Optimizing Cache Performance for Graph Analytics

Yunming Zhang 6.886 Presentation

Goals

- How to optimize in-memory graph applications
- How to go about performance engineering just about anything

In-memory Graph Processing

- Compare to Disk / DRAM boundary (GraphChi, BigSparse, LLAMA..), Cache / DRAM boundary has
    - Much smaller latency gap (L3 cache 10-30 ns, DRAM 80-100 ns, Flash 100,000 ns (100 ms))
    - Much larger memory bandwidth (DRAM >100GB/s, Flash 6GB/s)
    - Much smaller granularity (64 bytes cache lines vs 4k or 2 MB pages)

Outline

- Performance Analysis for Graph Applications
- Milk / Propagation Blocking
- Frequency based Clustering
- CSR Segmenting
- Summary

Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server

Scott Beamer, Krste Asanović, David Patterson

Mostly borrowed from the authors' IISWC presentation

Motivation

- What is the performance bottleneck for graph applications running in memory?
- How much performance can we gain?
- How can we achieve the performance gains?

Graph Algorithms Are Random?

Graph Algorithms Are Random?

"Thus, the low speedup of OOO execution is due solely to a lack of memory bandwidth required to service the repeated last level cache misses caused by the random access memory pattern of the algorithm."

PPoPP 2011

Graph Algorithms Are Random?

"Thus, the low speedup of OOO execution is due solely to a lack of memory bandwidth required to service the repeated last level cache misses caused by the random access memory pattern of the algorithm."

PPoPP 2011

Graph Algorithms Are Random?

"Thus, the low speedup of OOO execution is due solely to a lack of memory bandwidth required to service the repeated last level cache misses caused by the random access memory pattern of the algorithm."

PPoPP 2011

"First, the memory bandwidth of the system seems to limit performance"

SPAA 2010

Graph Algorithms Are Random?

"Thus, the low speedup of OOO execution is due solely to a lack of memory bandwidth required to service the repeated last level cache misses caused by the random access memory pattern of the algorithm."

PPoPP 2011

"First, the memory bandwidth of the system seems to limit performance"

SPAA 2010

Current Graph Architecture Wisdom?

Current Wisdom

Current Graph Architecture Wisdom?

Current Wisdom

- Random memory access pattern

Current Graph Architecture Wisdom?

Current Wisdom

- Random memory access pattern
- Limited by memory bandwidth

Current Graph Architecture Wisdom?

Current Wisdom

- Random memory access pattern
- Limited by memory bandwidth
- Will be plagued by low core utilization

Current Graph Architecture Wisdom?

Current Wisdom

- Random memory access pattern
- Limited by memory bandwidth
- Will be plagued by low core utilization

Cray XMT Design

Current Graph Architecture Wisdom?

Current Wisdom

- Random memory access pattern
- Limited by memory bandwidth
- Will be plagued by low core utilization

Cray XMT Design

- No caches

Current Graph Architecture Wisdom?

Current Wisdom

- Random memory access pattern
- Limited by memory bandwidth
- Will be plagued by low core utilization

Cray XMT Design

- No caches
- Heavy multithreading

- Are graph applications really memory bandwidth bounded?
  - Is cache really completely useless in graph computations?

Results from Characterization

Results from Characterization

- No single representative workload

Results from Characterization

- No single representative workload
  - need suite

Results from Characterization

- No single representative workload
  - need suite
- Out-of-order core not limited by memory bandwidth for most graph workloads

Results from Characterization

- No single representative workload
  - need suite
- Out-of-order core not limited by memory bandwidth for most graph workloads
  - can improve by changing only processor

Results from Characterization

- No single representative workload
  - need suite
- Out-of-order core not limited by memory bandwidth for most graph workloads
  - can improve by changing only processor
- Many graph workloads have good locality

Results from Characterization

- No single representative workload
  - need suite
- Out-of-order core not limited by memory bandwidth for most graph workloads
  - can improve by changing only processor
- Many graph workloads have good locality
  - caches help! try to avoid thrashing

Target Graph Algorithms

Most popular based on 45-paper literature survey:

- Breadth-First Search (BFS)
- Single-Source Shortest Paths (SSSP)
- PageRank (PR)
- Connected Components (CC)
- Betweenness Centrality (BC)

Target Graph Frameworks

Target Graph Frameworks

- **Galois** (custom parallel runtime) - UT Austin
  - specialized for irregular fine-grain tasks

Target Graph Frameworks

- **Galois** (custom parallel runtime) - UT Austin
  - specialized for irregular fine-grain tasks
- **Ligra** (Cilk) - CMU
  - applies algorithm in push or pull directions

Target Graph Frameworks

- **Galois** (custom parallel runtime) - UT Austin
  - specialized for irregular fine-grain tasks
- **Ligra** (Cilk) - CMU
  - applies algorithm in push or pull directions
- **GAP** Benchmark Suite (OpenMP) - UCB
  - written directly in most natural way for algorithm, not constrained by framework

Target Input Graphs

ASPIRE
UC Berkeley

Graph	# Vertices	# Edges	Degree	Diameter	Degree Dist.
Roads of USA	23.9M	58.3M	2.4	High	const
Twitter Follow Links	61.6M	1468.4M	23.8	Low	power
Web Crawl of .sk Domain	50.6M	1949.4M	38.5	Medium	power
Kronecker Synthetic Graph	128.0M	2048.0M	16.0	Low	power
Uniform Random Graph	128.0M	2048.0M	16.0	Low	normal

Graphs can have very different degree distributions, diameters and other structural characteristics.

How does the memory system work?

How does the memory system work?

- Executing a load that access DRAM requires:

How does the memory system work?

- Executing a load that access DRAM requires:

1. Execution reaches load instruction (**fetch**)

How does the memory system work?

- Executing a load that access DRAM requires:

1. Execution reaches load instruction (**fetch**)
2. Space in the instruction **window**

How does the memory system work?

- Executing a load that access DRAM requires:

1. Execution reaches load instruction (**fetch**)
2. Space in the instruction **window**
3. Register operands are available (**dataflow**)

How does the memory system work?

- Executing a load that access DRAM requires:

1. Execution reaches load instruction (**fetch**)
2. Space in the instruction **window**
3. Register operands are available (**dataflow**)
4. Memory **bandwidth** is available

How does the memory system work?

- Executing a load that access DRAM requires:

1. Execution reaches load instruction (**fetch**)
2. Space in the instruction window
3. Register operands are available (**dataflow**)
4. Memory **bandwidth** is available

- Bandwidth ~ # outstanding requests

How does the memory system work?

- Executing a load that access DRAM requires:

  1. Execution reaches load instruction (**fetch**)
  3. Register operands are available (**dataflow**)
  4. Memory **bandwidth** is available

Memory bandwidth (#4) matters only if (#1-3) satisfied

- Bandwidth ~ # outstanding requests

- Little's Law

- Little's Law

effective
MLP

(MLP = memory level parallelism)

- Little's Law

$$\text{effective MLP} = \text{average memory bandwidth} \times \text{average memory latency}$$

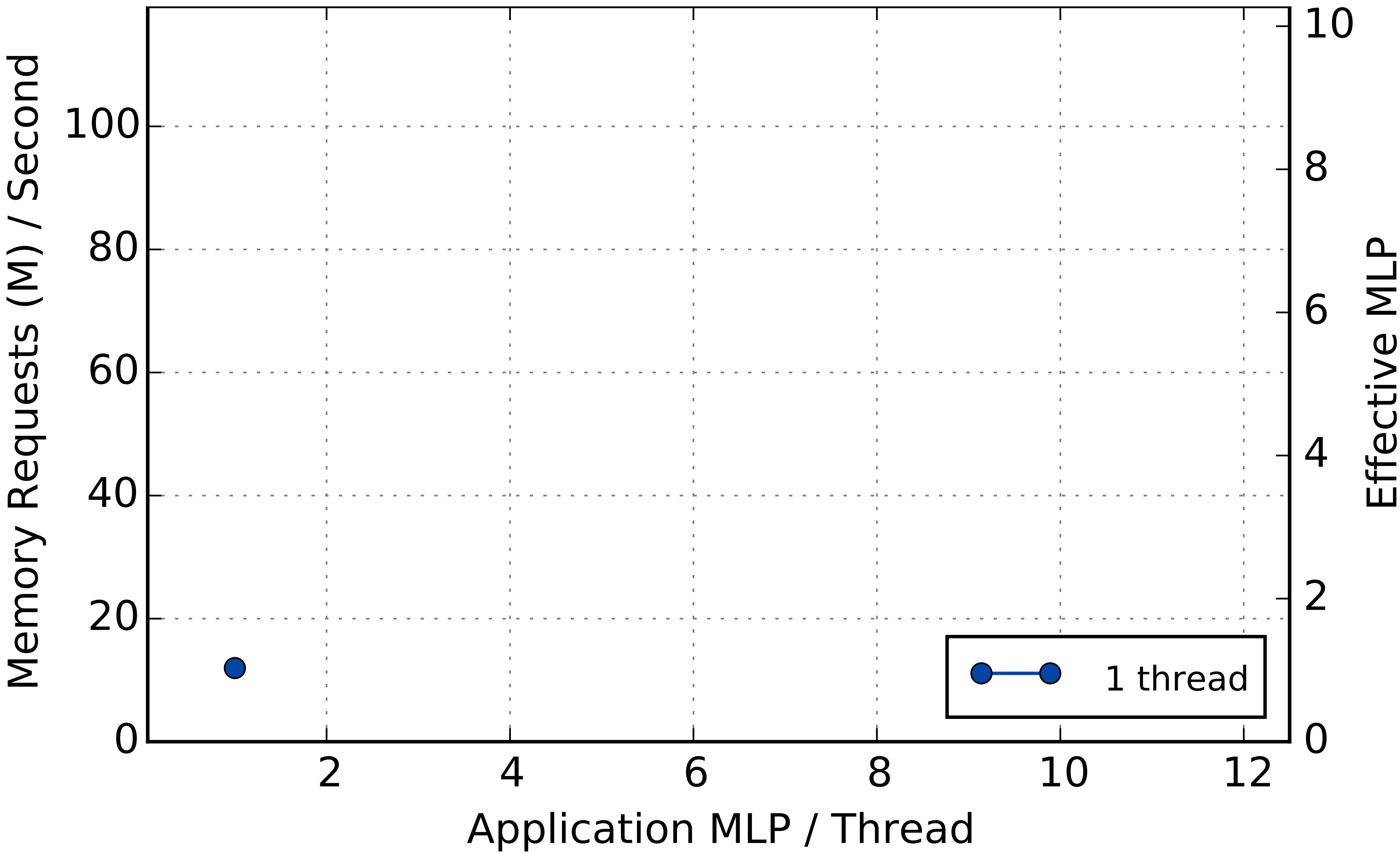
(MLP = memory level parallelism)

- Little's Law

$$\text{effective MLP} = \text{average memory bandwidth} \times \text{average memory latency} \leq \text{application MLP}$$

(MLP = memory level parallelism)

