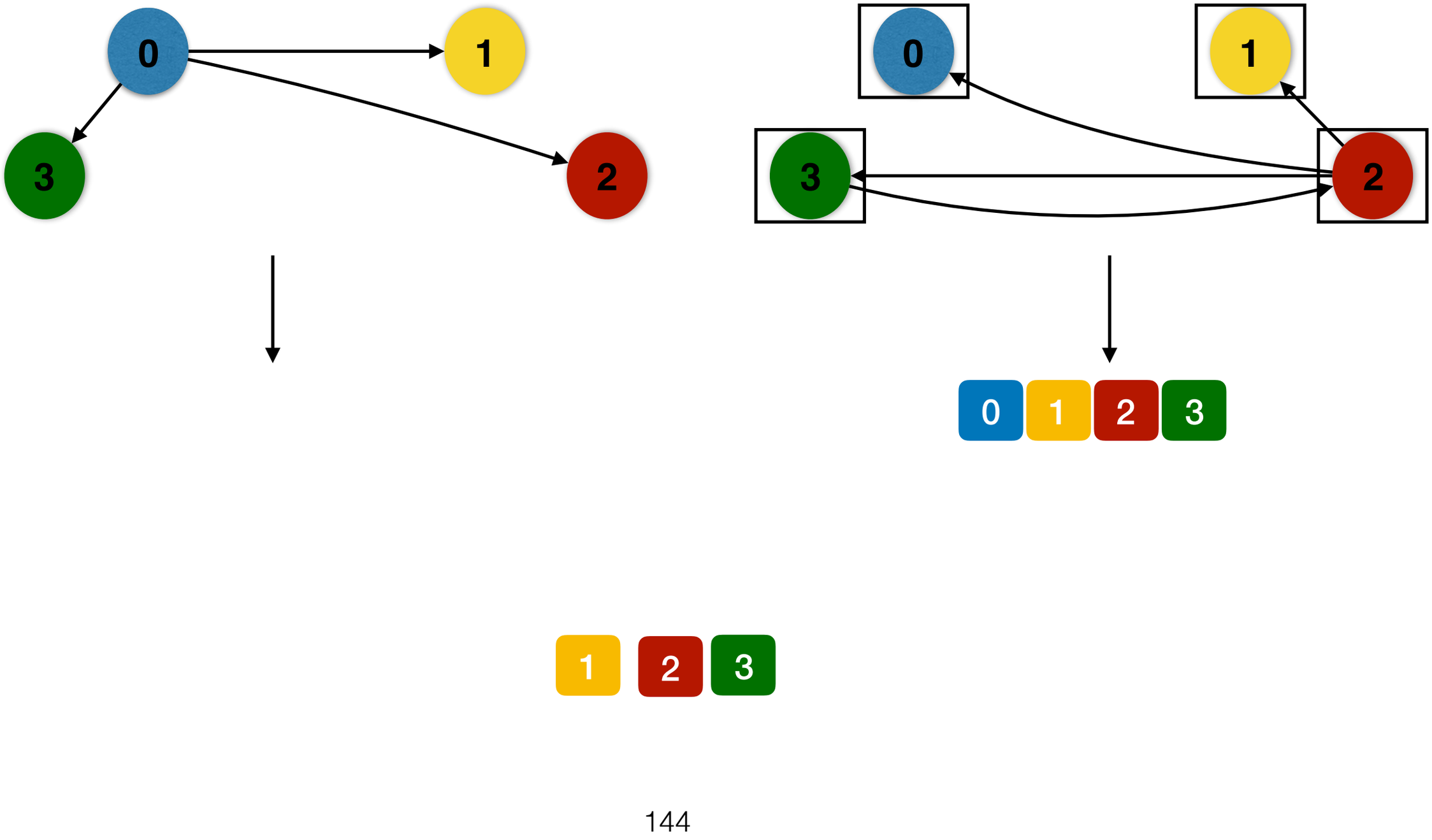
# Cache-aware Merge



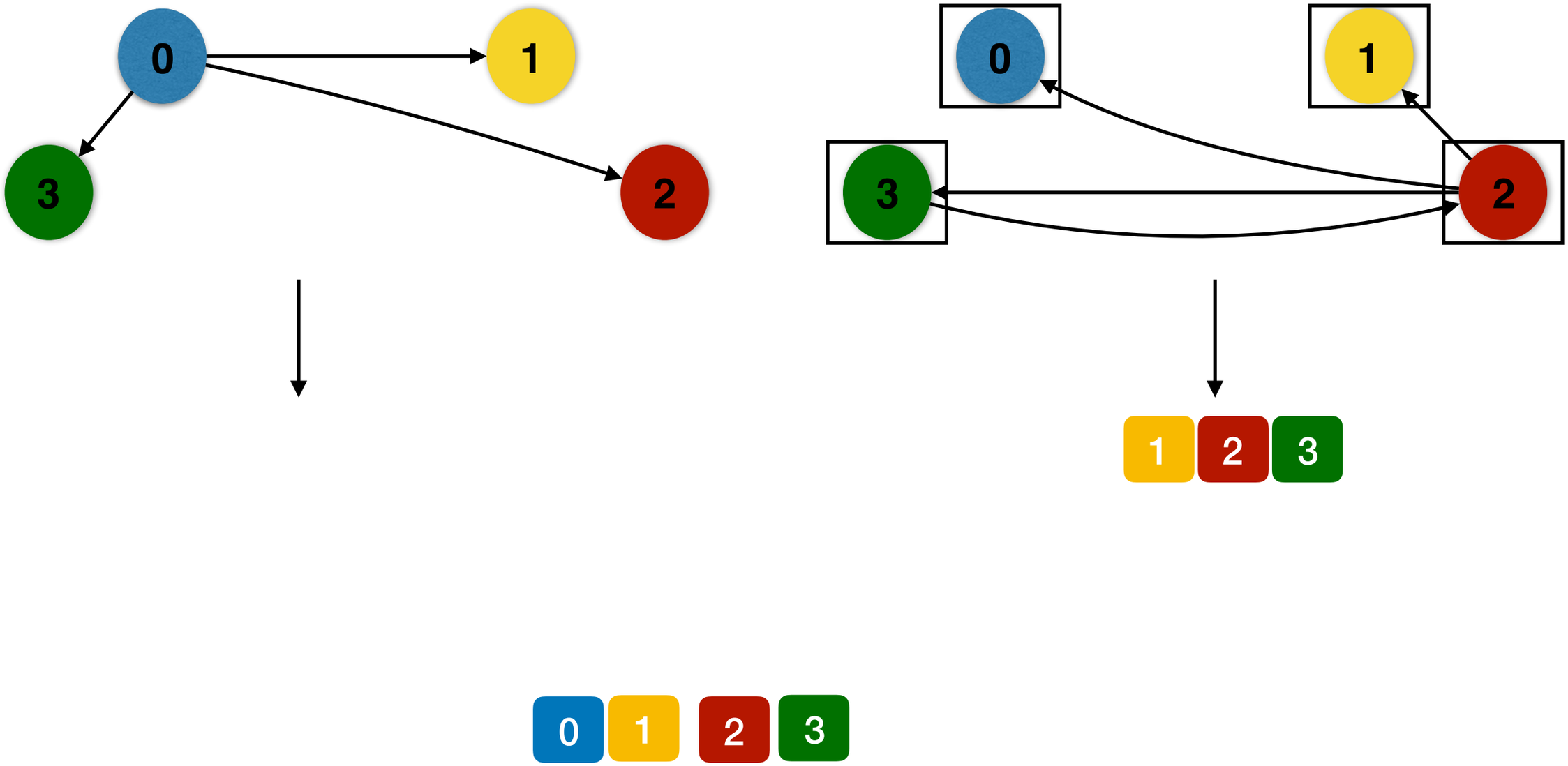
# Cache-aware Merge



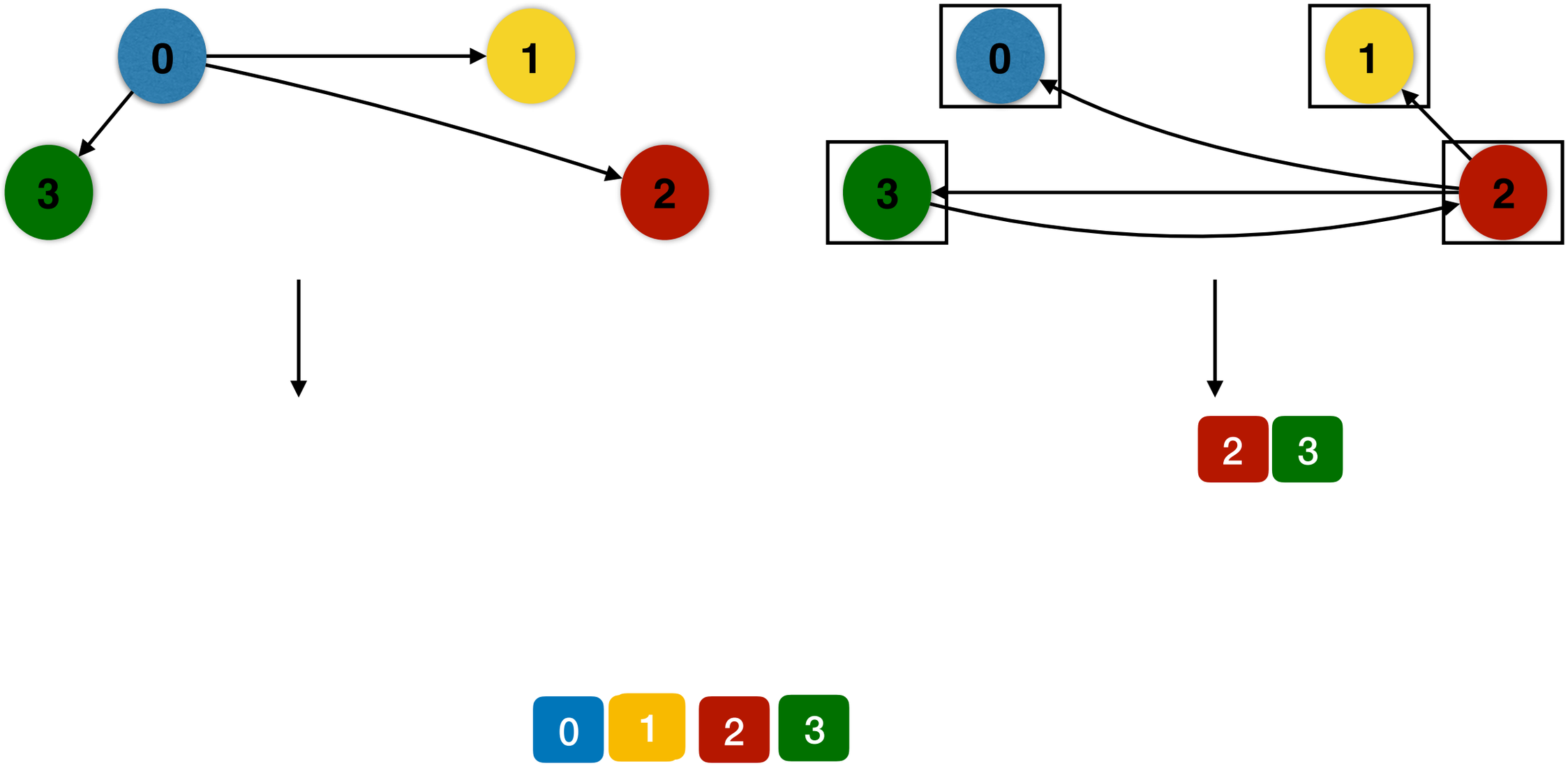
# Cache-aware Merge



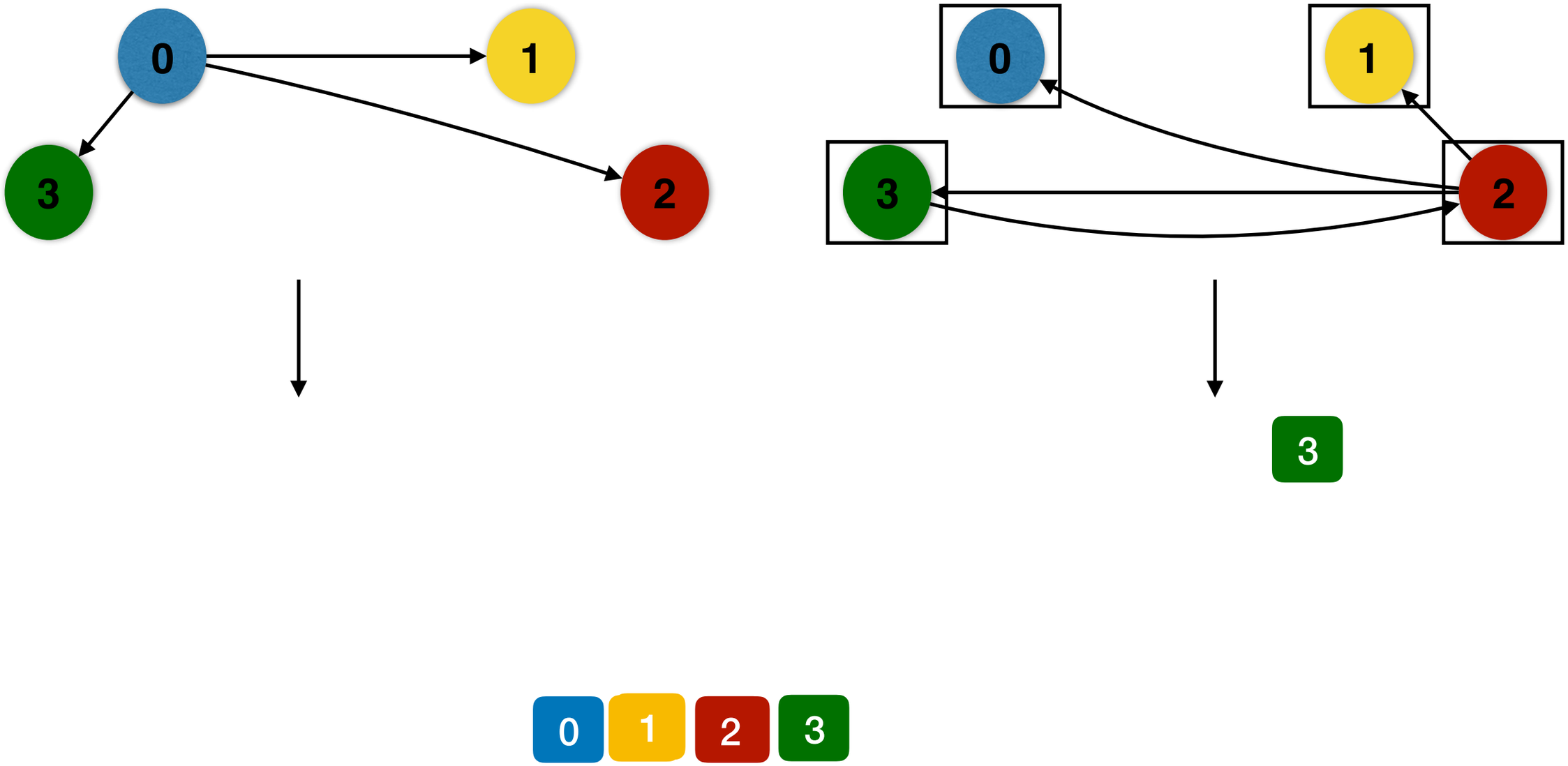
# Cache-aware Merge



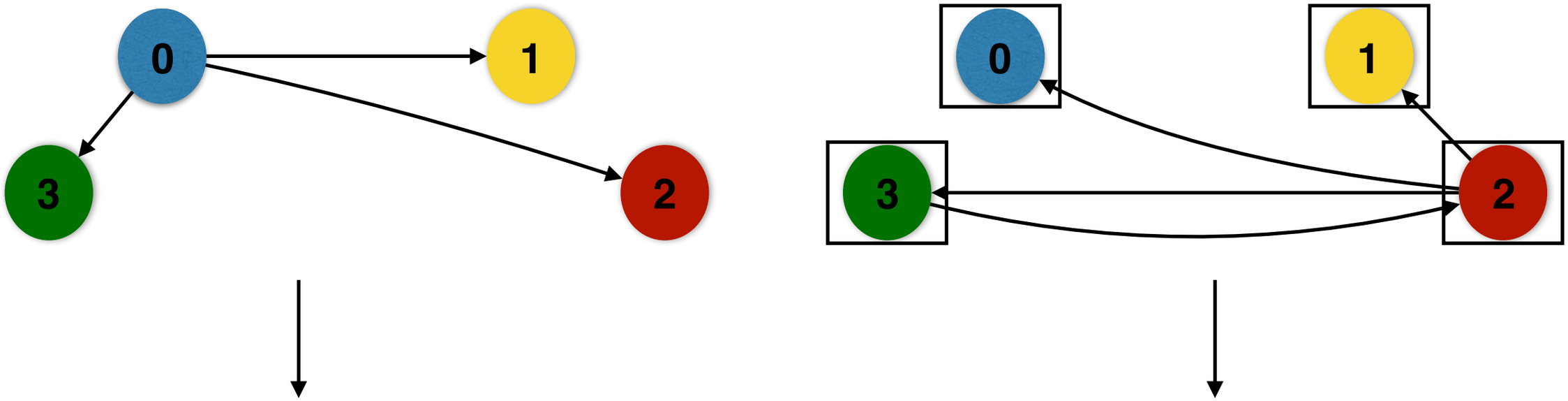