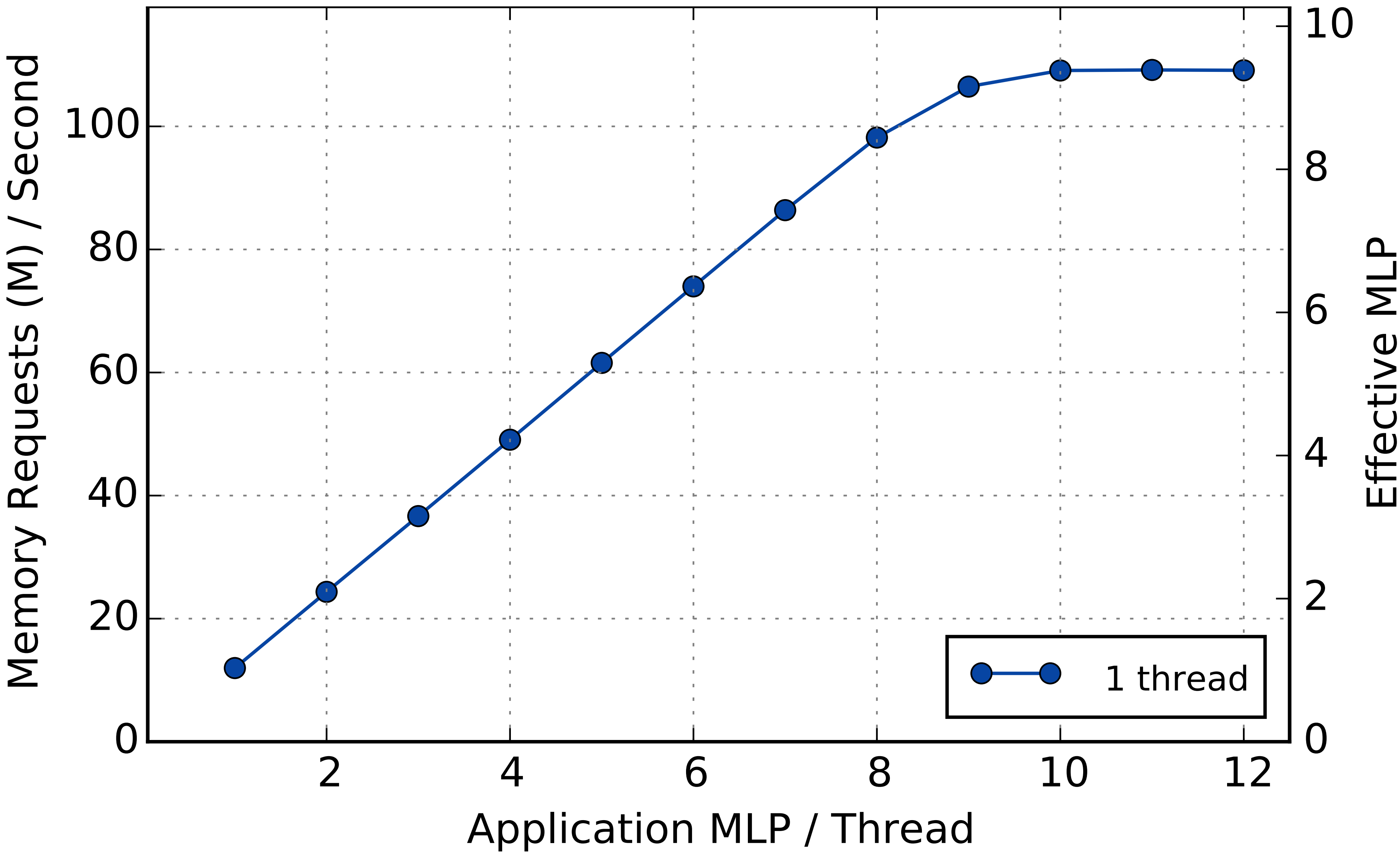
Single-Core Memory Bandwidth



1 core

Pointer Chasing Microbenchmark with varying number of parallel pointer chases

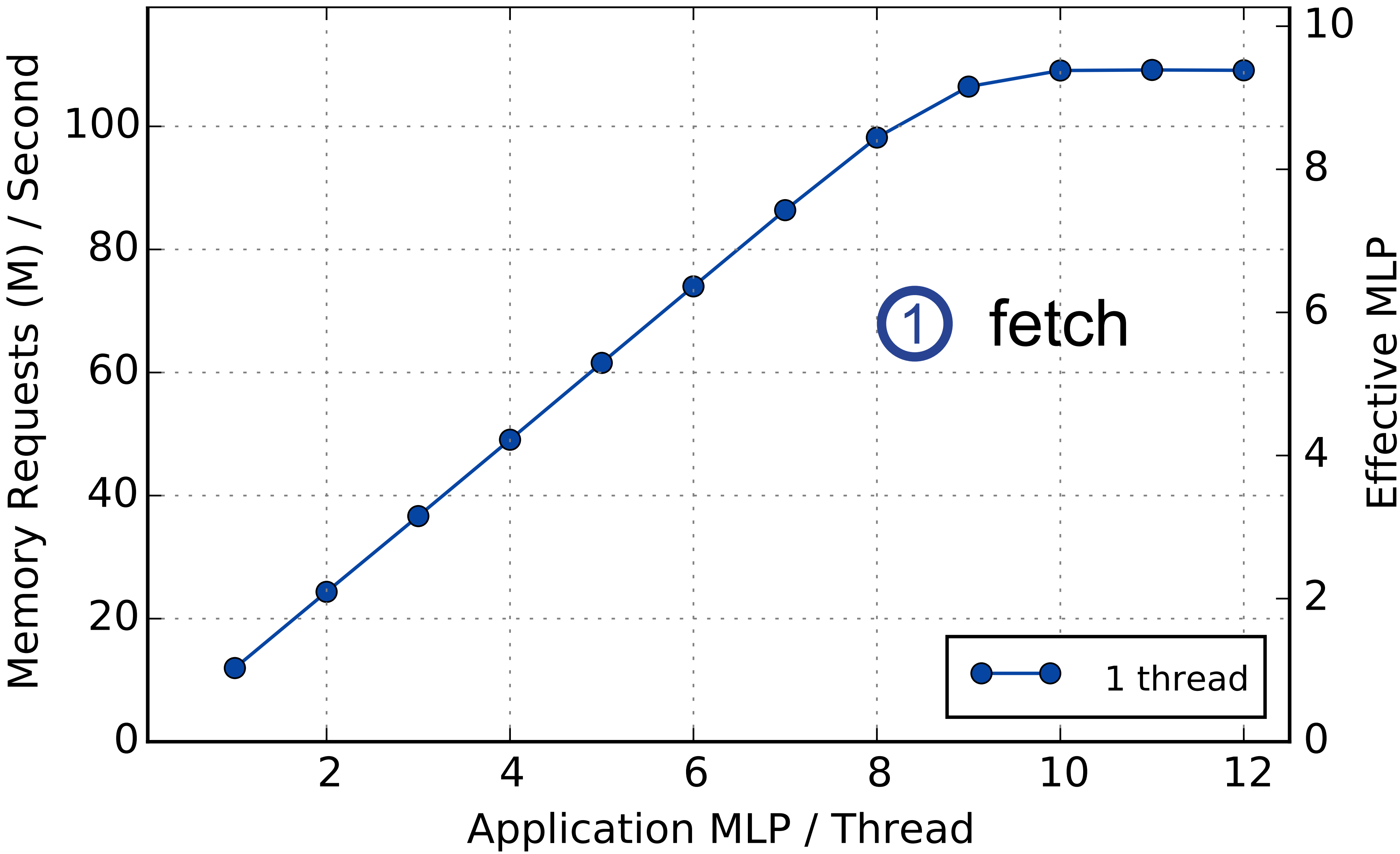
Single-Core Memory Bandwidth



1 core

Pointer Chasing Microbenchmark with varying number of parallel pointer chases

Single-Core Memory Bandwidth

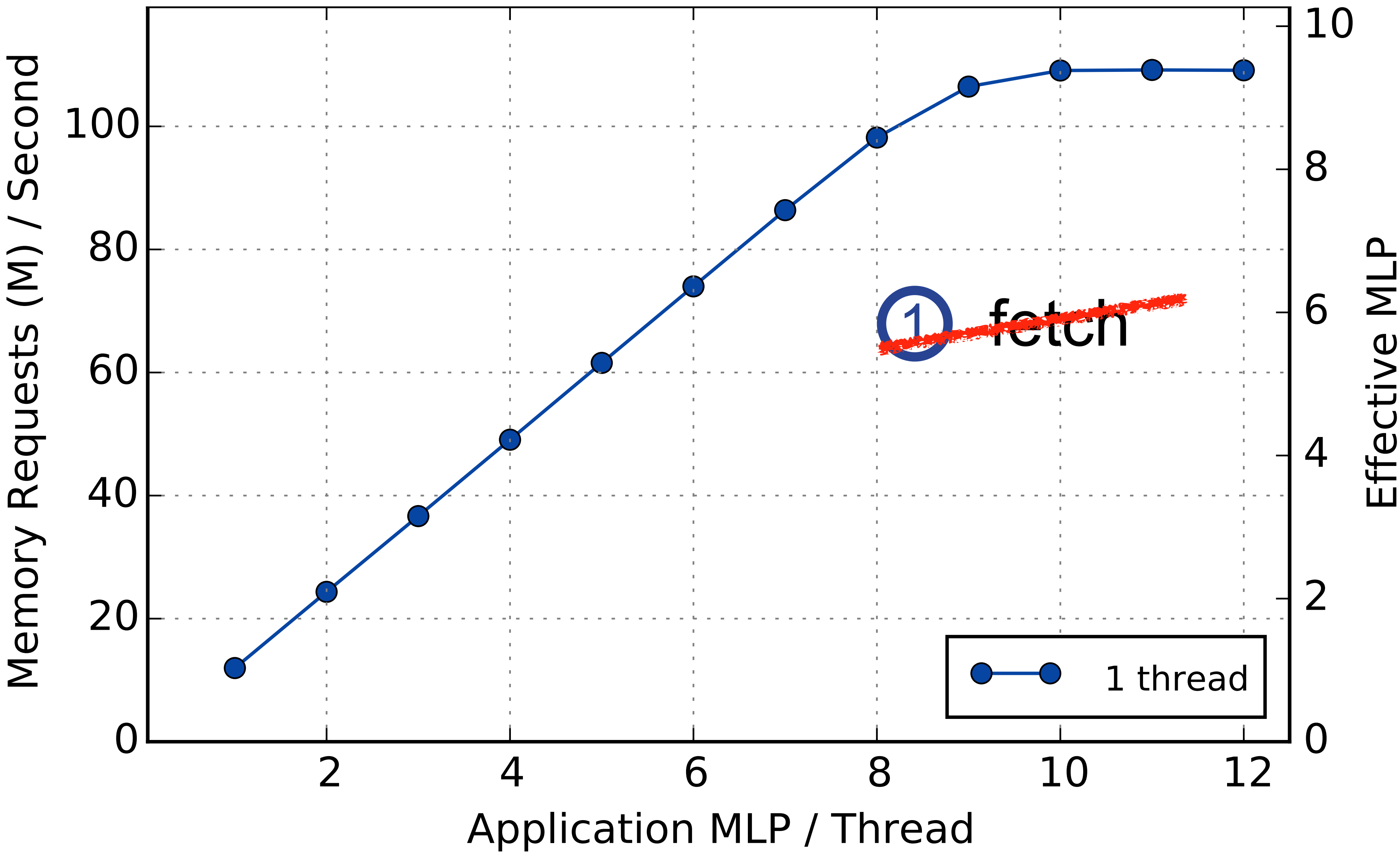


1 core

Pointer Chasing Microbenchmark with varying number of parallel pointer chases

Single-Core Memory Bandwidth

ASPIRE
UC Berkeley

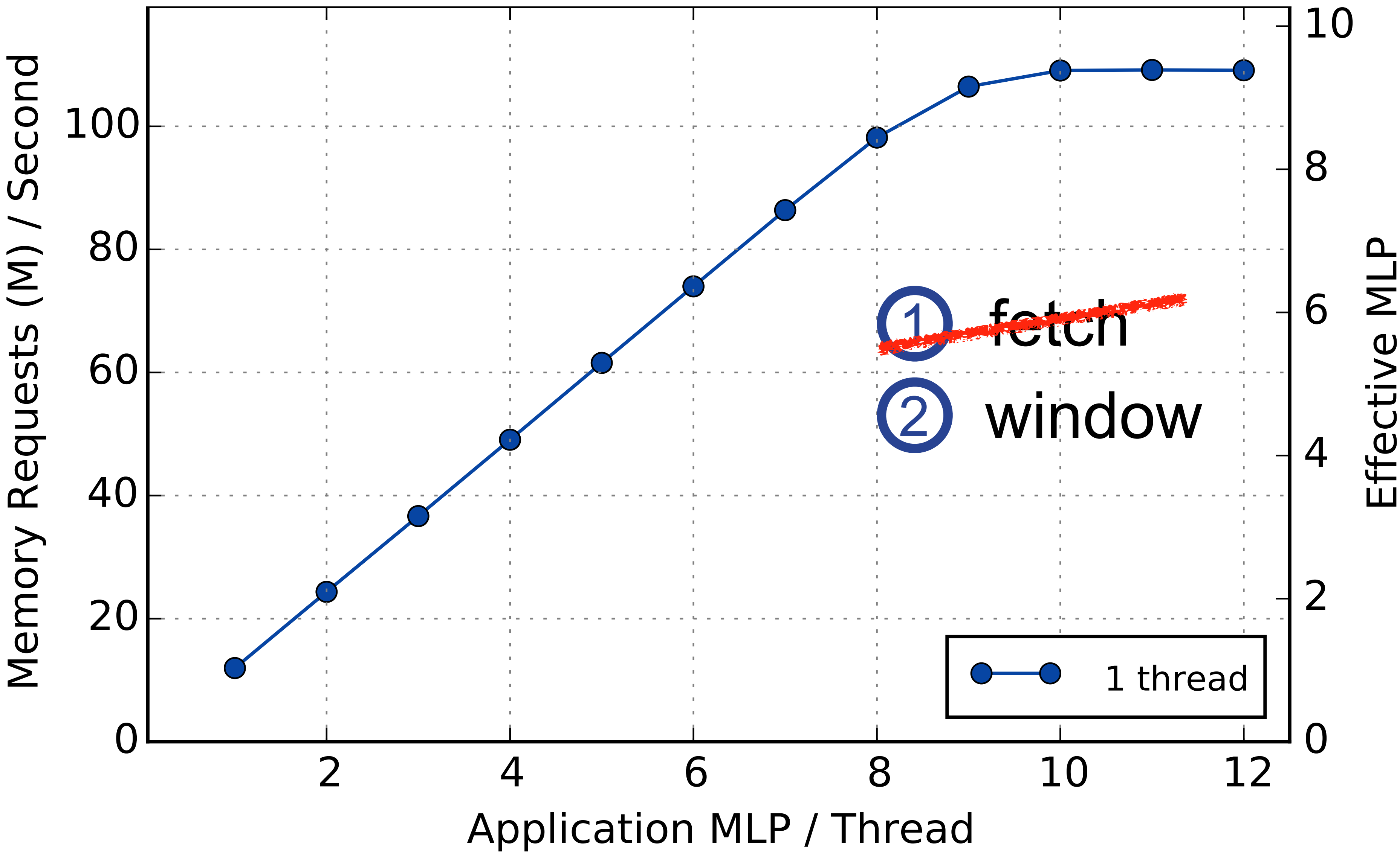


1 core

Pointer Chasing Microbenchmark with varying number of parallel pointer chases

Single-Core Memory Bandwidth

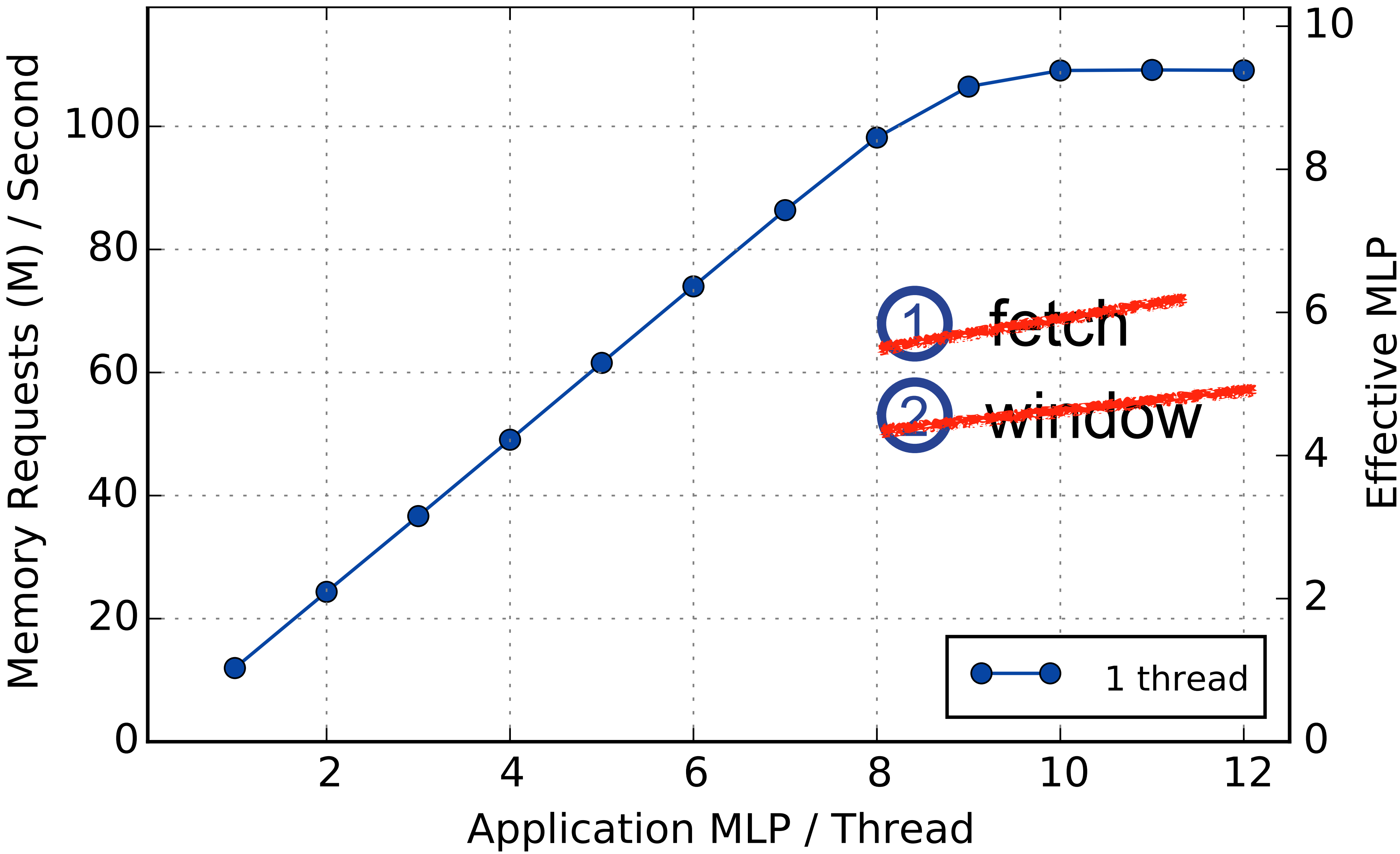
ASPIRE UC Berkeley



1 core

Pointer Chasing Microbenchmark with varying number of parallel pointer chases

Single-Core Memory Bandwidth

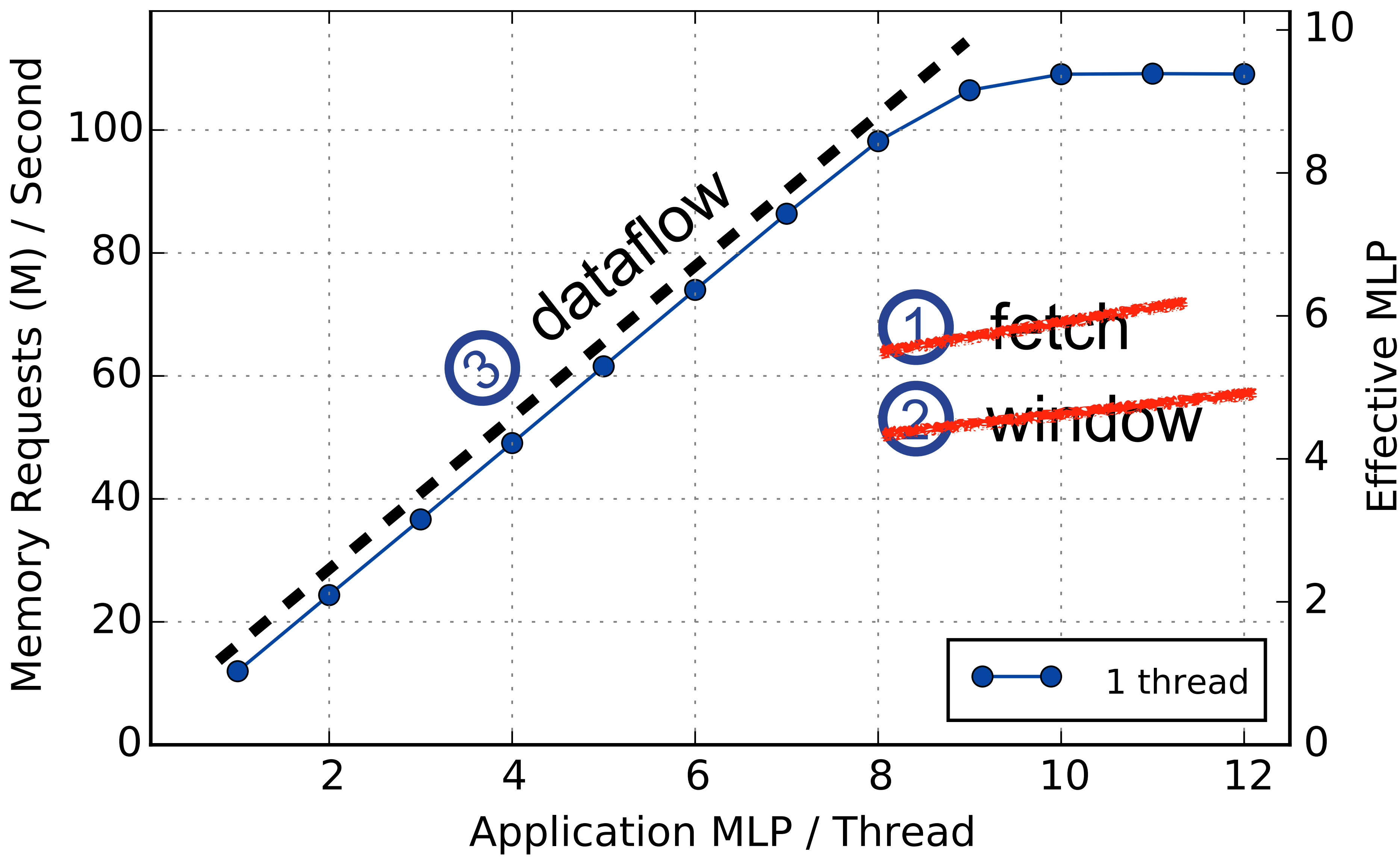


1 core

Pointer Chasing Microbenchmark with varying number of parallel pointer chases

Single-Core Memory Bandwidth

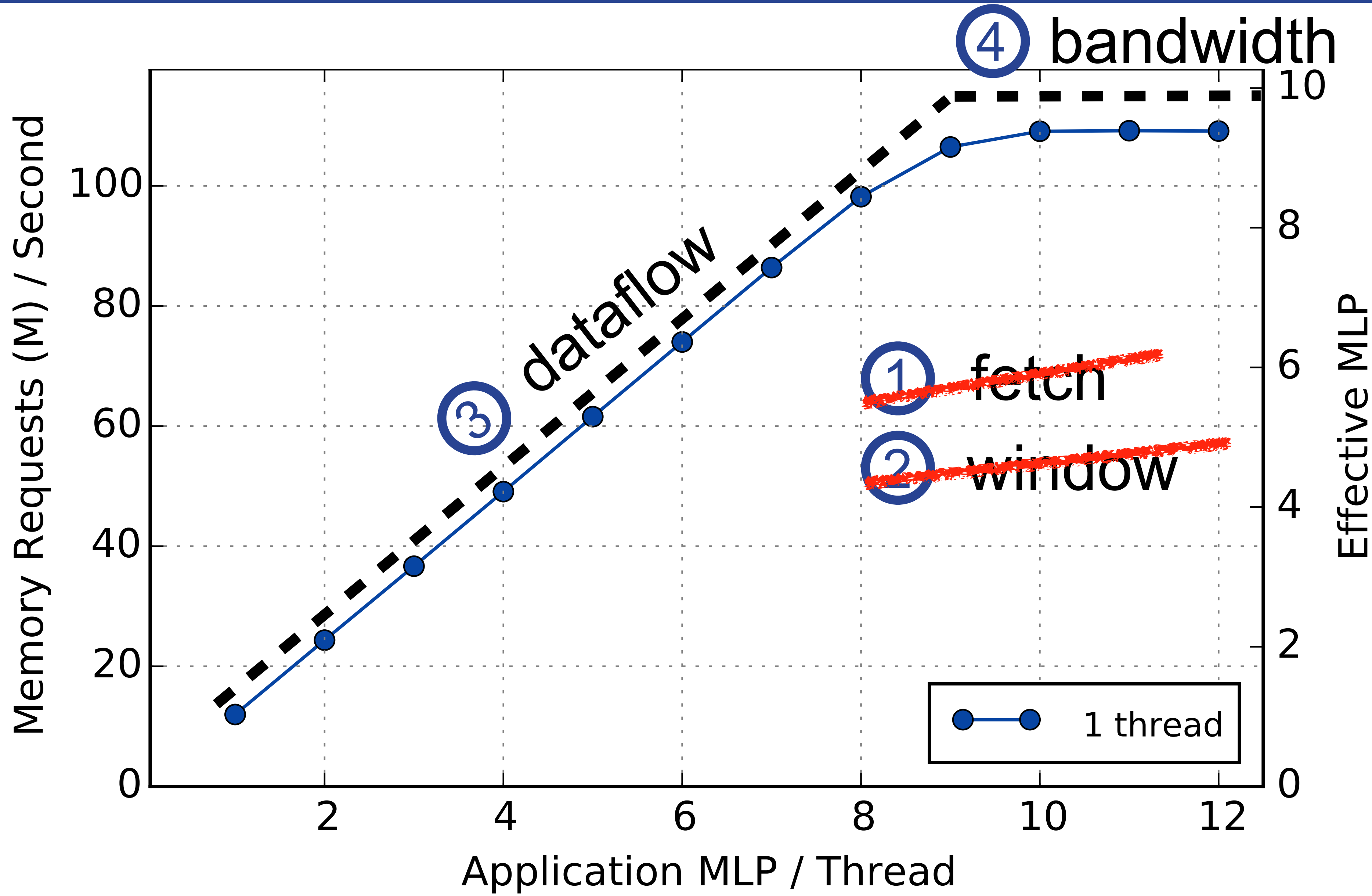
ASPIRE
UC Berkeley



1 core

Pointer Chasing Microbenchmark with varying number of parallel pointer chases

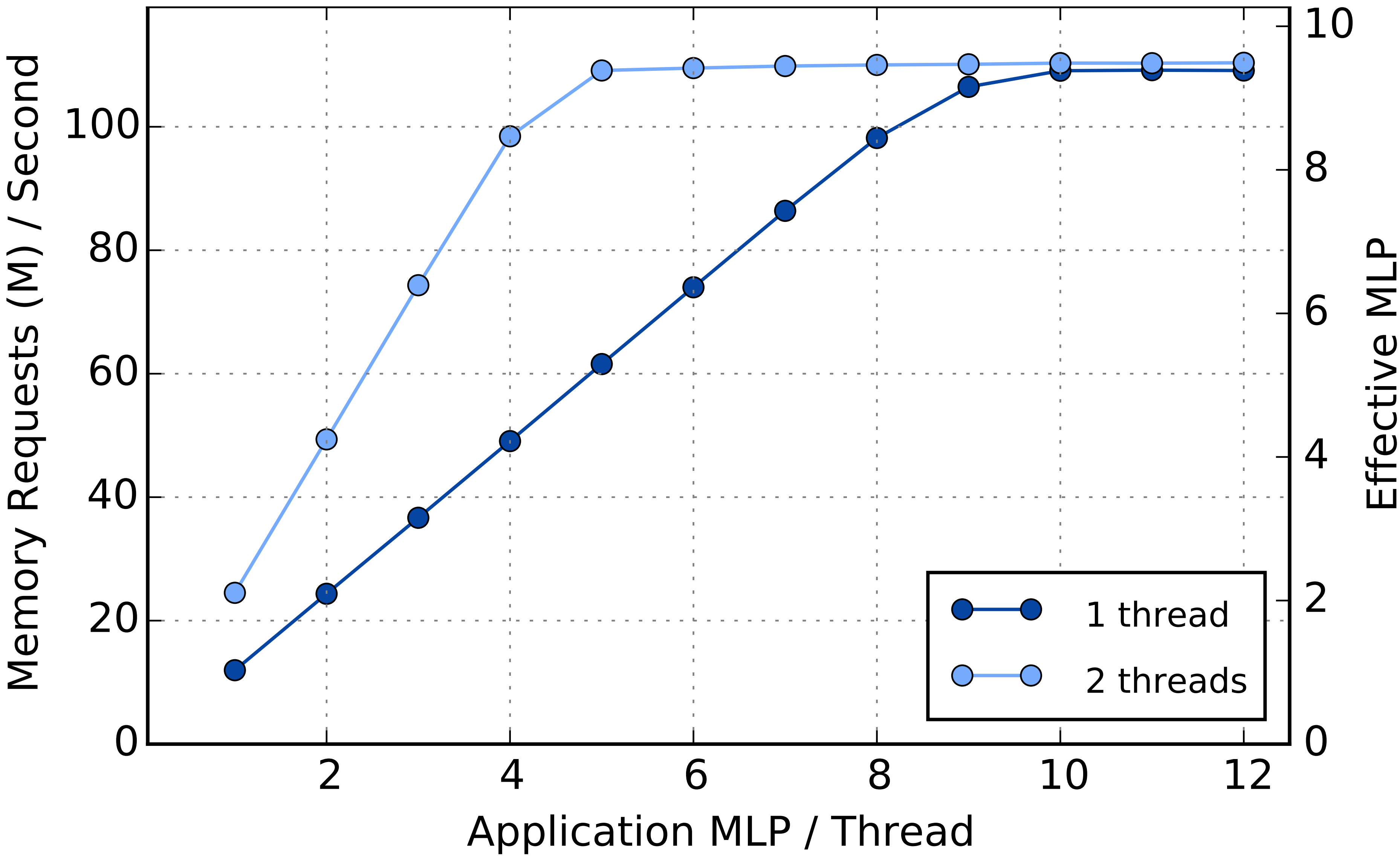
Single-Core Memory Bandwidth



1 core

Pointer Chasing Microbenchmark with varying number of parallel pointer chases

Single-Core Memory Bandwidth



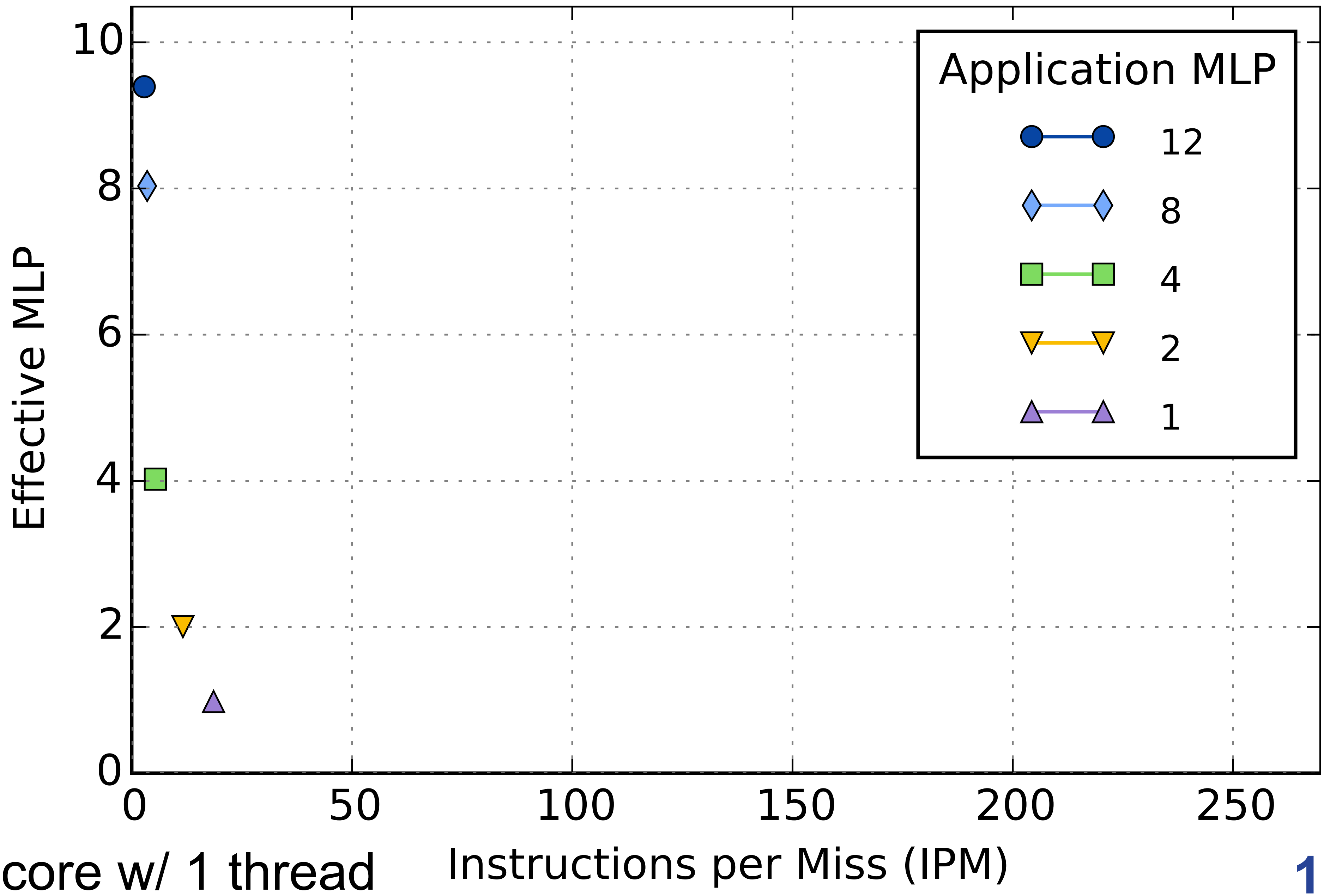
1 core

Pointer Chasing Microbenchmark with varying number of parallel pointer chases

Outline

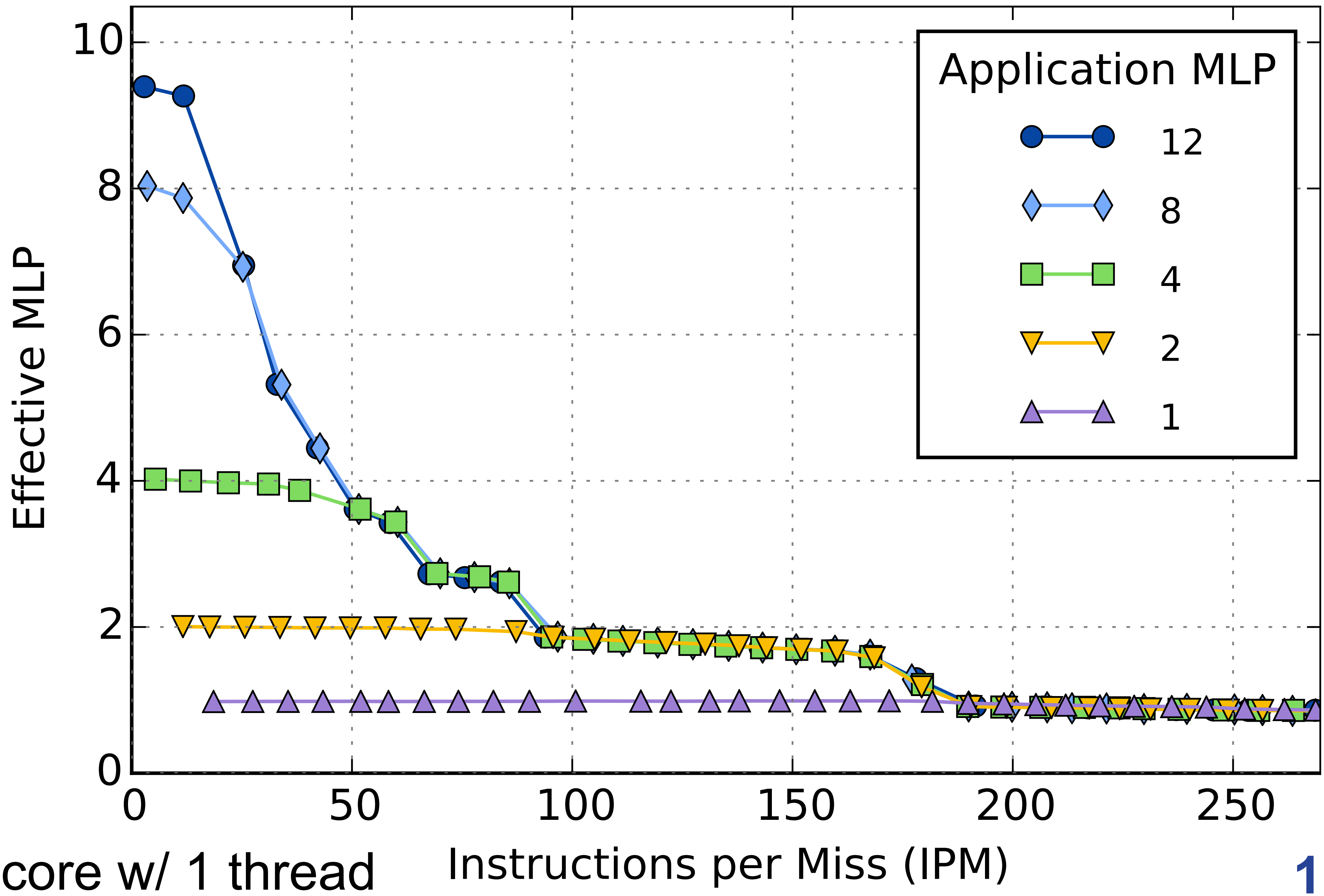
- Methodology
- Platform Memory Bandwidth Availability
- **Single-core Results**
- Parallel Results
- GAP Benchmark Suite
- Conclusion

Importance of Window Size



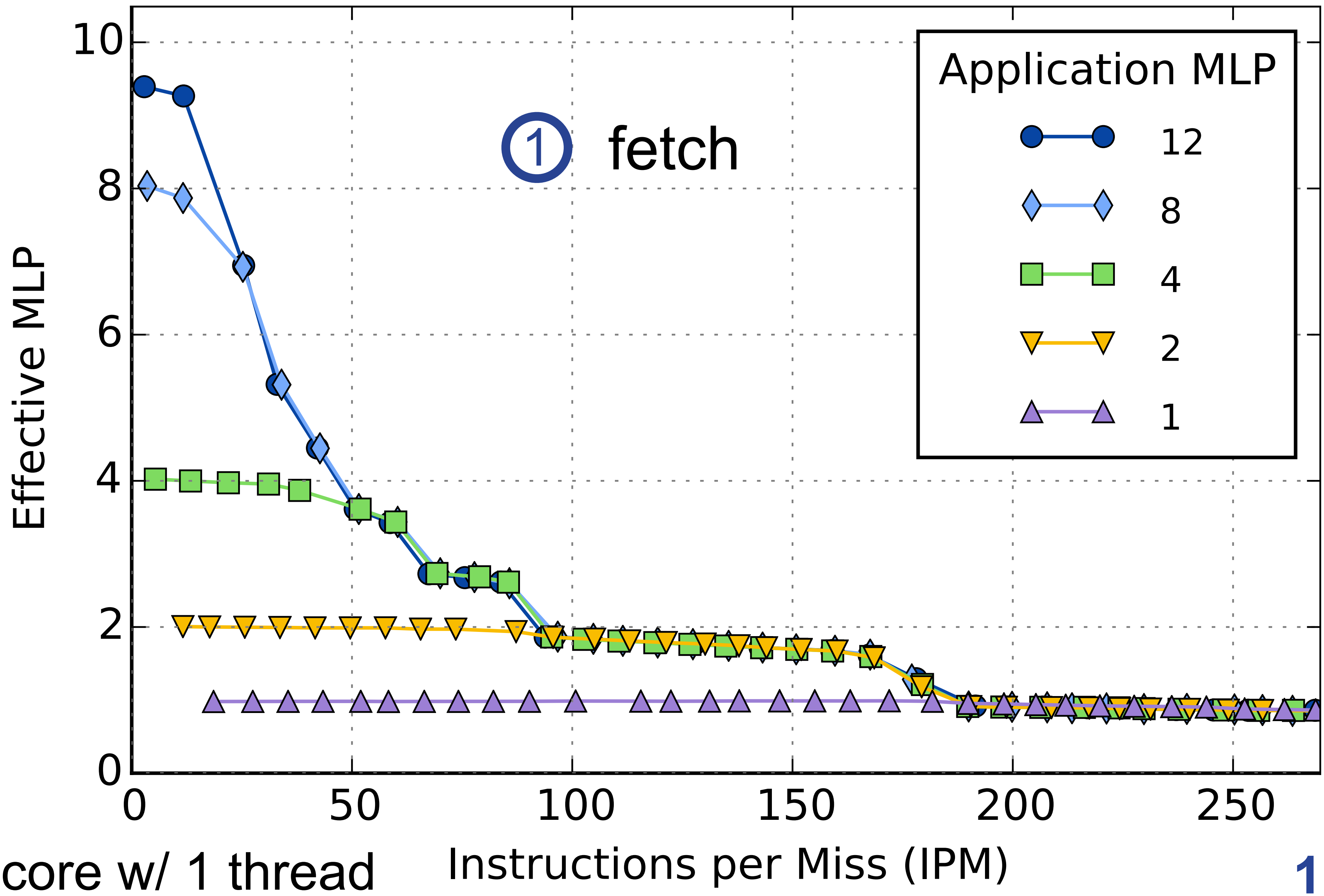
1 core w/ 1 thread

Importance of Window Size



1 core w/ 1 thread

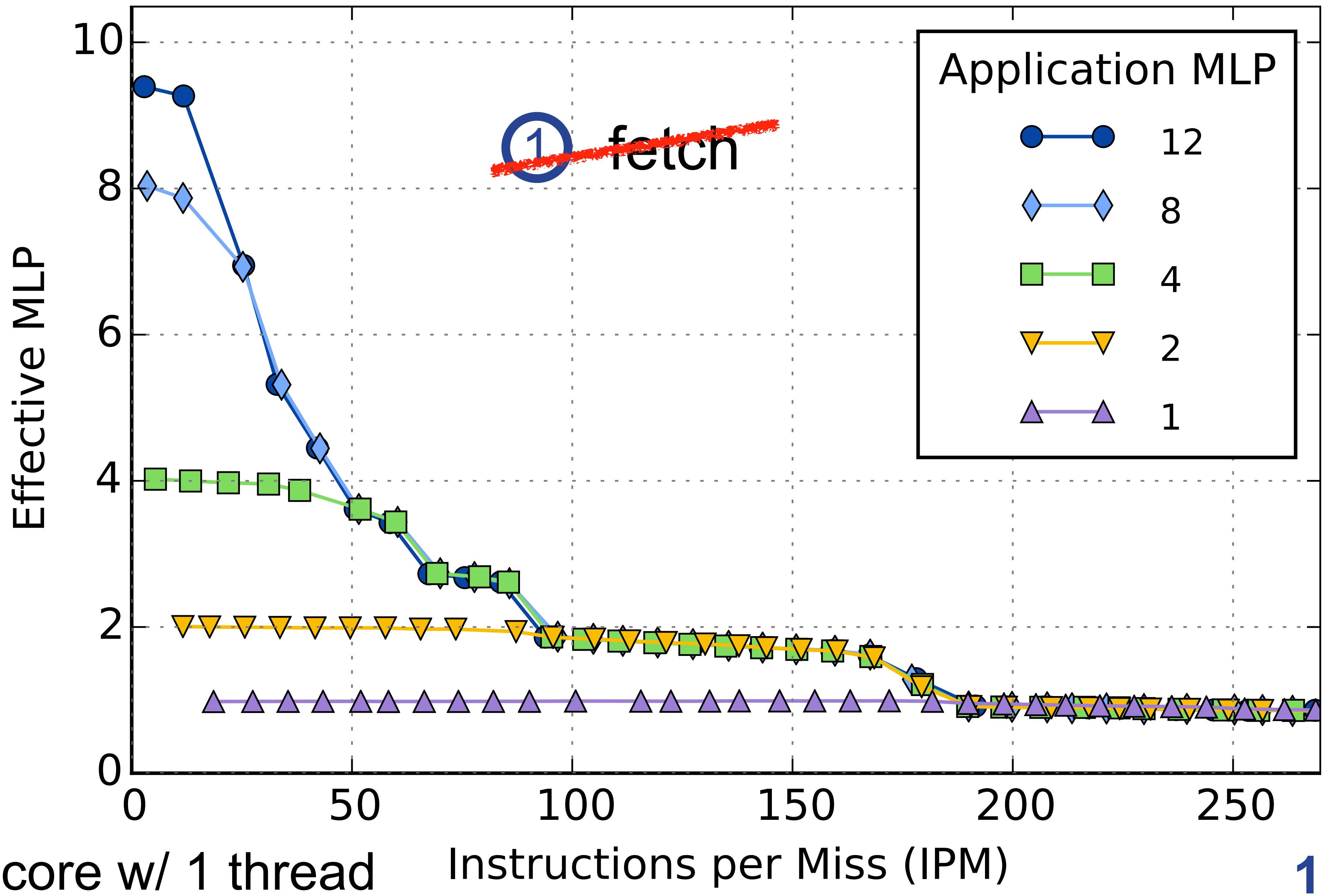
Importance of Window Size



1 core w/ 1 thread

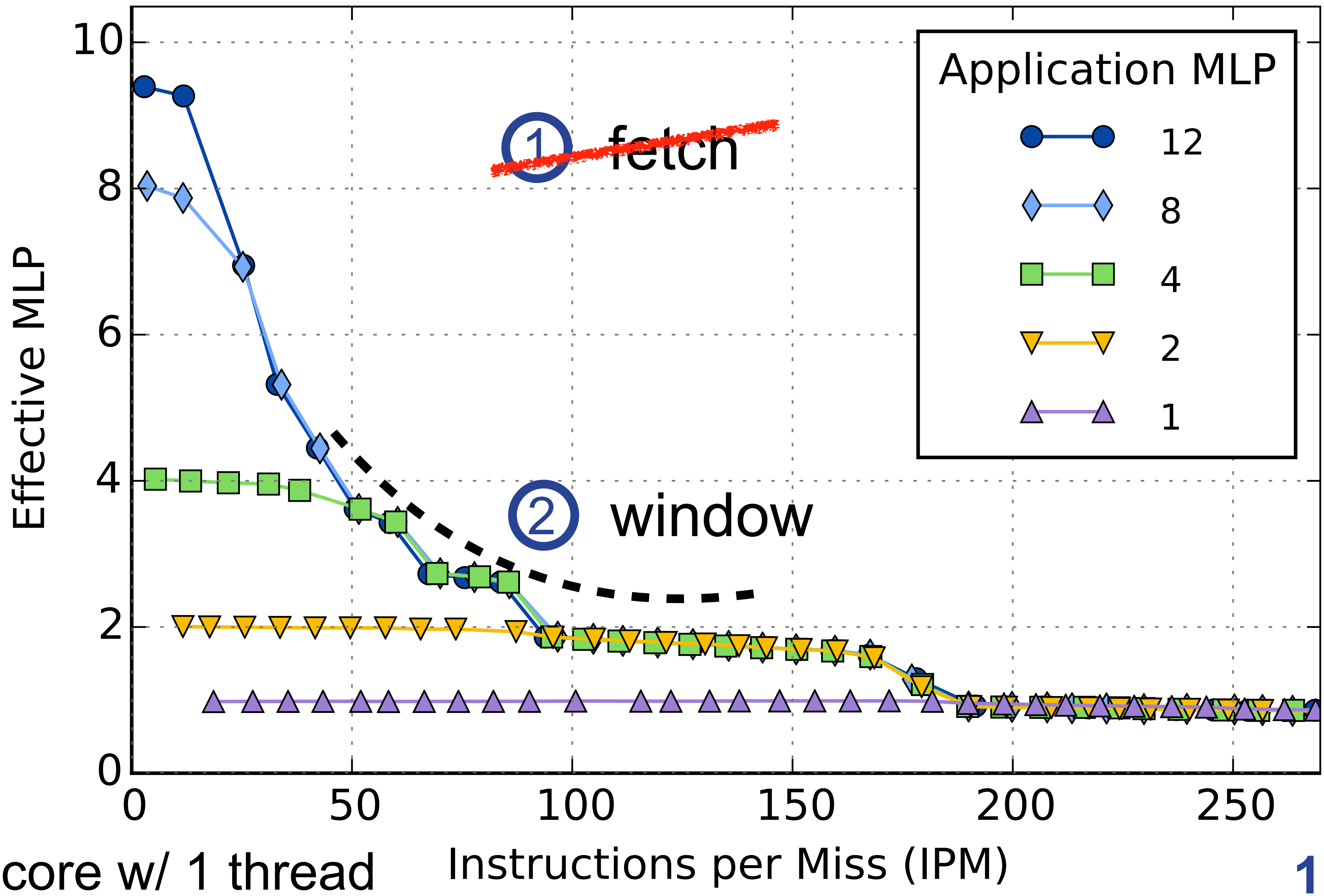
Importance of Window Size

ASPIRE
UC Berkeley



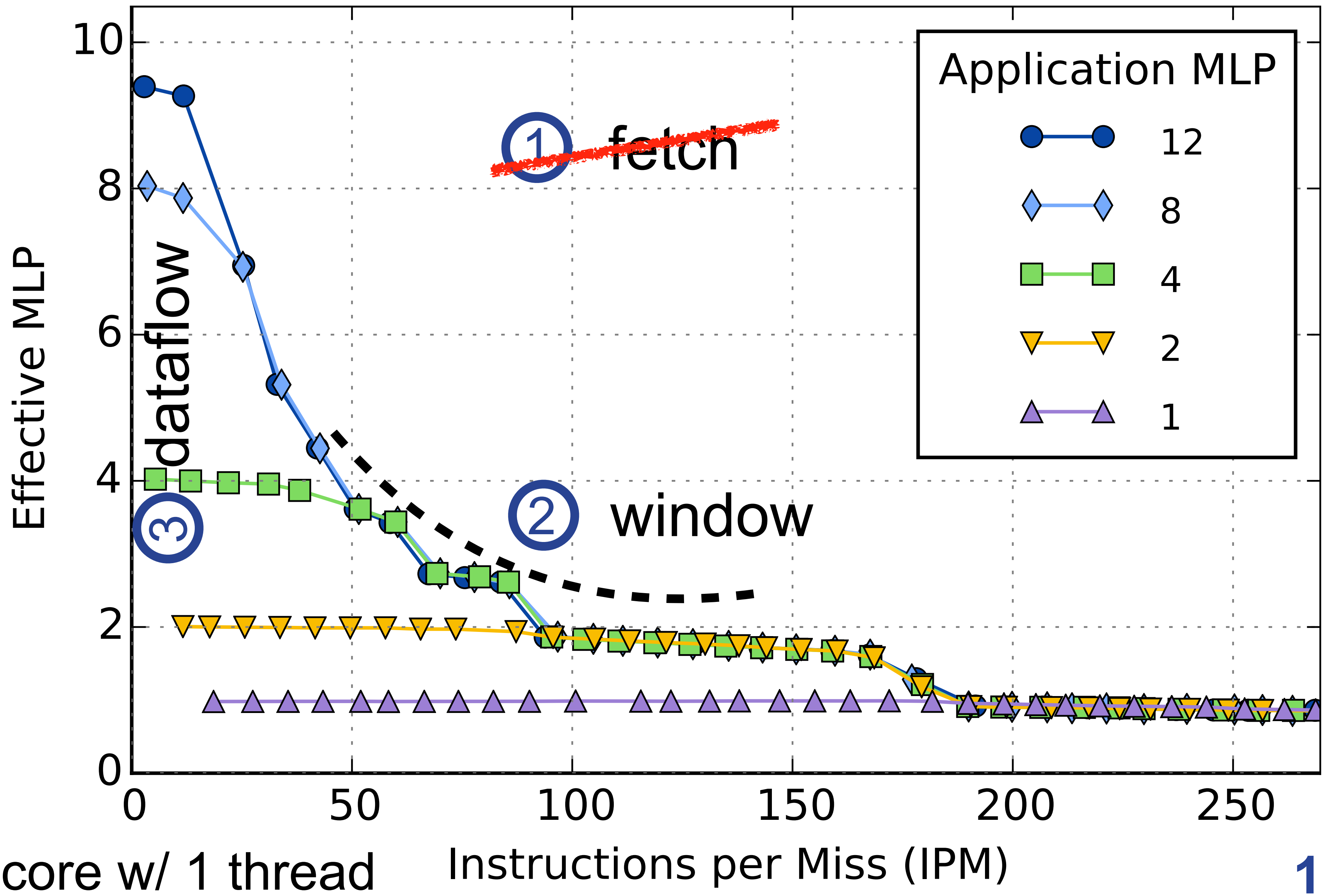
1 core w/ 1 thread

Importance of Window Size



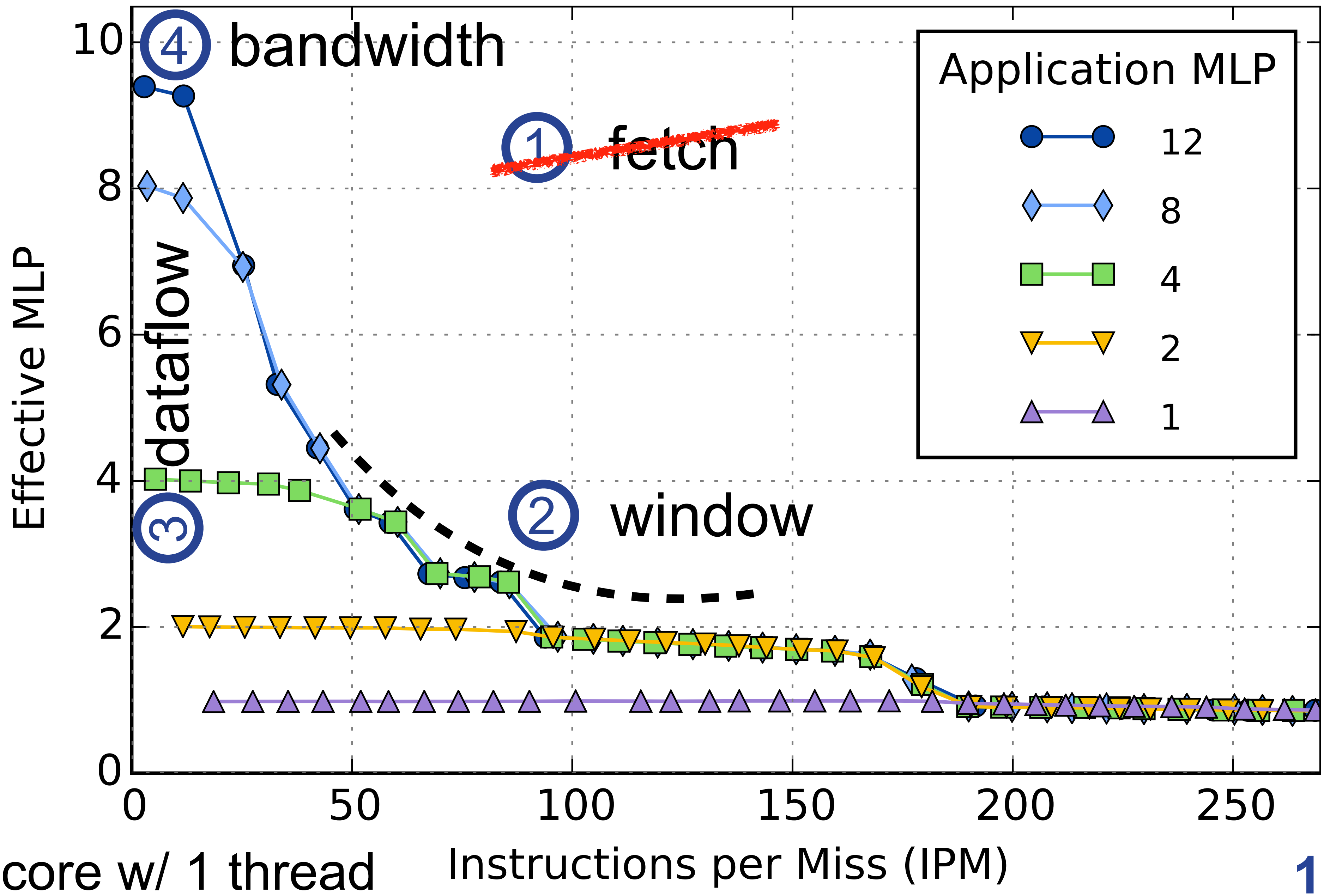
1 core w/ 1 thread

Importance of Window Size



1 core w/ 1 thread

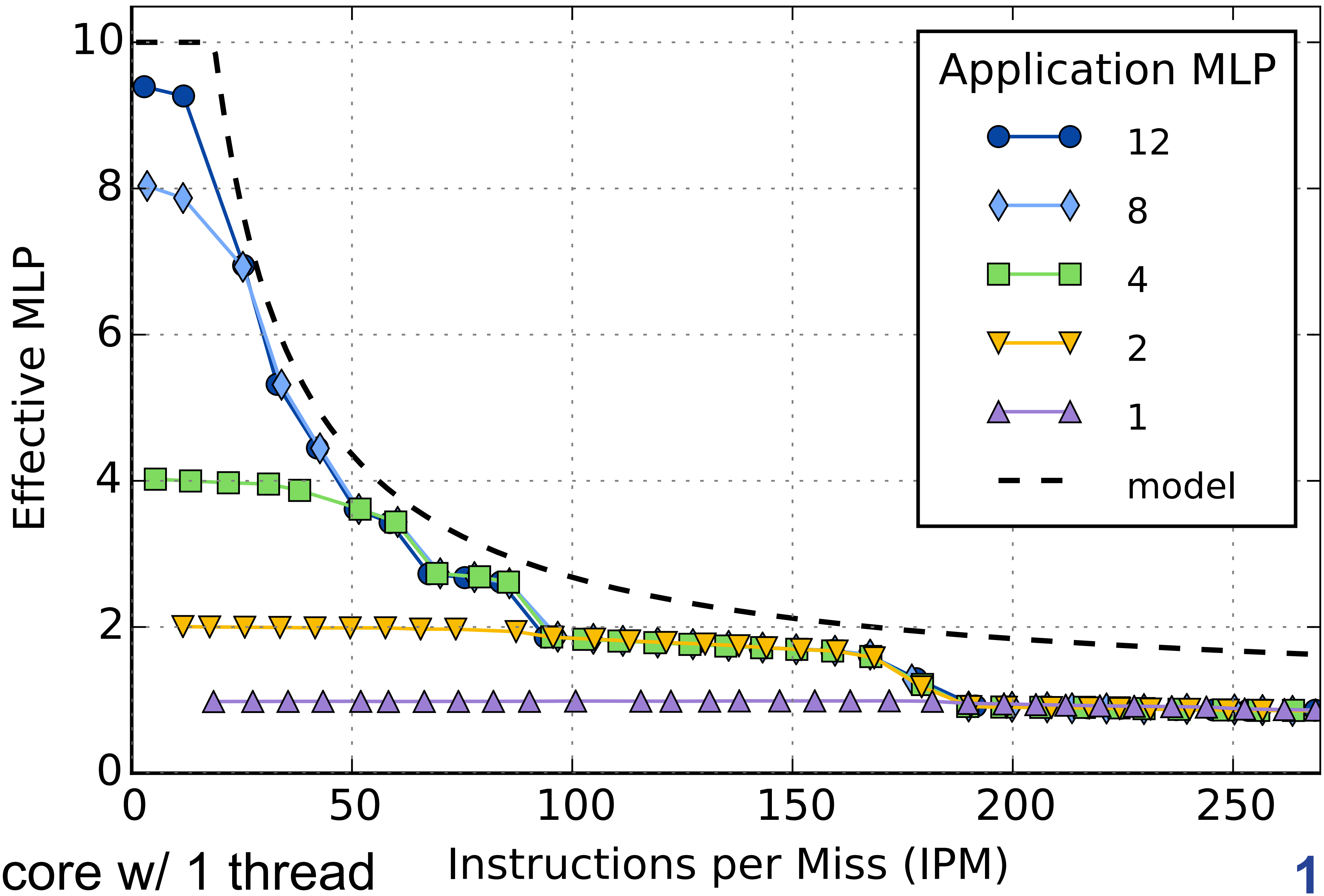
Importance of Window Size



1 core w/ 1 thread

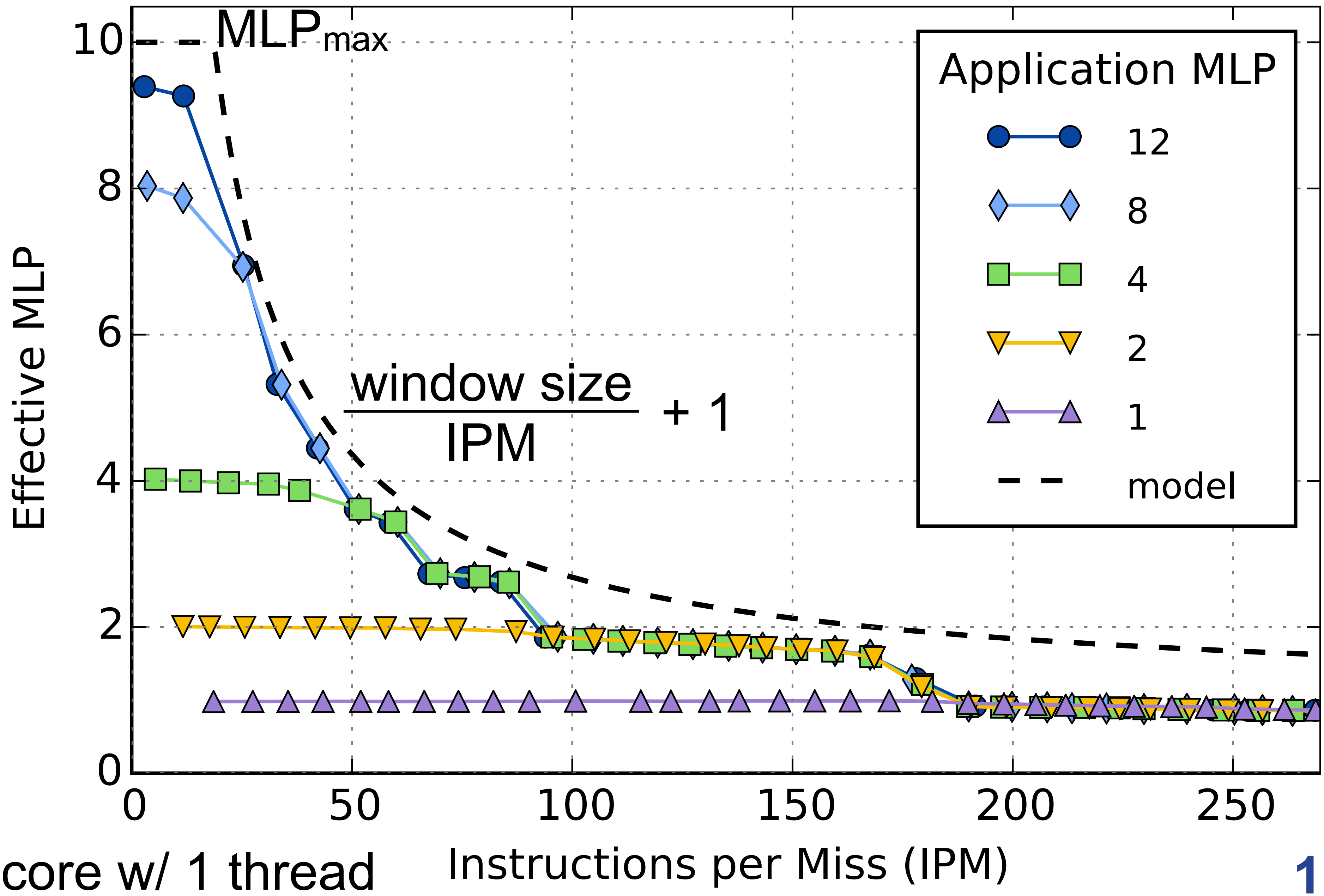
Importance of Window Size

ASPIRE UC Berkeley



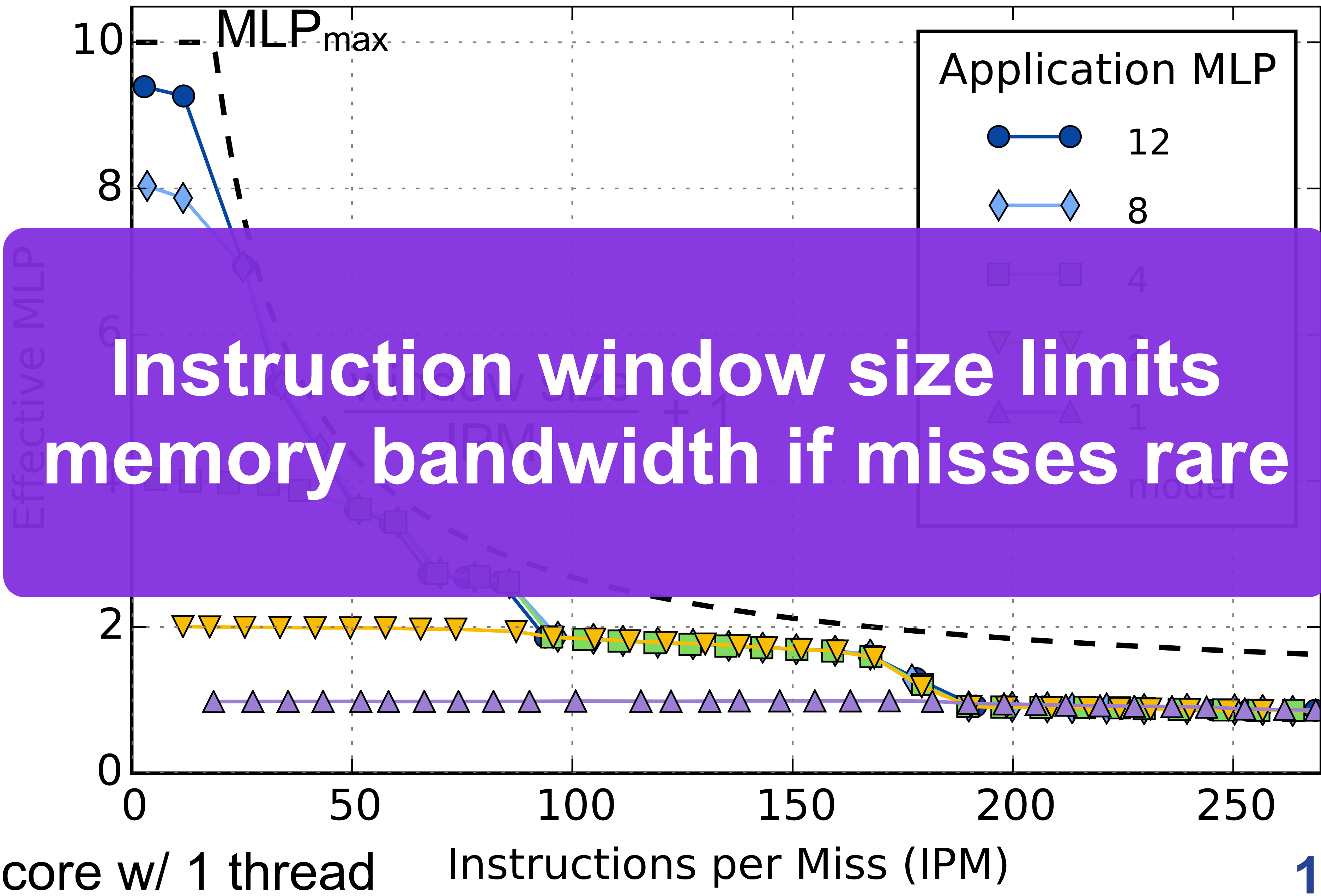
1 core w/ 1 thread

Importance of Window Size



1 core w/ 1 thread

Importance of Window Size



Instruction window size limits memory bandwidth if misses rare

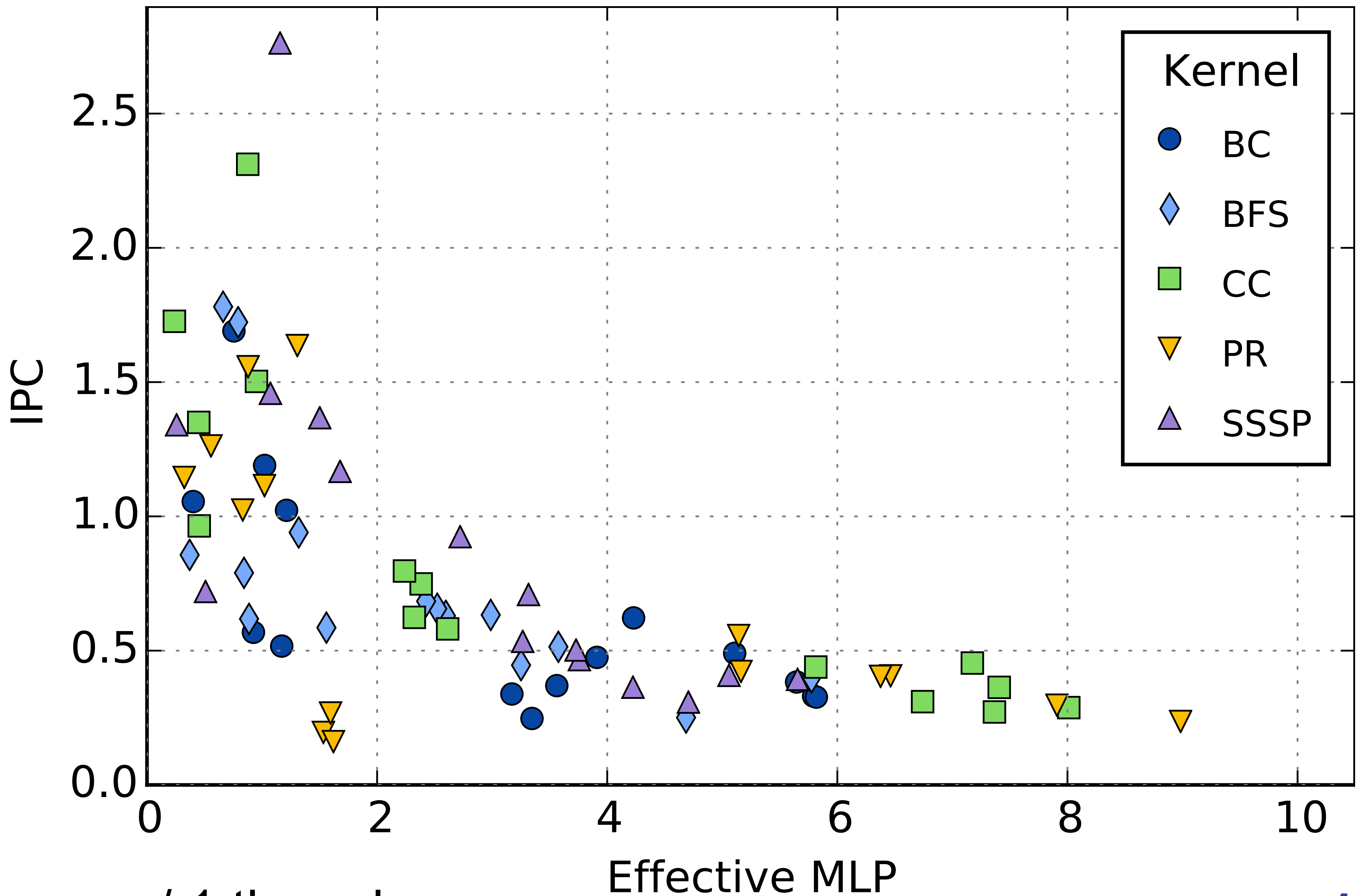
1 core w/ 1 thread

Biggest Influence on Single-Core?

1 core w/ 1 thread

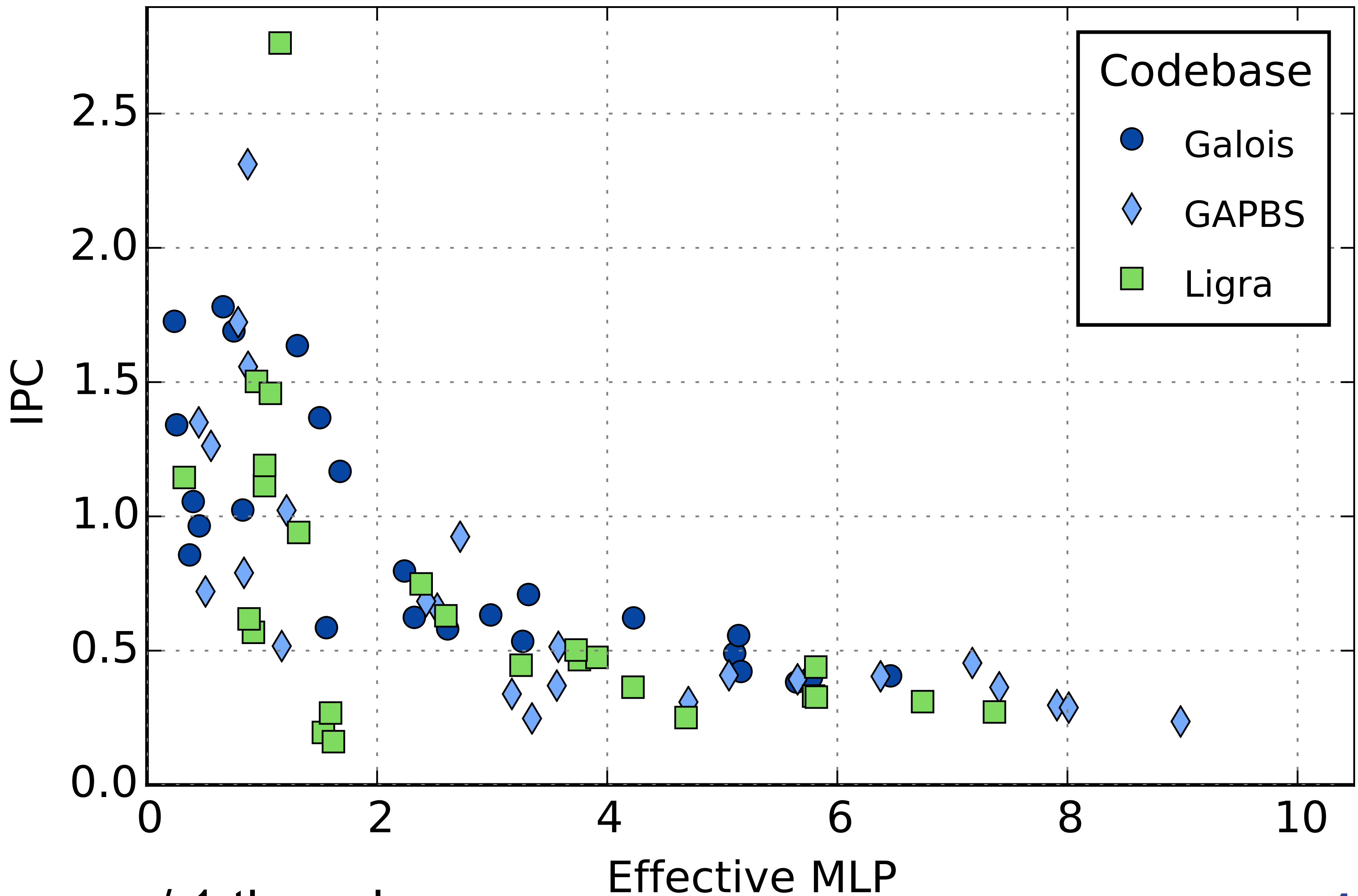
Biggest Influence on Single-Core?

ASPIRE
UC Berkeley



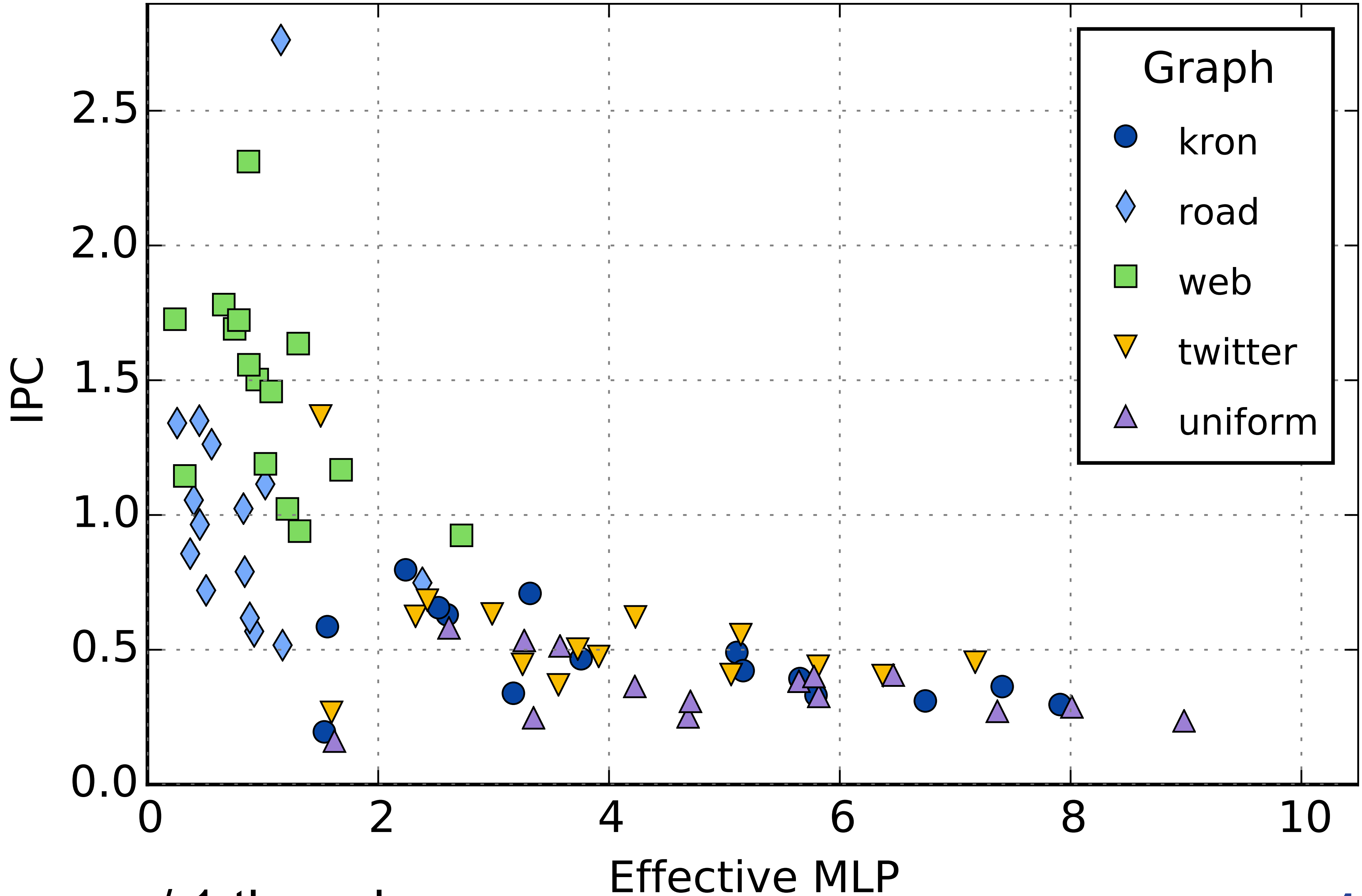
1 core w/ 1 thread

Biggest Influence on Single-Core?



1 core w/ 1 thread

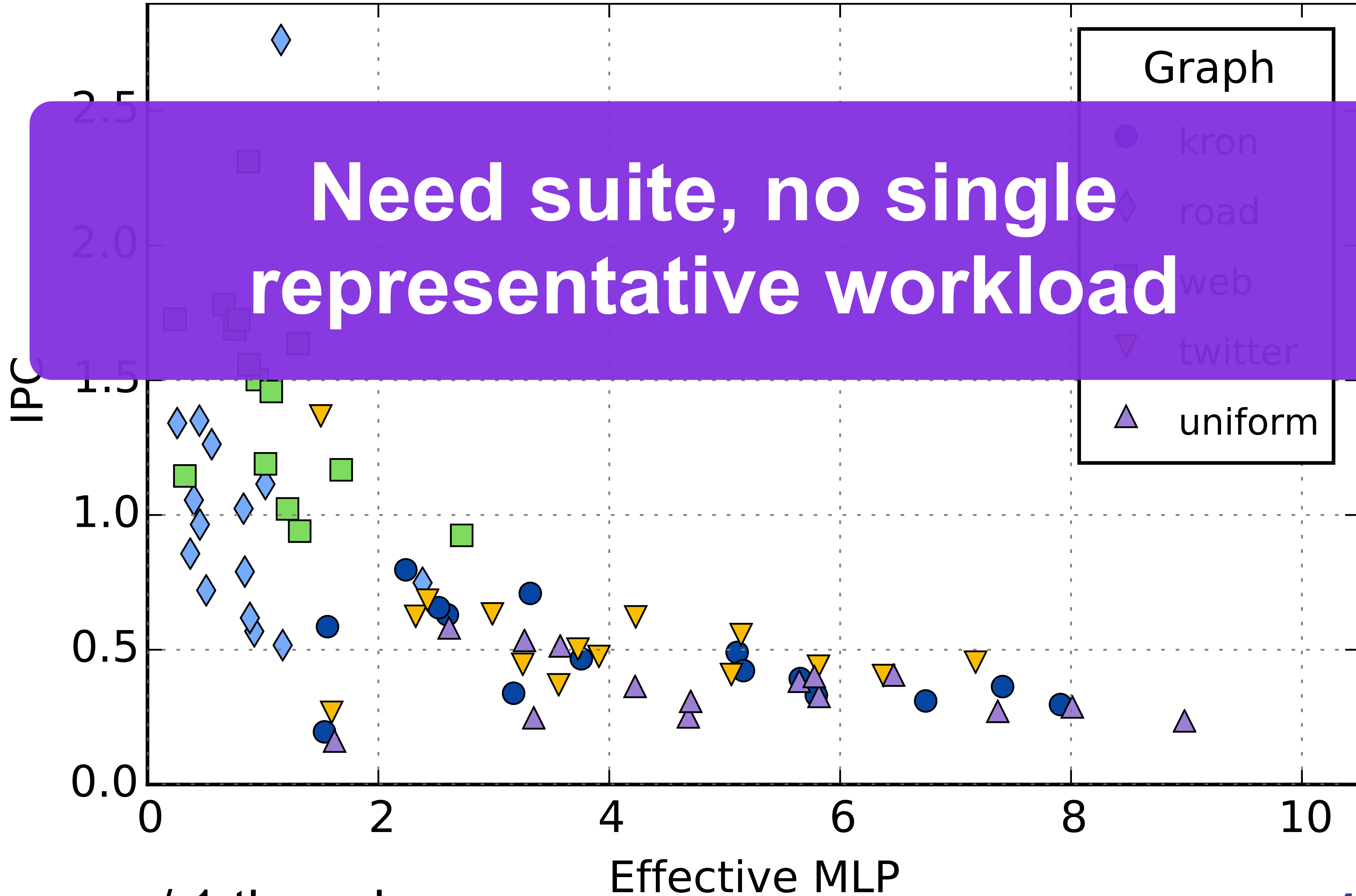
Biggest Influence on Single-Core?



1 core w/ 1 thread

Biggest Influence on Single-Core?

ASPIRE
UC Berkeley



Need suite, no single representative workload

1 core w/ 1 thread

Biggest Influence on Single-Core?

Need suite, no single representative workload

Only few workloads near memory bandwidth limit

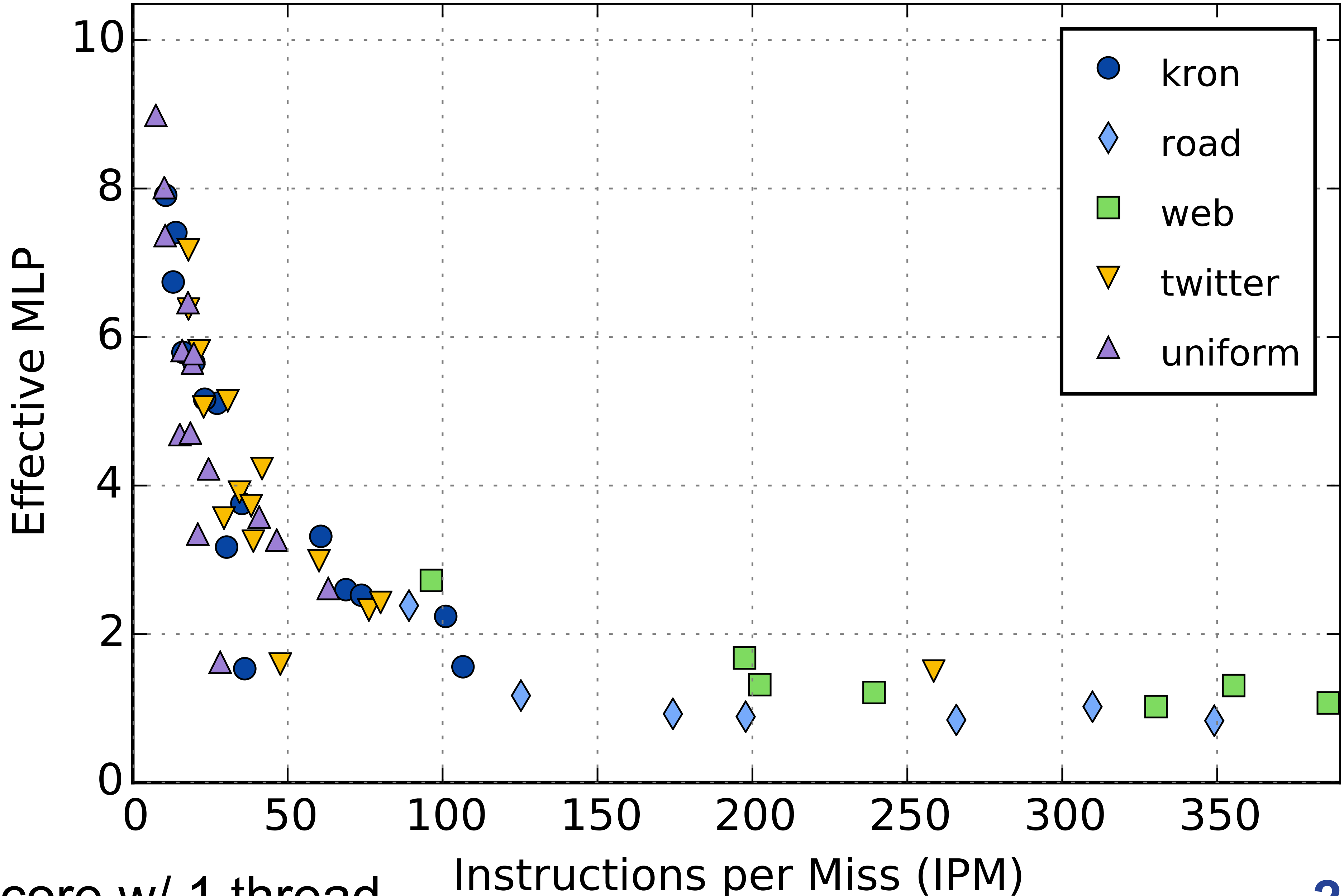
1 core w/ 1 thread

Instruction Window Limits BW

1 core w/ 1 thread

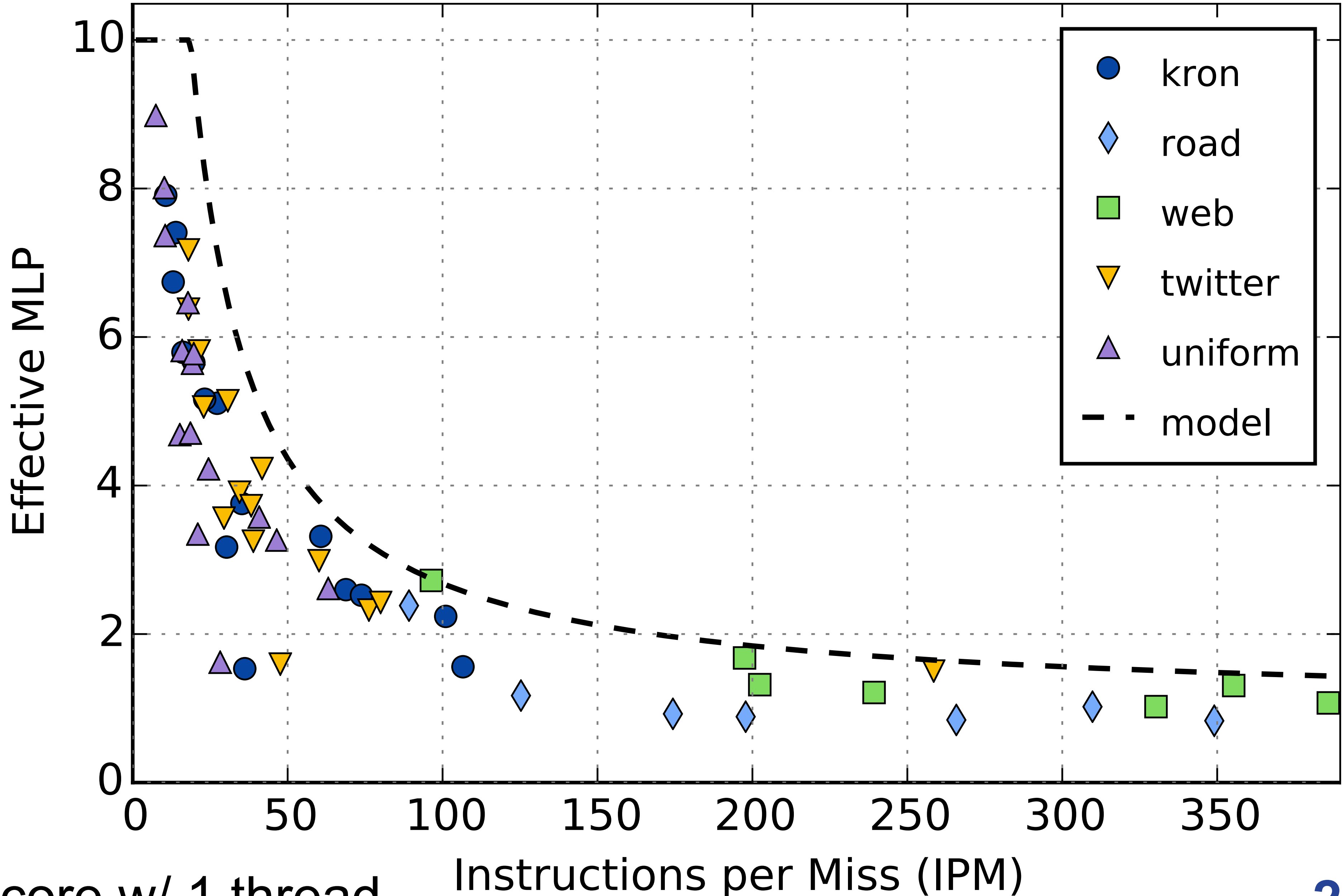
Instruction Window Limits BW

ASPIRE
UC Berkeley



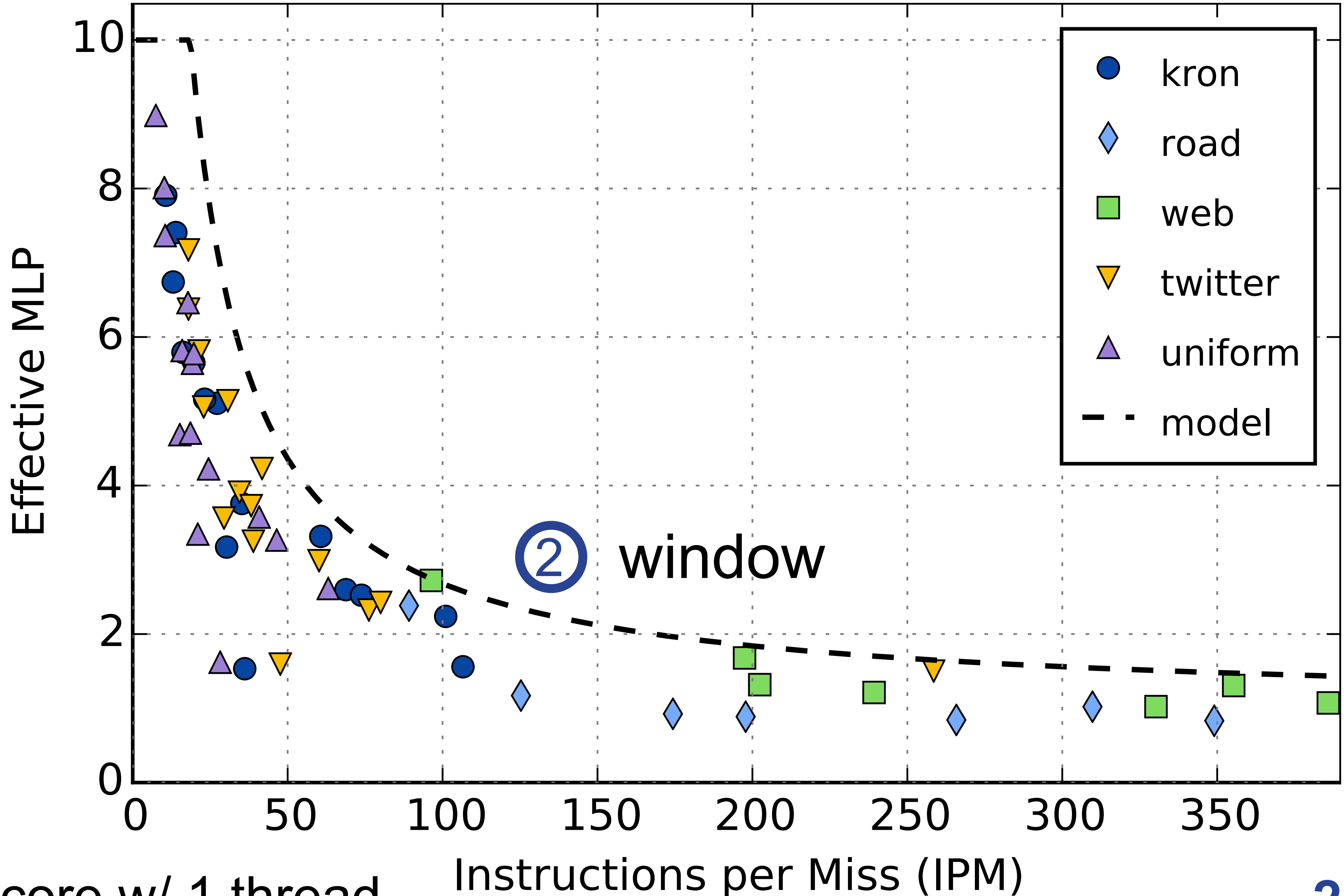
1 core w/ 1 thread

Instruction Window Limits BW



1 core w/ 1 thread

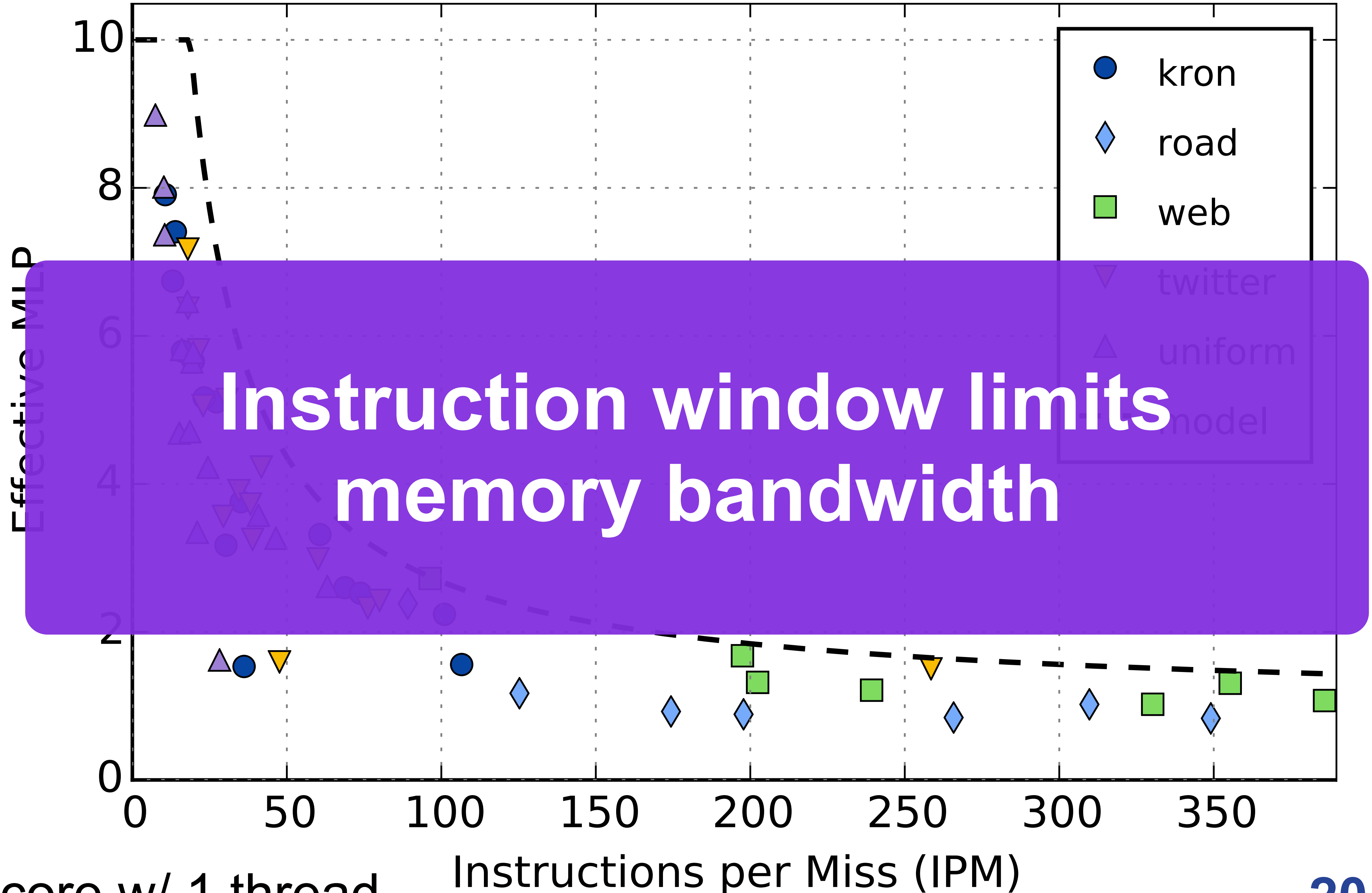
Instruction Window Limits BW



1 core w/ 1 thread

Instruction Window Limits BW

ASPIRE UC Berkeley



1 core w/ 1 thread

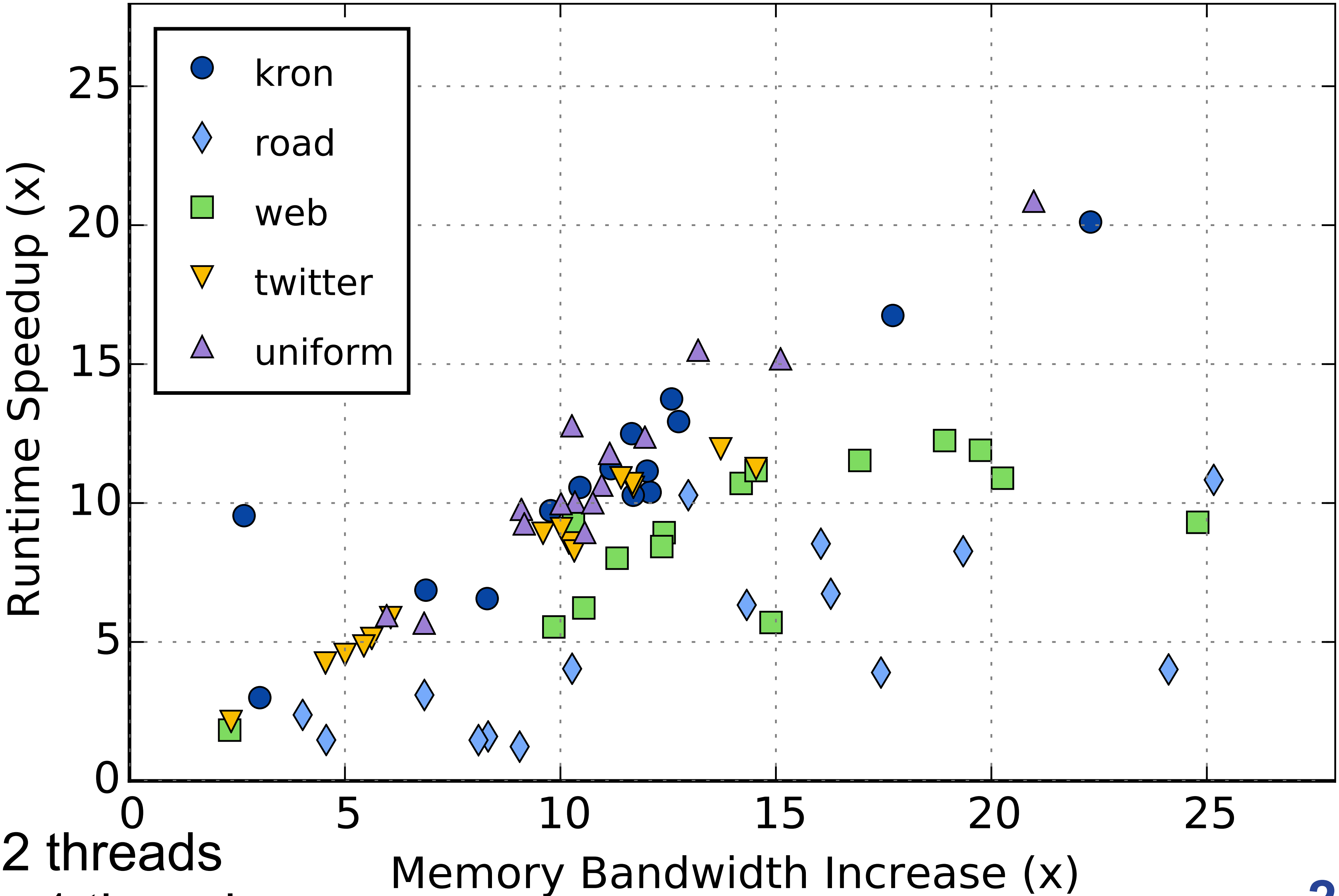
Outline

ASPIRE
UC Berkeley

- Methodology
- Platform Memory Bandwidth Availability
- Single-core Results
- **Parallel Results**
- GAP Benchmark Suite
- Conclusion

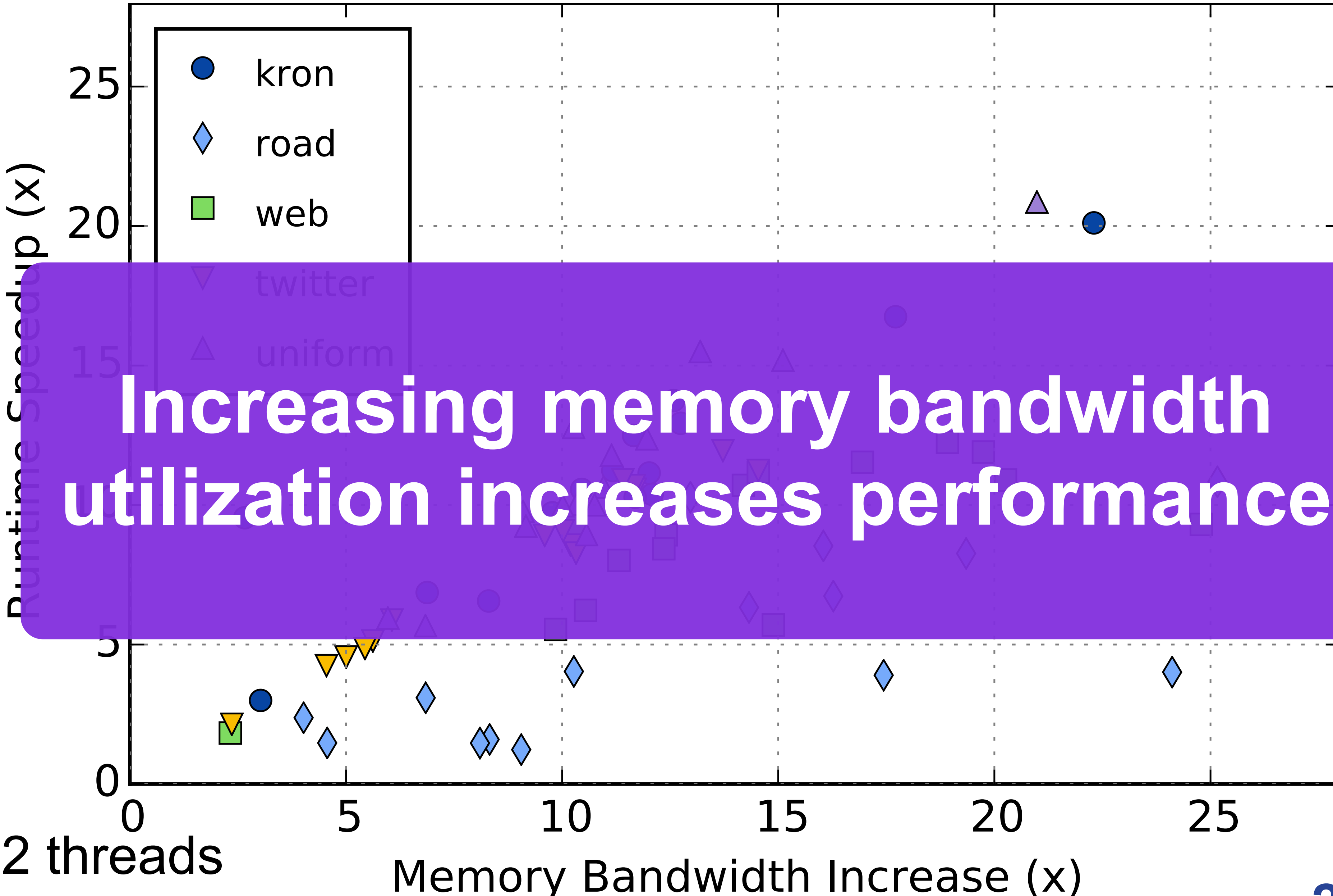
Memory Bandwidth ~ Performance

ASPIRE UC Berkeley



32 threads
vs. 1 thread

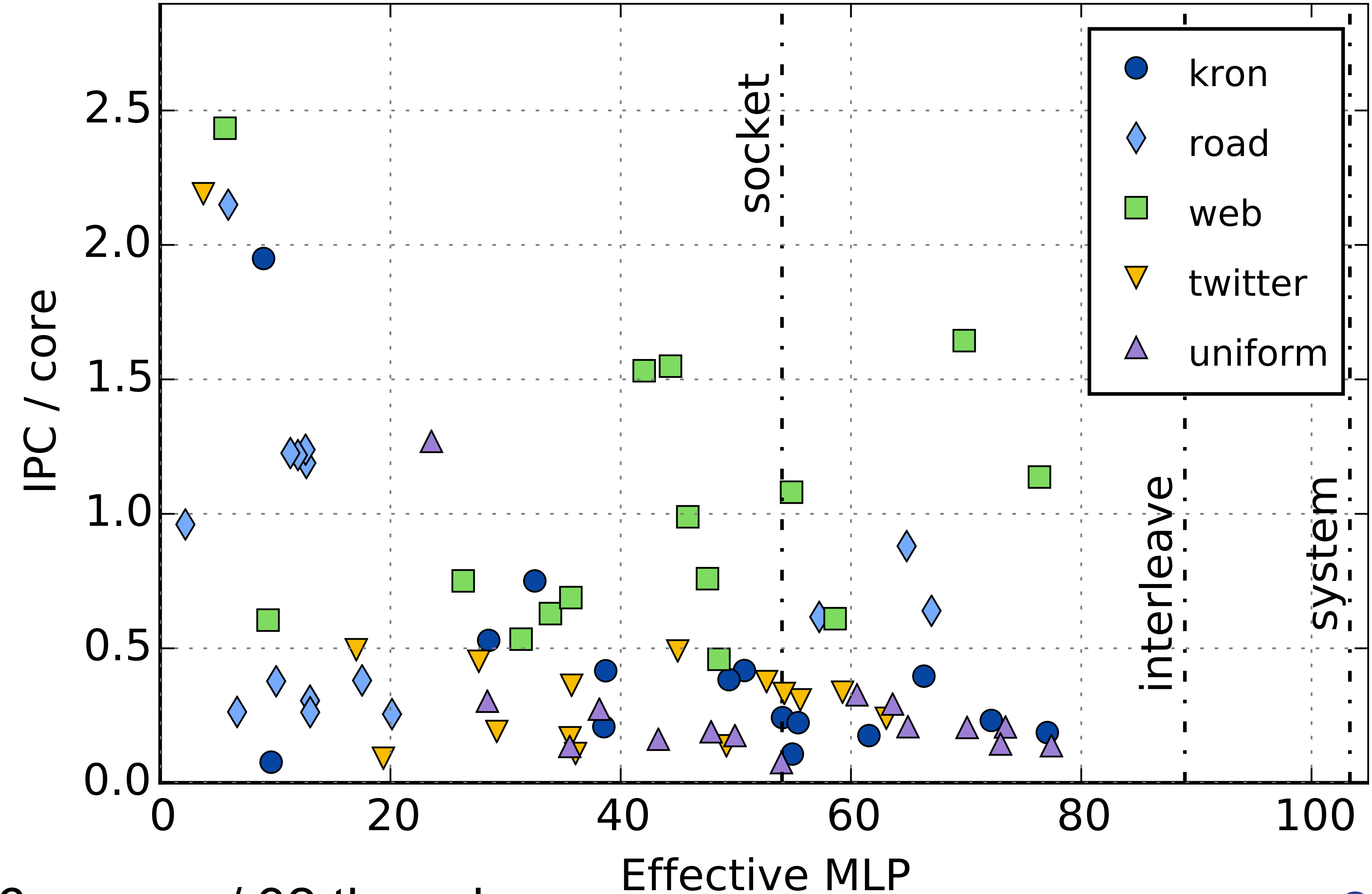
Memory Bandwidth ~ Performance



Increasing memory bandwidth utilization increases performance

32 threads
vs. 1 thread

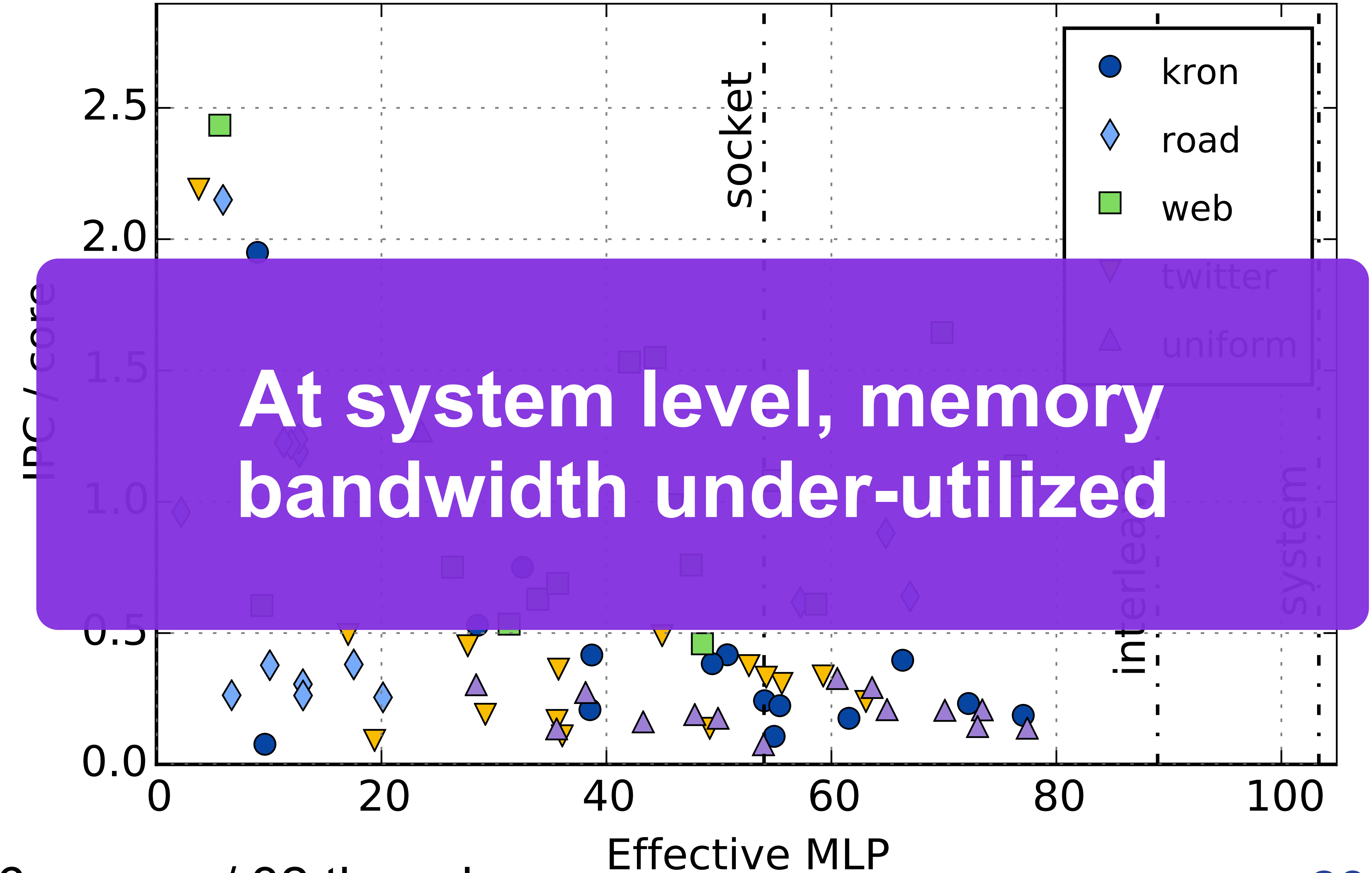
Parallel Utilization



16 cores w/ 32 threads

Parallel Utilization

ASPIRE UC Berkeley



At system level, memory bandwidth under-utilized

16 cores w/ 32 threads

Multithreading Opportunity

Multithreading Opportunity

1 fetch

2 window

3 dataflow

4 bandwidth

Multithreading Opportunity

ASPIRE
UC Berkeley

① fetch

② ~~window~~ same hardware (shared)