# Cache-aware Merge



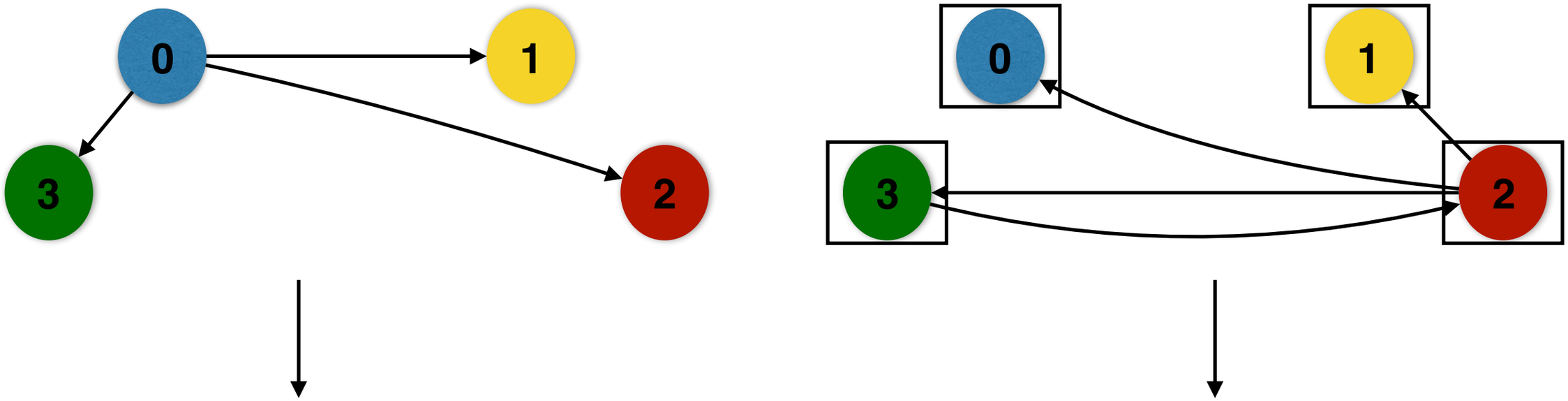
# Cache-aware Merge



# Cache-aware Merge



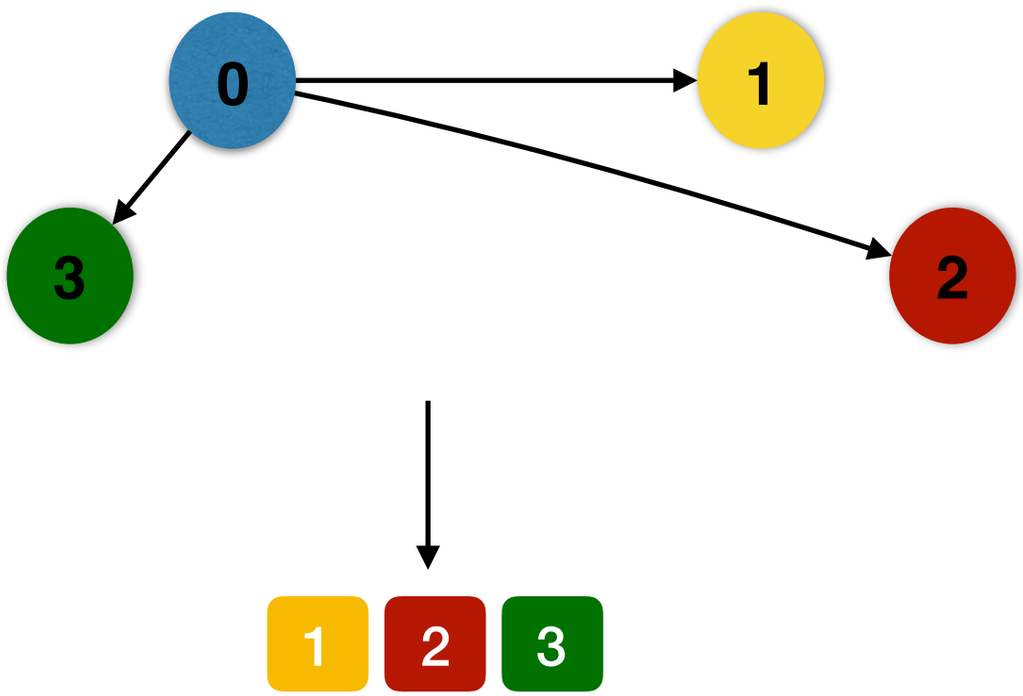

# Cache-aware Merge

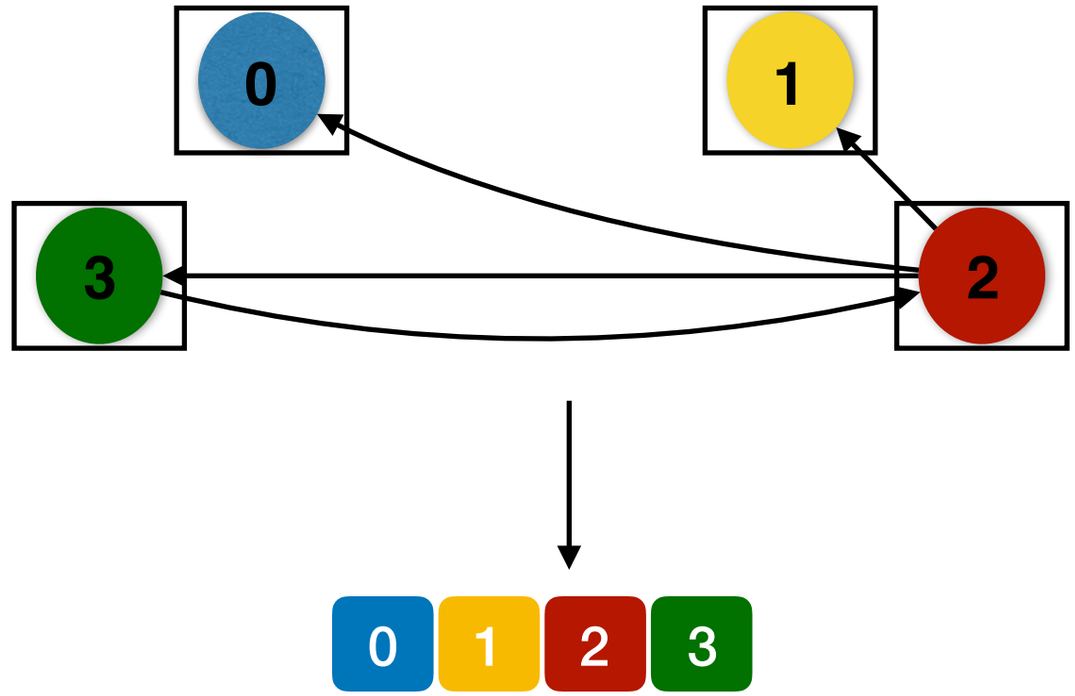




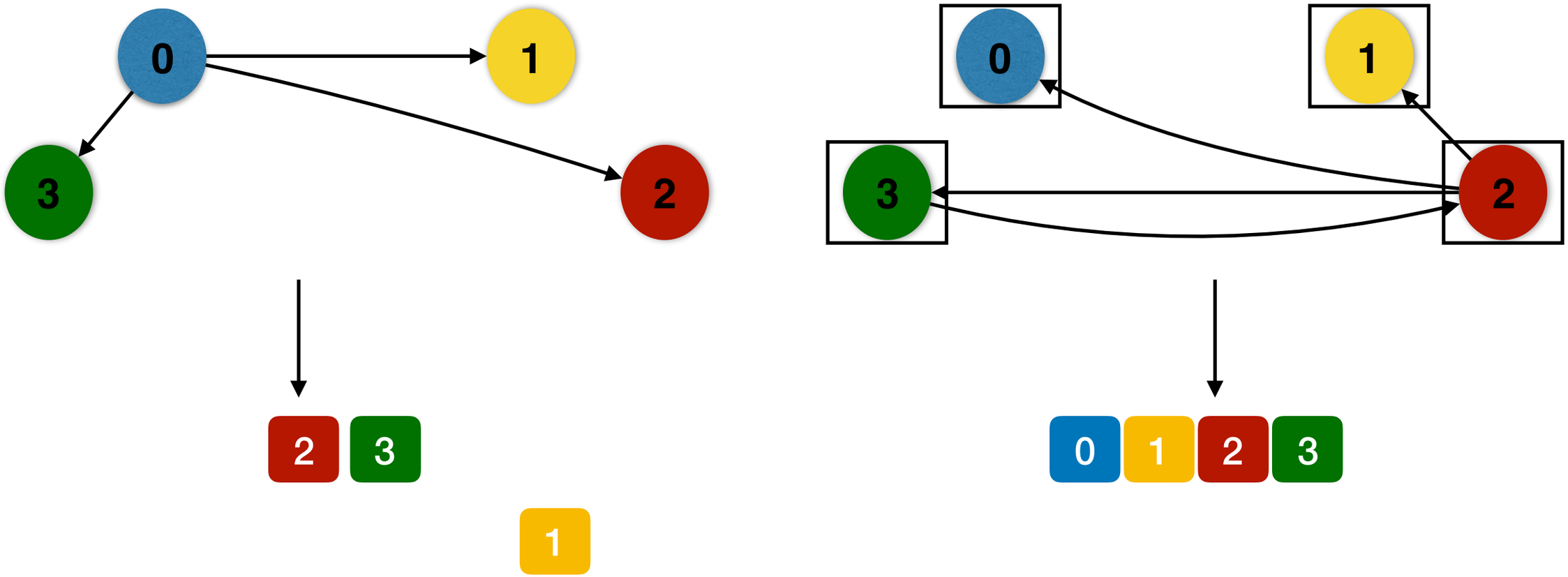
**The naive approach incurs random DRAM accesses**