③ dataflow

④ bandwidth

Multithreading Opportunity

ASPIRE
UC Berkeley

1 fetch

2 ~~window~~ same hardware (shared)

3 dataflow

4 ~~bandwidth~~ same hardware (shared)

Multithreading Opportunity

ASPIRE UC Berkeley

1 fetch	+ fewer instructions in flight
2 ~~window~~	same hardware (shared)
3 dataflow	
4 ~~bandwidth~~	same hardware (shared)

Multithreading Opportunity

ASPIRE
UC Berkeley

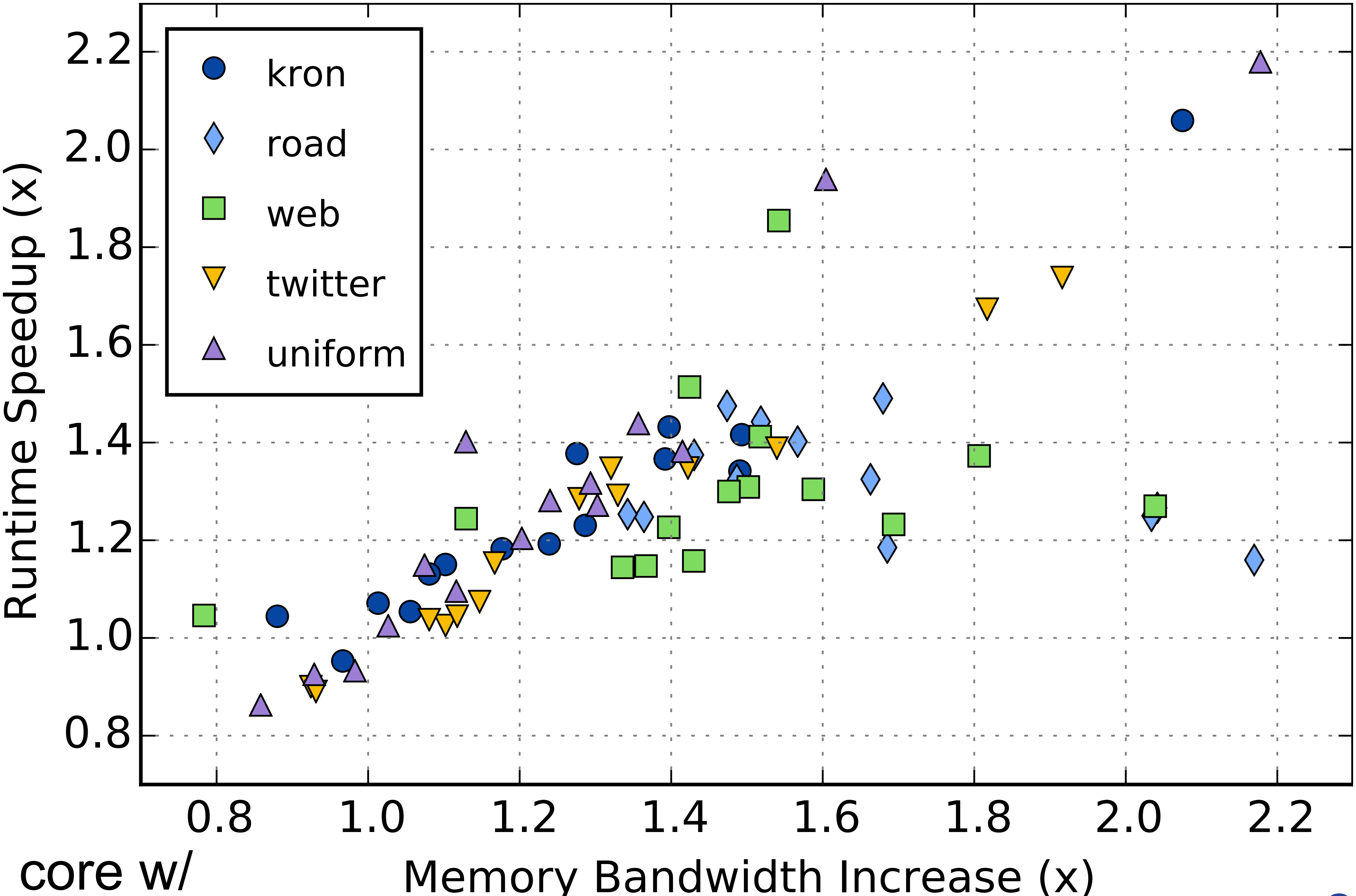
1 fetch + fewer instructions in flight

2 ~~window~~ same hardware (shared)

3 dataflow ++ more application MLP

4 ~~bandwidth~~ same hardware (shared)

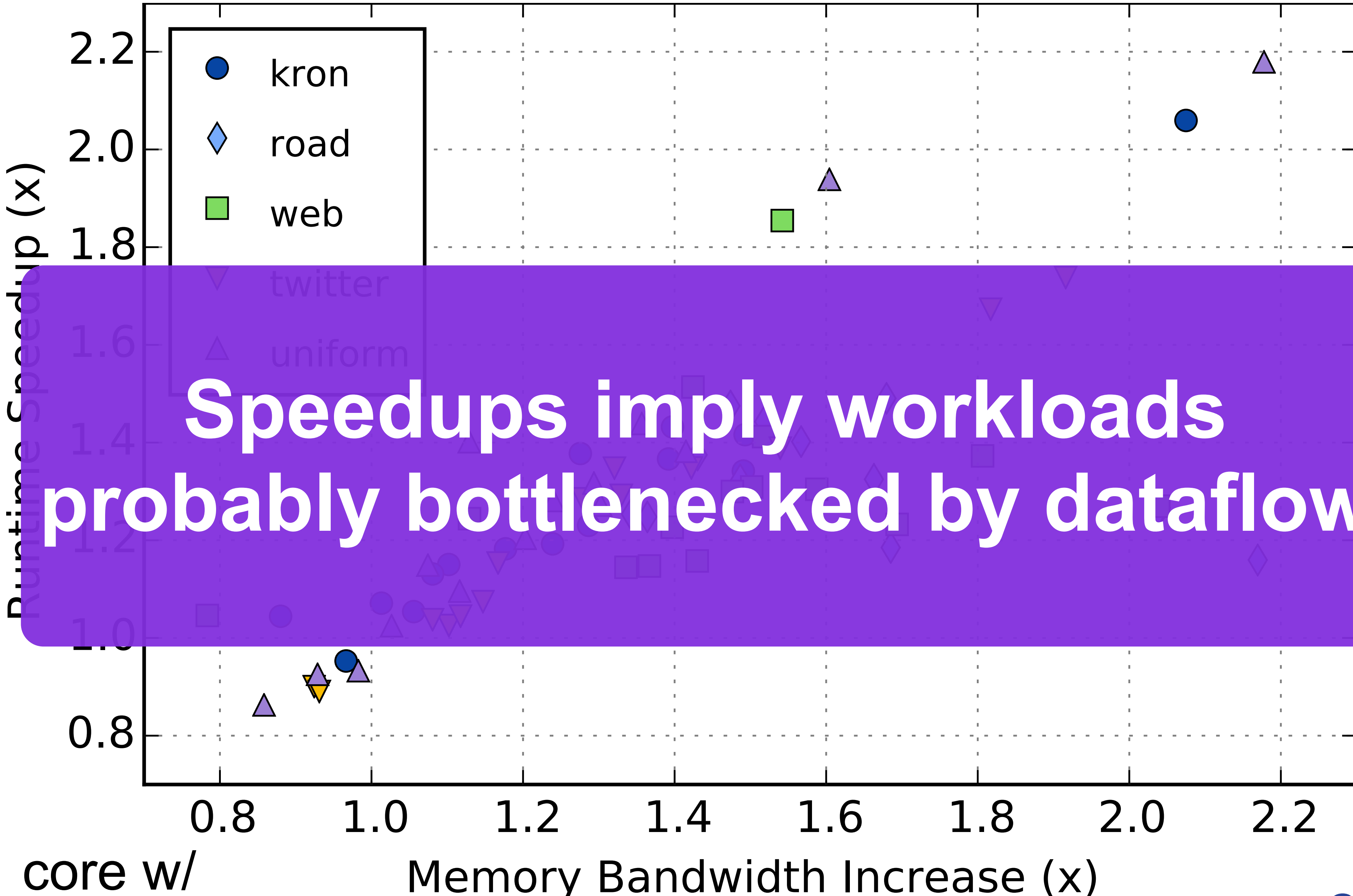
Multithreading Increases Bandwidth



1 core w/
1-2 threads

Multithreading Increases Bandwidth

ASPIRE
UC Berkeley



Speedups imply workloads probably bottlenecked by dataflow

1 core w/
1-2 threads

Conclusions

Conclusions

ASPIRE
UC Berkeley

- Most graph workloads do not utilize a large fraction of memory bandwidth

Conclusions

ASPIRE UC Berkeley

- Most graph workloads do not utilize a large fraction of memory bandwidth
  - Many graph workloads have decent locality

Conclusions

- Most graph workloads do not utilize a large fraction of memory bandwidth
  - Many graph workloads have decent locality
  - Cache misses too infrequent to fit in window

Conclusions

- Most graph workloads do not utilize a large fraction of memory bandwidth
  - Many graph workloads have decent locality
  - Cache misses too infrequent to fit in window
  - Changing processor alone could help

Conclusions

- Most graph workloads do not utilize a large fraction of memory bandwidth
  - Many graph workloads have decent locality
  - Cache misses too infrequent to fit in window
  - Changing processor alone could help
- Sub-linear parallel speedups cast doubt on gains from multithreading on OoO core

Outline

- Performance Analysis for Graph Applications
- Milk / Propagation Blocking
- Frequency based Clustering
- CSR Segmenting
- Summary

Milk / Propagation Blocking

- Is changing the architecture really the only way to improve the performance of graph applications running in memory?
  - Boosting MLP
- Other approaches
  - Reducing the amount of communication

Optimizing Indirect Memory References with `milk`

Vladimir Kiriansky, Yunming Zhang, Saman Amarasinghe

MIT

CSAIL

PACT '16

September 13, 2016, Haifa, Israel

Indirect Accesses

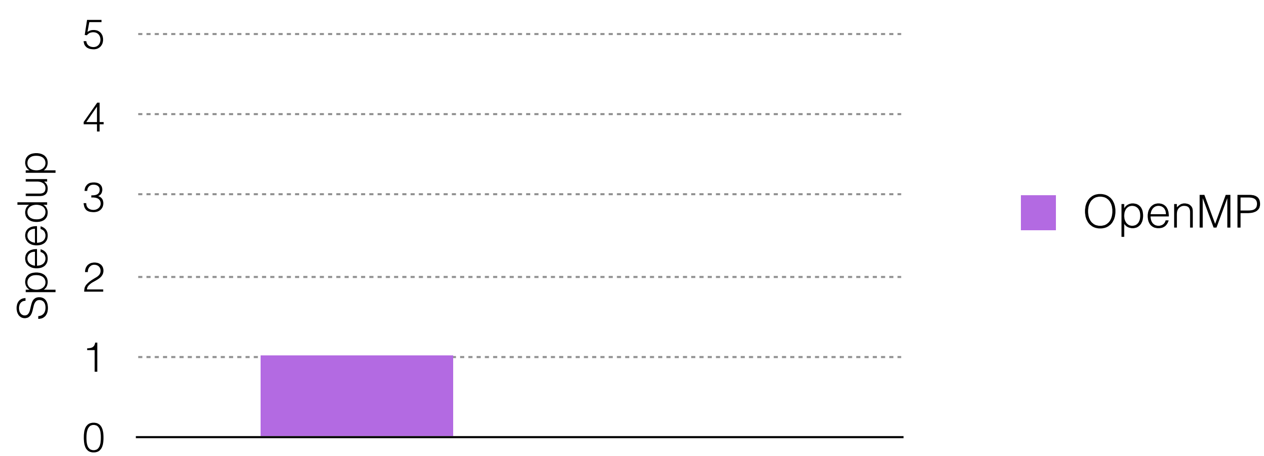
```
for(int i=0; i<N; i++)
    count[d[i]]++;
```

Indirect Accesses with OpenMP

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

Indirect Accesses with OpenMP

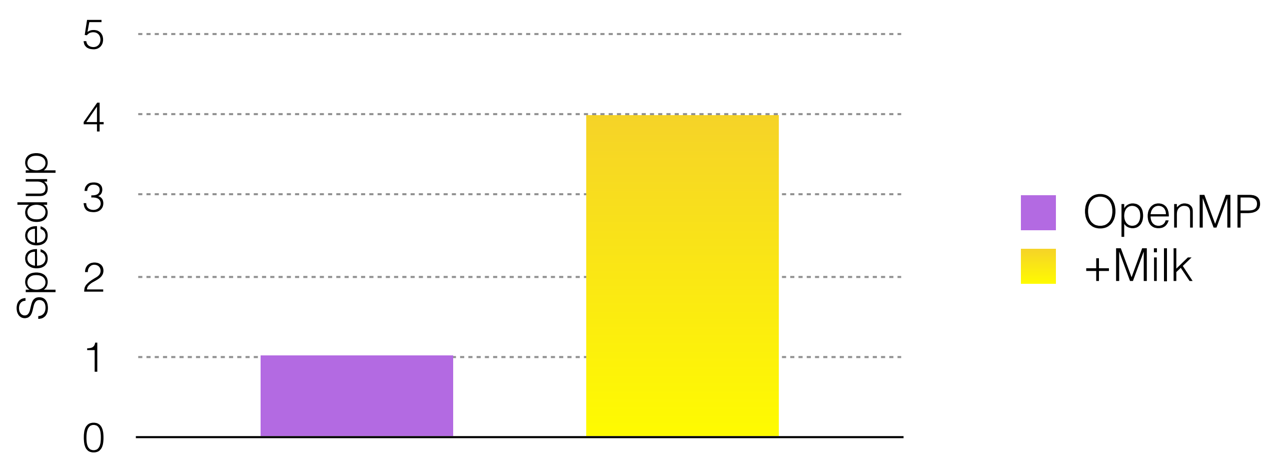
```
#pragma omp parallel for
for(int i=0; i<N; i++)
    #pragma omp atomic
    count[d[i]]++;
```



uniform [0..100M)
8 threads, 8MB L3

Indirect Accesses with **milk**

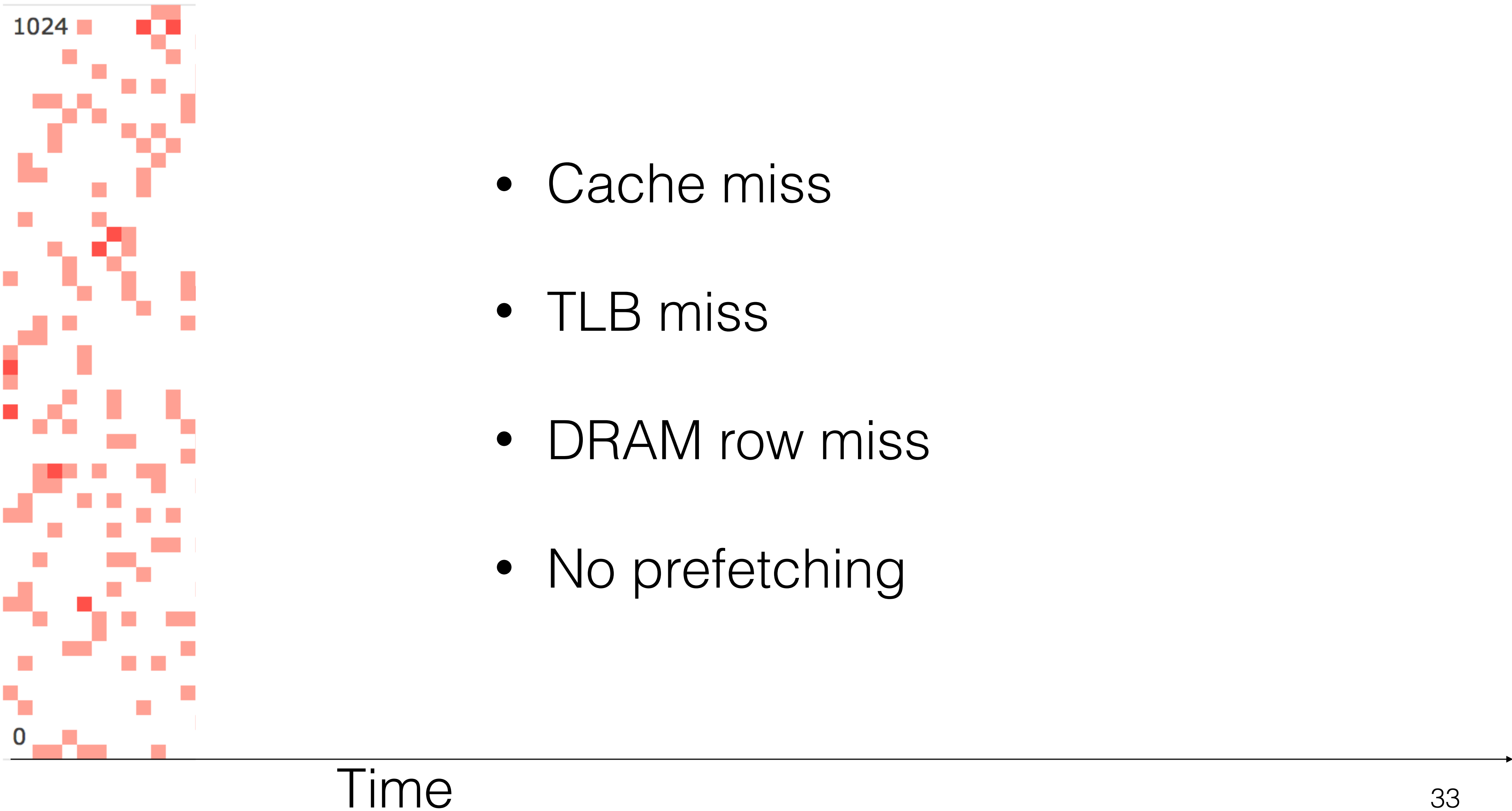
```
#pragma omp parallel for milk
for(int i=0; i<N; i++)
    #pragma omp atomic if(!milk)
    count[d[i]]++;
```



uniform [0..100M)

8 threads, 8MB L3

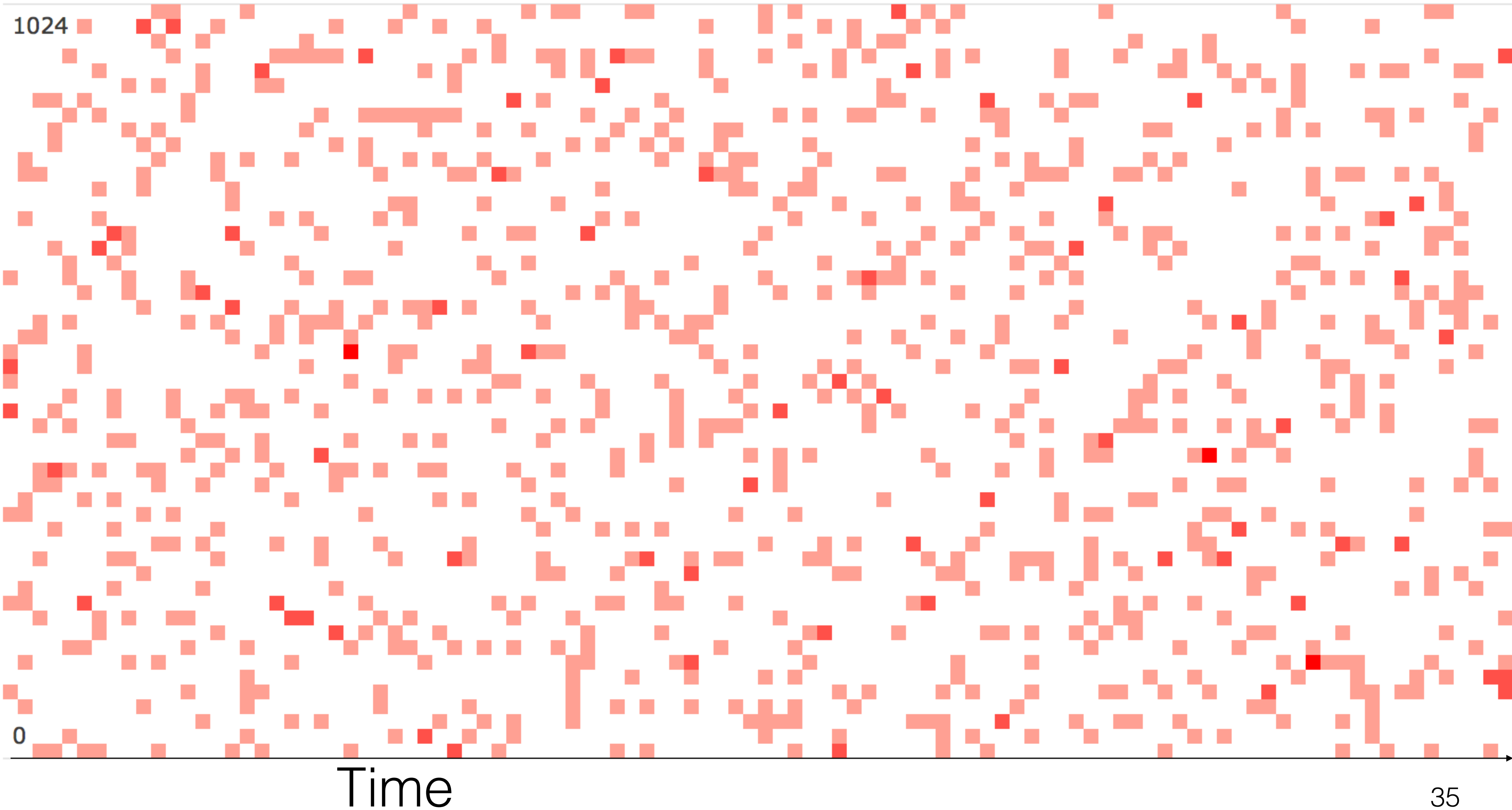
No Locality?



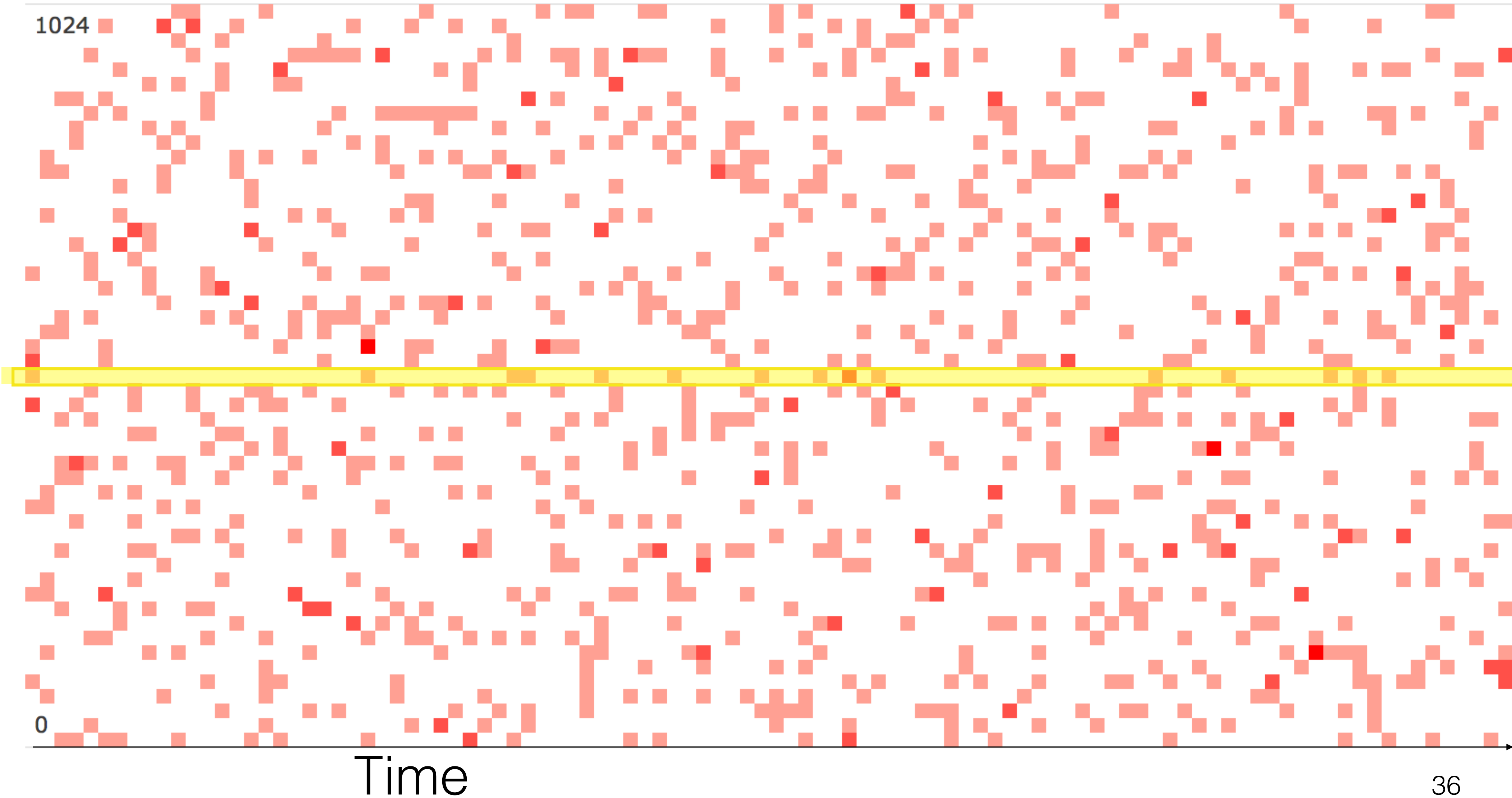
- Cache miss
- TLB miss
- DRAM row miss
- No prefetching

No Locality?

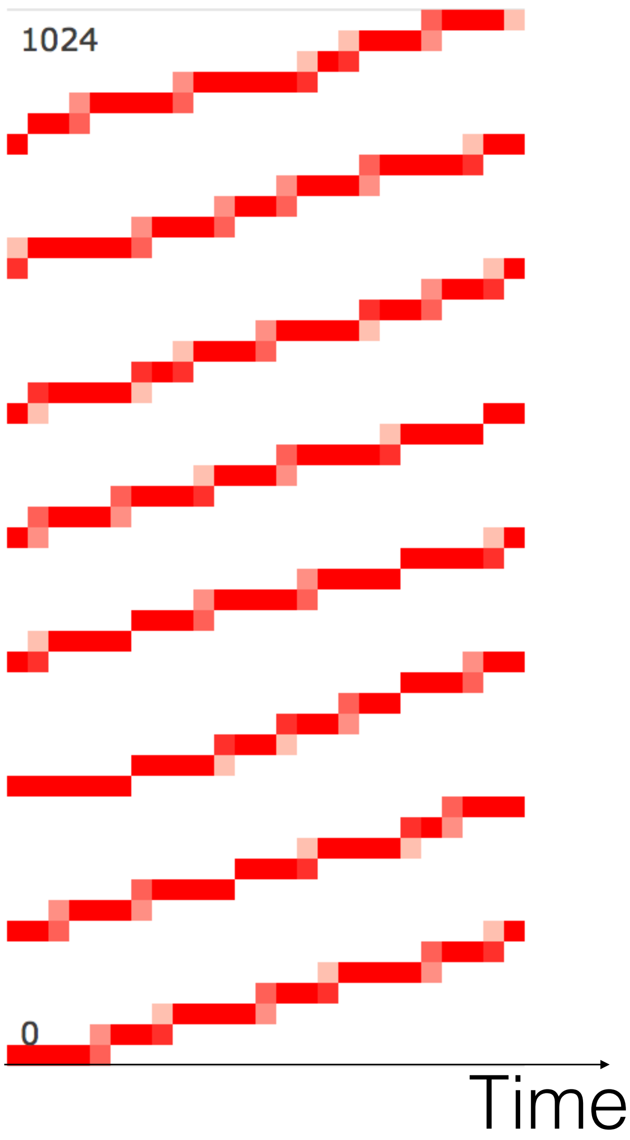
No Locality?



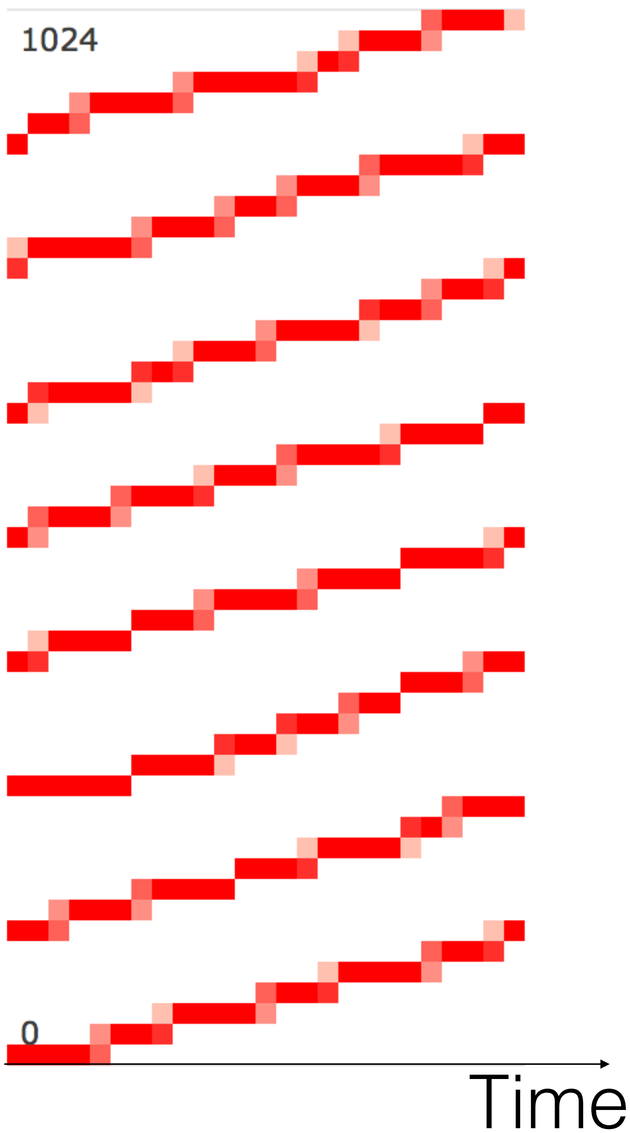
No Locality?



Milk Clustering

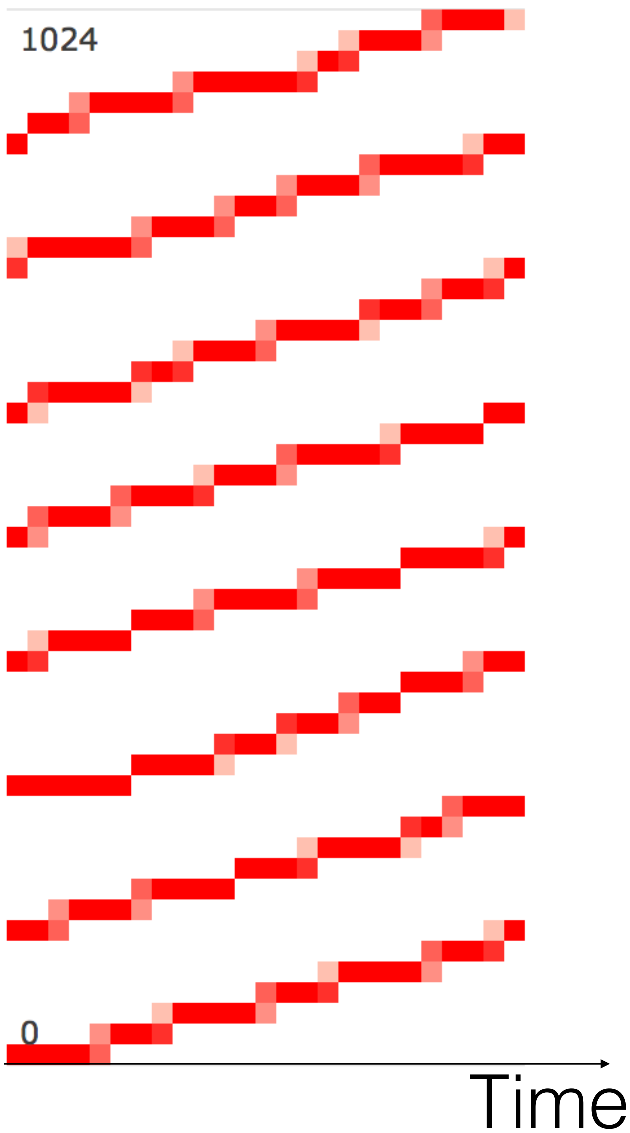


Milk Clustering



- 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

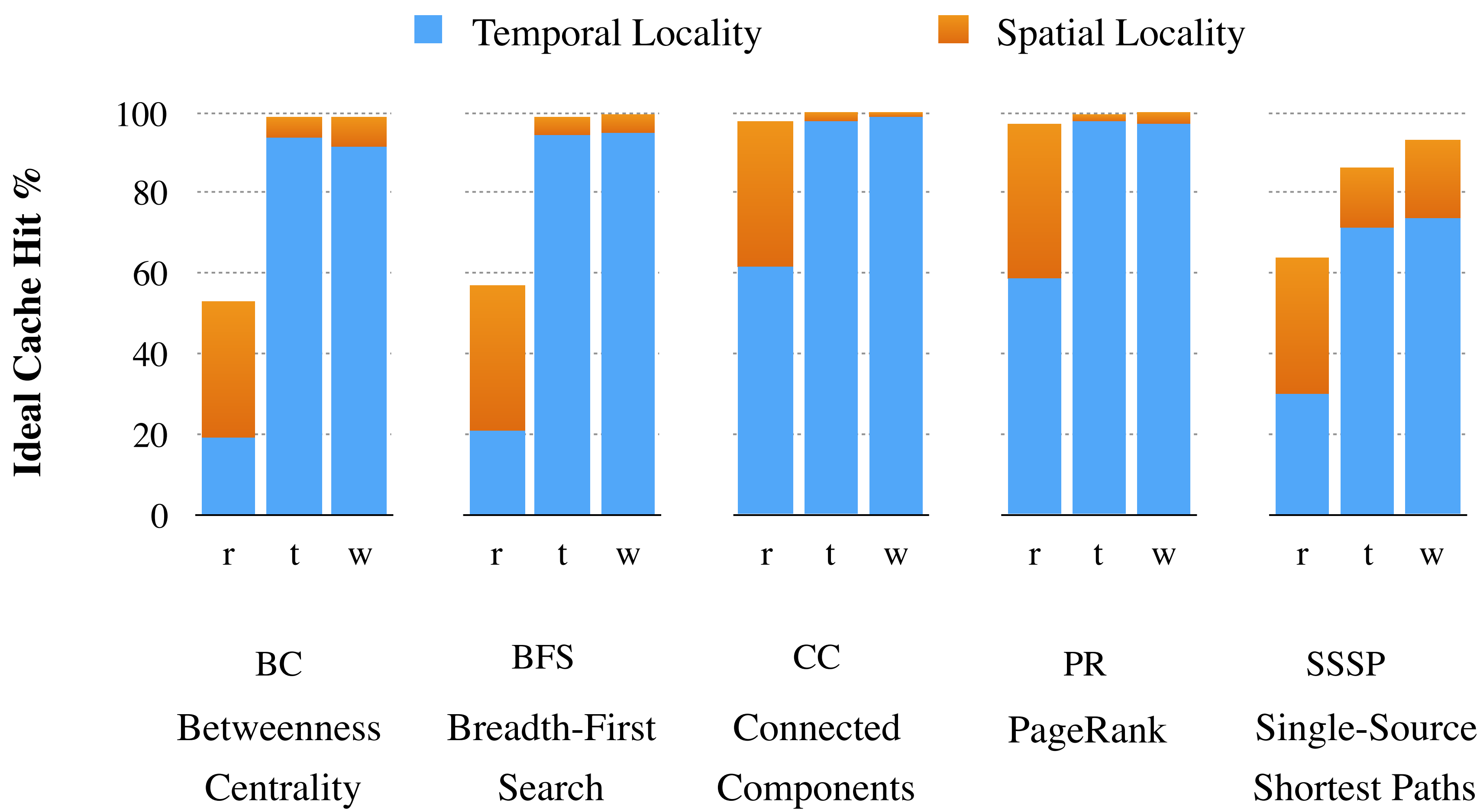
Foundations

- Milk programming model — extending BSP
- **`milk`** syntax — OpenMP for C/C++
- MILK compiler and runtime — LLVM/Clang

Big (sparse) Data

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

Infinite Cache Locality in Graph Applications



Road (d=2.4)
Twitter (d=24)
Web (d=39)

[GAPBS]

Milk Execution Model

- Collection
- Distribution
- Delivery

Propagation Blocking:
Binning
(Collection + Distribution),
Deliver
(Accumulation)

Collection

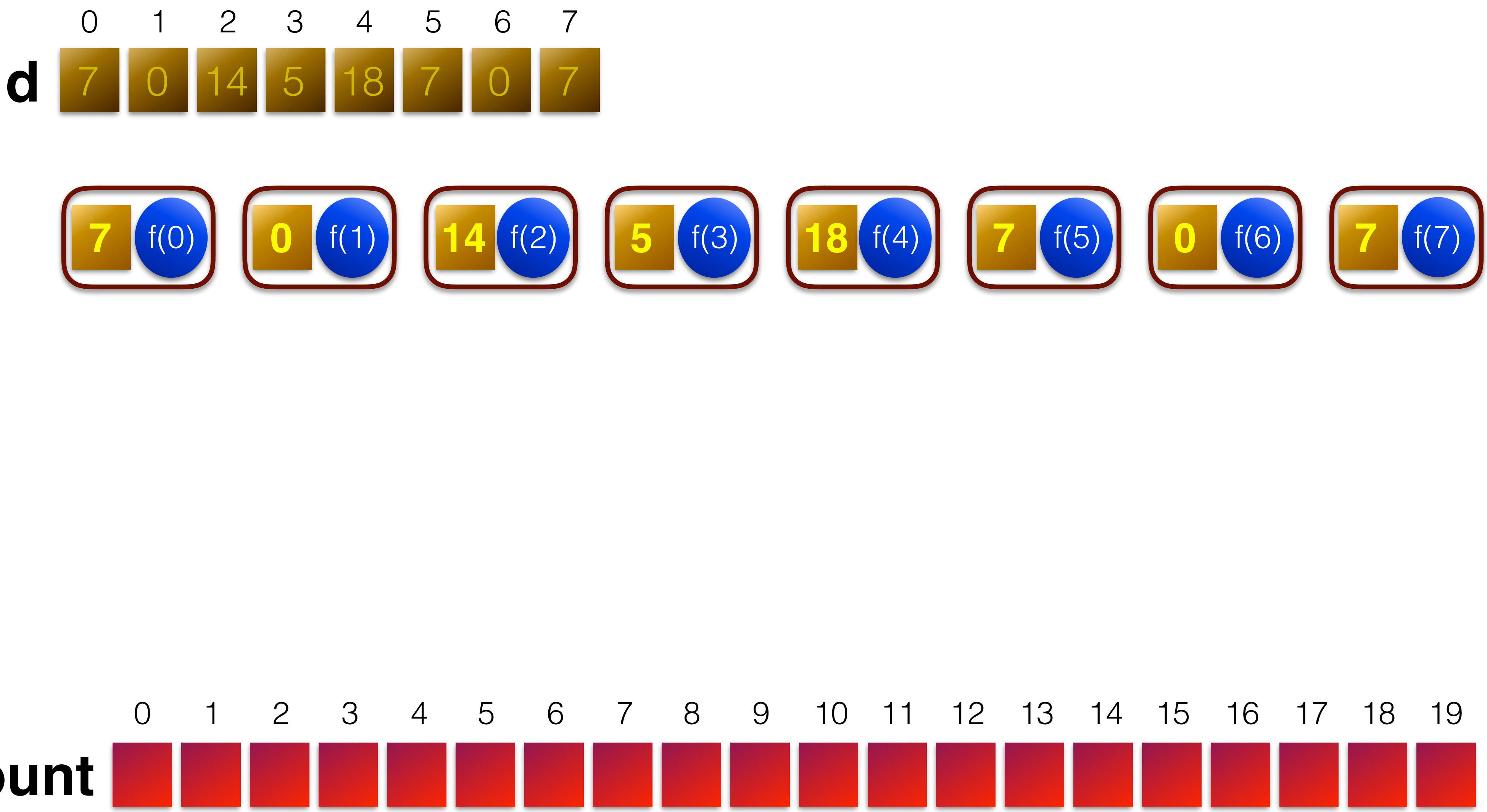
```
#pragma omp parallel for
for(int i=0; i<N; i++)
    #pragma omp atomic
    count[d[i]] += f(i);
```

	0	1	2	3	4	5	6	7
d	7	0	14	5	18	7	0	7

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
count																				

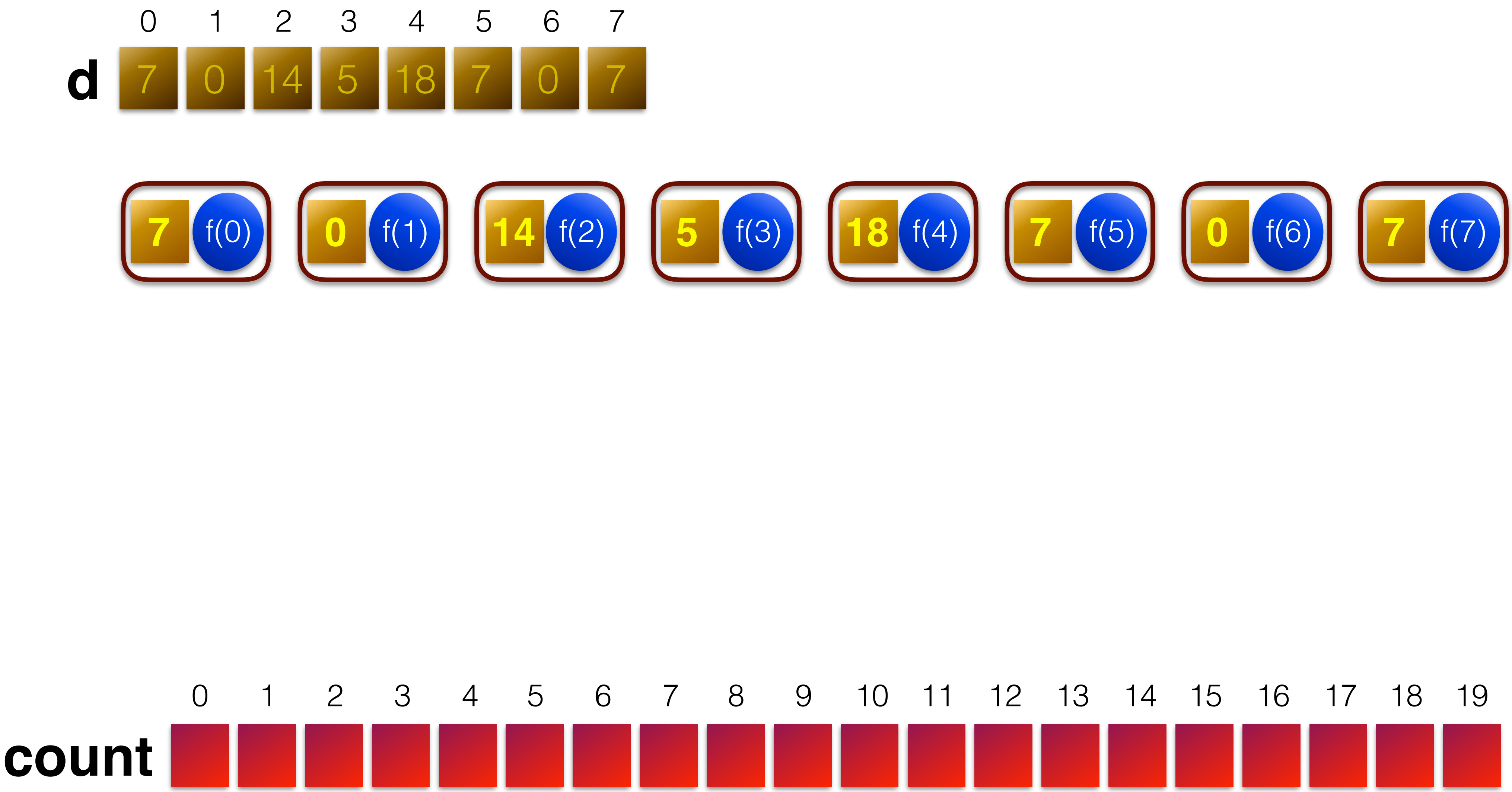
Collection

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



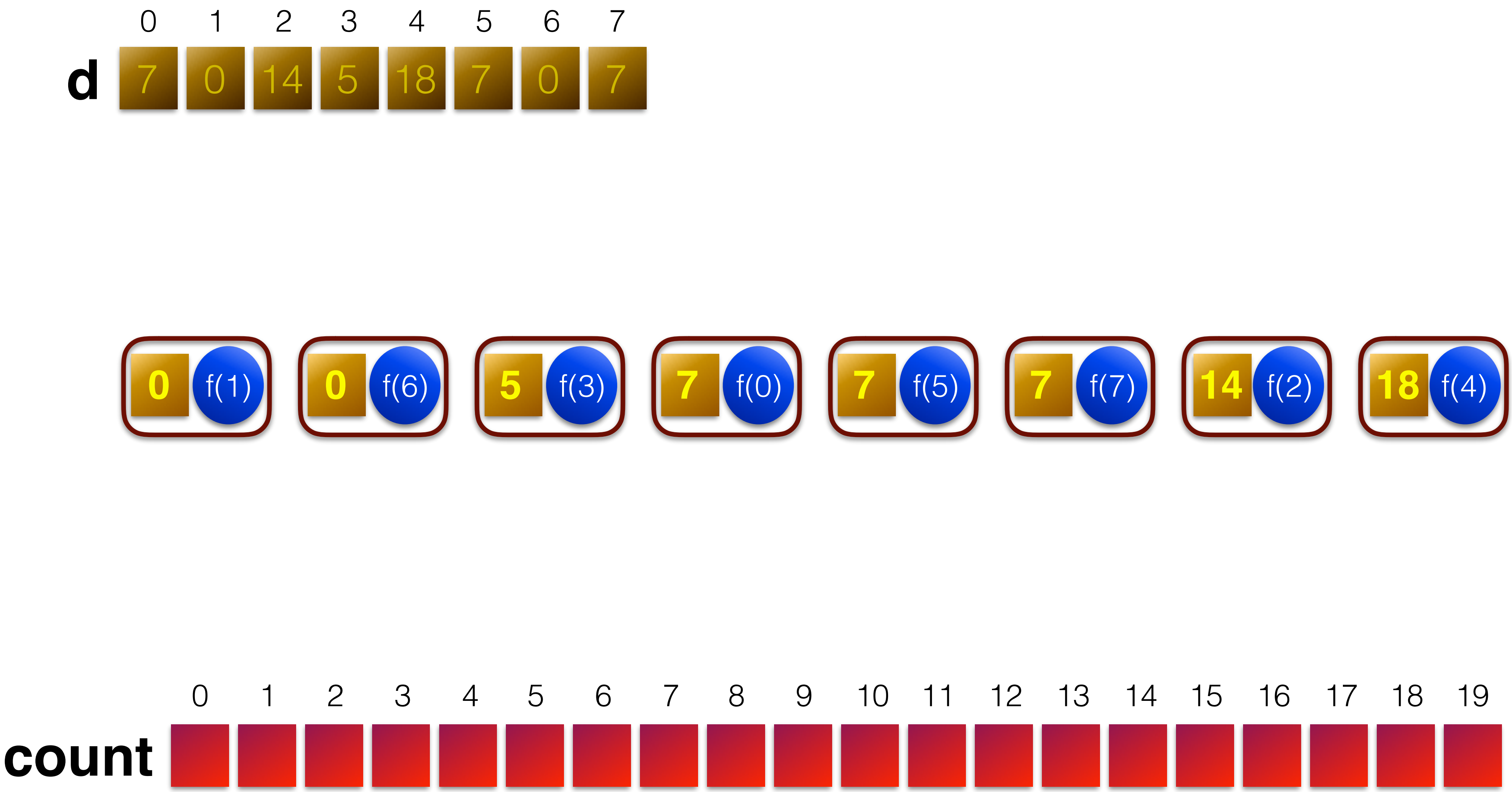
Distribution

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



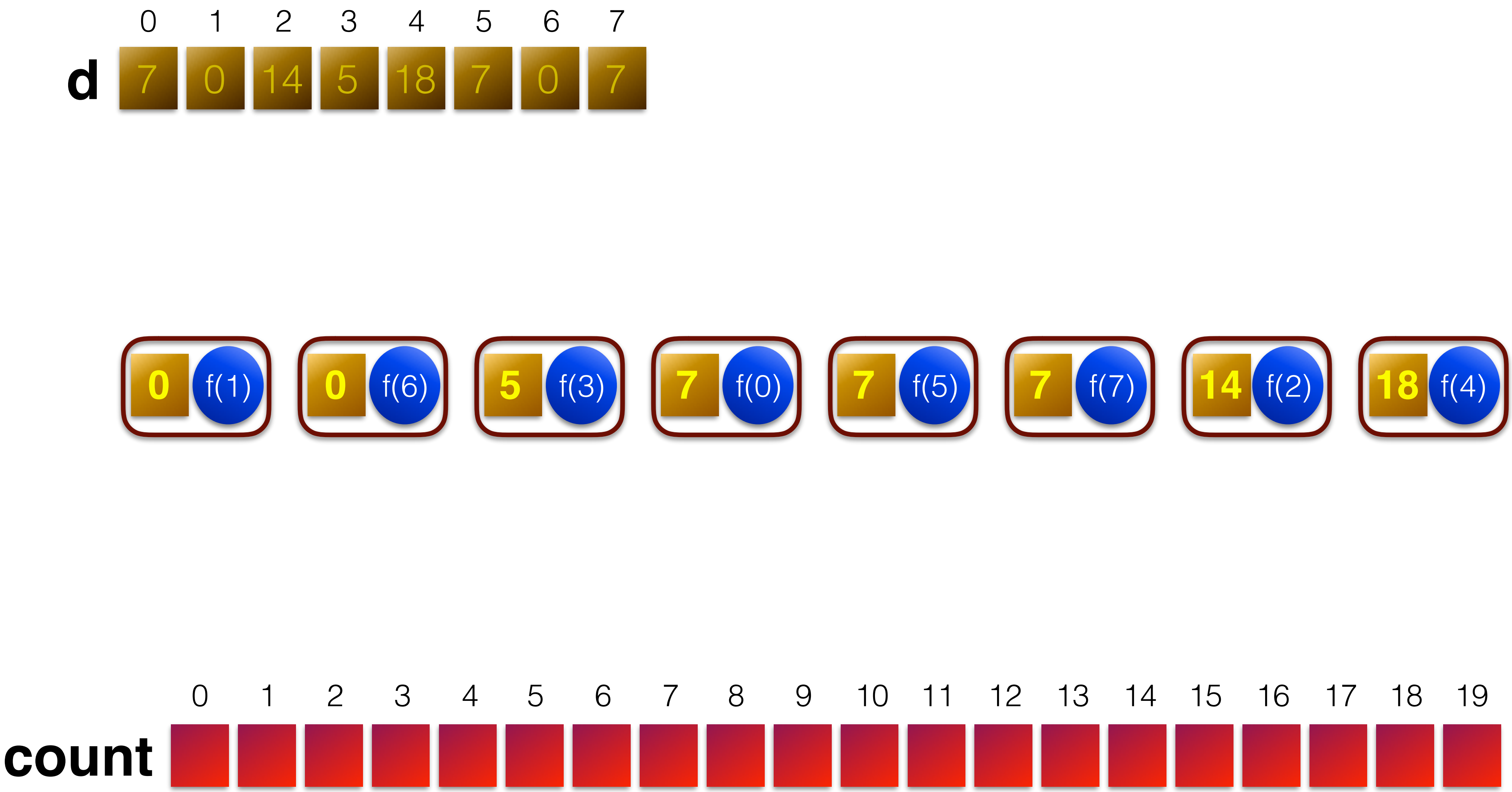
Distribution

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



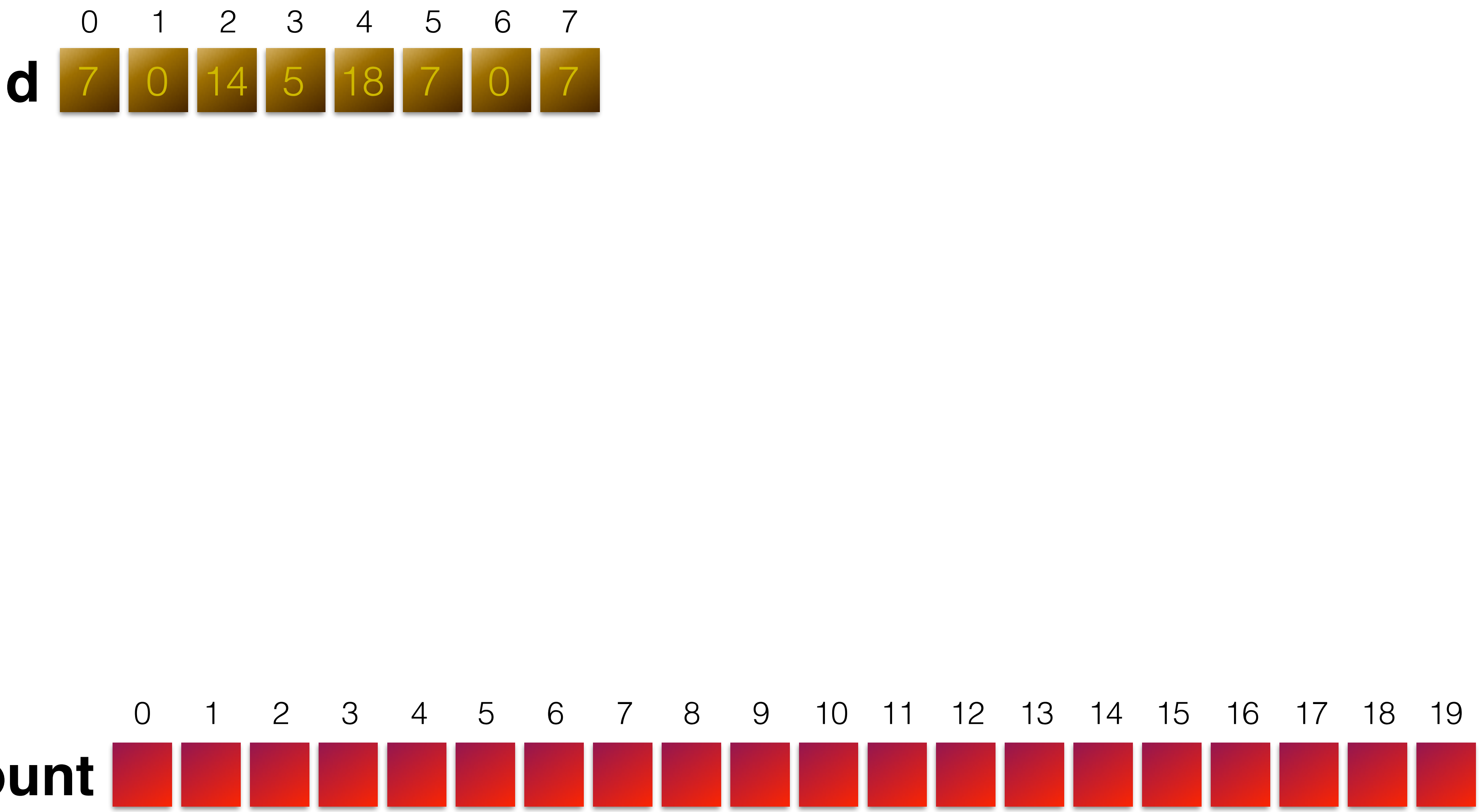
Delivery

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



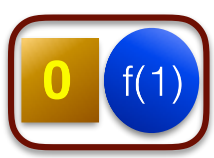
Delivery

```
#pragma omp parallel for
for(int i=0; i<N; i++)
    #pragma omp atomic
    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

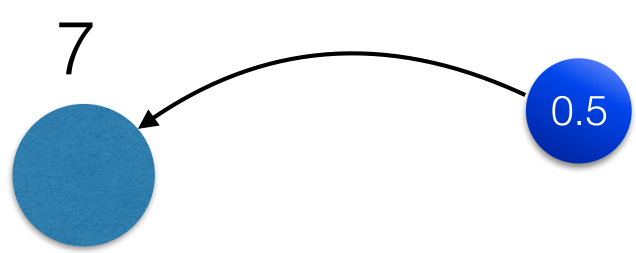


`pack` Combiners

```
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**

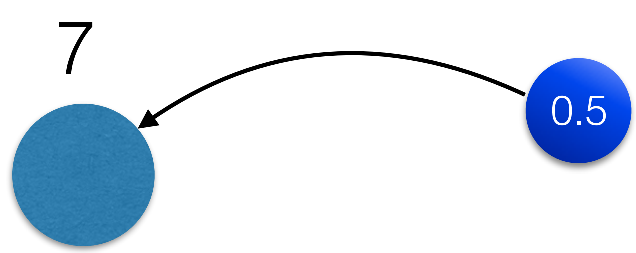
```
vector<float> contrib, new_rank;

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

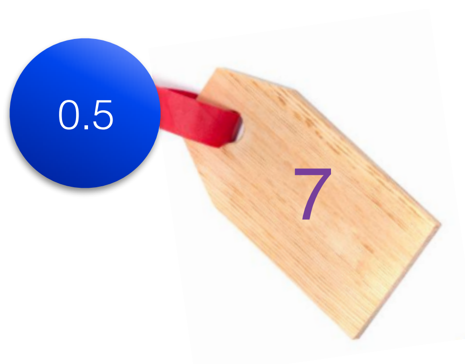
PageRank with **milk**

```
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;
  }
}
```

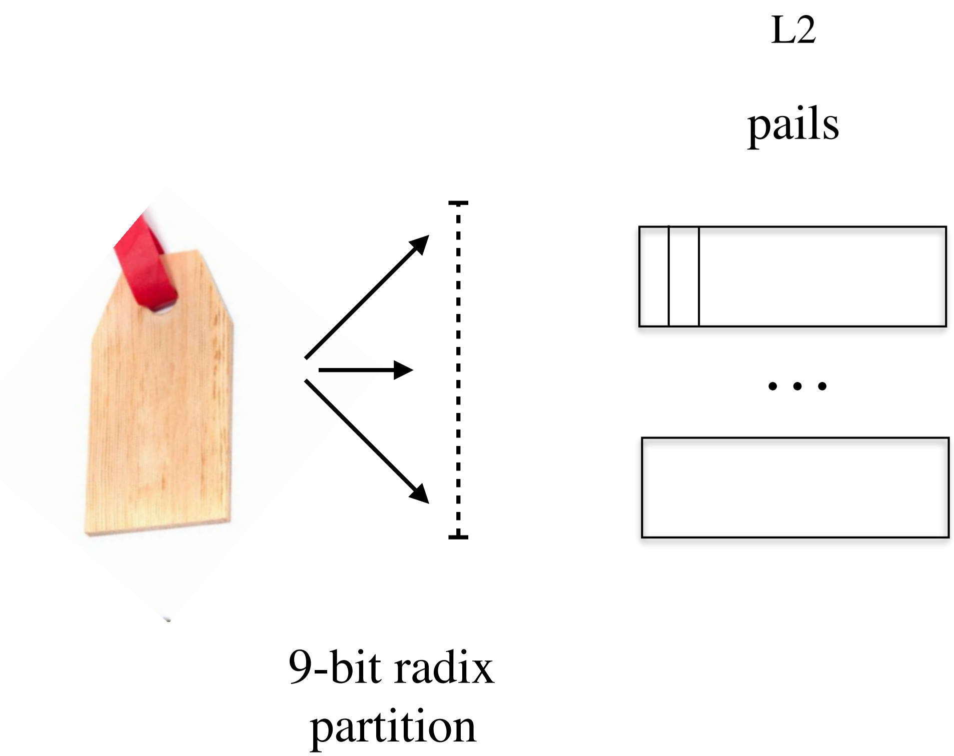


PageRank: Collection

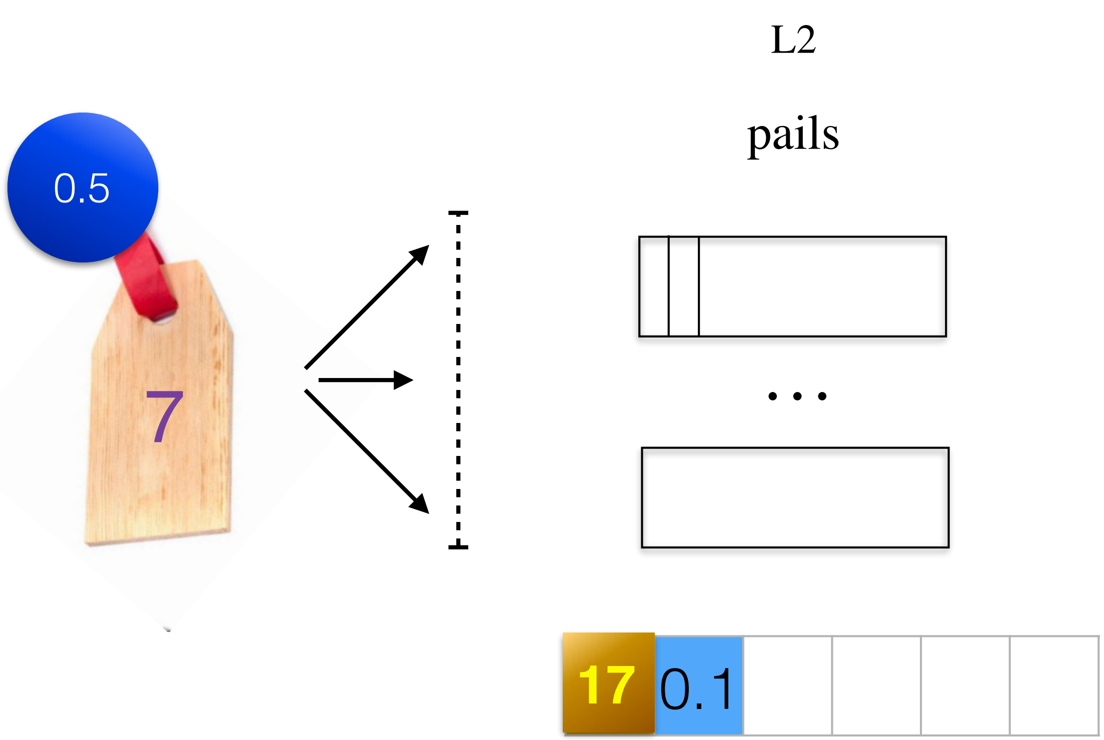
```
vector<float> contrib, new_rank;

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)


  }
}
```



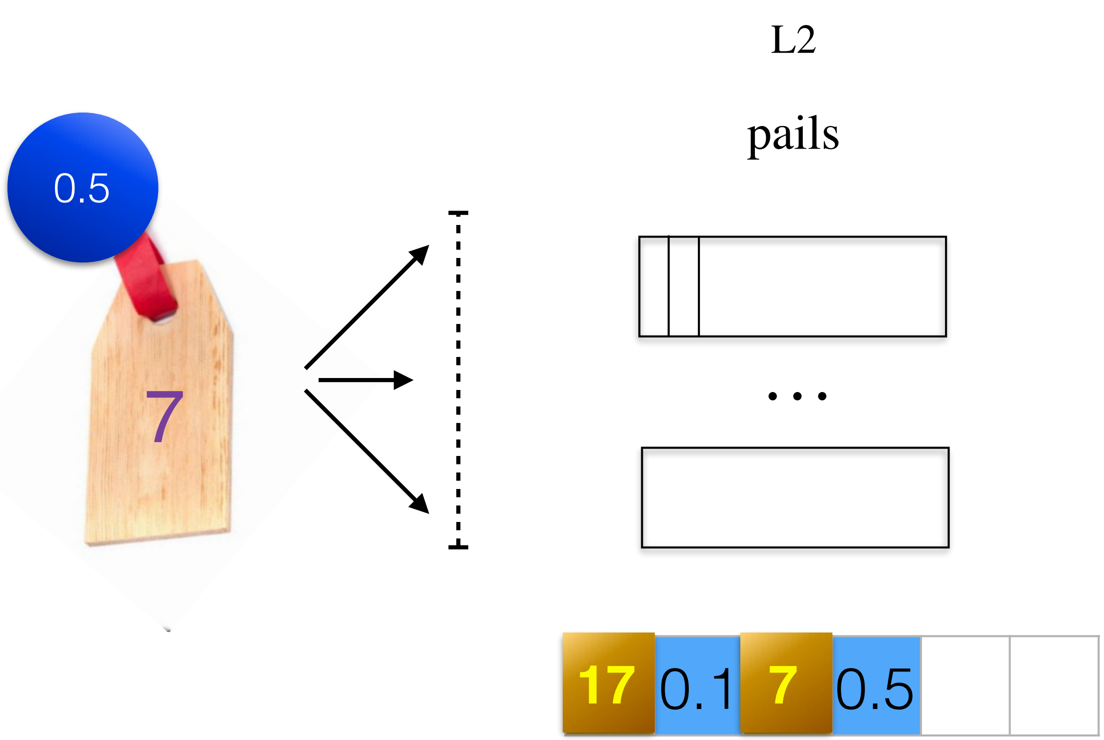
Tag Distribution



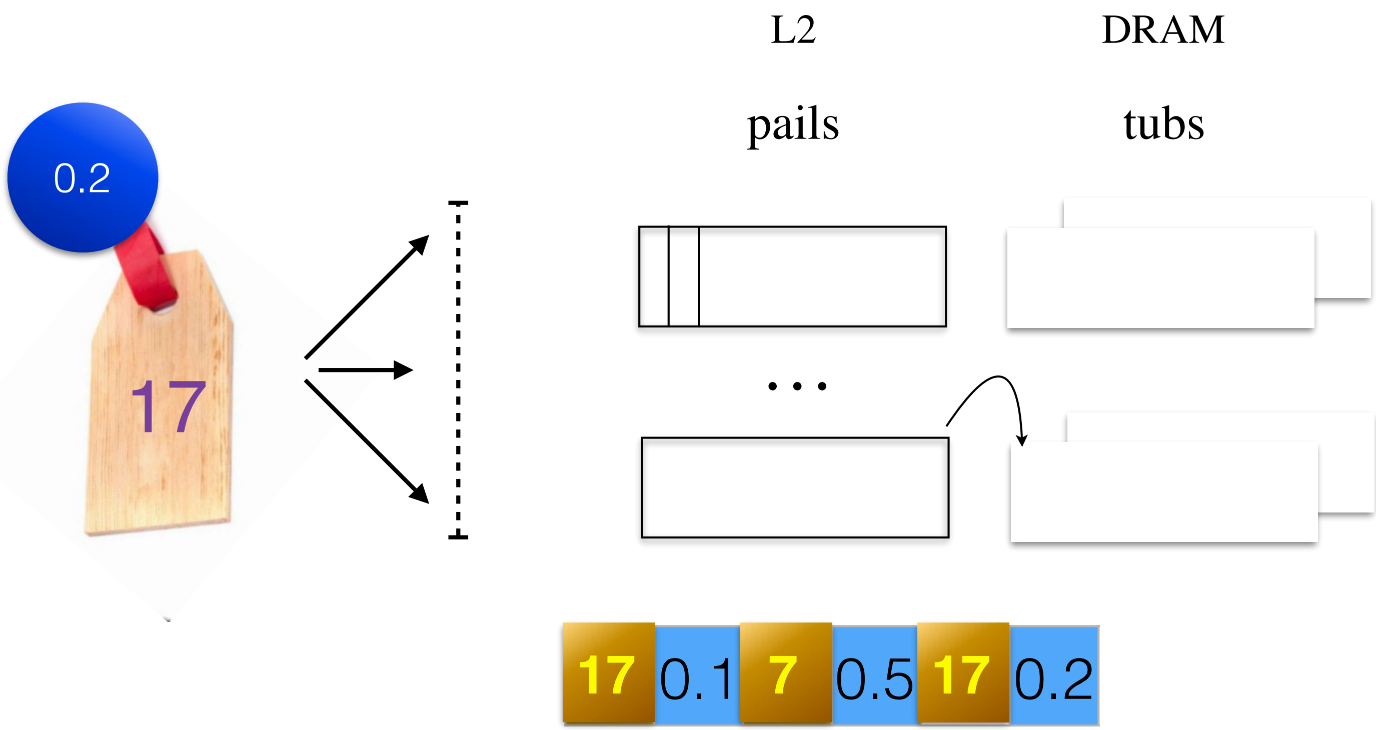
Tag Distribution



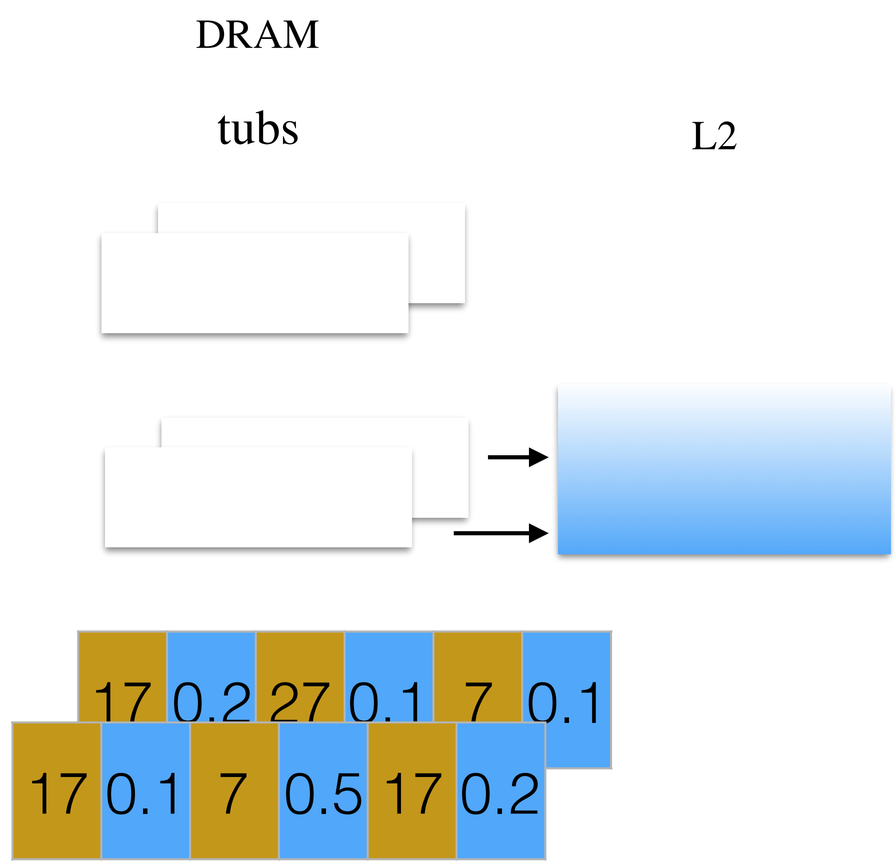
Tag Distribution



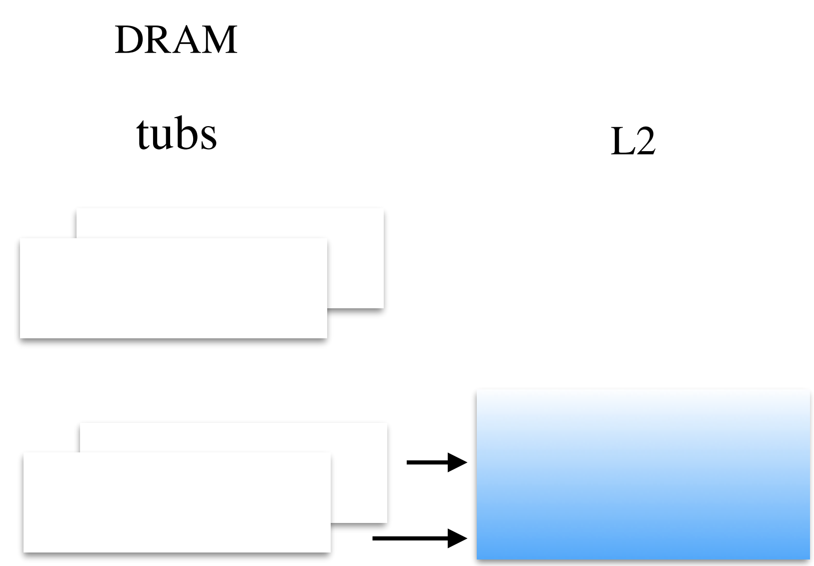
Distribution: Pail Overflow



Milk Delivery



Milk Delivery

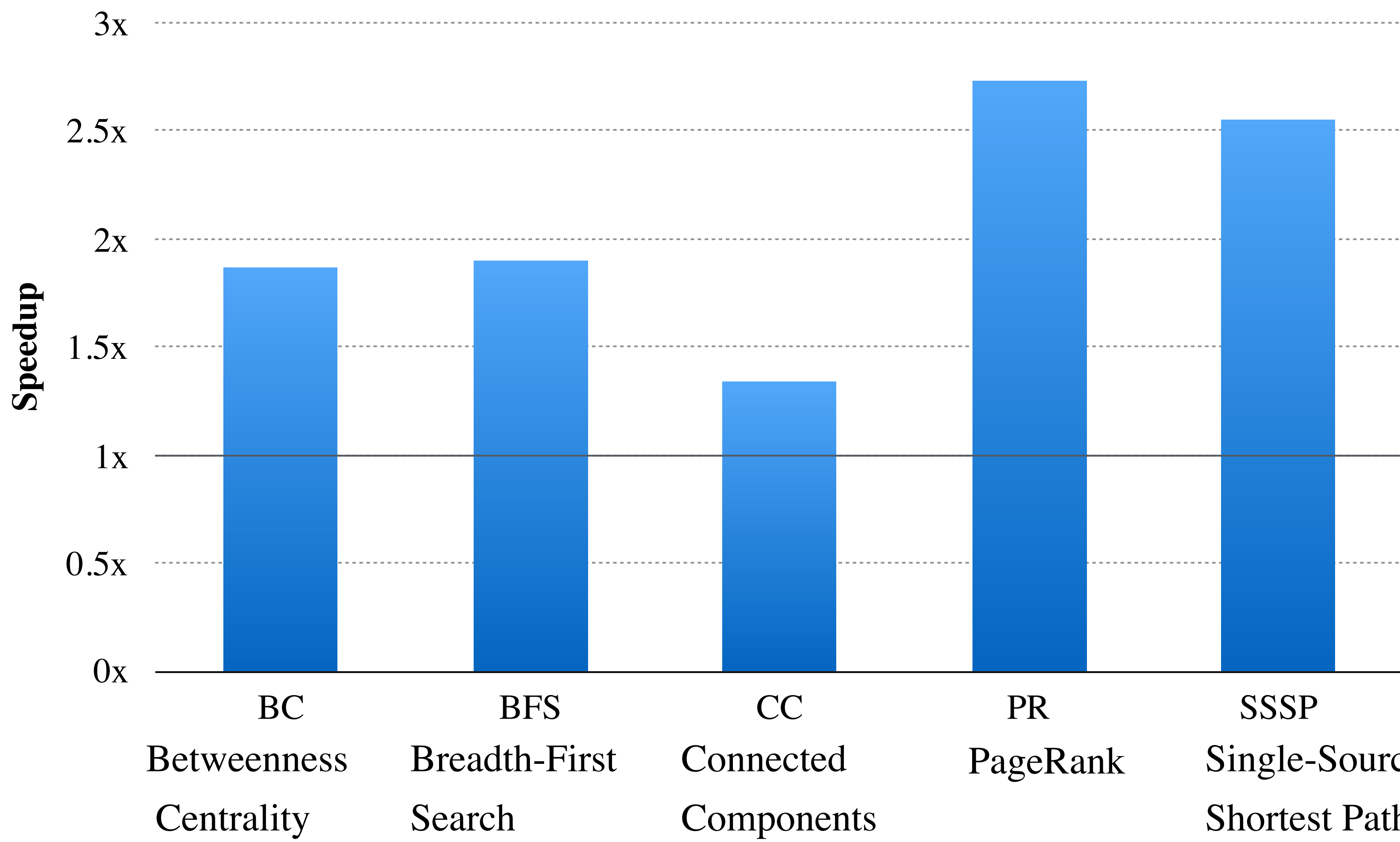