# Cache-aware Merge

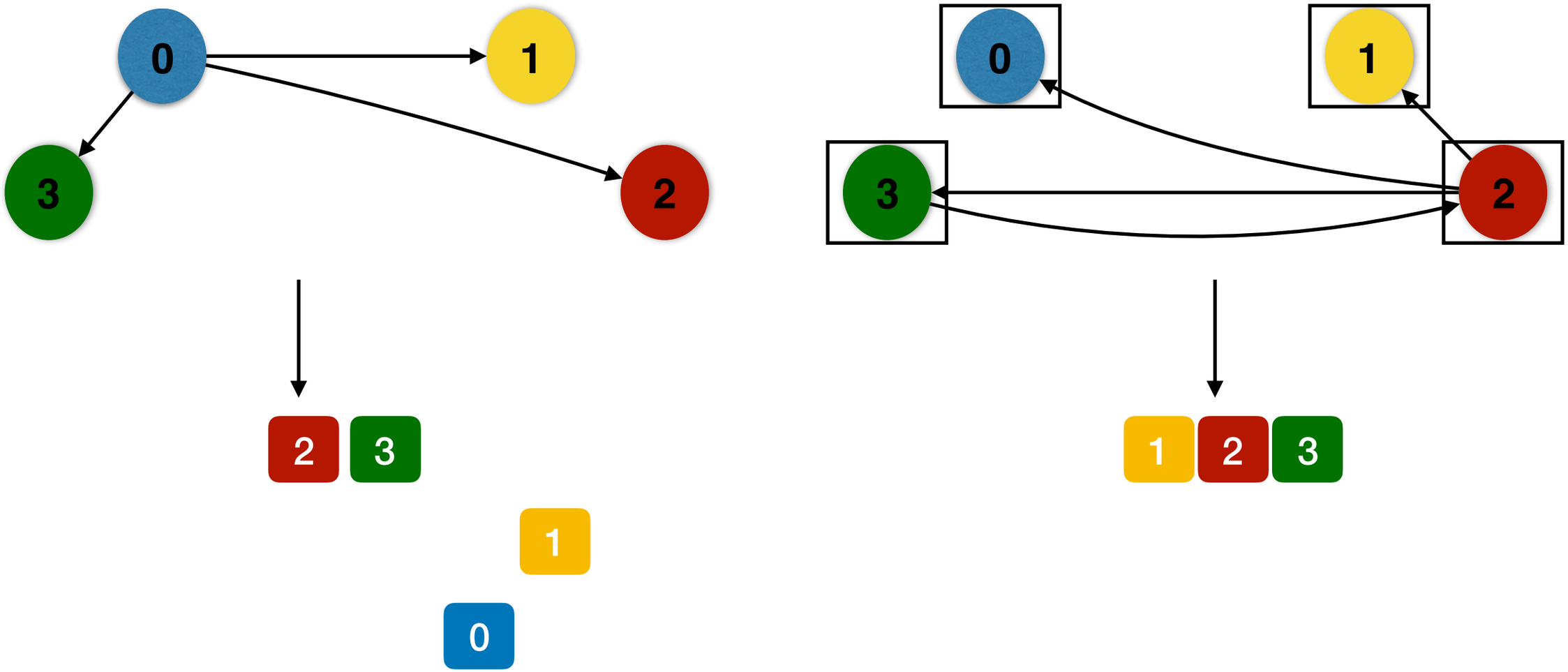


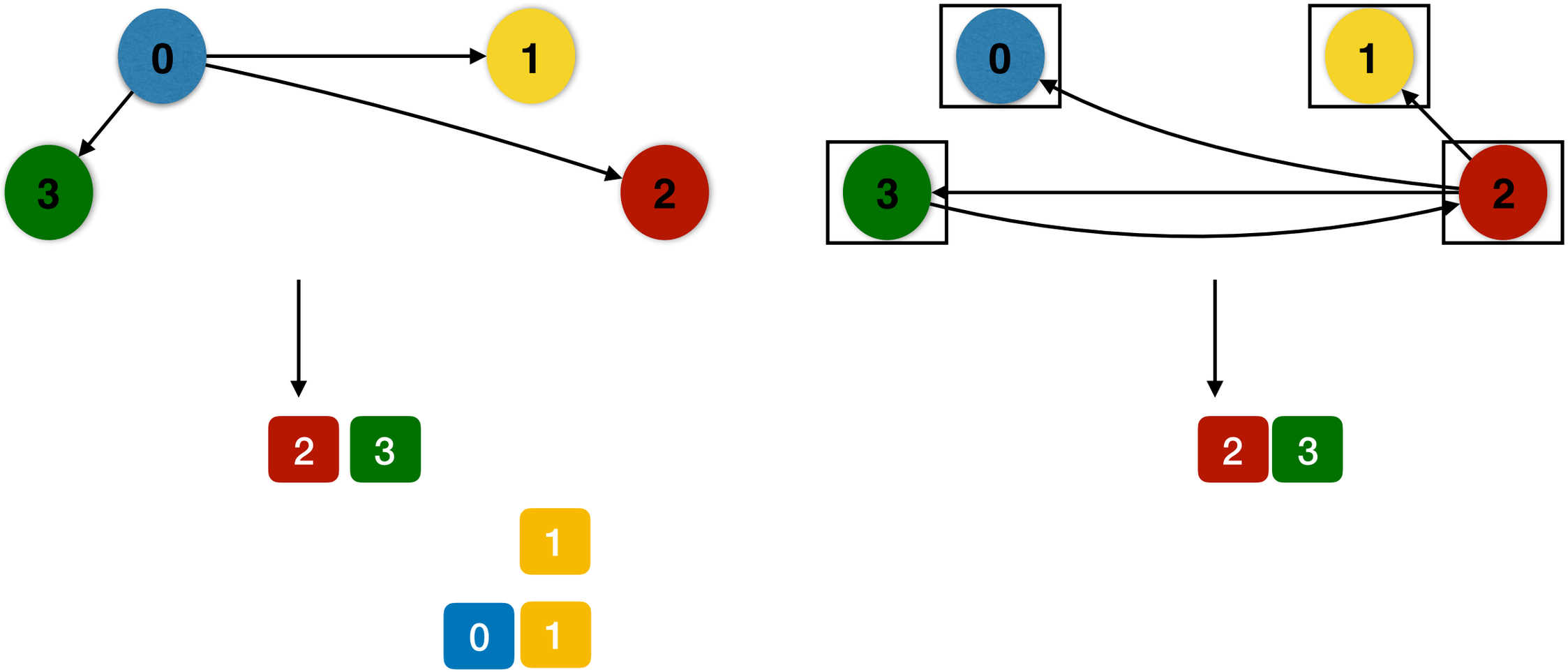
# Cache-aware Merge



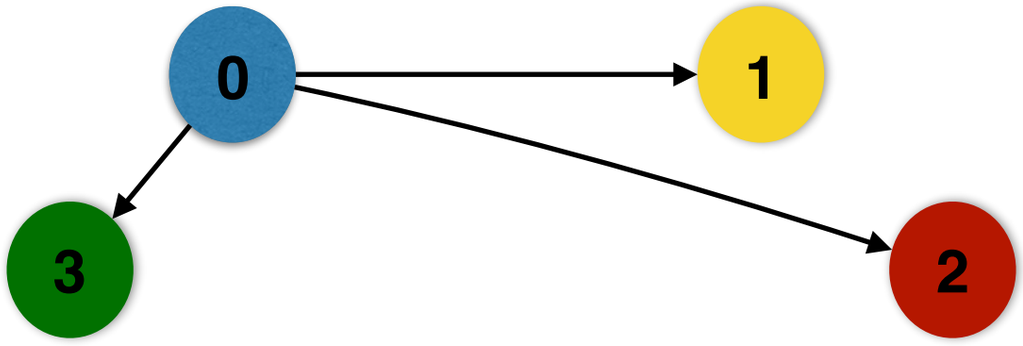
# Cache-aware Merge

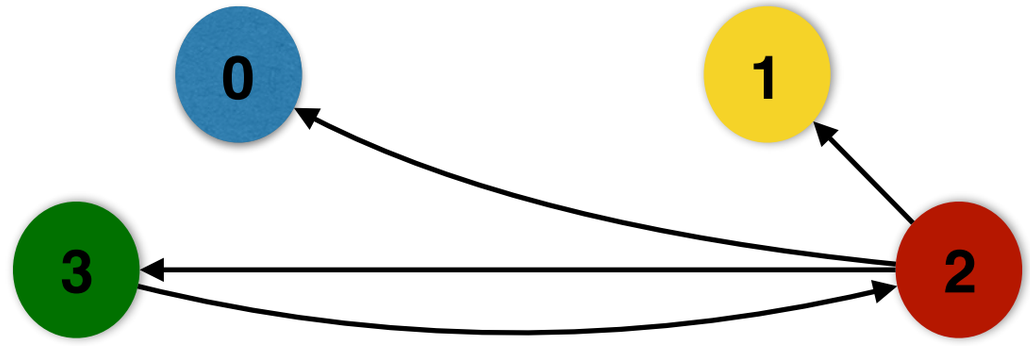


# Cache-aware Merge



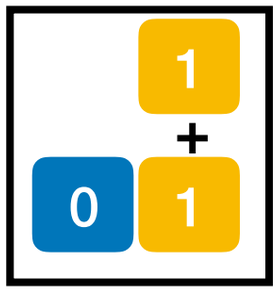
# Cache-aware Merge



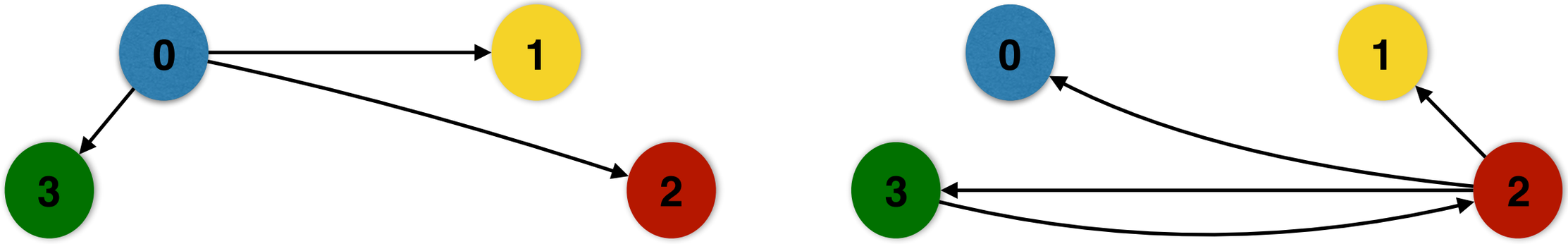






**Break down into chunks that fit in cache**

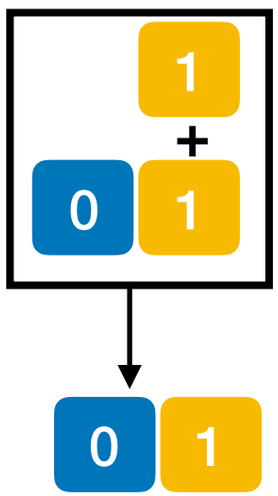


# Cache-aware Merge

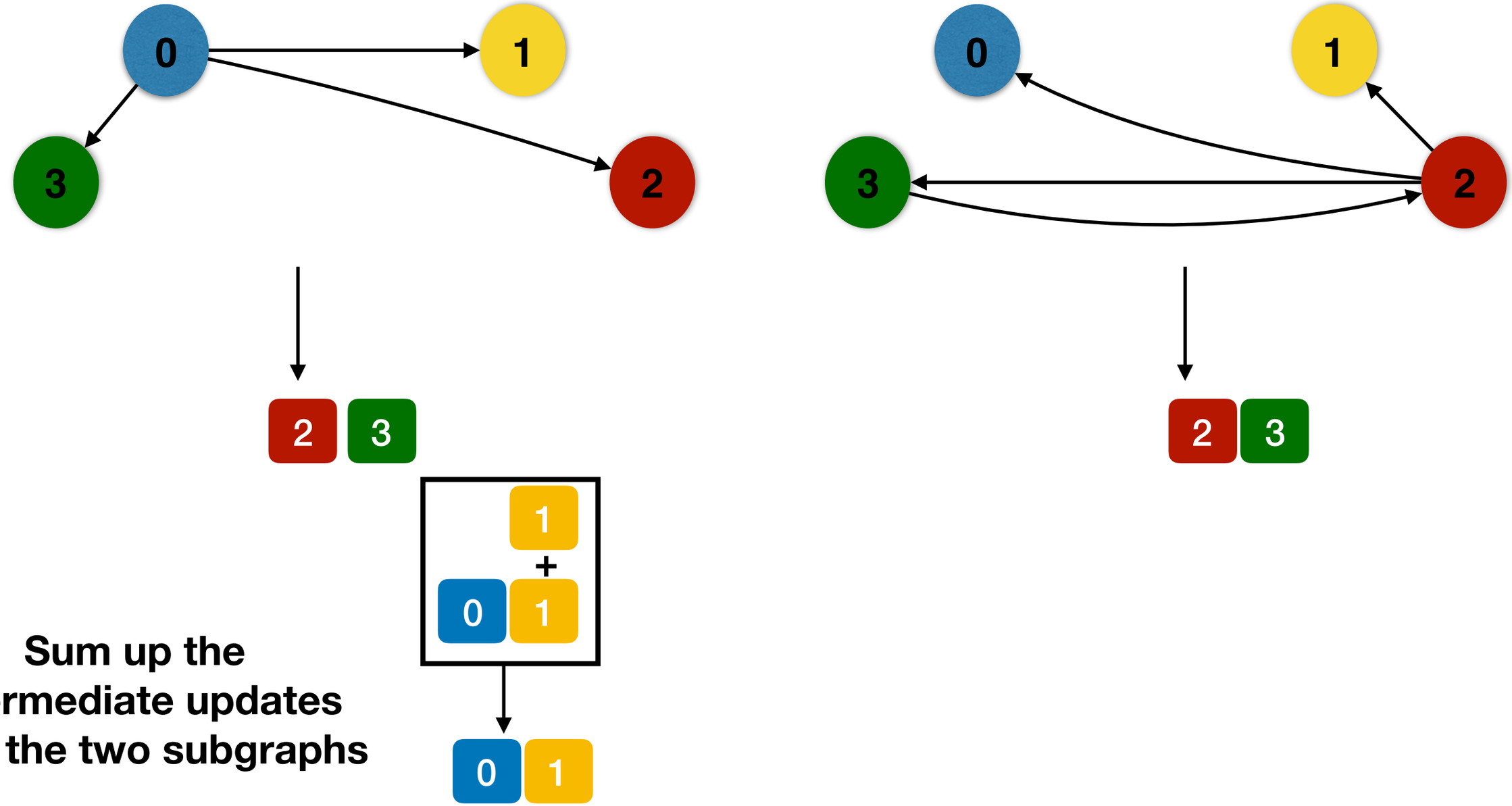




**Sum up the intermediate updates from the two subgraphs**

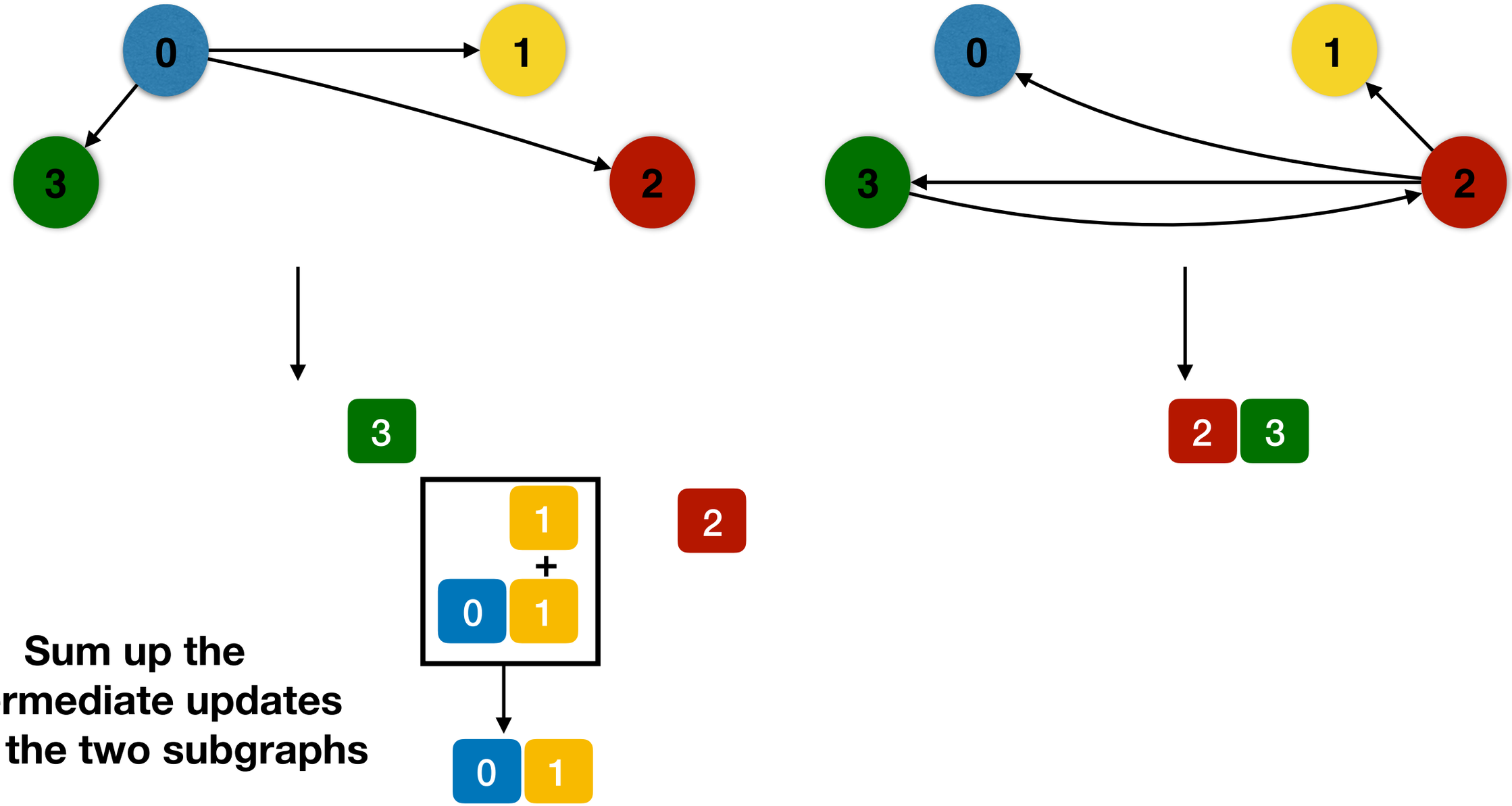


# Cache-aware Merge

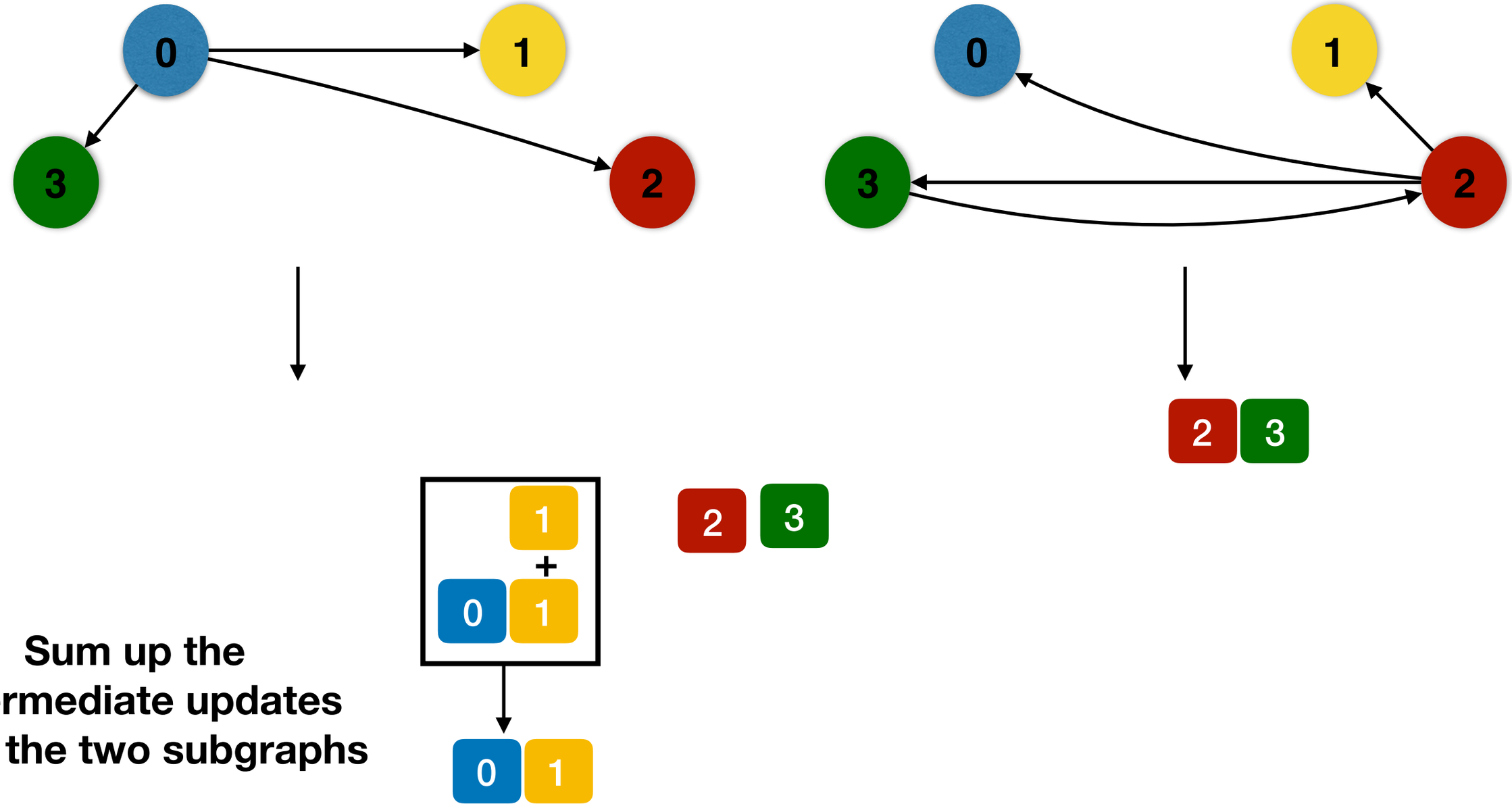



**Sum up the intermediate updates from the two subgraphs**

# Cache-aware Merge

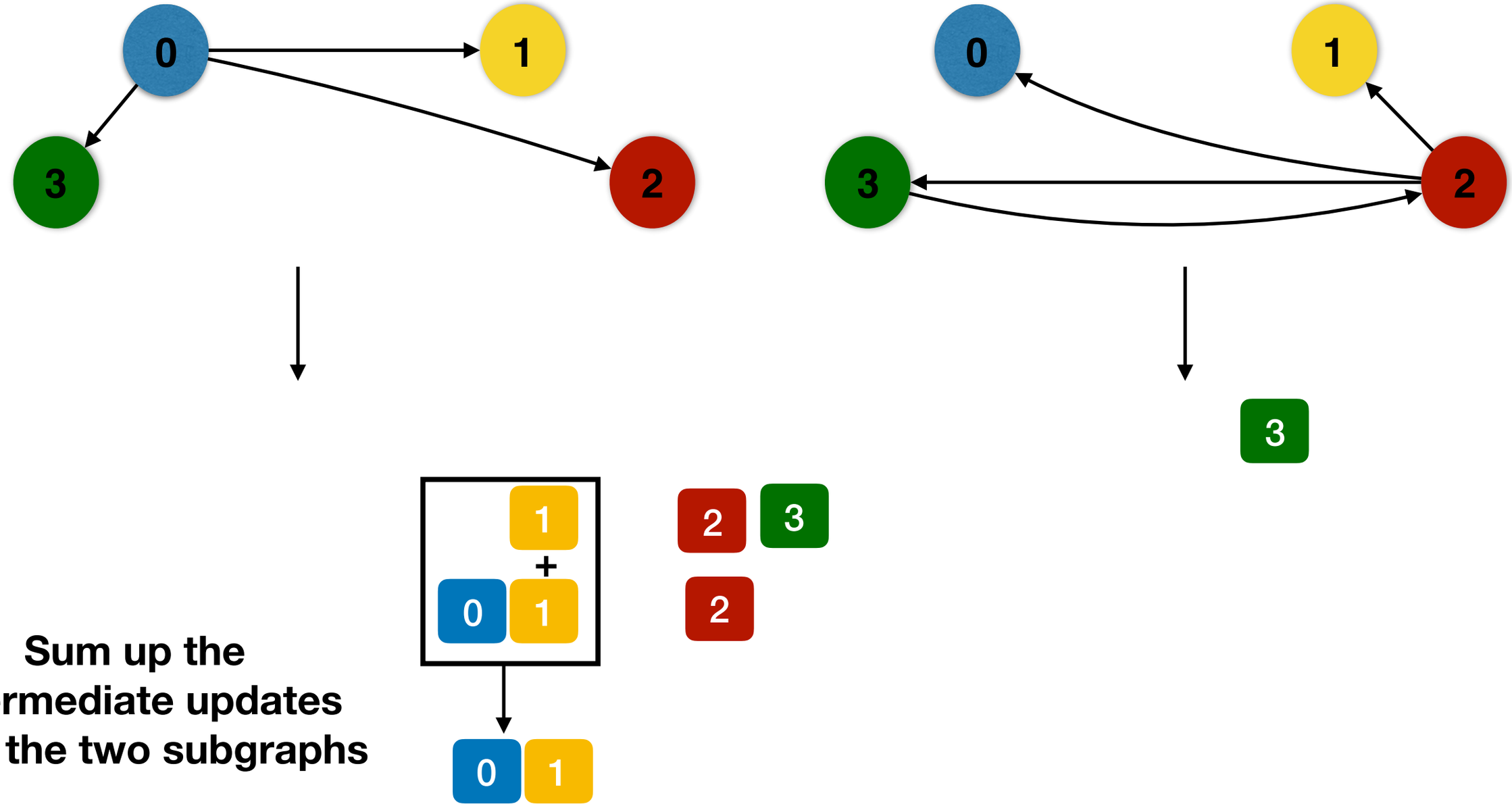


# Cache-aware Merge



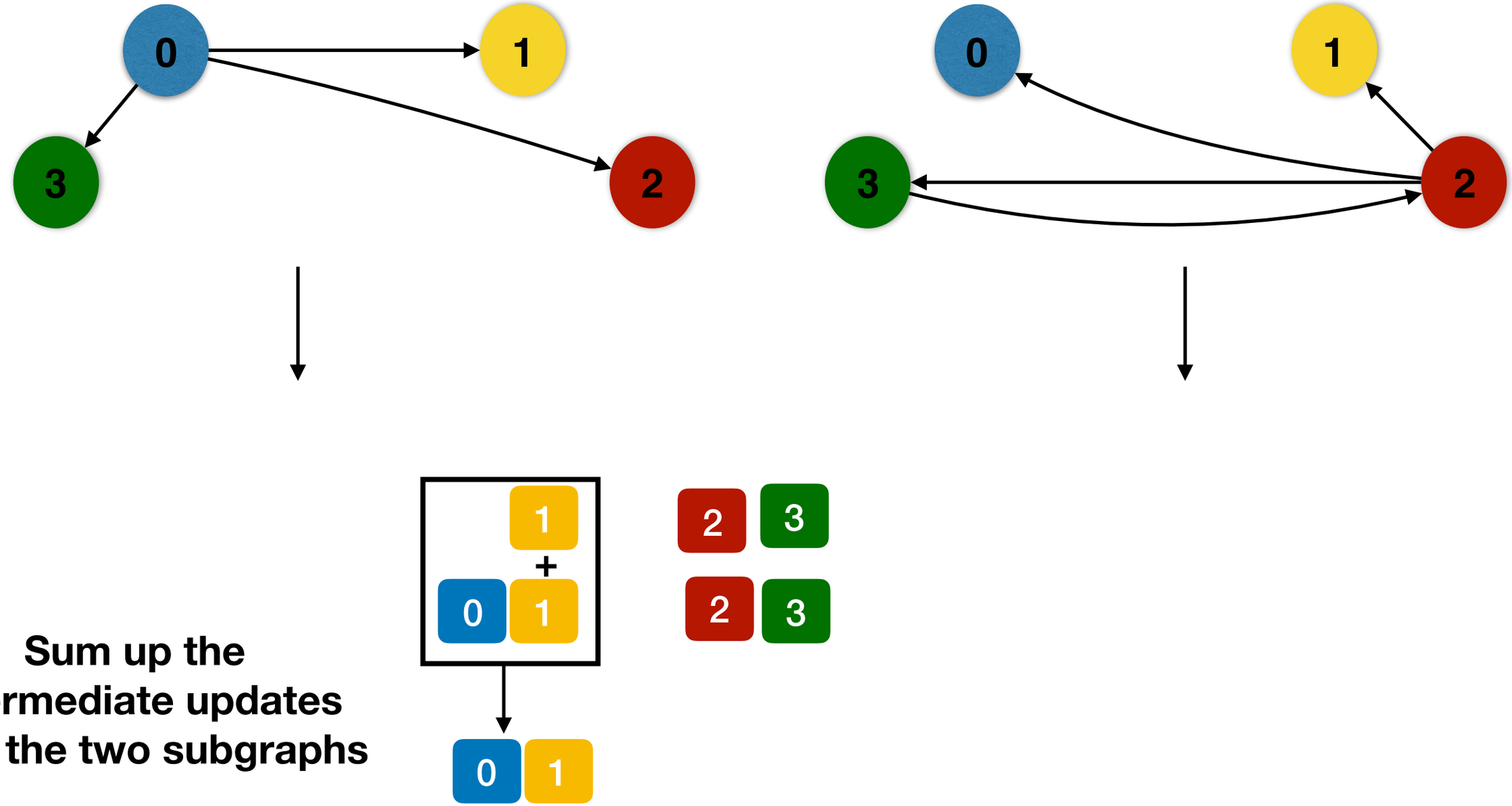