```
#pragma milk pack(contribU : +) tag(v)
#pragma omp atomic if(!milk)
      new_rank[v] += contribU;
```

Related Work

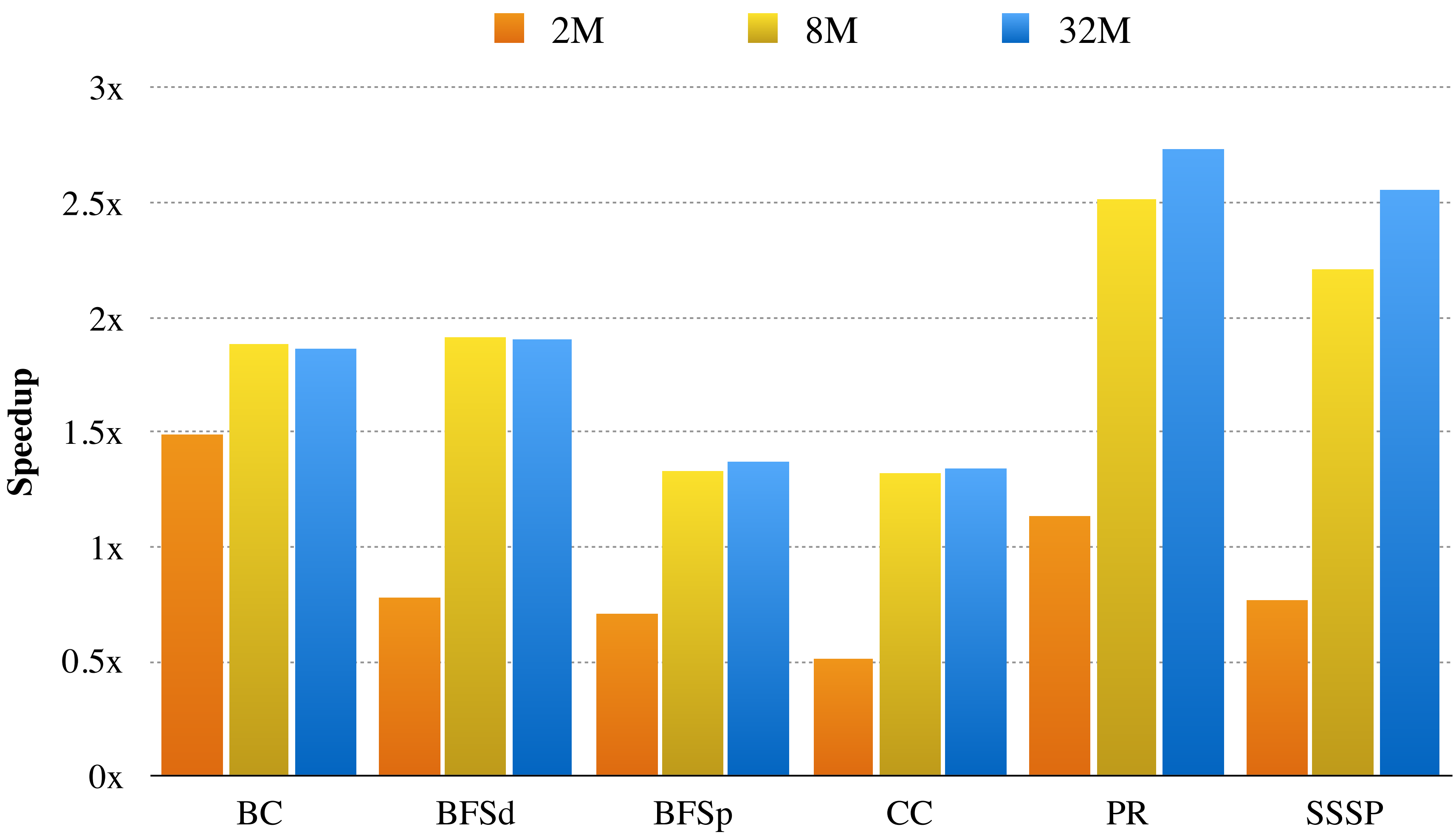
- Database JOIN optimizations
  - [Shatdal94] cache partitioning
  - [Manegold02, Kim09, Albutiu12, Balkesen15] TLB, SIMD, NUMA, non-temporal writes, software write buffers

Overall Speedup with **`milk`**



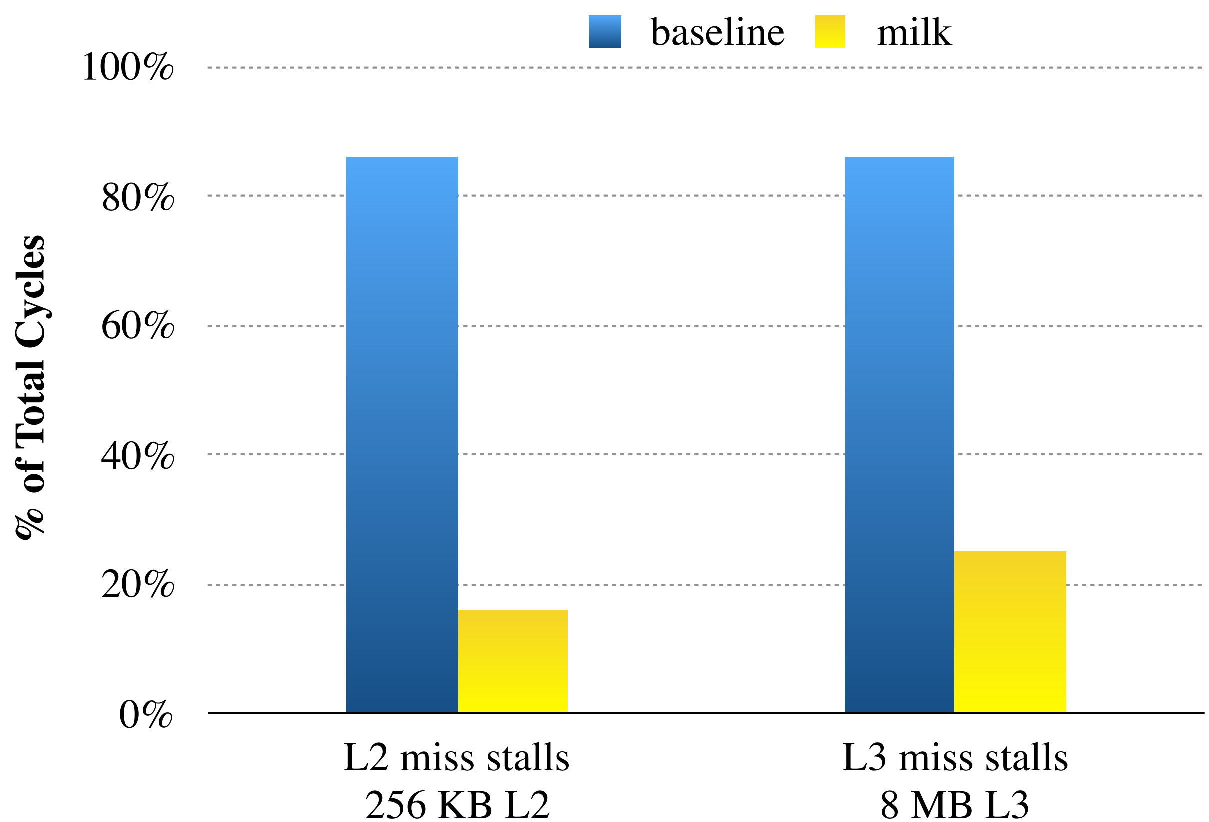
V=32M
8 MB L3

Overall Speedup with **`milk`**



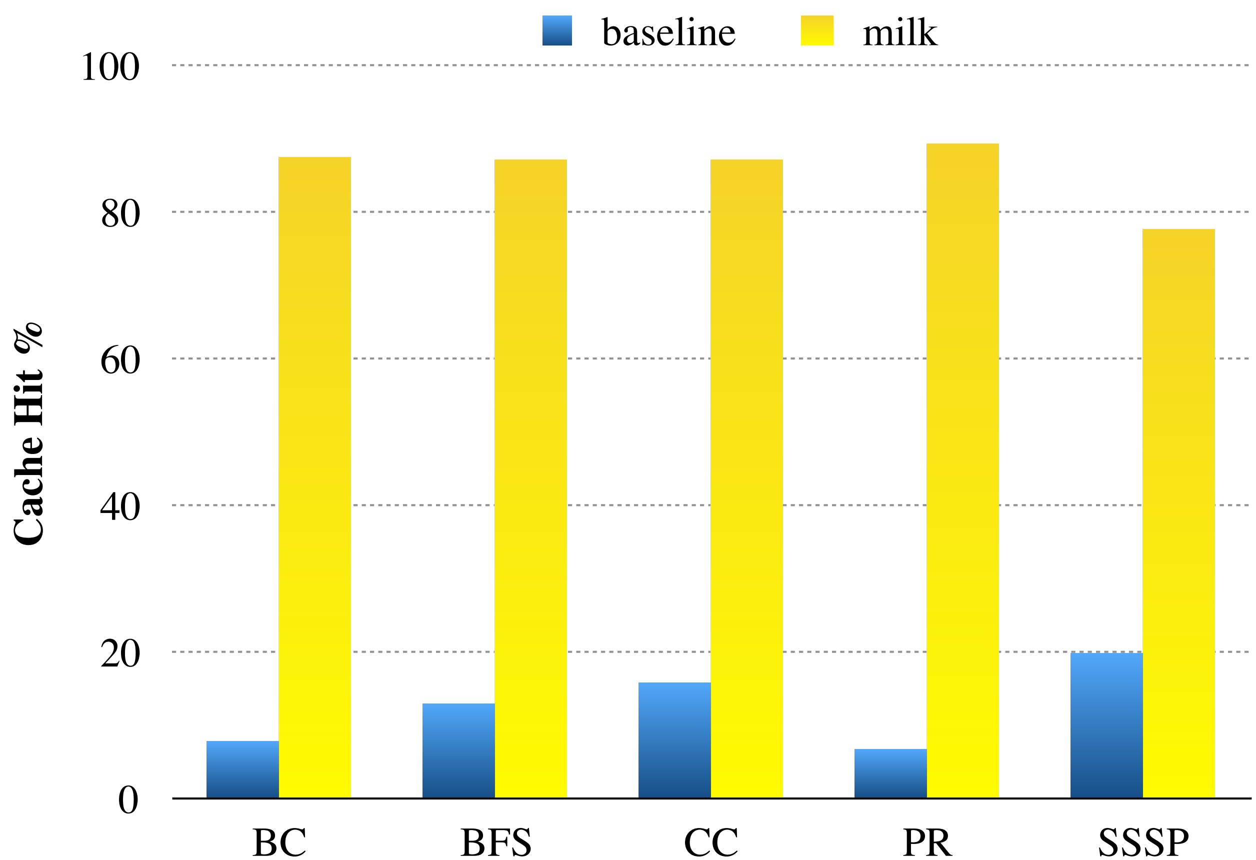
8 MB L3

Stall Cycle Reduction



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

Indirect Access Cache Hit%

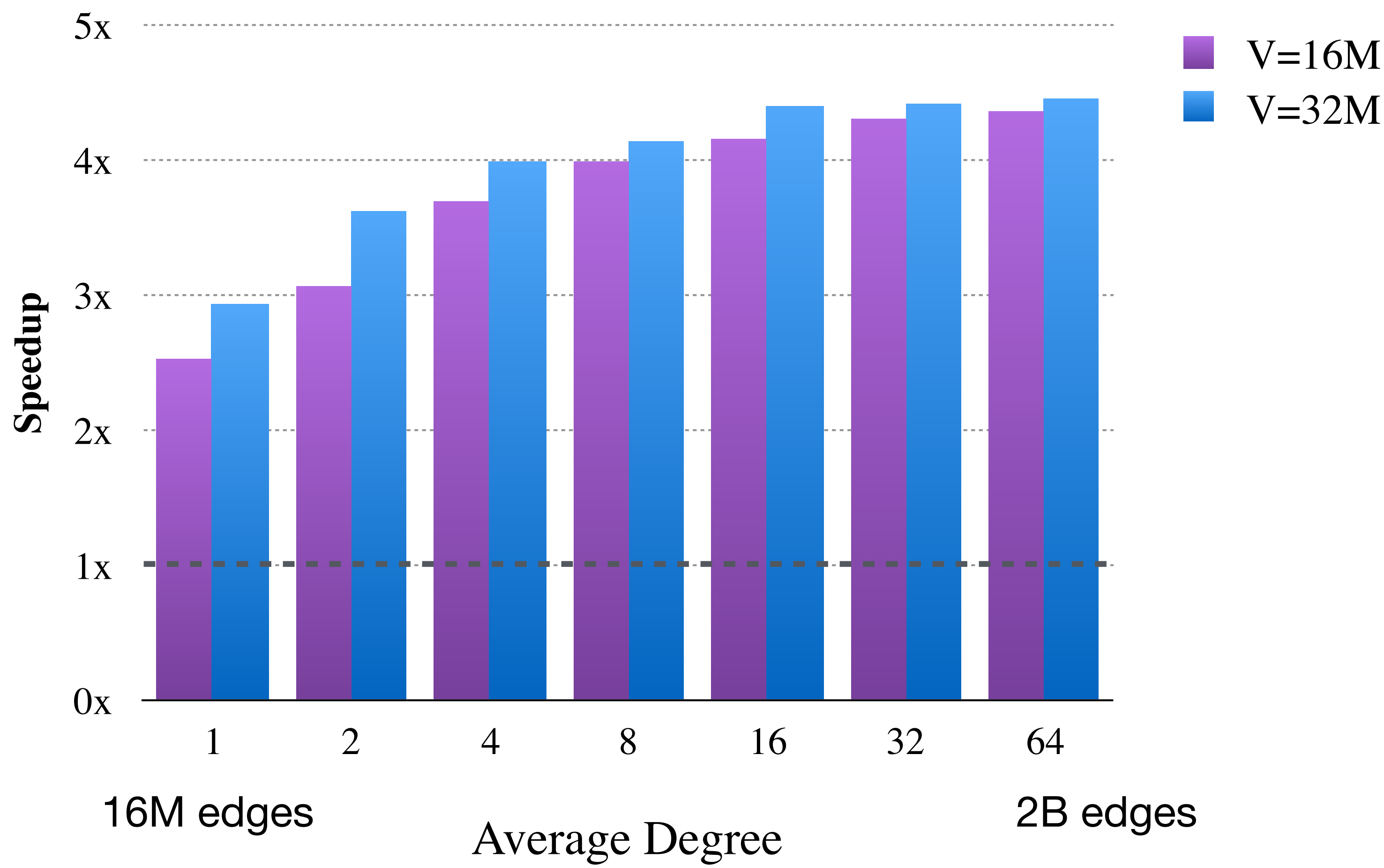


V=32M

8 MB L3

256KB L2

Higher Degree
→ Higher Locality



Related Works

- How is Milk different from BigSparse and Propagation Blocking?

Related Work

- Big Sparse
  - Big Sparse can work on both graphs with good and bad locality (Milk and Propagation Blocking both work on low locality graphs)
  - Can afford to do global sort instead of bucketing
- Propagation Blocking
  - Milk doesn't have two separate phases for Binning and Accumulate (Collection, Distribution, Delivery are all fused together using coroutines)
  - PB reuses the tags to save memory bandwidth assuming the application is iterative
  - Milk has a more general programming model for various applications

Outline

- Performance Analysis for Graph Applications
- Milk / Propagation Blocking
- Frequency based Clustering
- CSR Segmenting
- Summary

Making Caches Work for Graph Analytics

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

MIT CSAIL and *Stanford InfoLab

CSAIL

MIT COMPUTER SCIENCE AND ARTI

LELAND STANFORD JUNIOR UNIVERSITY

DIE LUFT DER FREIHEIT WEHT

1891

Outline

- PageRank
- 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

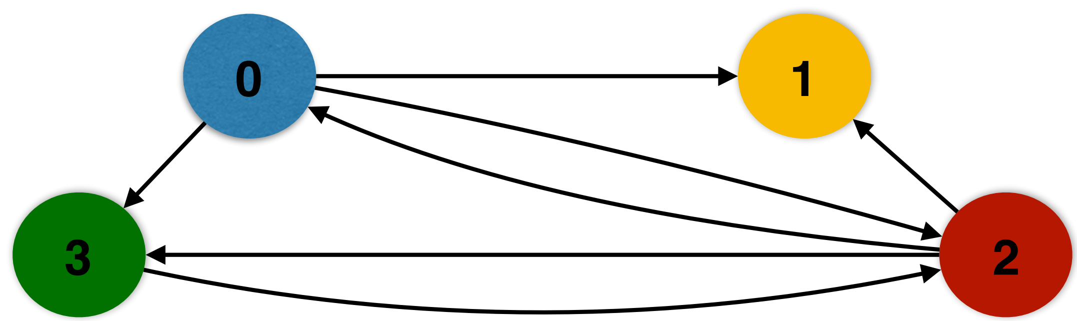
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

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
```

Cache

#hits: 0
#misses: 0



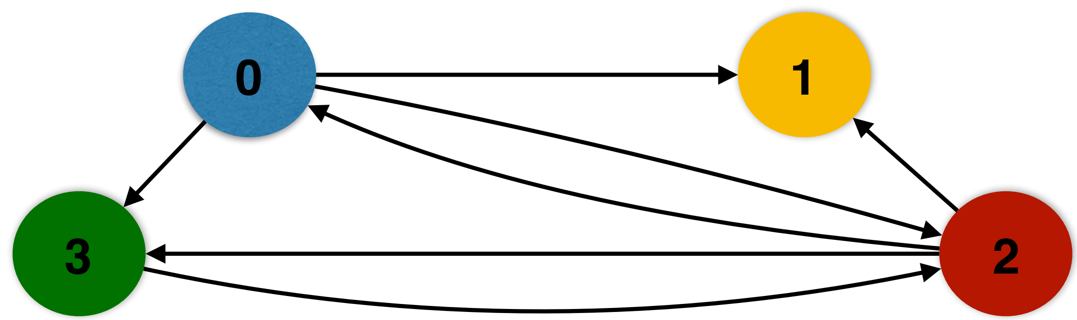
PageRank

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

Focus on the random memory accesses on ranks array

Cache

#hits: 0
#misses: 0



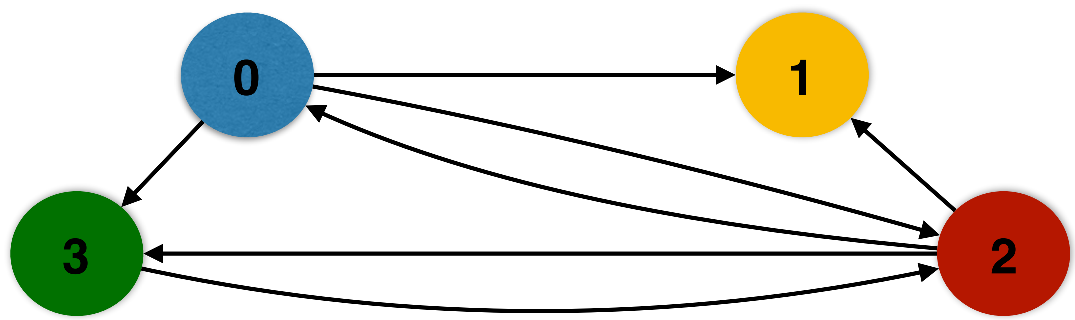
PageRank

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

Focus on the random memory accesses on ranks array

Cache ← **holds one cache line**

#hits: 0
#misses: 0



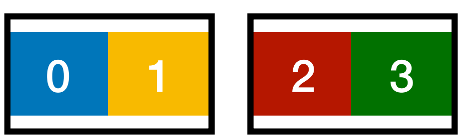
PageRank

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

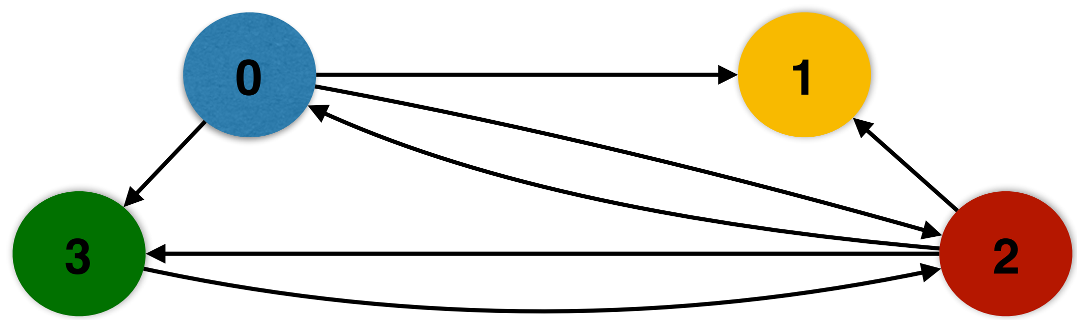
Focus on the random memory accesses on ranks array

Cache ← **holds one cache line**

#hits: 0
#misses: 0



← **stored in two cache lines**

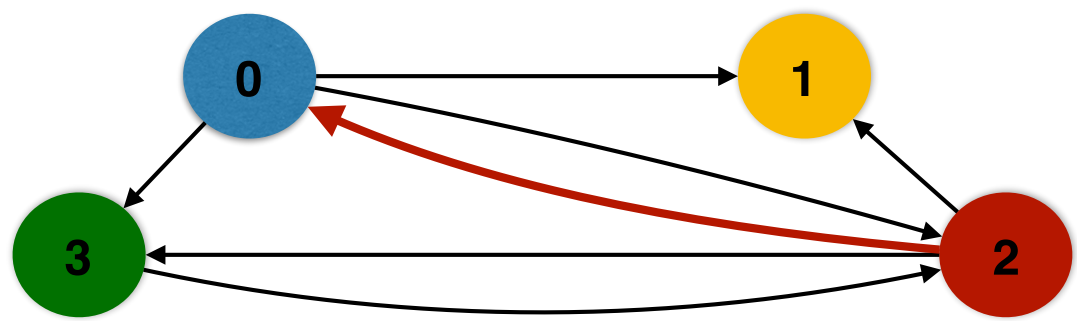


PageRank

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

Cache

#hits: 0
#misses: 0

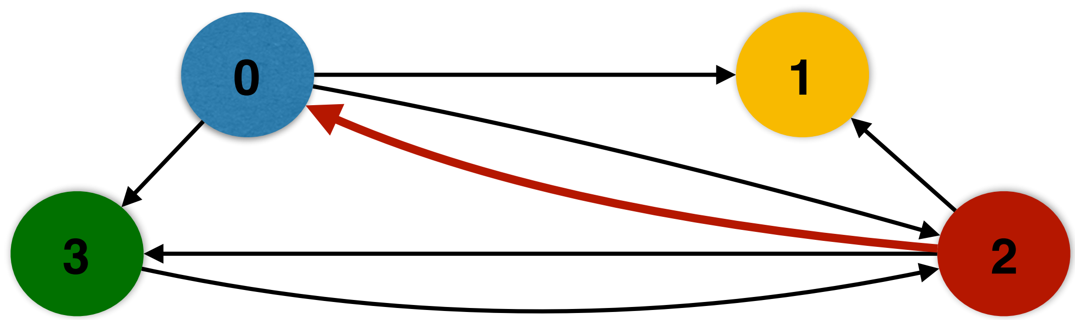


PageRank

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

Cache

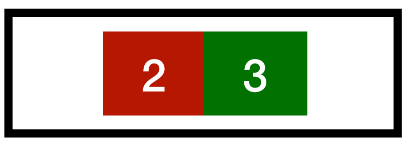
#hits: 0
#misses: 0



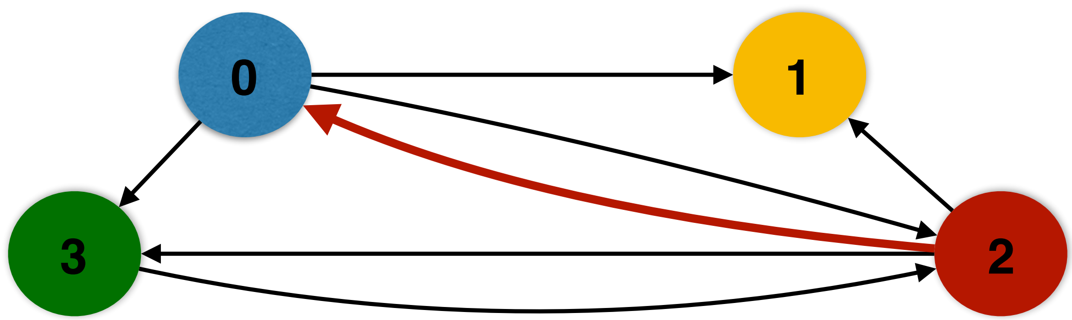
PageRank

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

Cache

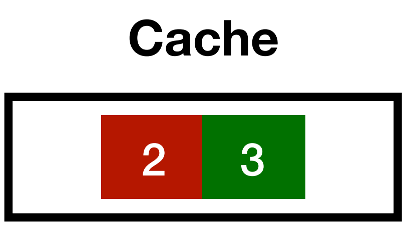


#hits: 0
#misses: 1

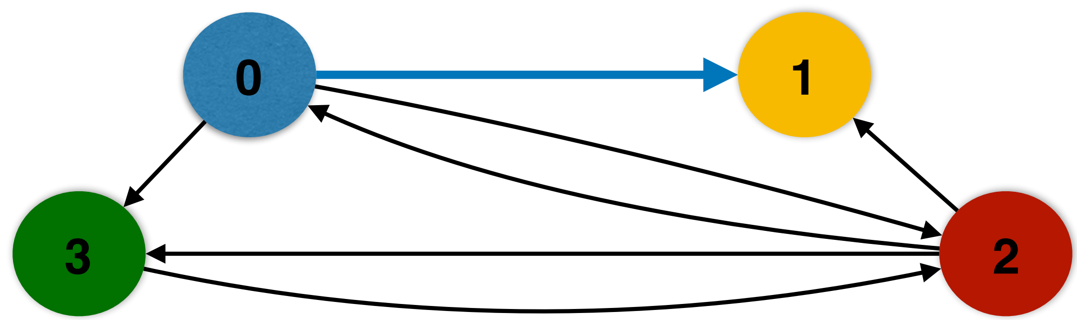


PageRank

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



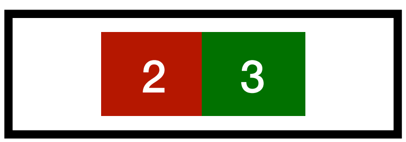
#hits: 0
#misses: 1



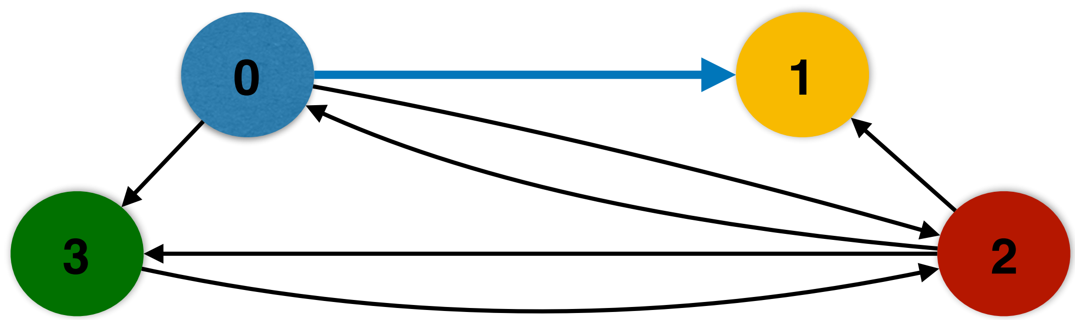
PageRank

```
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

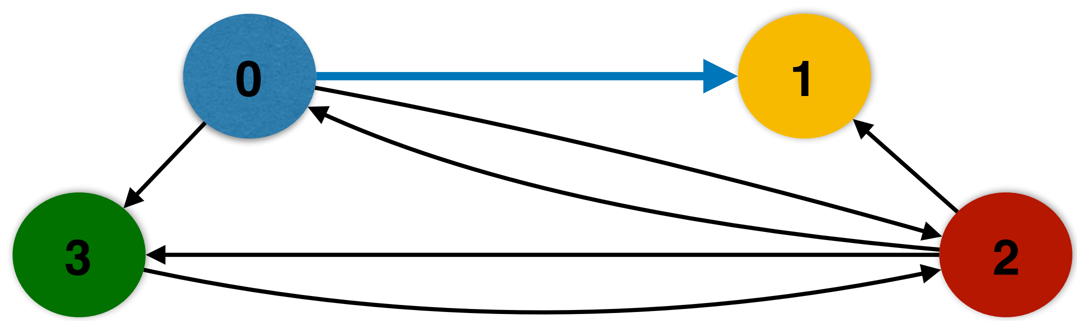


PageRank

```
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

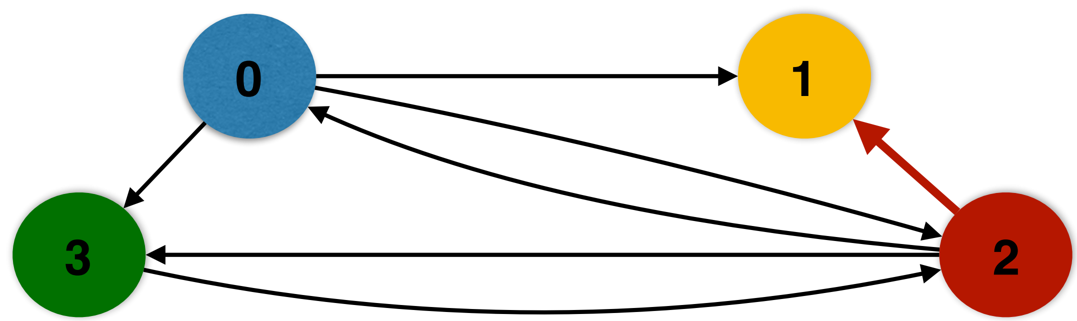


PageRank

```
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

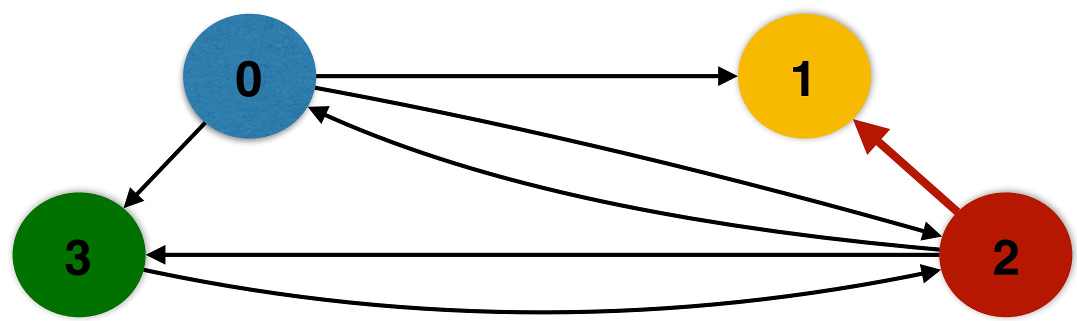


PageRank

```
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

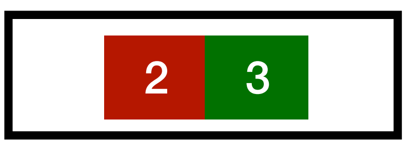
#hits: 0
#misses: 2



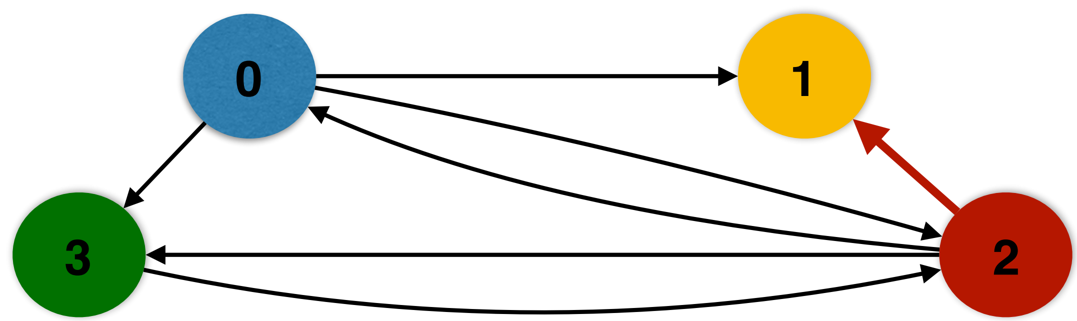
PageRank

```
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



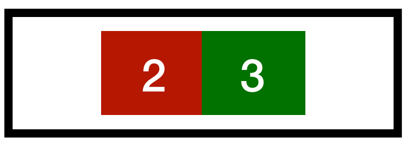
#hits: 0
#misses: 3



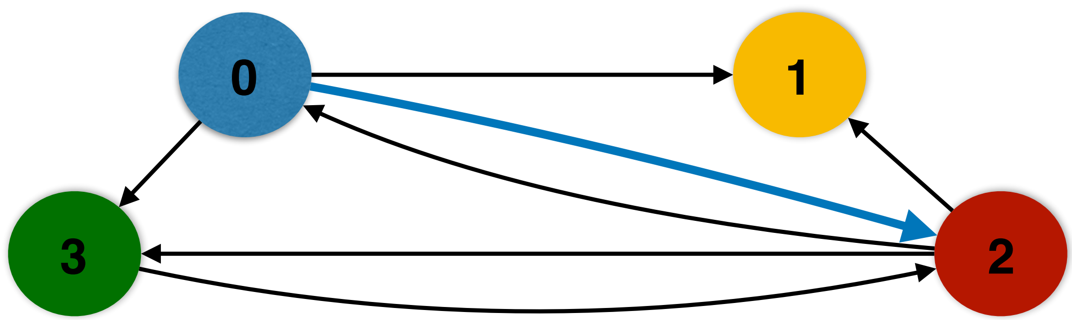
PageRank

```
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

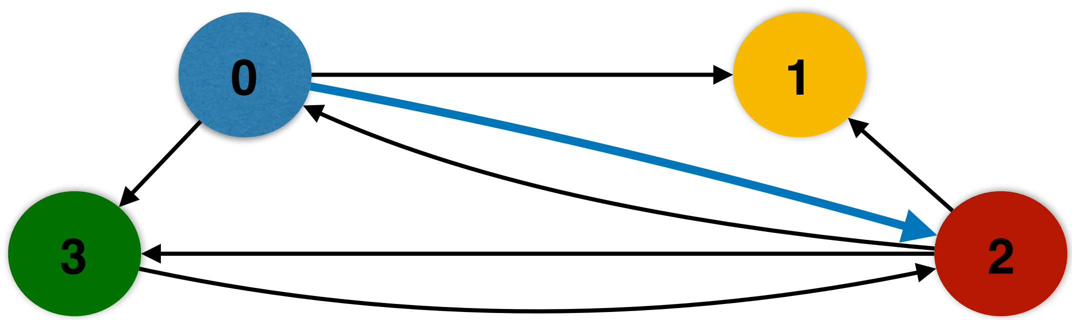


PageRank

```
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

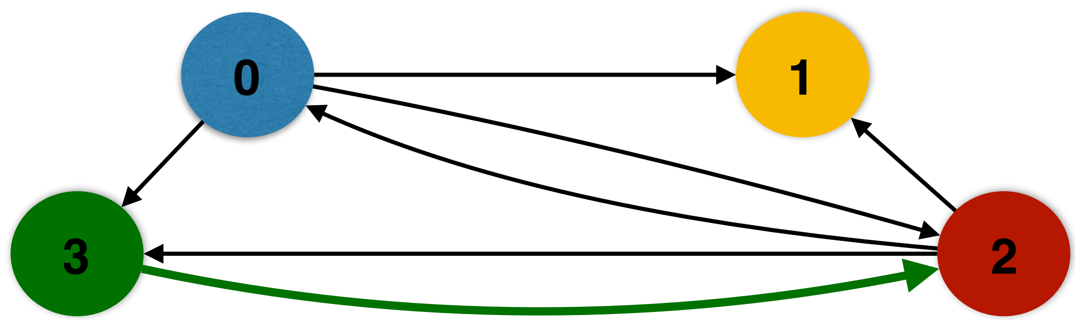


PageRank

```
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

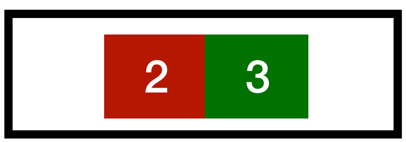
#hits: 0
#misses: 4



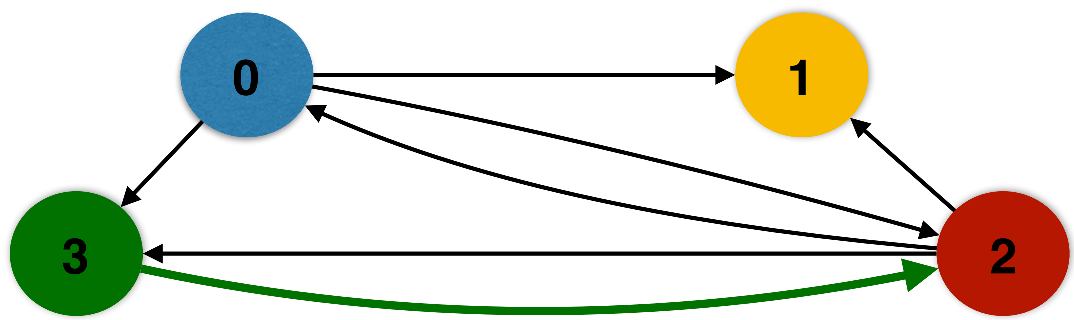
PageRank

```
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



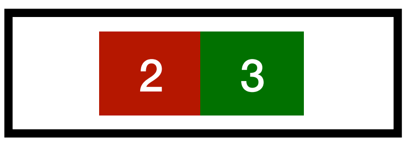
#hits: 0
#misses: 5



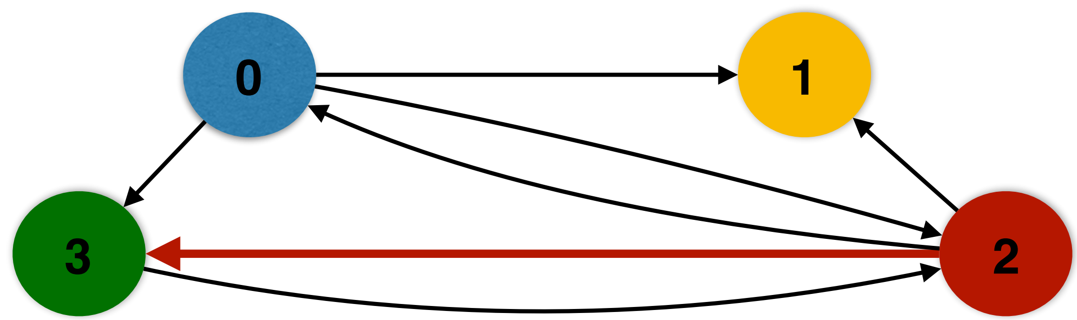
PageRank

```
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



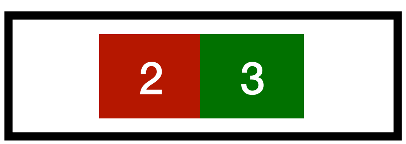
#hits: 0
#misses: 5



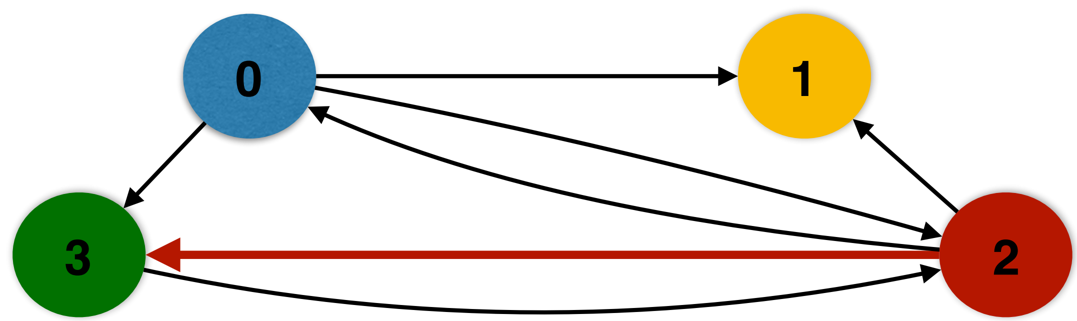
PageRank

```
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



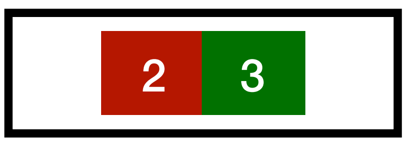
#hits: 1
#misses: 5



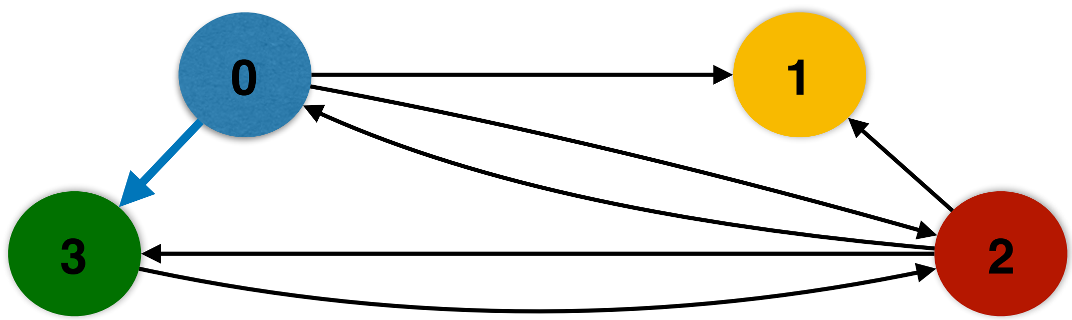
PageRank

```
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

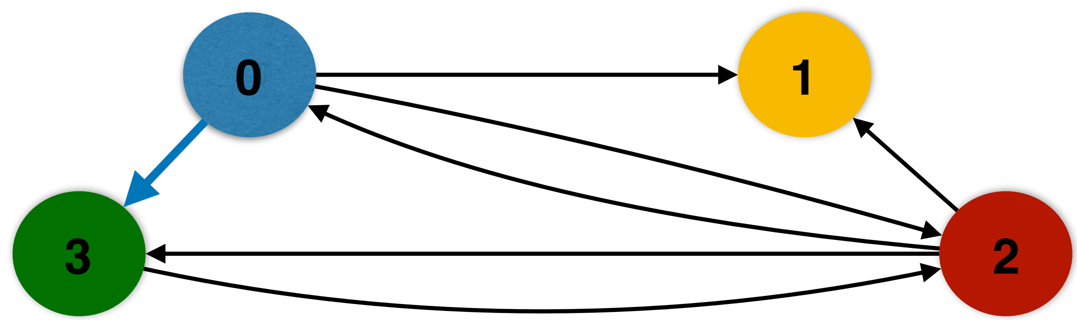
```
while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

0	1

#hits: 1
#misses: 6

0	1	2	3



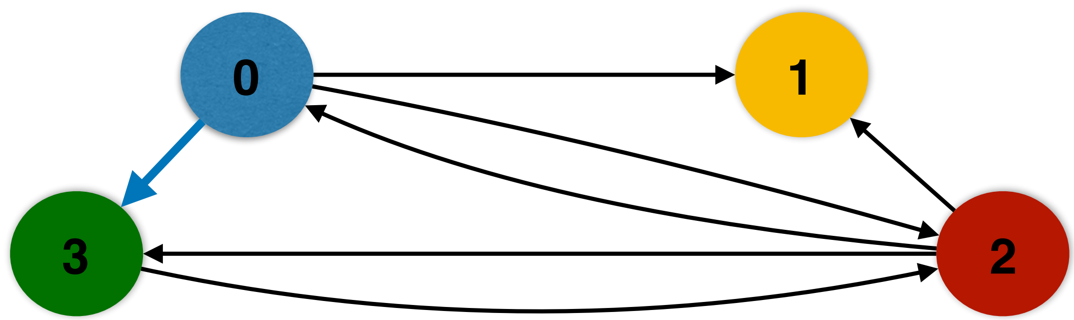
PageRank

```
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: 1
#misses: 6

A very high miss rate



PageRank

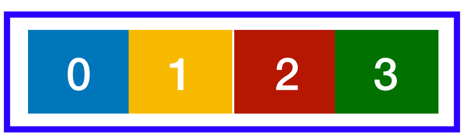
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