**Sum up the intermediate updates from the two subgraphs**

# Cache-aware Merge



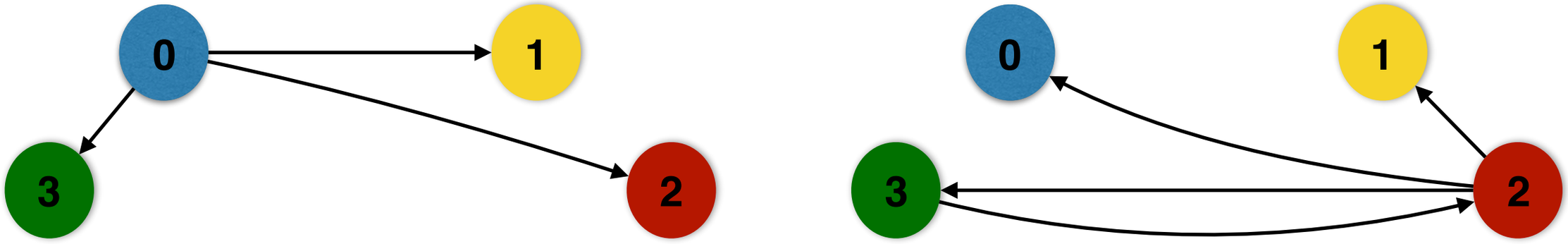

**Sum up the intermediate updates from the two subgraphs**

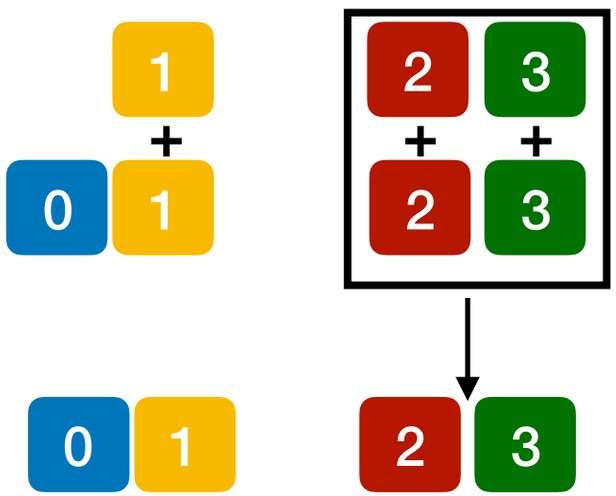
# Cache-aware Merge




**Sum up the intermediate updates from the two subgraphs**

# Cache-aware Merge

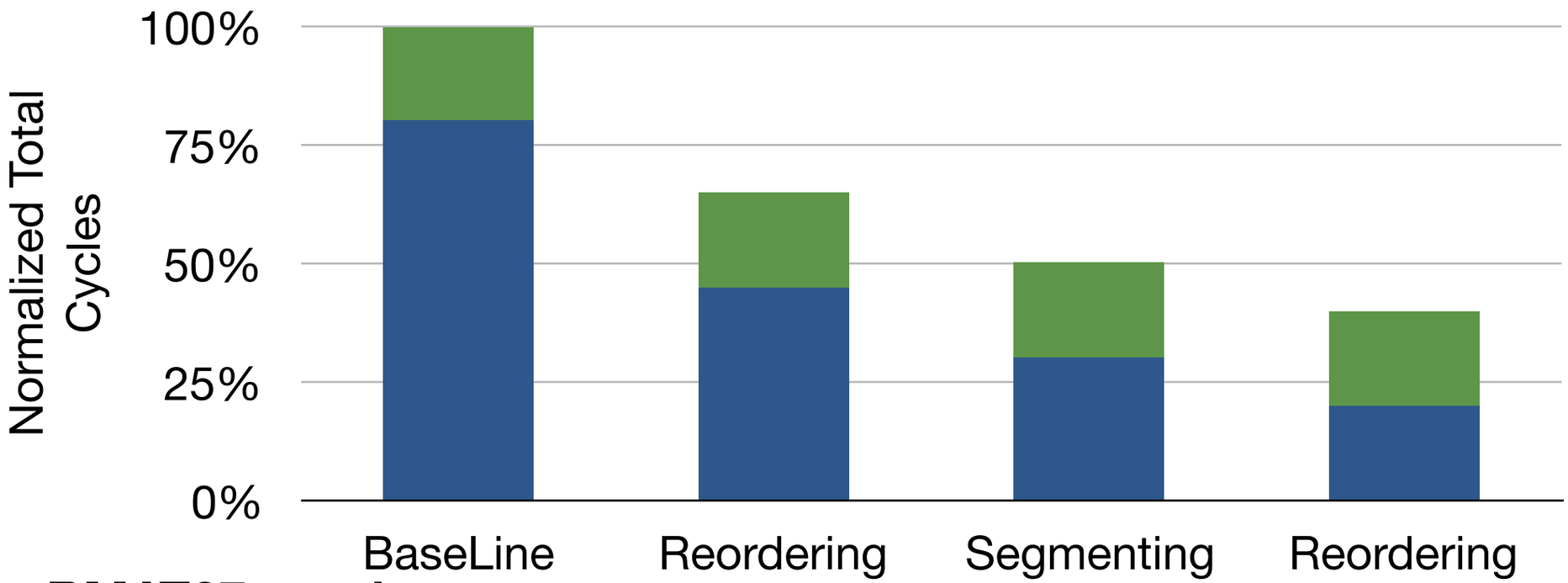



**Sum up the intermediate updates from the two subgraphs**

# PageRank

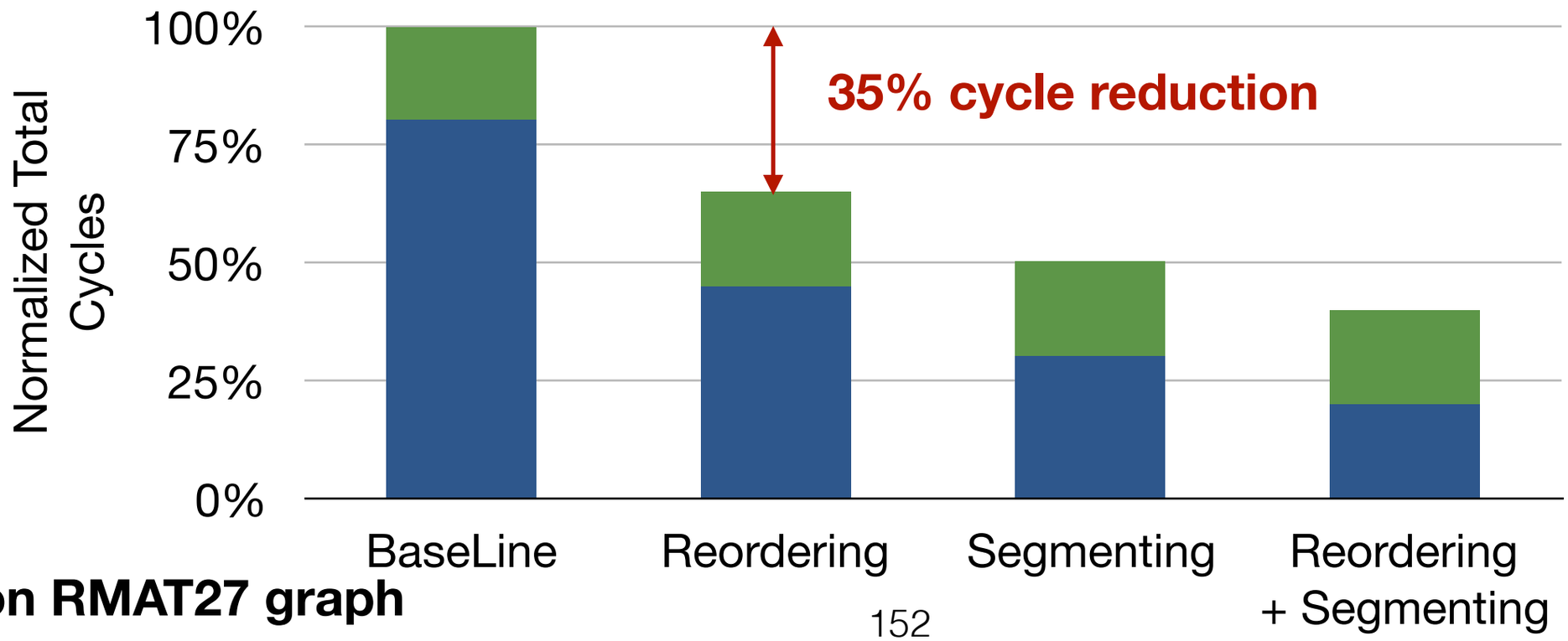
```
while …
      for node : graph.vertices
          for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
          newRanks[node] = baseScore +
damping*newRanks[node];
          swap ranks and newRanks
```




**on RMAT27 graph**

# PageRank

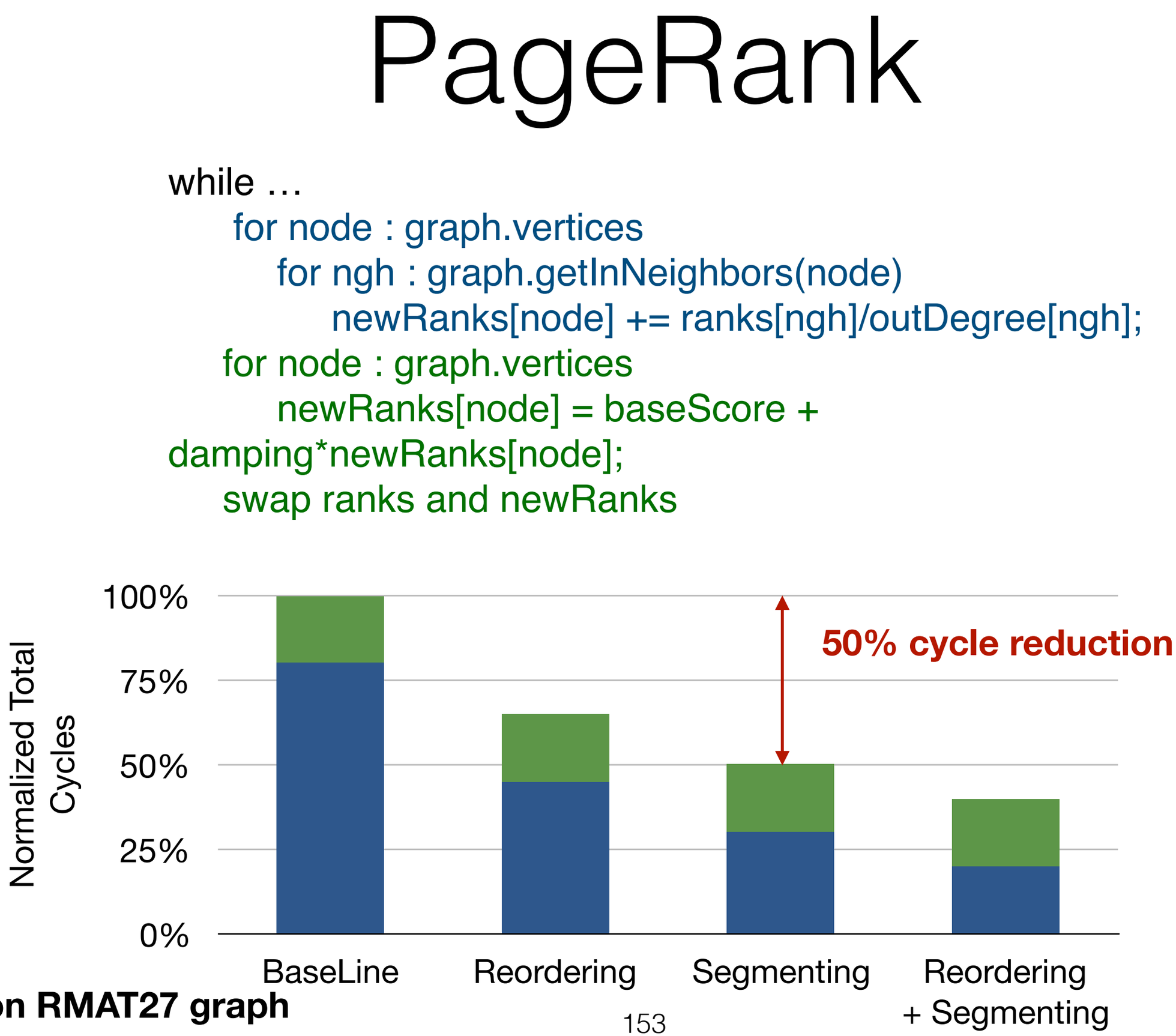
```
while …
      for node : graph.vertices
          for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
          newRanks[node] = baseScore +
damping*newRanks[node];
swap ranks and newRanks
```