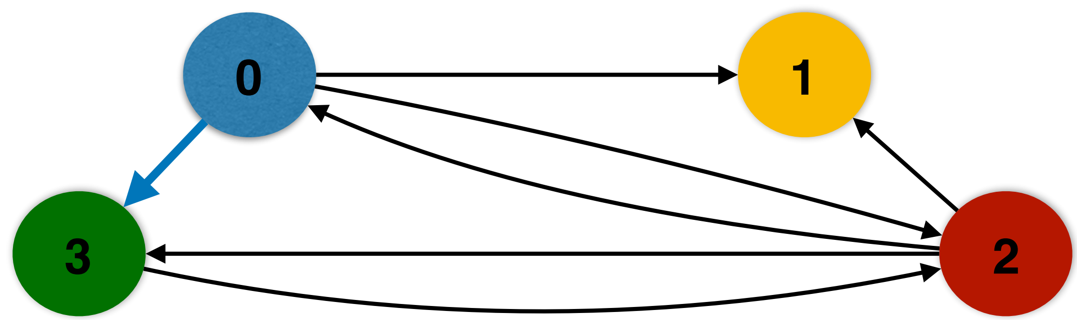
Cache

#hits: 1

#misses: 6

Working set larger than cache



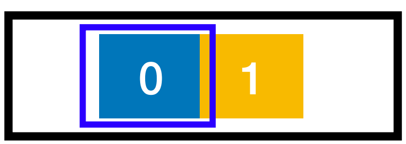


PageRank

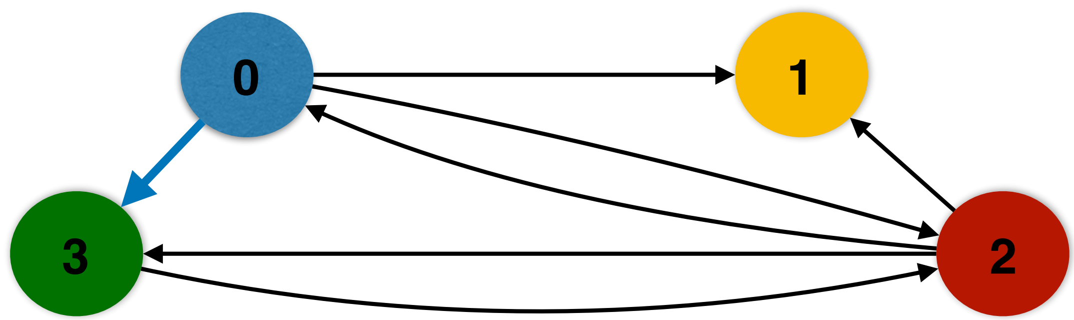
```
while ...
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

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 random
  - Often uses part of the cache line
  - Can not 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 random
    - Often uses part of the cache line
    - Can not benefit from hardware prefetching
    - TLB miss, DRAM row miss (hundreds of cycles)

Performance Bottleneck

- Working set much larger than cache size
- Access pattern is random
  - Often uses part of the cache line
  - Can not benefit from hardware prefetching
  - TLB miss, DRAM row miss (hundreds of cycles)

Performance Bottleneck

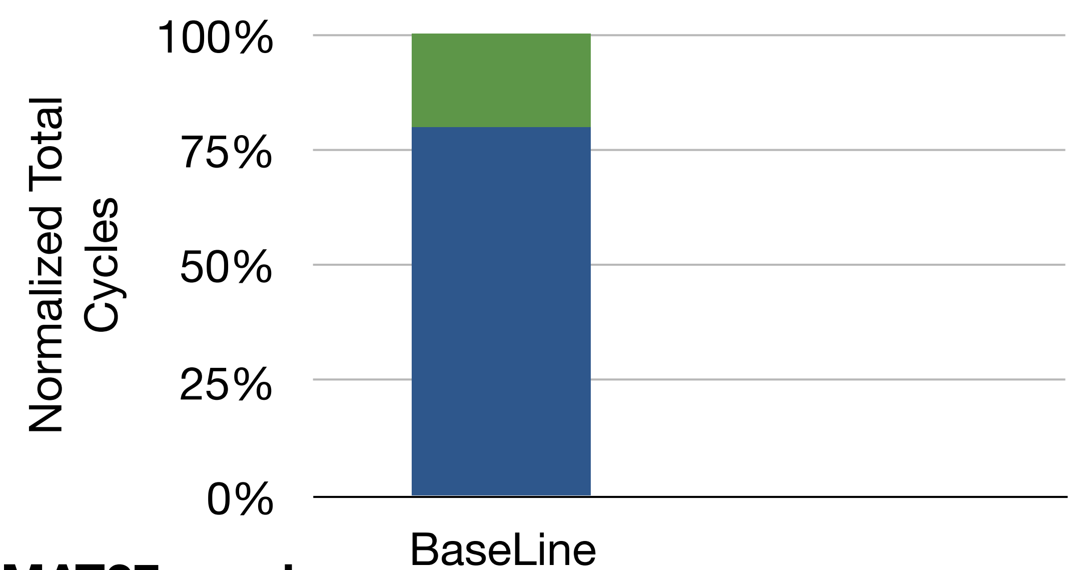
- Working set much larger than cache size
- Access pattern is random

Often only use 1/16 - 1/8 of a cache line in modern hardware

  - Often uses part of the cache line
  - Can not benefit from hardware prefetching
  - TLB miss, DRAM row miss (hundreds of cycles)

PageRank

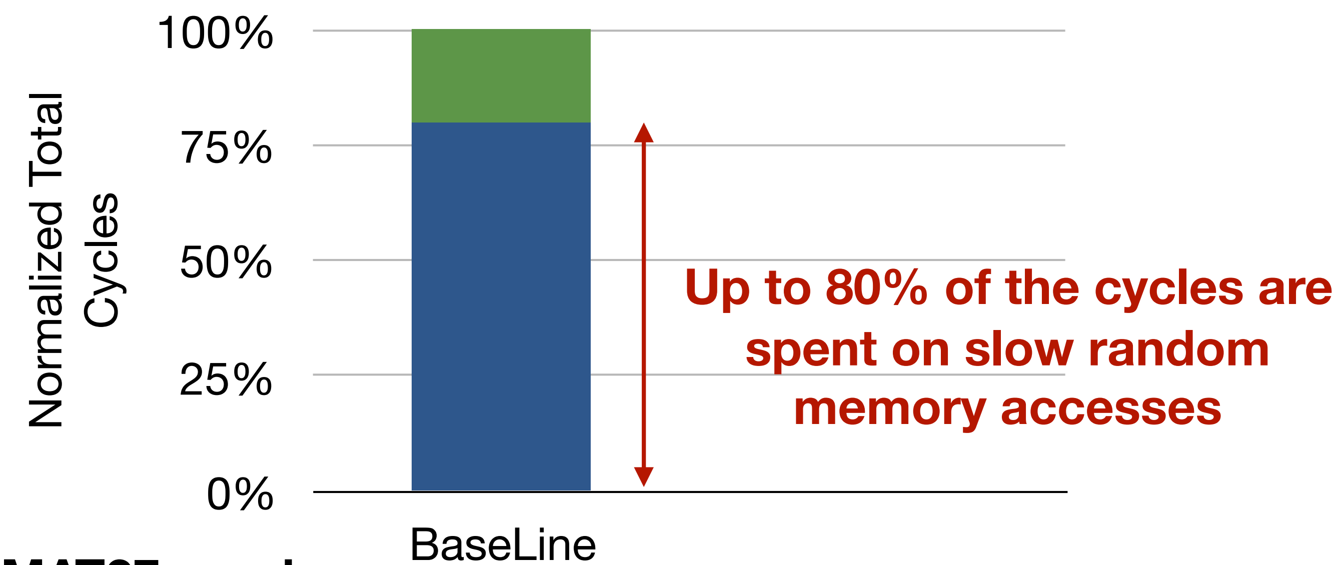
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



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
```

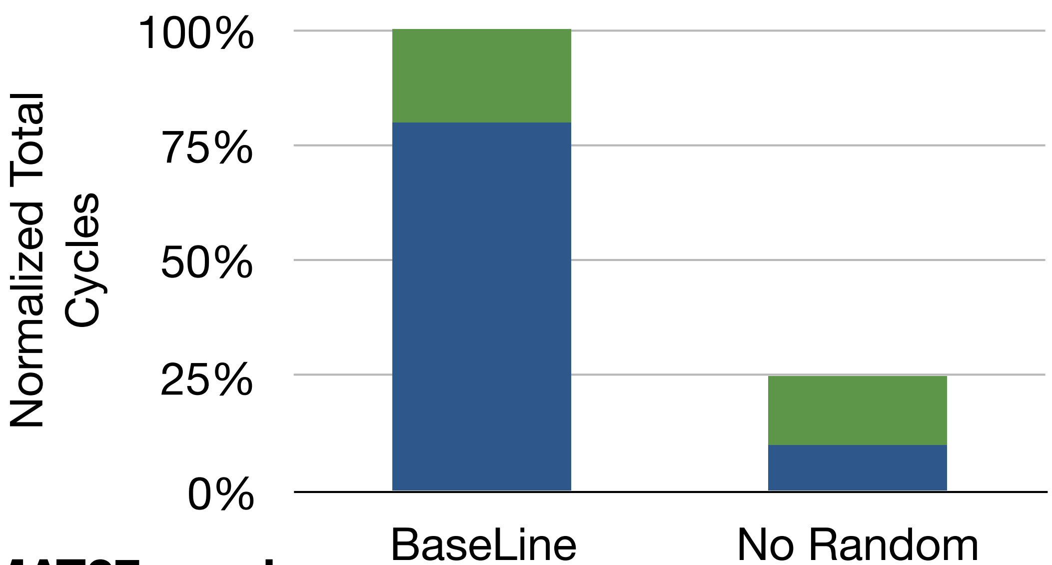


on RMAT27 graph

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)

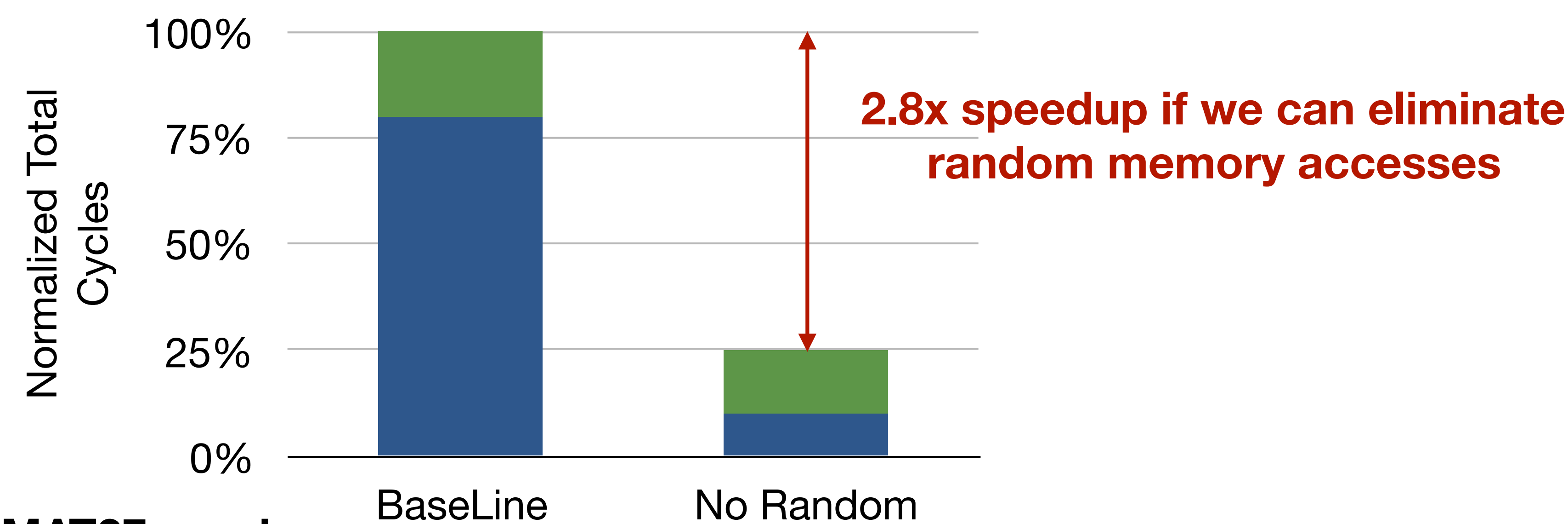


on RMAT27 graph

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
```

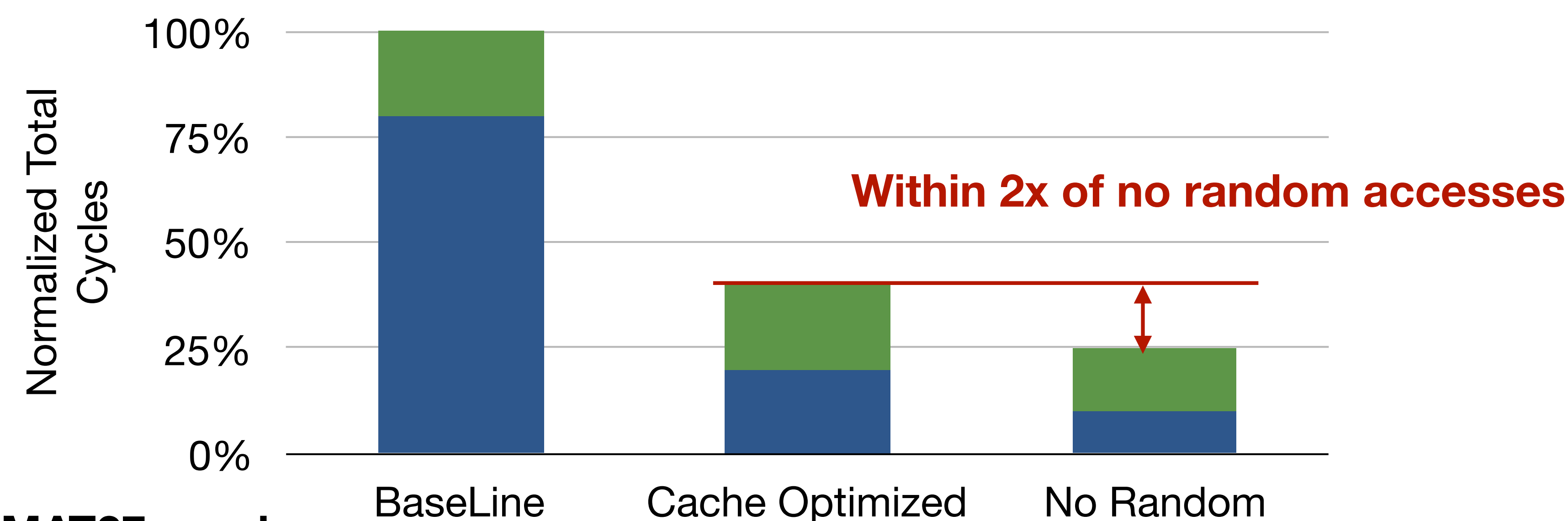
Removing Random Accesses (Incorrect)



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

- PageRank
- Frequency based Vertex Reordering
- Cache-aware Segmenting
- Evaluation

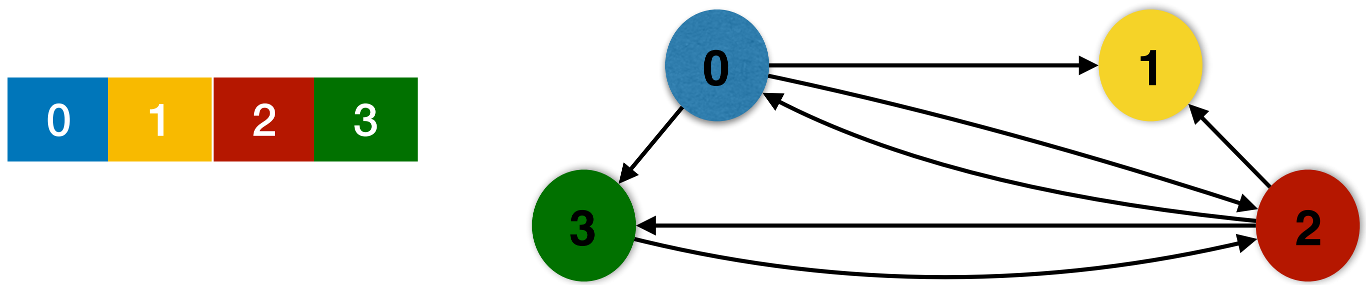
Frequency based Vertex Reordering

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

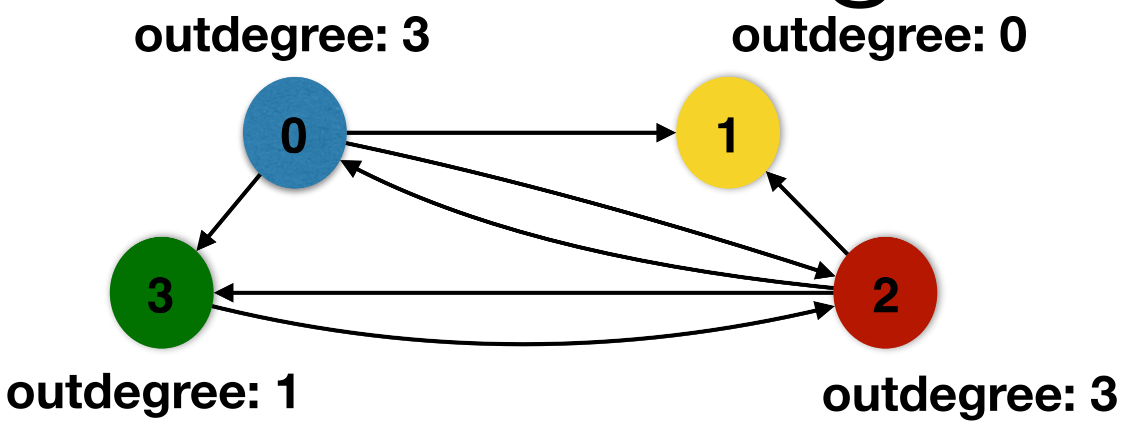
Frequency based Vertex Reordering

- 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

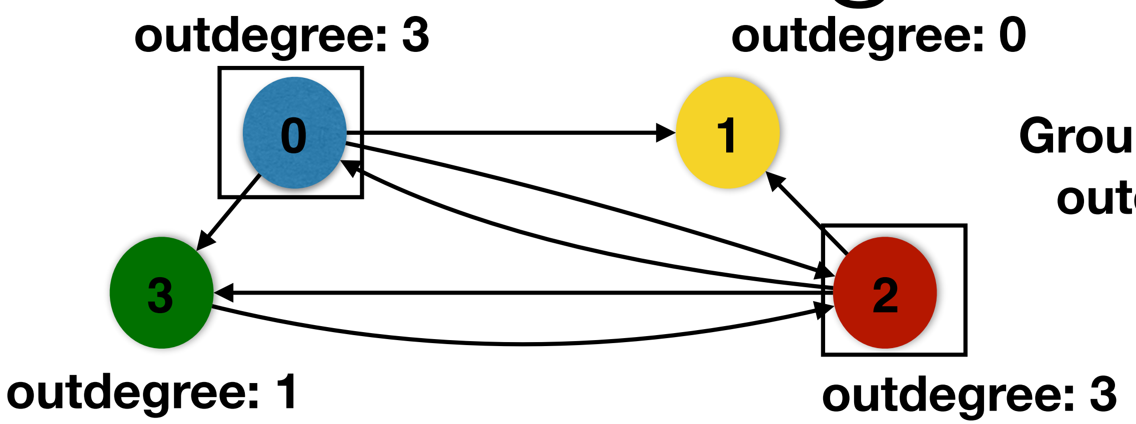
Frequency based Vertex Reordering



Frequency based Vertex Reordering

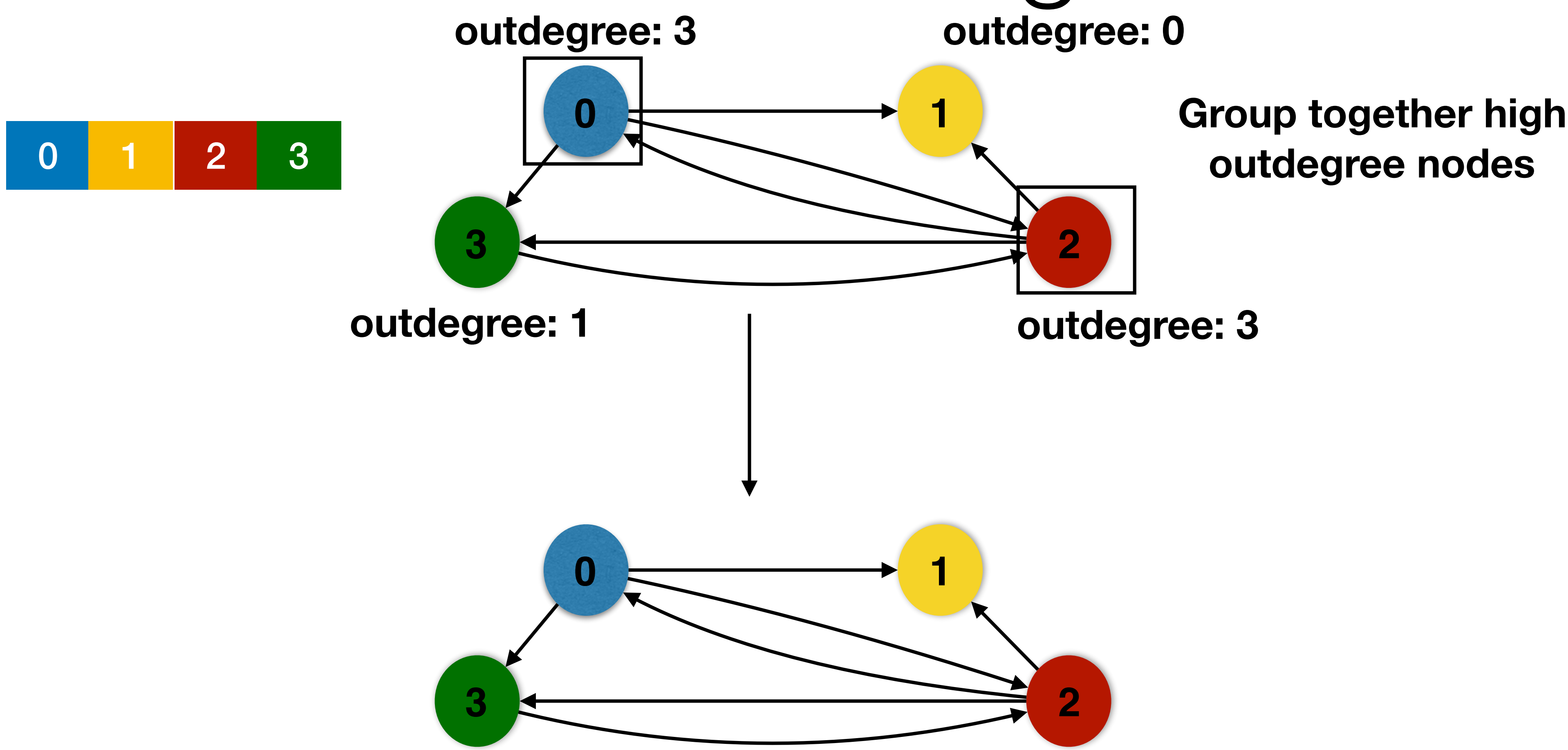


Frequency based Vertex Reordering

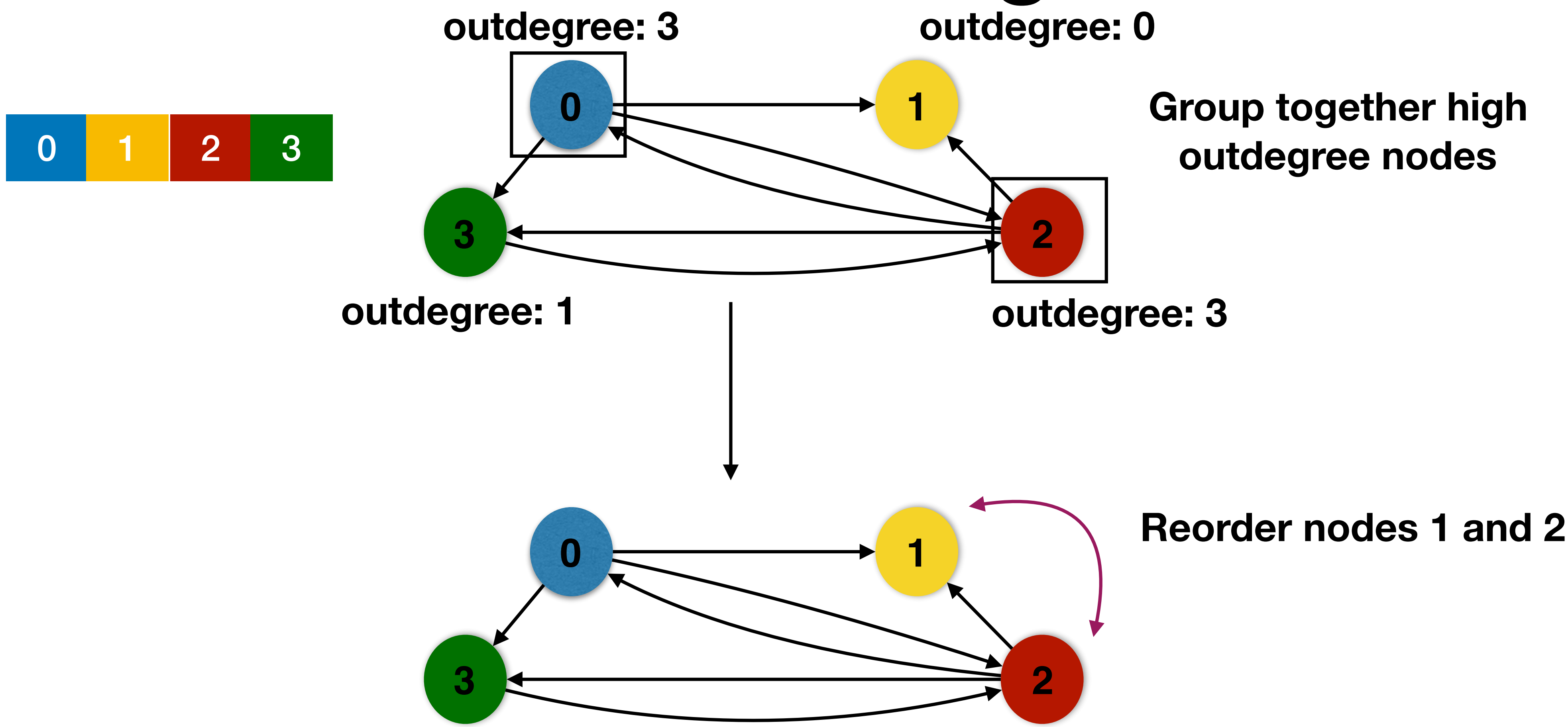


Group together high outdegree nodes

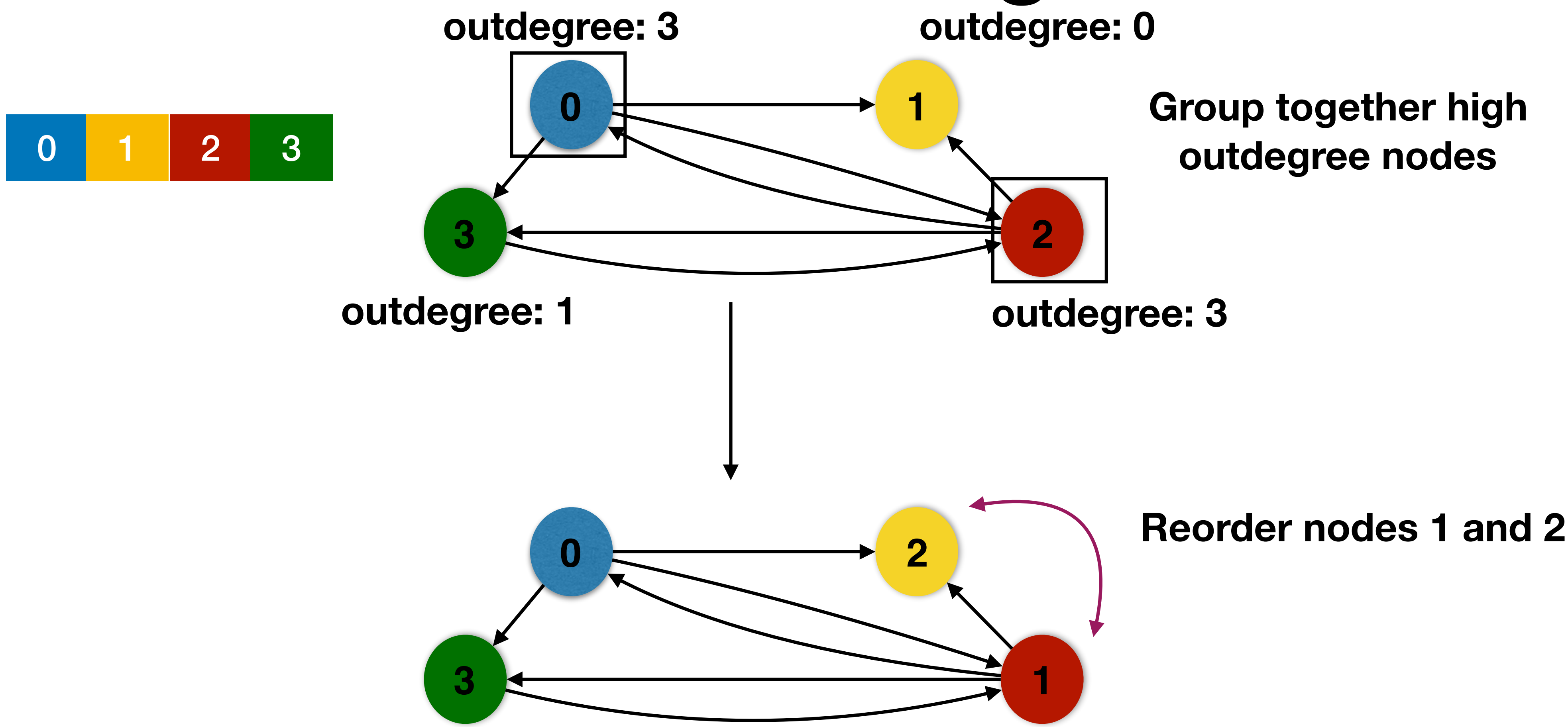
Frequency based Vertex Reordering



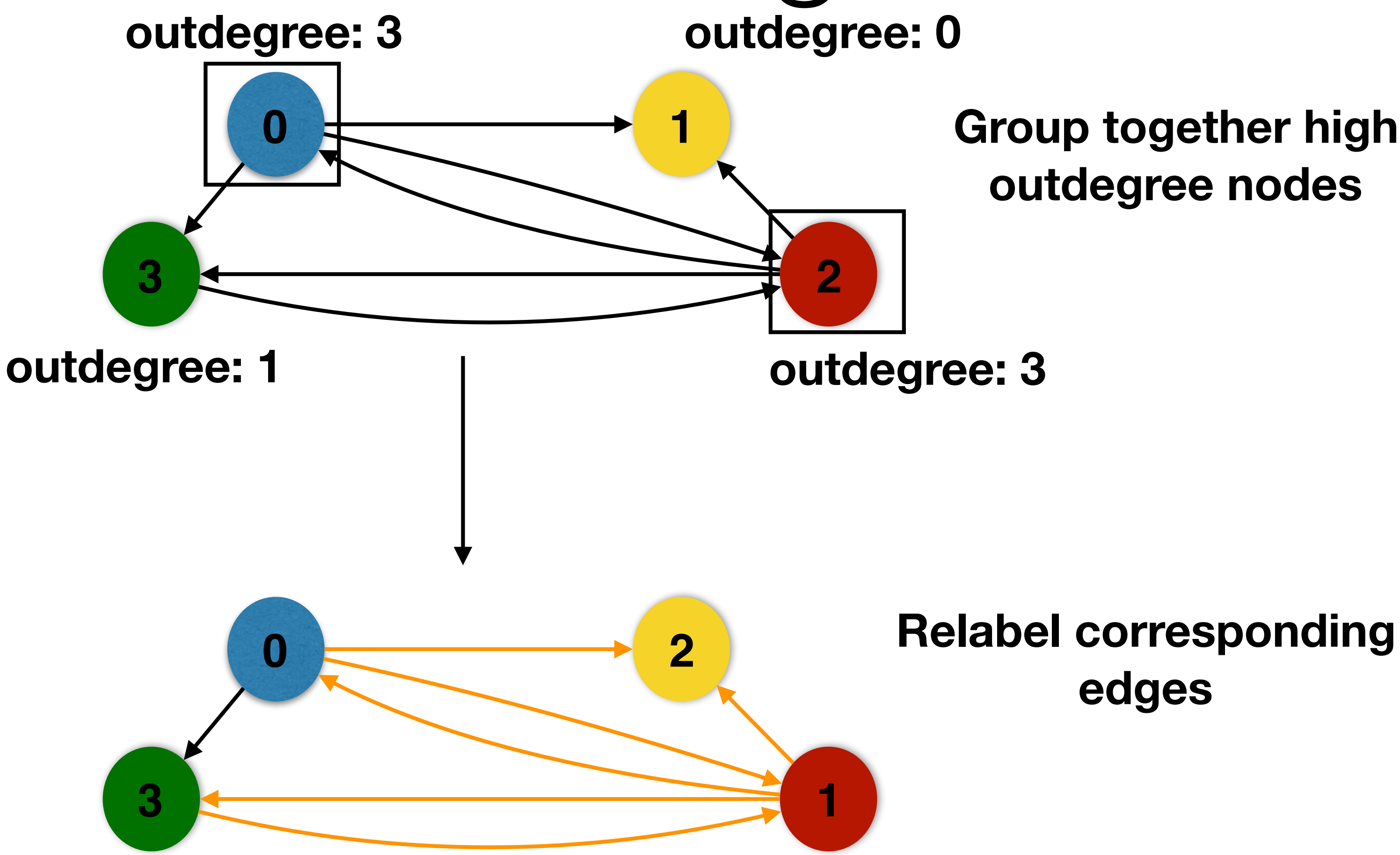
Frequency based Vertex Reordering



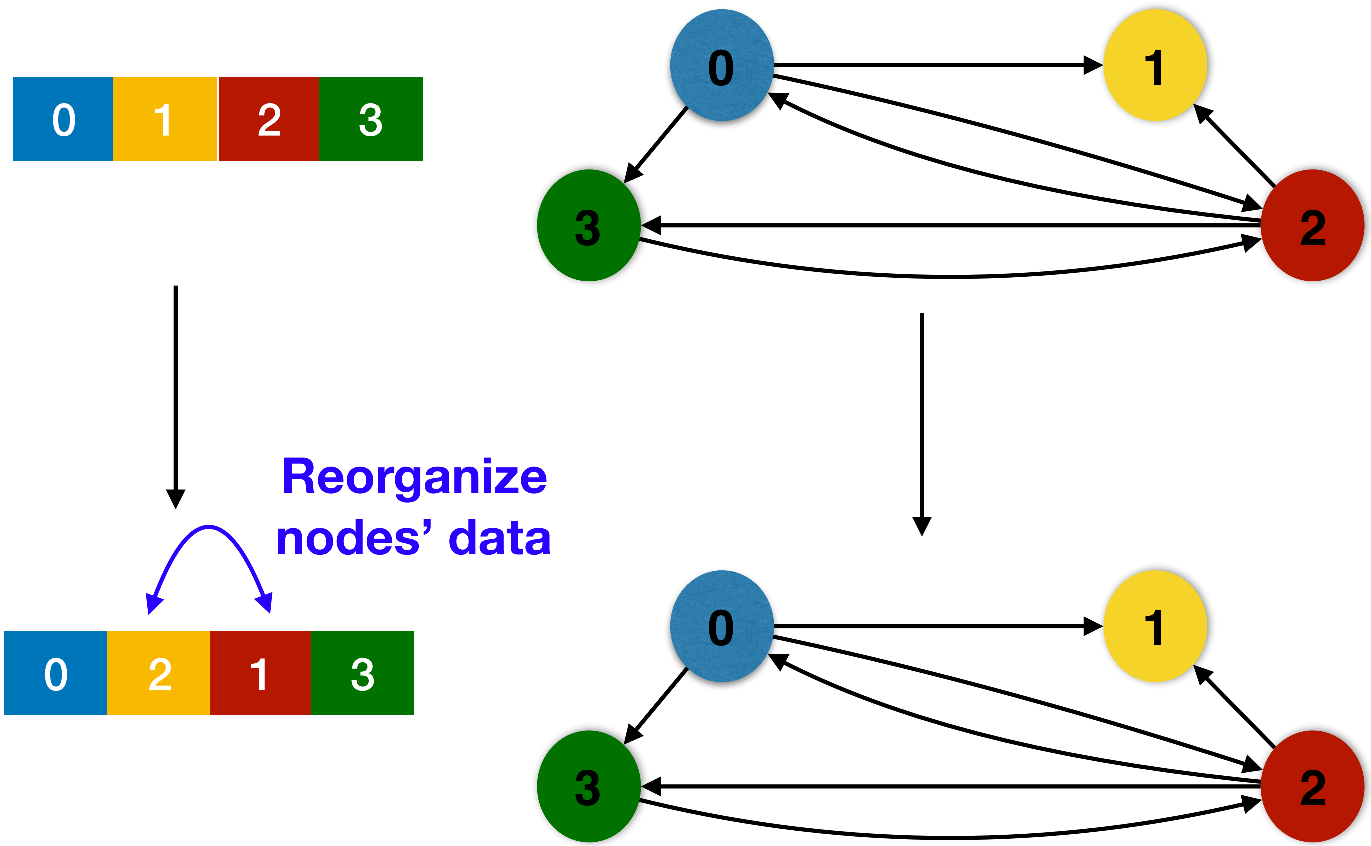
Frequency based Vertex Reordering



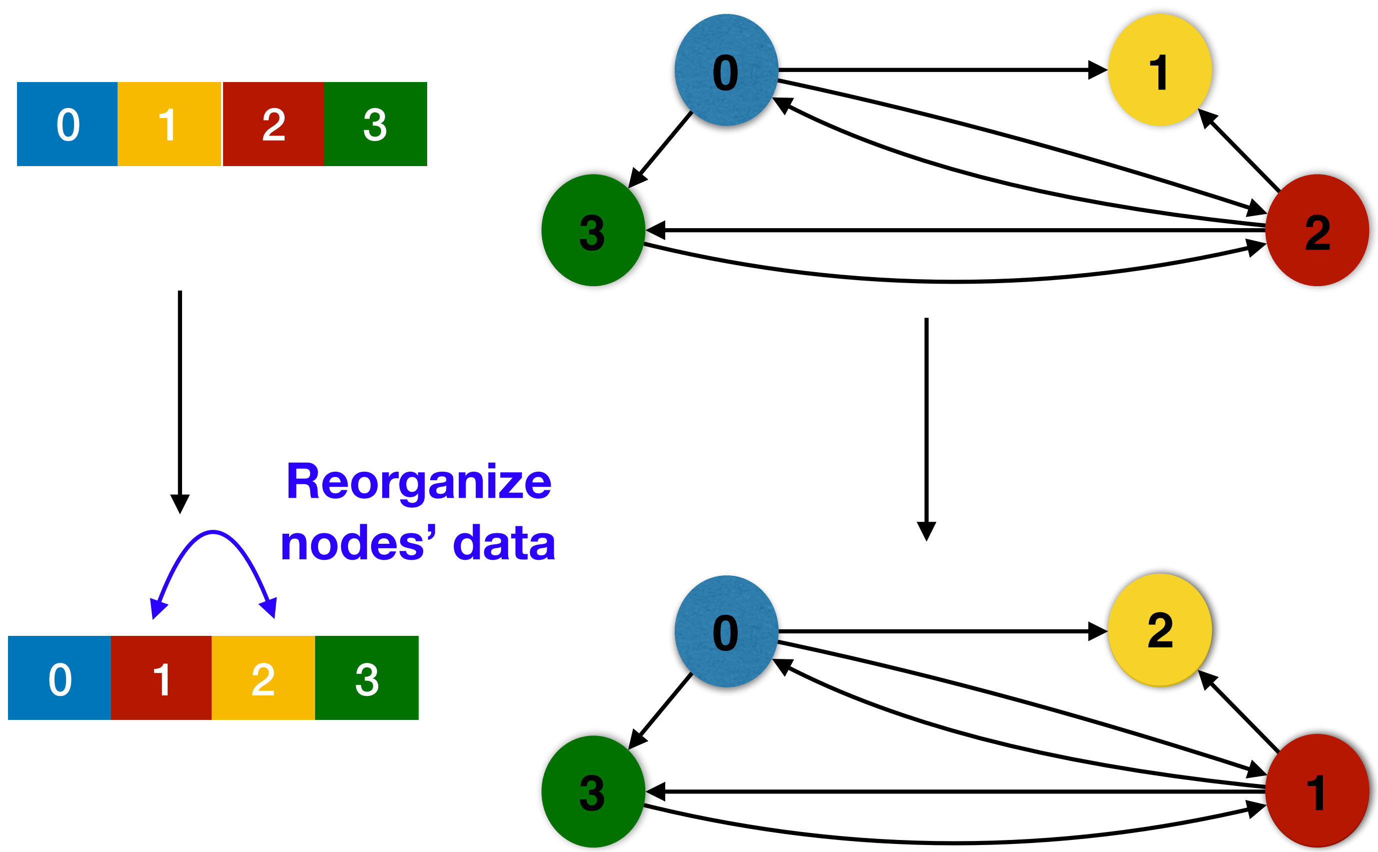
Frequency based Vertex Reordering



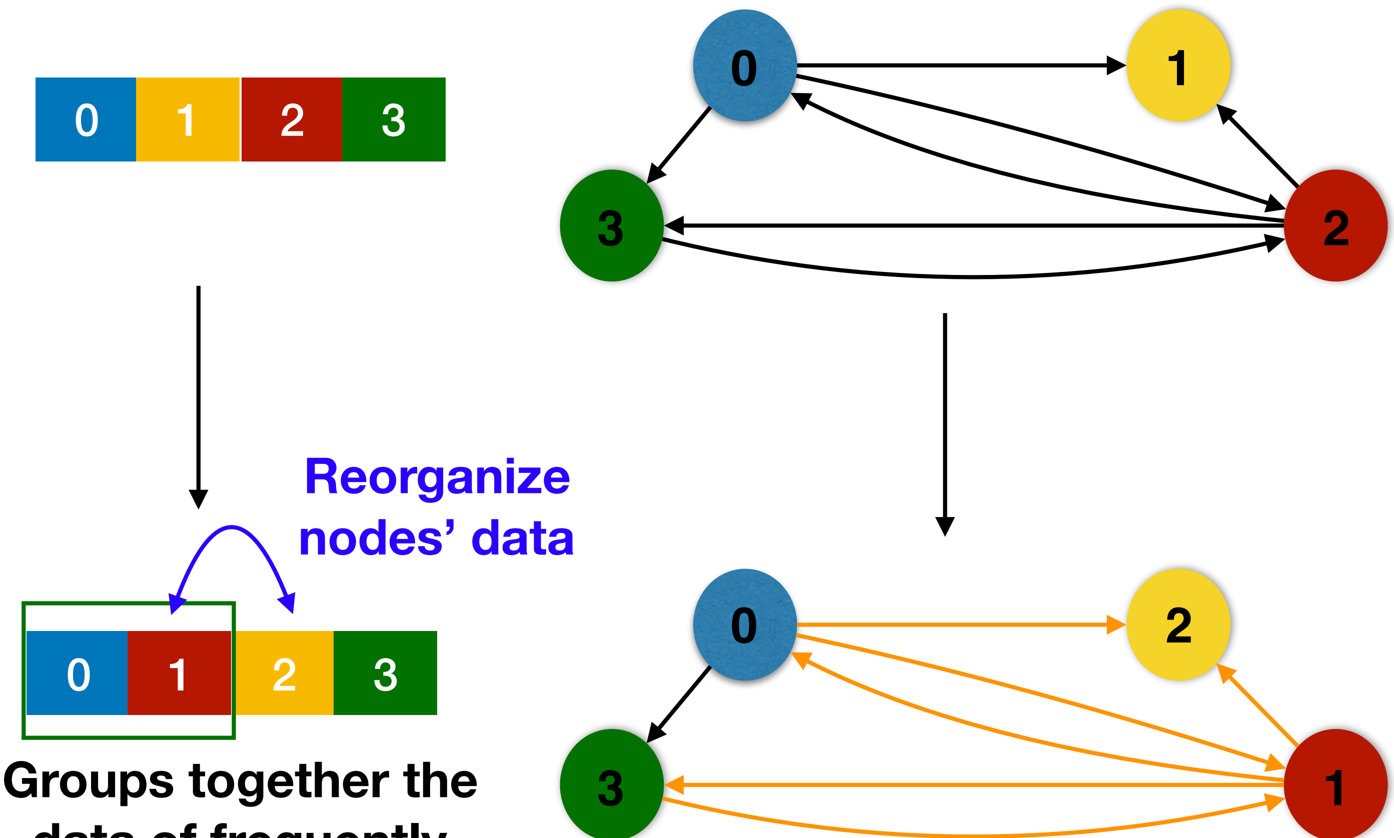
Frequency based Vertex Reordering



Frequency based Vertex Reordering



Frequency based Vertex Reordering



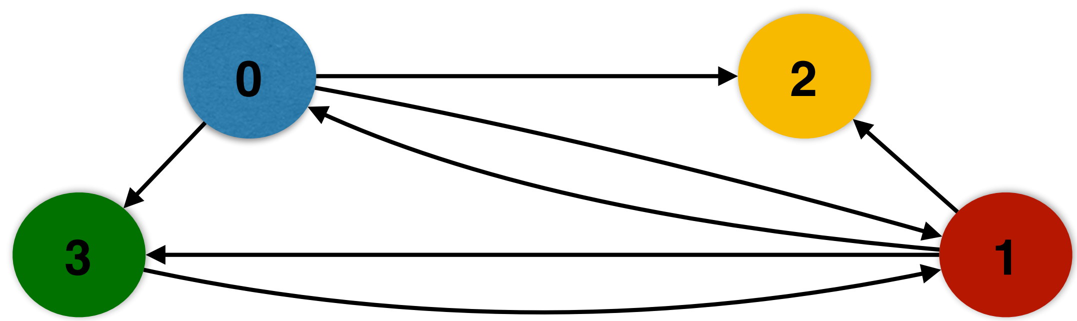
Groups together the data of frequently accessed nodes in one cache line

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



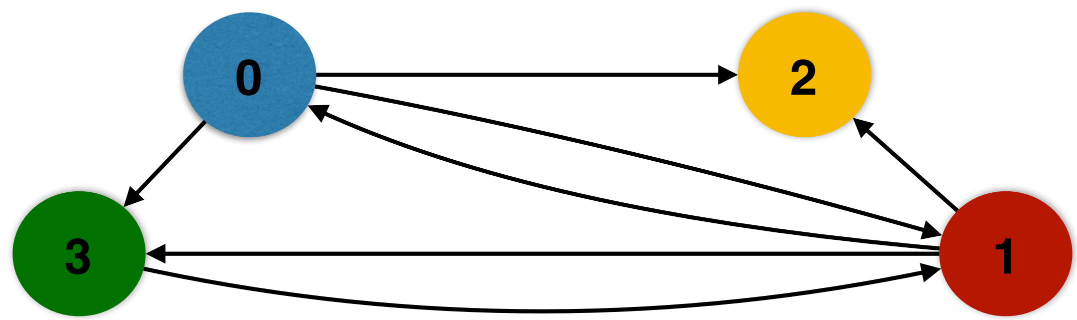
PageRank