**on RMAT27 graph**

# 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
```

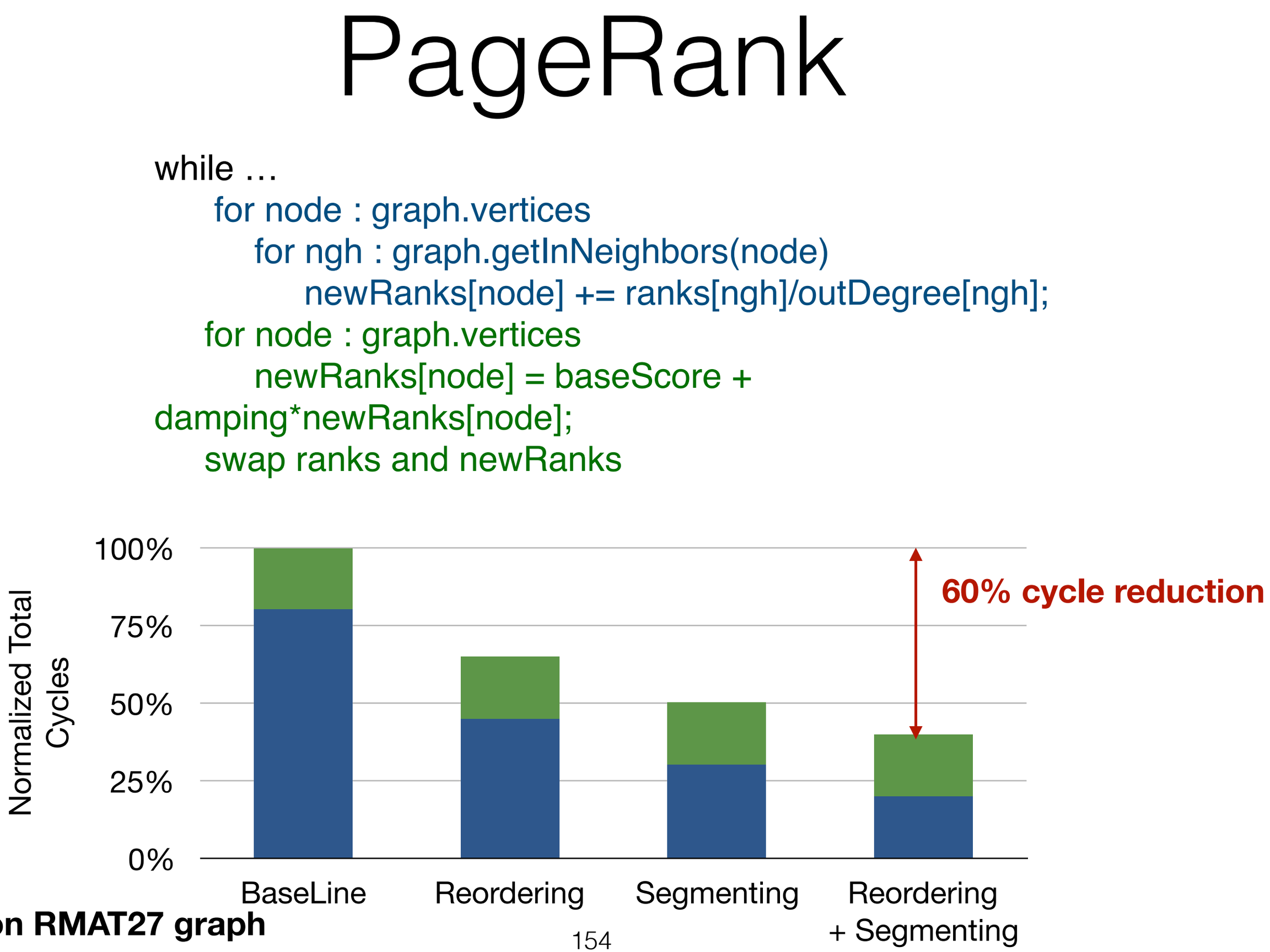



**on RMAT27 graph**

# 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




**on RMAT27 graph**

# Outline

- Motivation
- Related Works
- Frequency based Vertex Reordering
- Cache-aware Segmenting
- Evaluation

# Evaluation

| | PageRank (20 iter) | Label Propagation (per iter) | Betweenness Centrality (per start node) |
|---|---|---|---|
| **Twitter** | **5.8s** | **0.27s** | **1.21s** |
| **RMAT27** | **11.6s** | **0.52s** | **1.825s** |
| **Web Graph** | **8.6s** | **0.34s** | **0.0875s** |

**Absolute Running Times on 24 core Intel Xeon E5 servers**

# Evaluation

**In a single machine, we can complete 20 iterations of PageRank on 40 million nodes Twitter graph within 6s**

| | PageRank (20 iter) | Label Propagation (per iter) | Betweenness Centrality (per start node) |
|---|---|---|---|
| Twitter | 5.8s | 0.27s | 1.21s |
| RMAT27 | 11.6s | 0.52s | 1.825s |
| Web Graph | 8.6s | 0.34s | 0.0875s |

**Absolute Running Times on 24 core Intel Xeon E5 servers**

# Evaluation

| | PageRank (20 iter) | Label Propagation (per iter) | Betweenness Centrality (per start node) |
|---|---|---|---|
| Twitter | 5.8s | 0.27s | 1.21s |
| RMAT27 | 11.6s | 0.52s | 1.825s |
| Web Graph | 8.6s | 0.34s | 0.0875s |

**In a single machine, we can complete 20 iterations of PageRank on 40 million nodes Twitter graph within 6s**

**The best published results so far is 12.7s (Gemini OSDI 2017)**

**Absolute Running Times on 24 core Intel Xeon E5 servers**

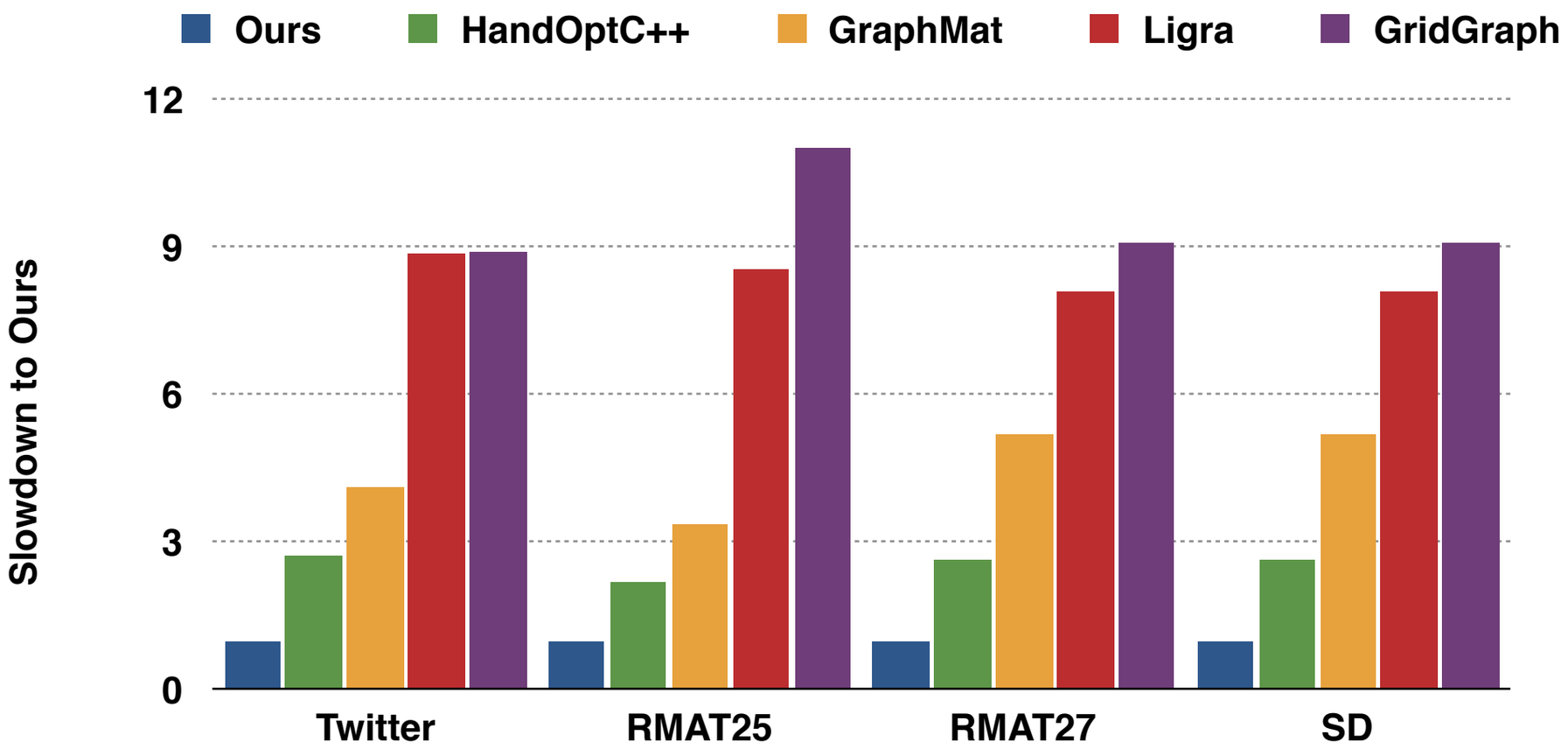
# Evaluation

| | PageRank (20 iter) | Label Propagation (per iter) | Betweenness Centrality (per start node) |
|---|---|---|---|
| Twitter | 5.8s | 0.27s | 1.21s |
| RMAT27 | 11.6s | 0.52s | 1.825s |
| Web Graph | 8.6s | 0.34s | 0.0875s |

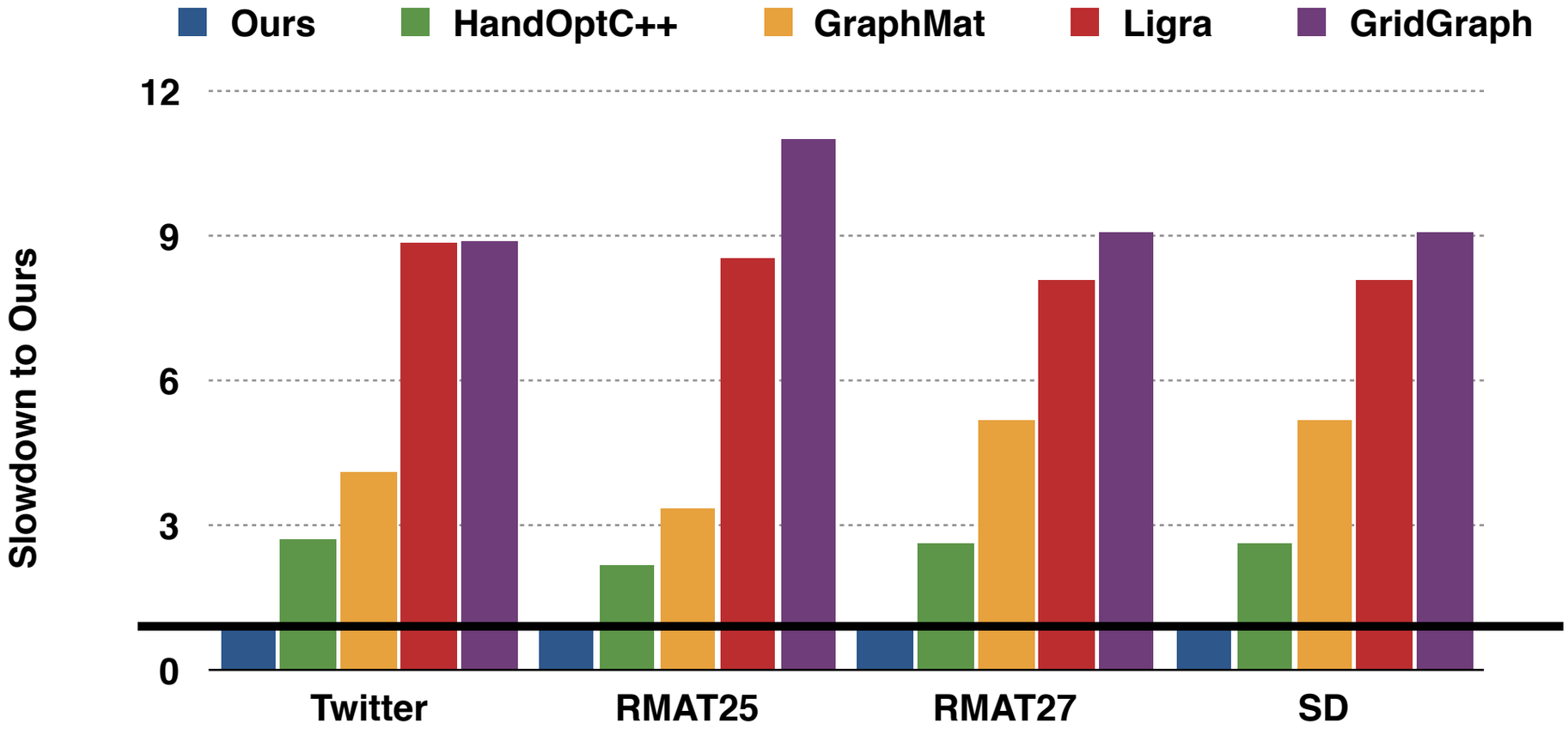
**Very fast execution on label propagation used in Connected Components and SSSP (Bellman-Ford)**

**Absolute Running Times on 24 core Intel Xeon E5 servers**

# PageRank

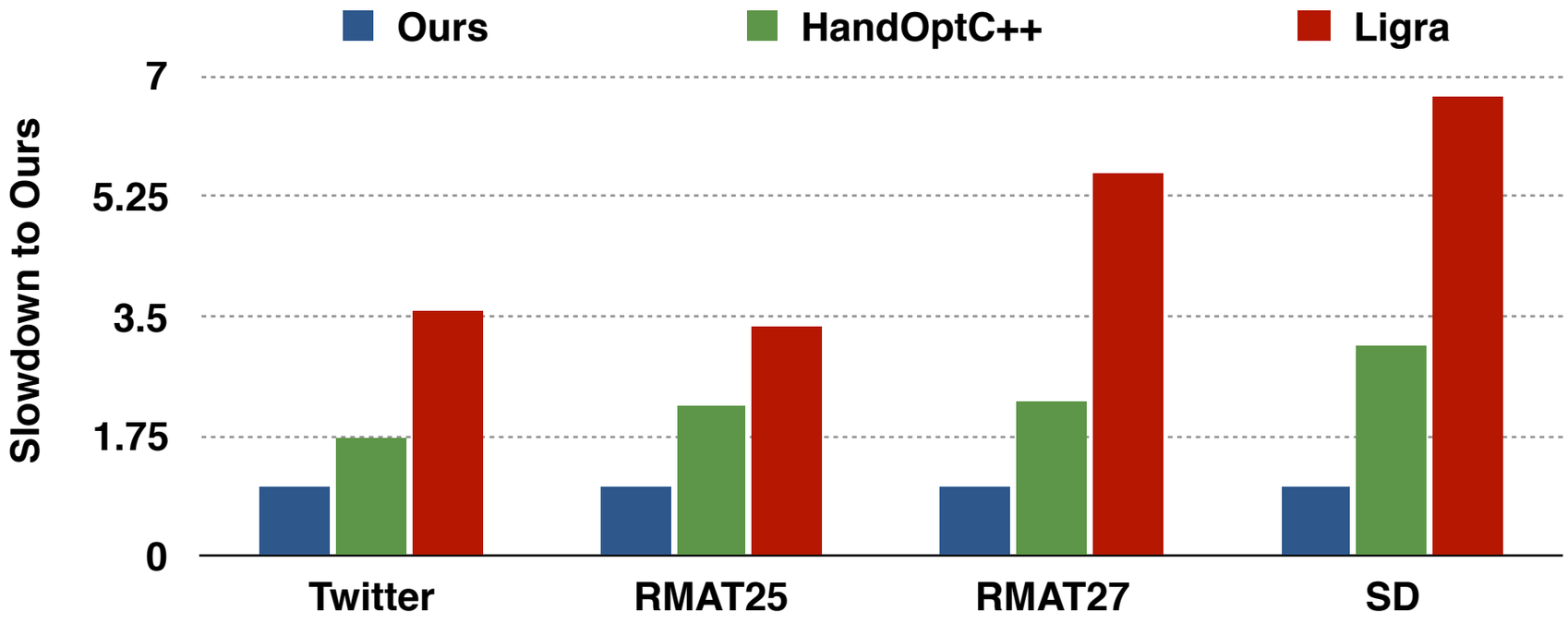

Ours
HandOptC++
GraphMat
Ligra
GridGraph
Slowdown to Ours
12
9
6
3
0
Twitter
RMAT25
RMAT27
SD

# PageRank

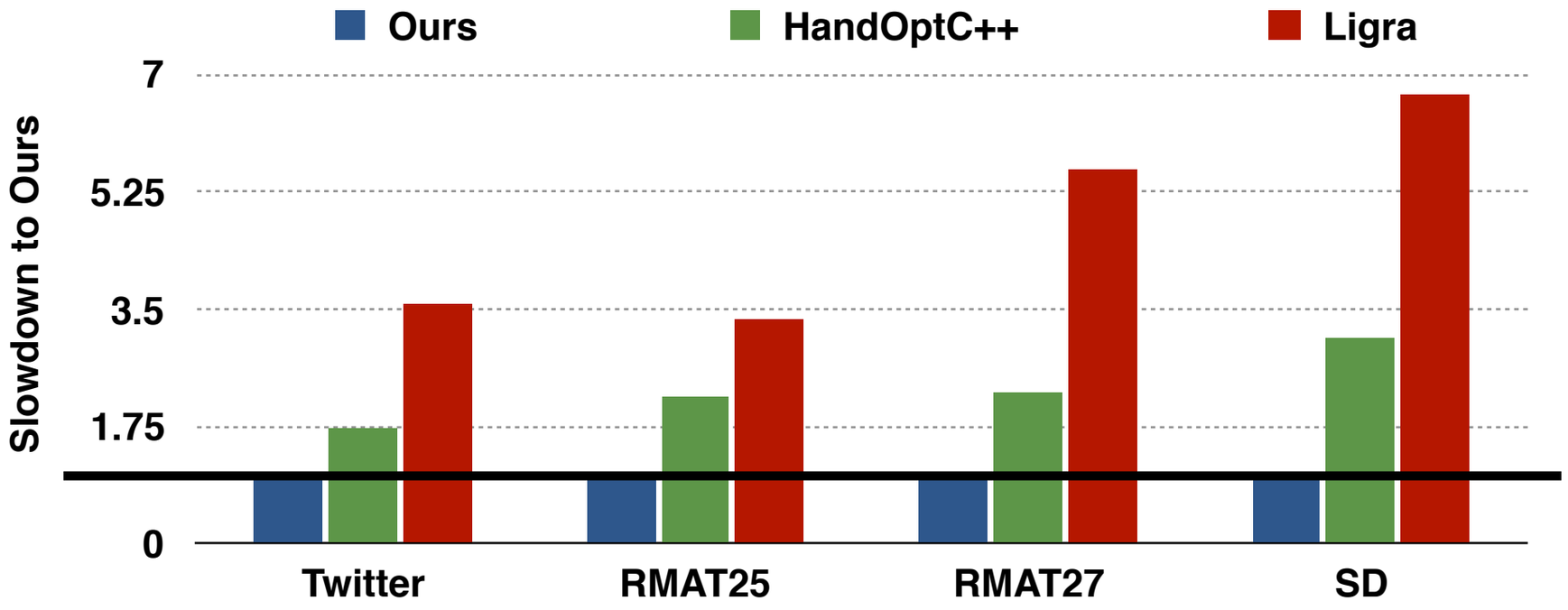



**Intel expert hand optimized version and state-of-the art graph frameworks are 2.2-11x slower than our version**

# Label Propagation

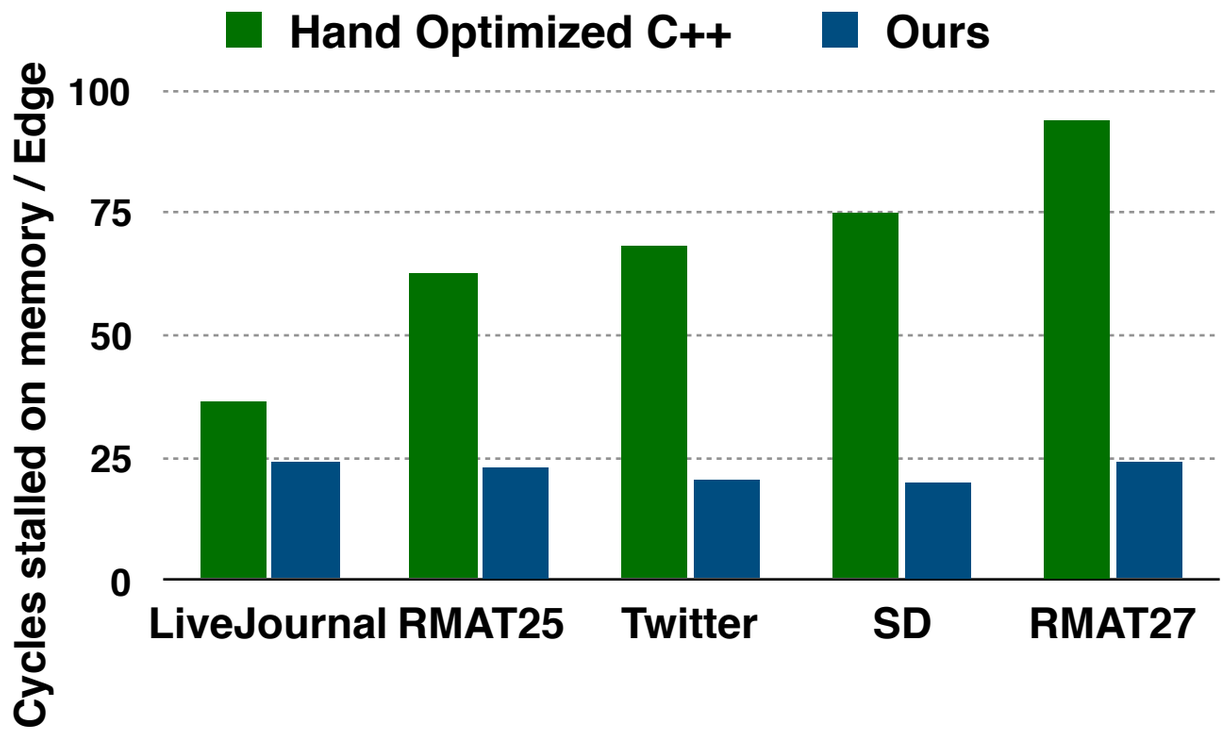


# Label Propagation

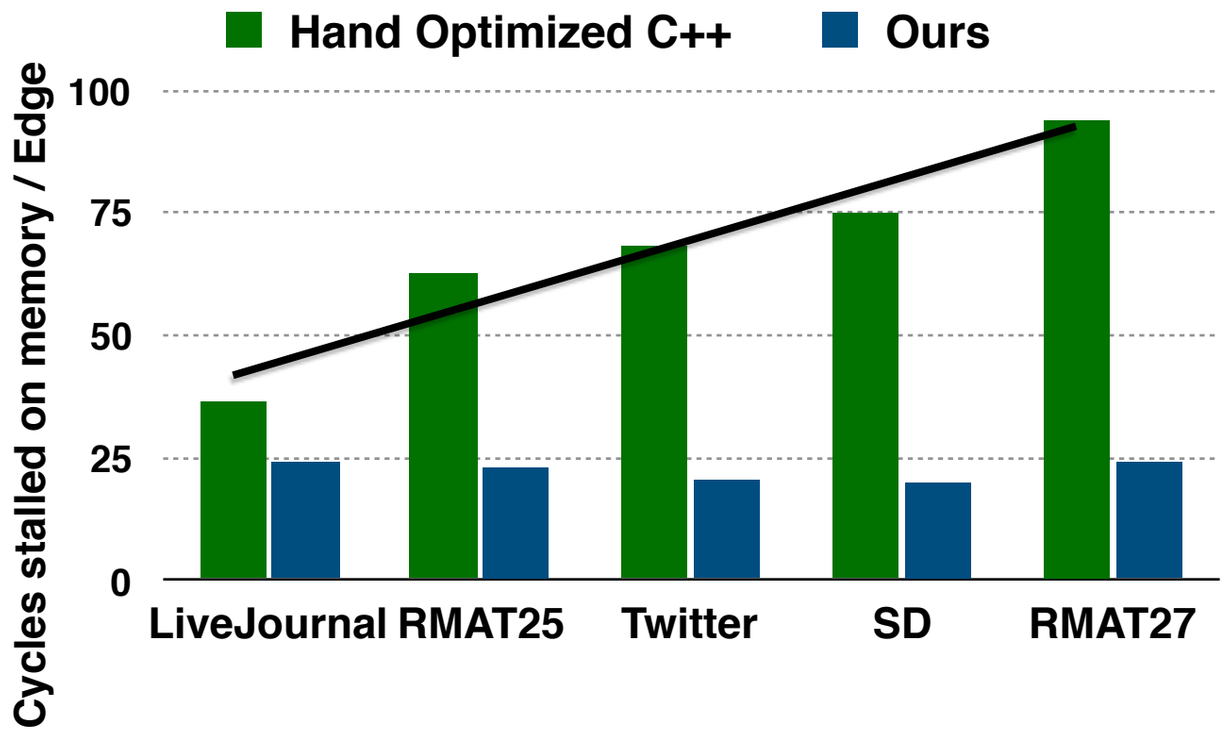



**Intel expert hand optimized version and state-of-the art graph frameworks are 1.7-6.7x slower than our version**

# Evaluation

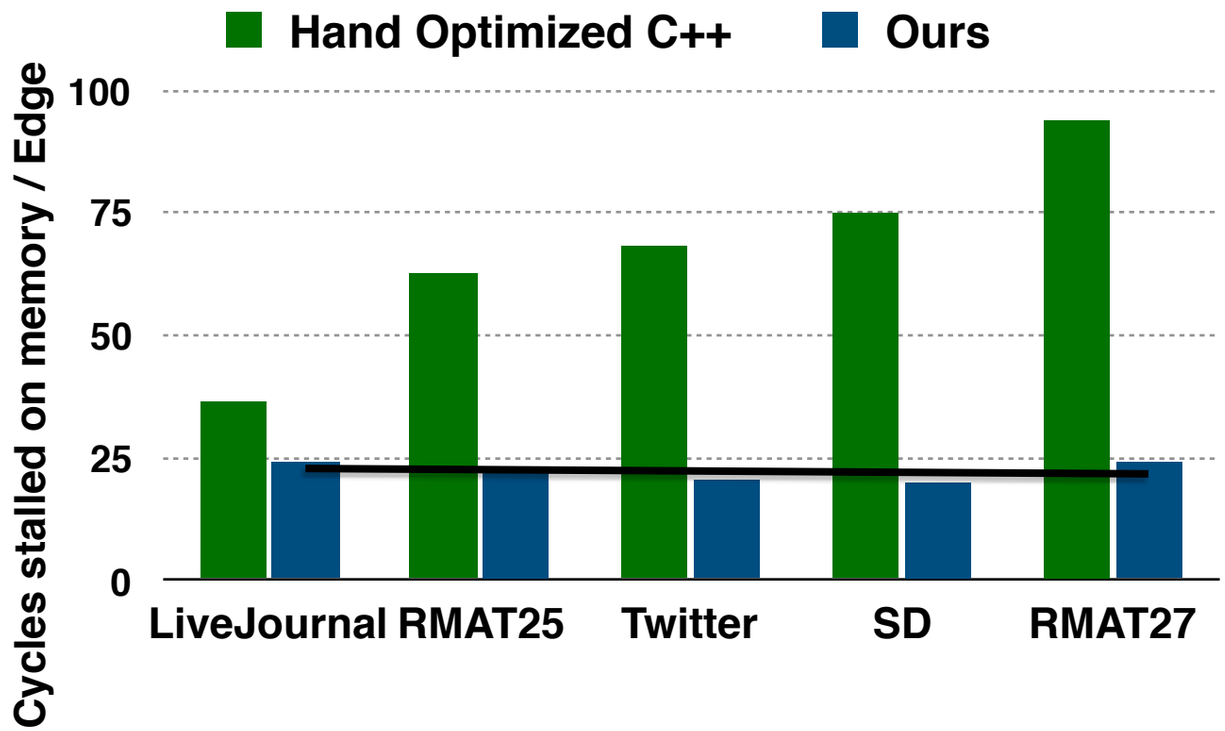

Hand Optimized C++
Ours
Cycles stalled on memory / Edge
100
75
50
25
0
LiveJournal
RMAT25
Twitter
SD
RMAT27

# Evaluation




**Cycles stalled on memory per edge increases as the size of the graph increases**

# Evaluation




**Cycles stalled on memory per edge stays constant as the size of the graph increases**

# Summary

- Performance Bottleneck of Graph Applications
- Frequency based Vertex Reordering
- Cache-aware Segmenting

# Outline

- Overview
- Milk
- Cagra
- GraphIt
- Conclusion

# Outline

- Overview
- Milk
- Cagra
- GraphIt
- Conclusion

# Conclusion

- Locality is very important
- Performance optimizations are all about tradeoffs (always engineer for your specific applications and data)
- Separation of algorithm from optimization (gets good programmability without sacrificing too much performance)

# Ongoing Projects

- Optimizing Performance for Ordered Graph Algorithms
- Running Graph Algorithms on GPU
- Applying Graph Optimizations to Sparse Matrix Linear Algebra
- Subgraph Matching