```
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



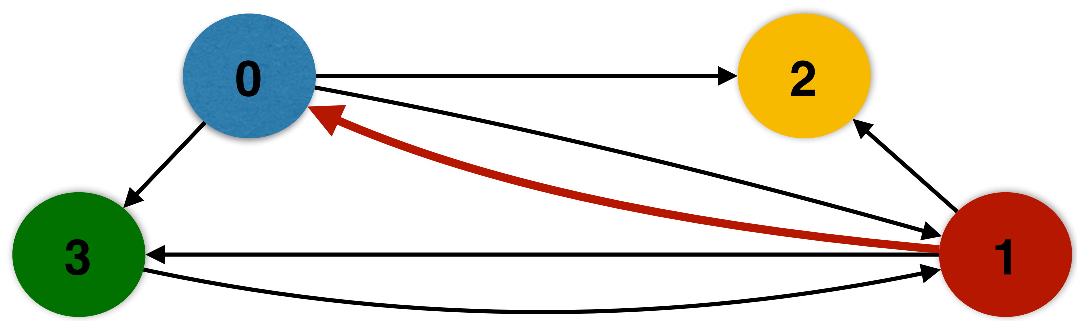
PageRank

```
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

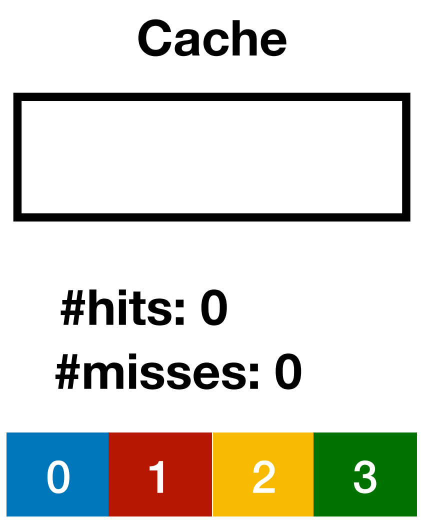
#hits: 0
#misses: 0

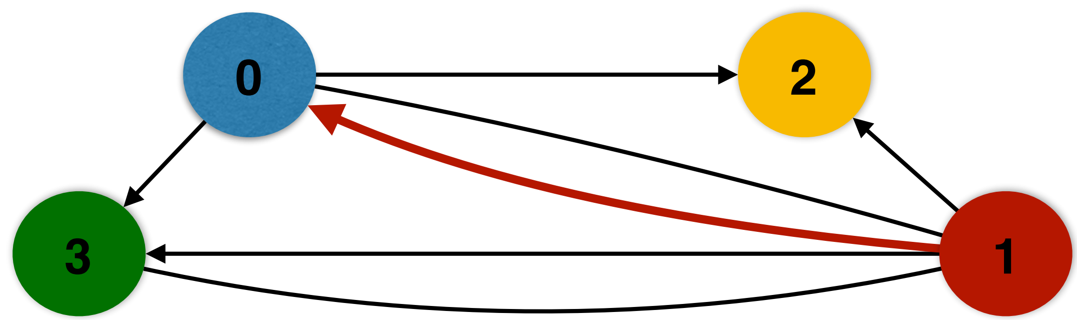
0	1	2	3



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
```

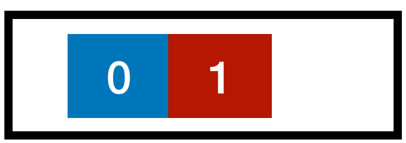




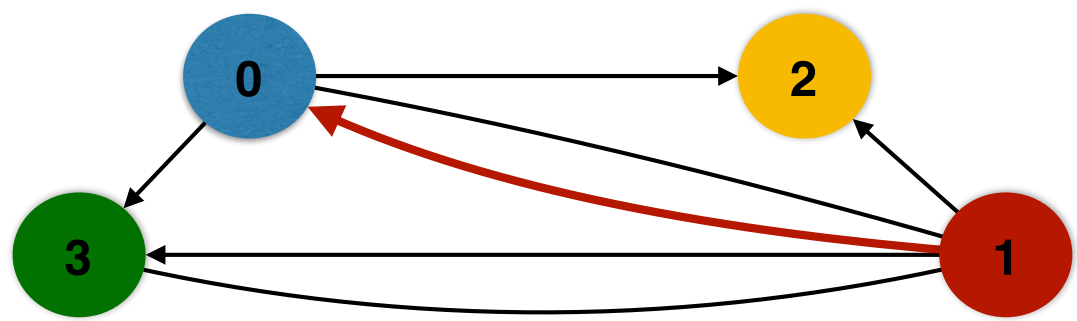
PageRank

```
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



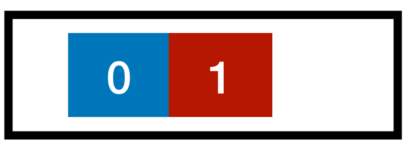
#hits: 0
#misses: 1



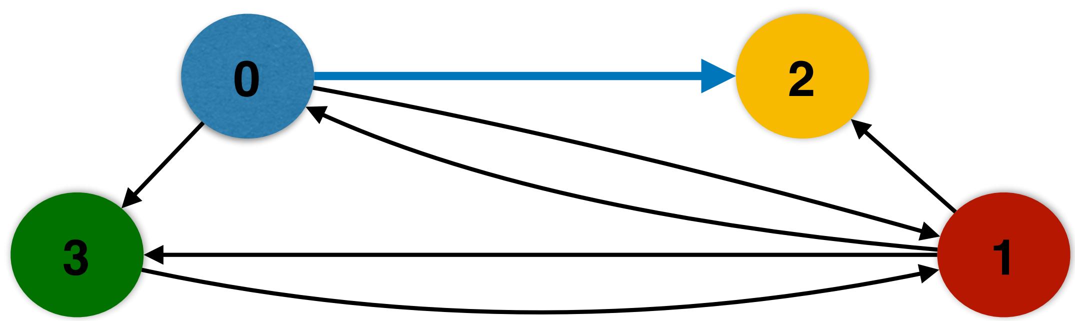
PageRank

```
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



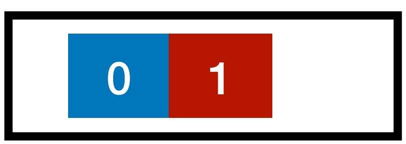
#hits: 0
#misses: 1



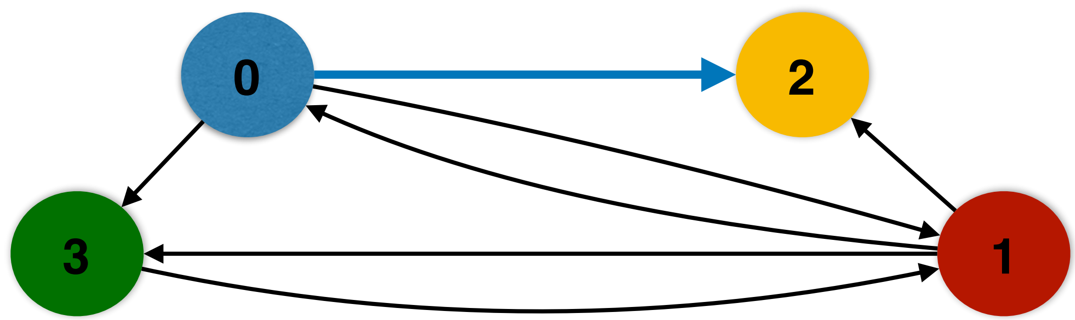
PageRank

```
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



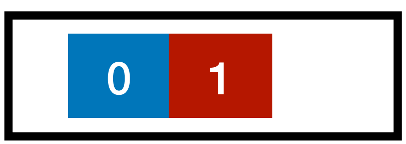
#hits: 1
#misses: 1



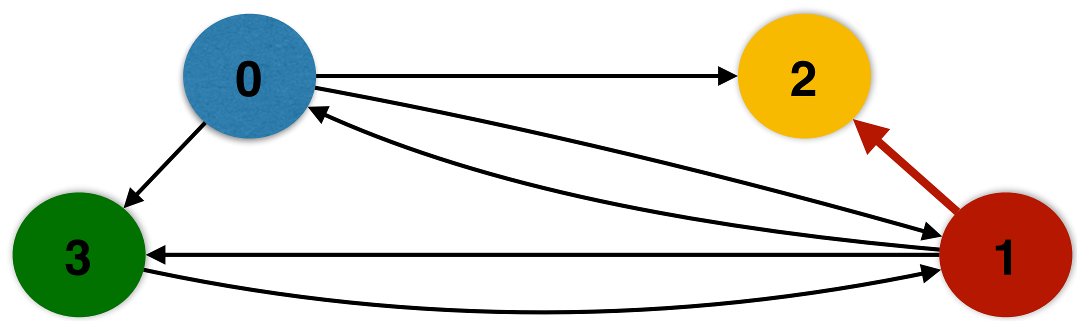
PageRank

```
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



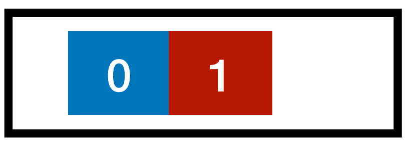
#hits: 1
#misses: 1



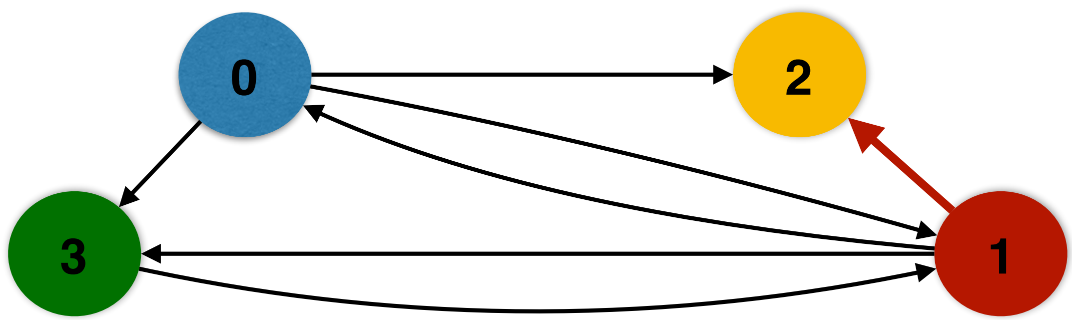
PageRank

```
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

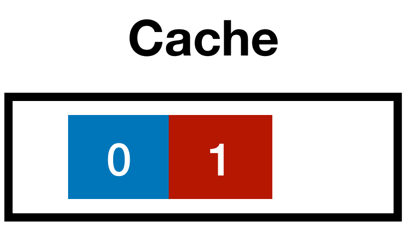


#hits: 2
#misses: 1

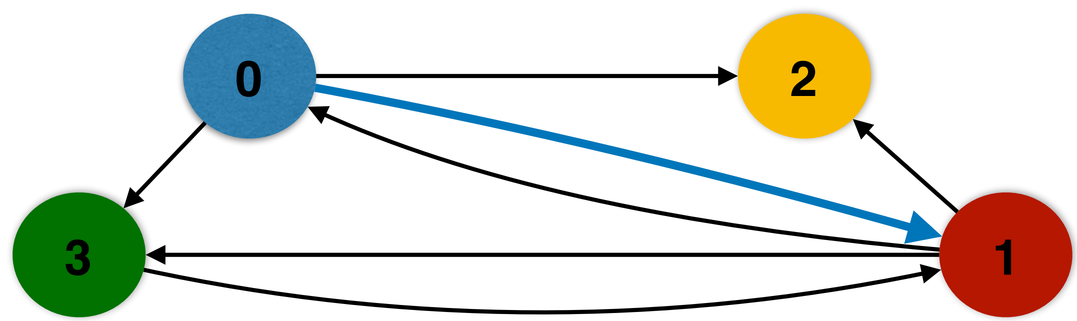


PageRank

```
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
```



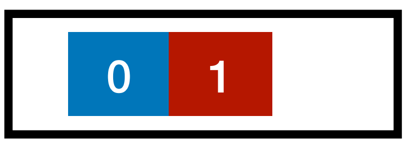
#hits: 2
#misses: 1



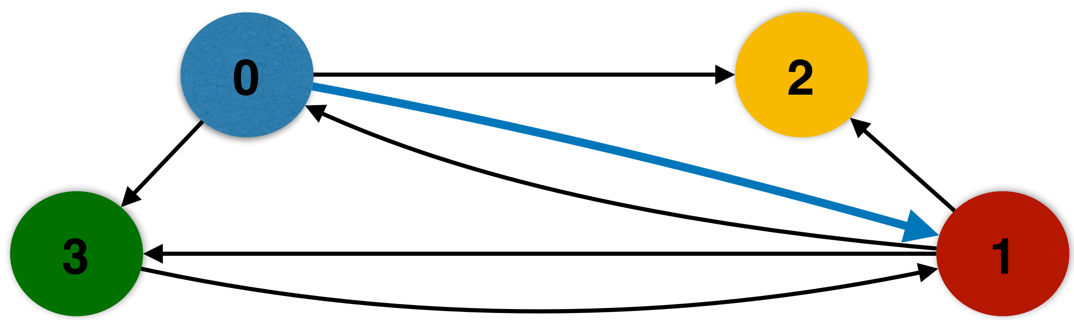
PageRank

```
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



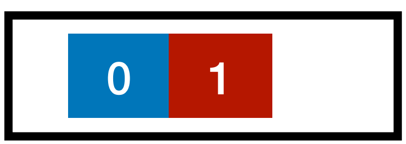
#hits: 3
#misses: 1



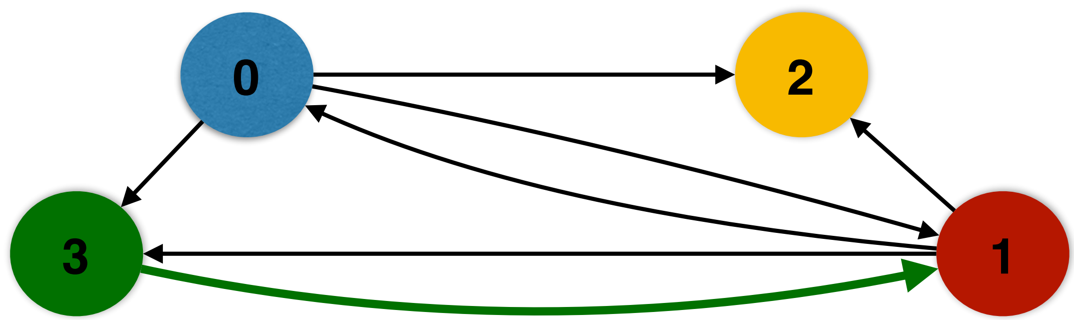
PageRank

```
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: 3
#misses: 1

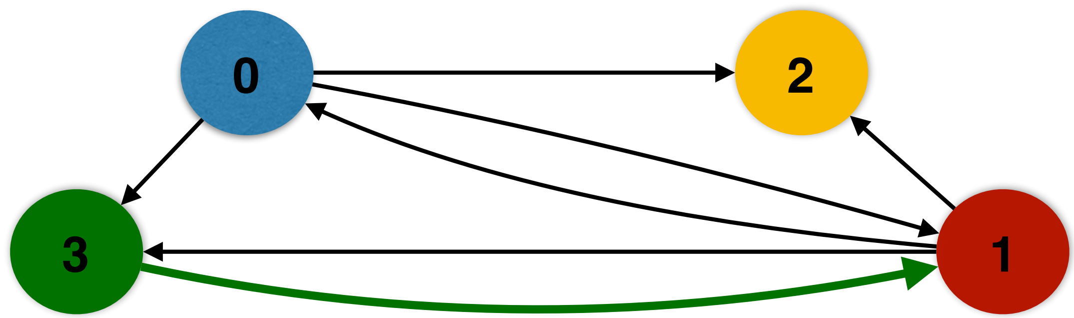


PageRank

```
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: 3
#misses: 2

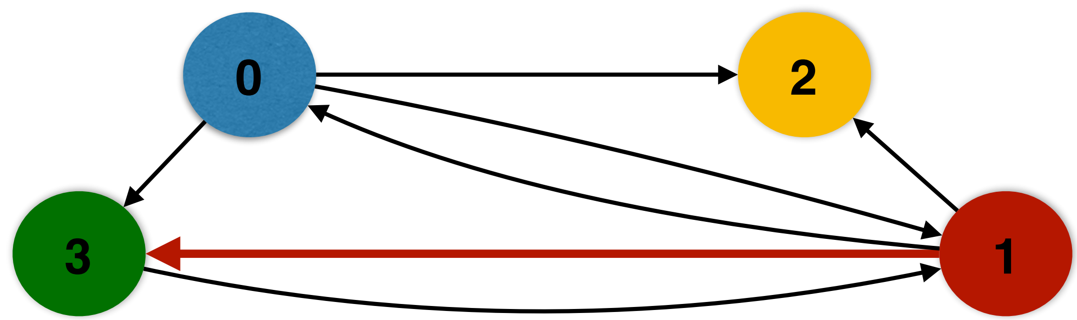


PageRank

```
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

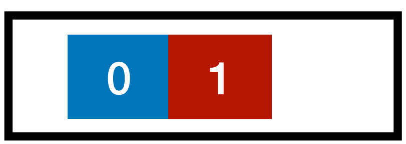
#hits: 3
#misses: 2



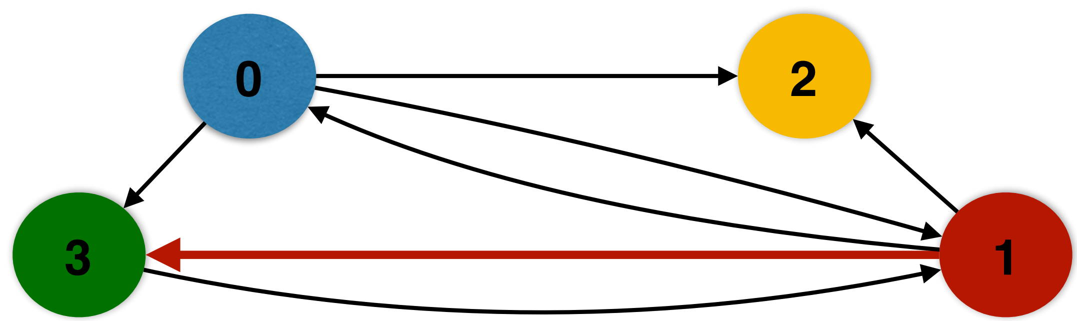
PageRank

```
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



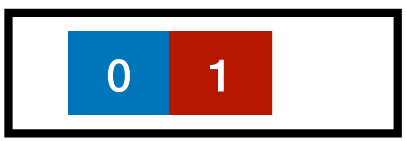
#hits: 3
#misses: 3



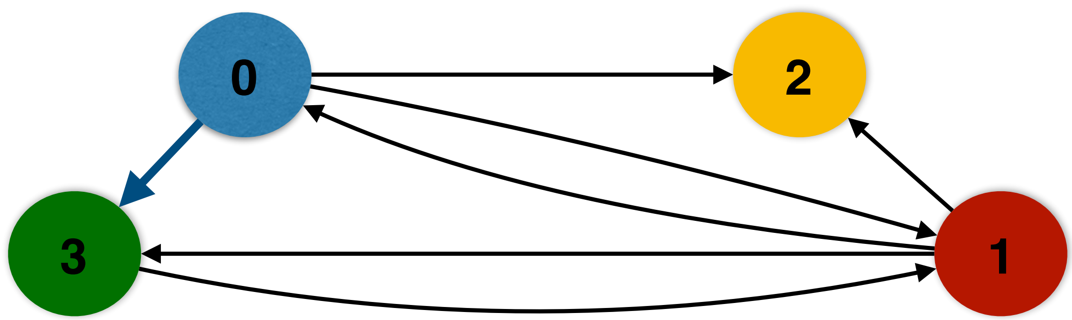
PageRank

```
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



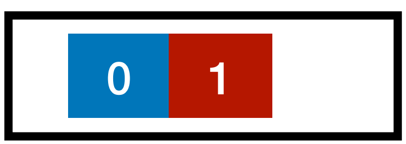
#hits: 3
#misses: 3



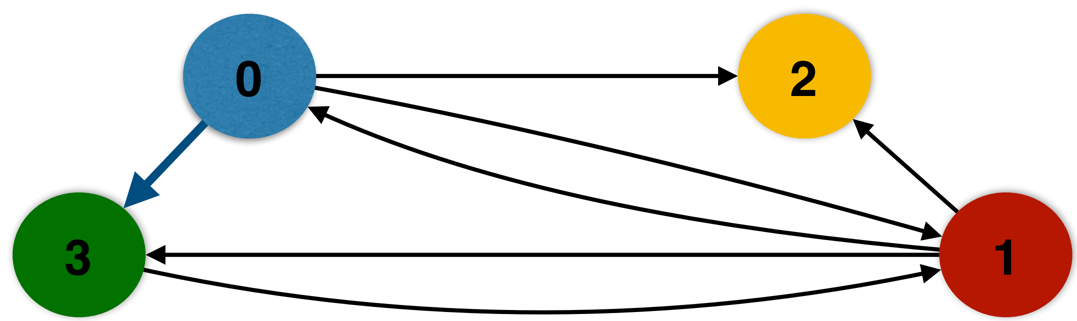
PageRank

```
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

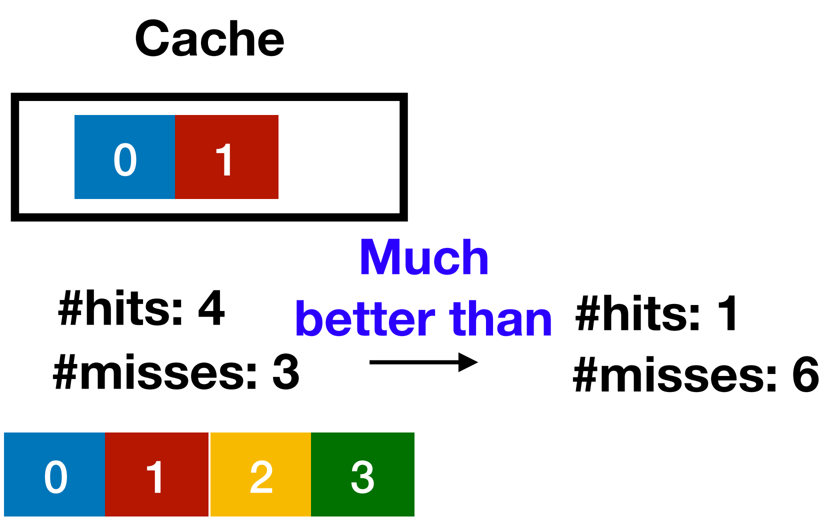


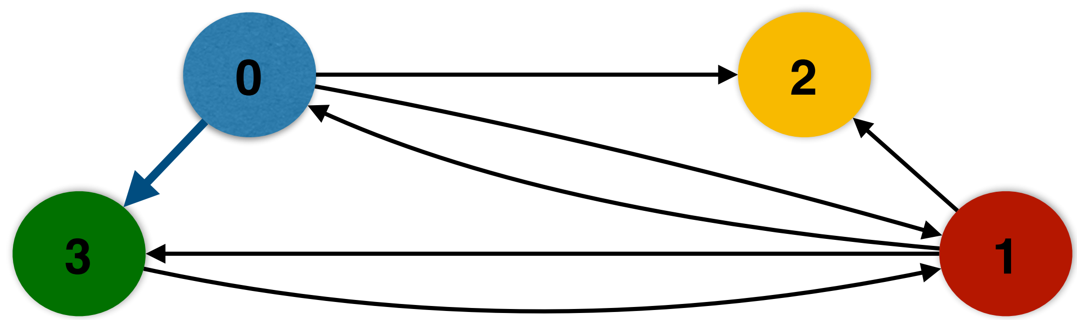
#hits: 4
#misses: 3



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
```

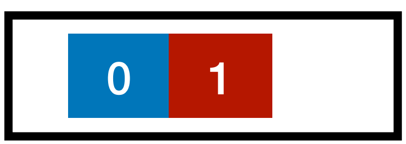




PageRank

```
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
```

Cache

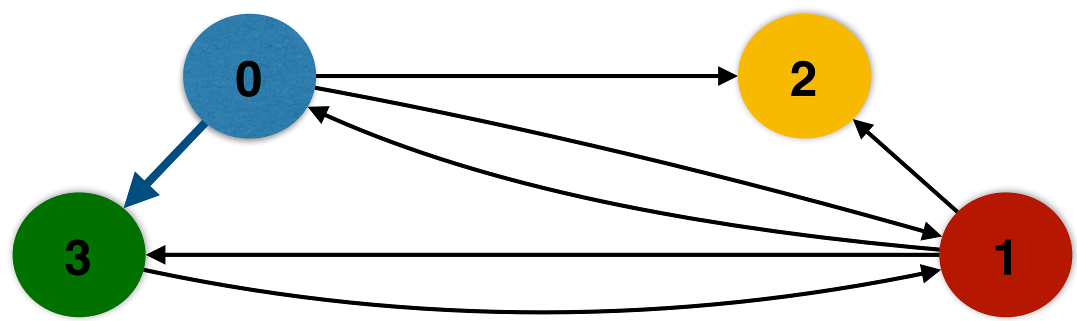


#hits: 4
#misses: 3

→

#hits: 1
#misses: 6

Better cache line utilization



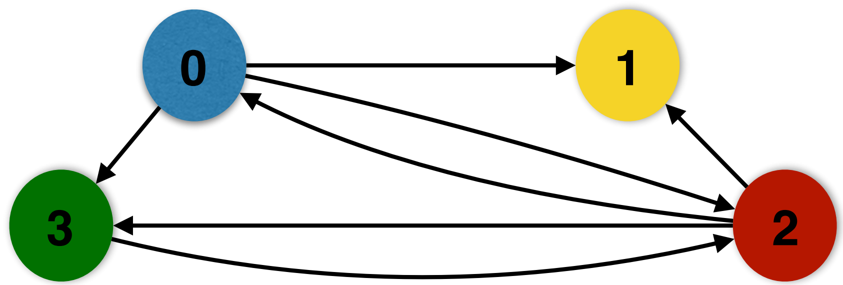
Outline

- PageRank
- Frequency based Vertex Reordering
- Cache-aware Segmenting
- Evaluation

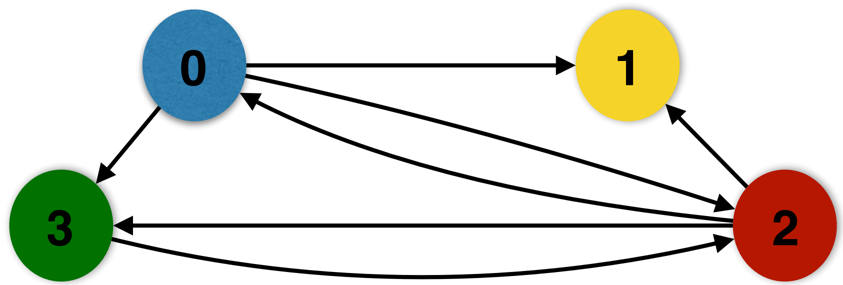
Cache-aware Segmenting

- 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

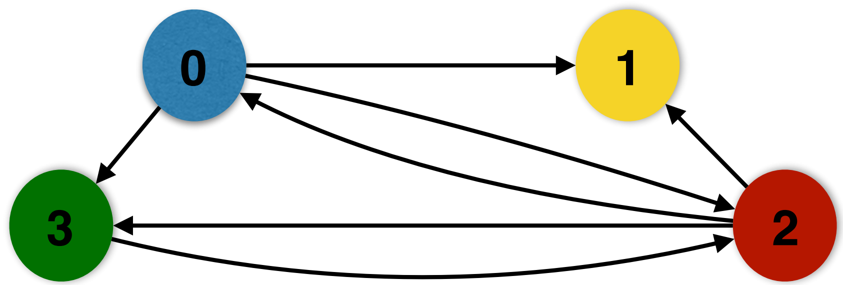


Graph Partitioning



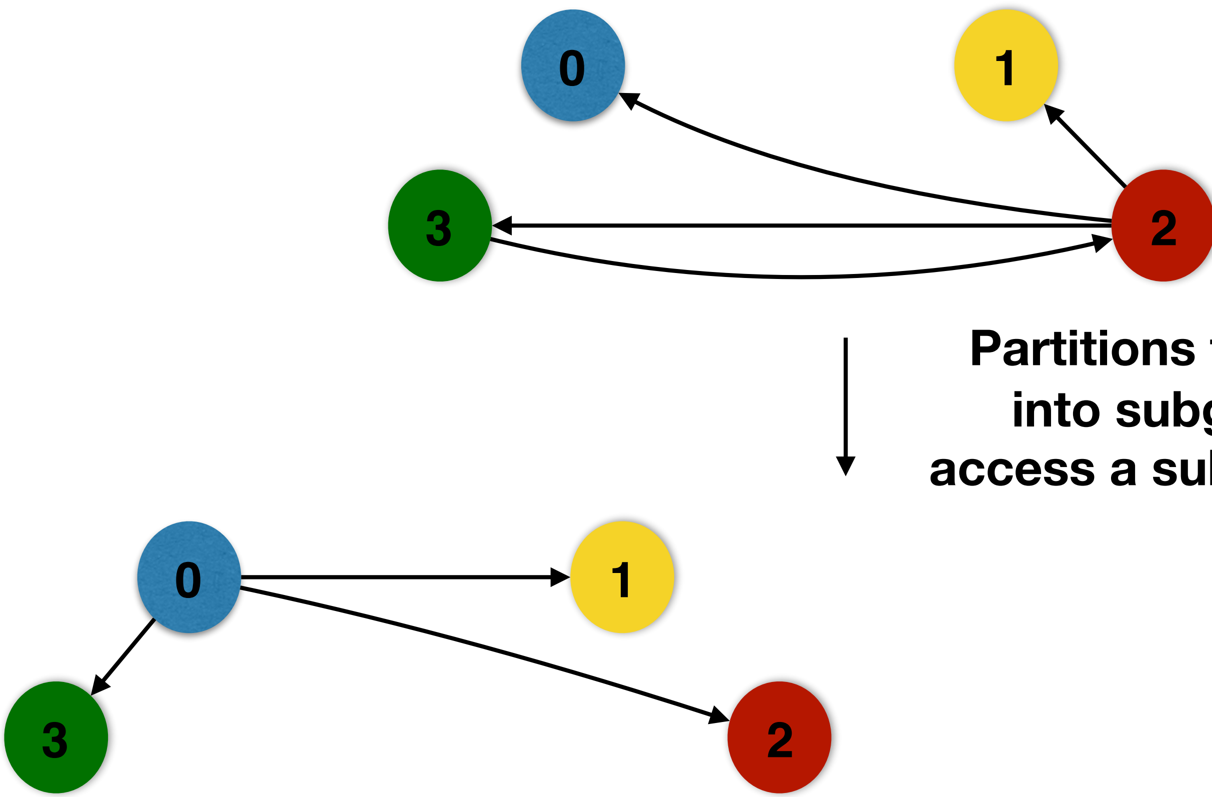
Partitions the original graph into subgraphs that only access a subset of nodes' data

Graph Partitioning



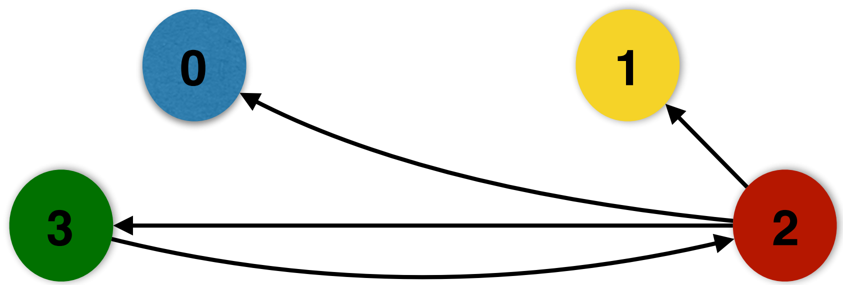
Partitions the original graph into subgraphs that only access a subset of nodes' data

Graph Partitioning

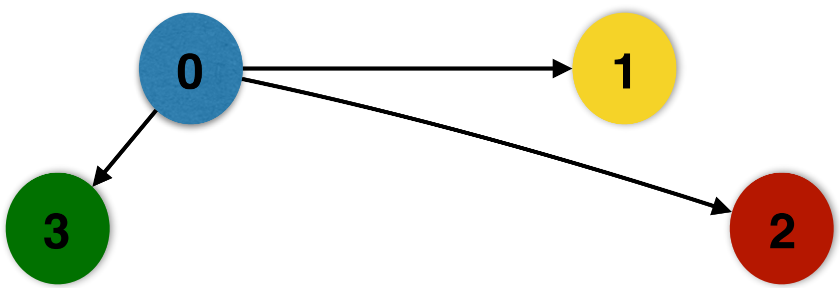


Partitions the original graph into subgraphs that only access a subset of nodes' data

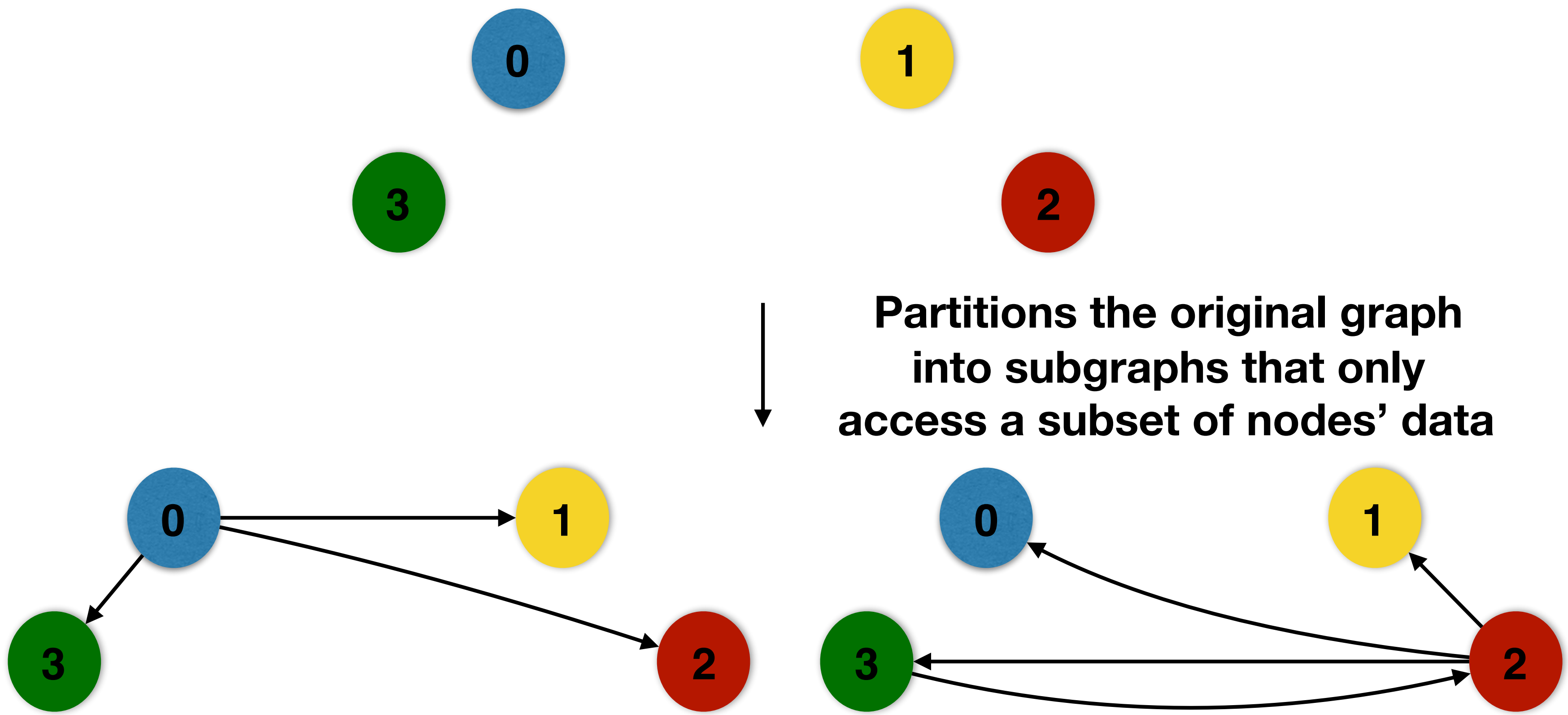
Graph Partitioning



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

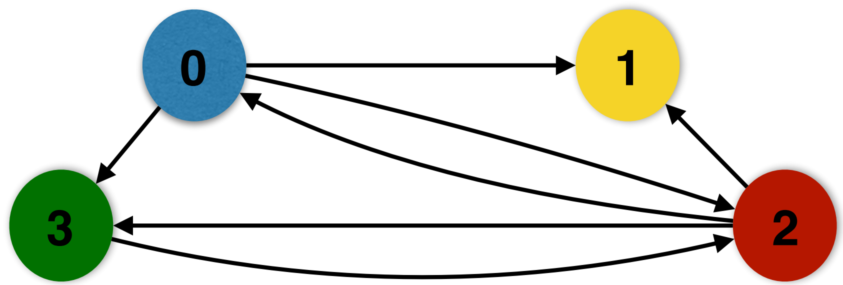


Graph Partitioning

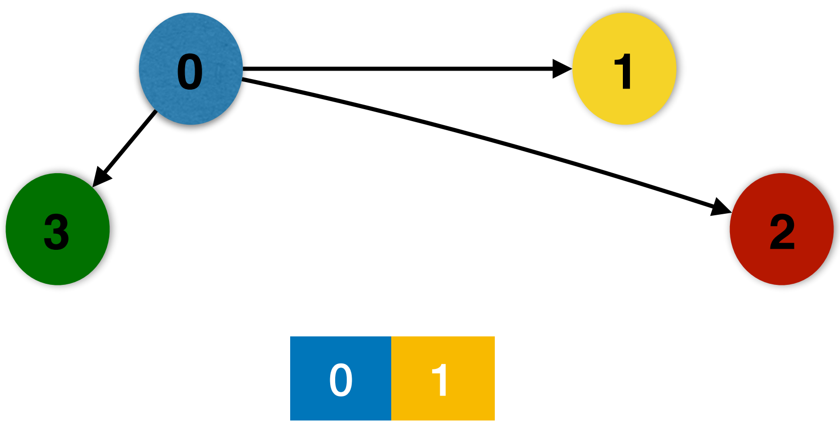


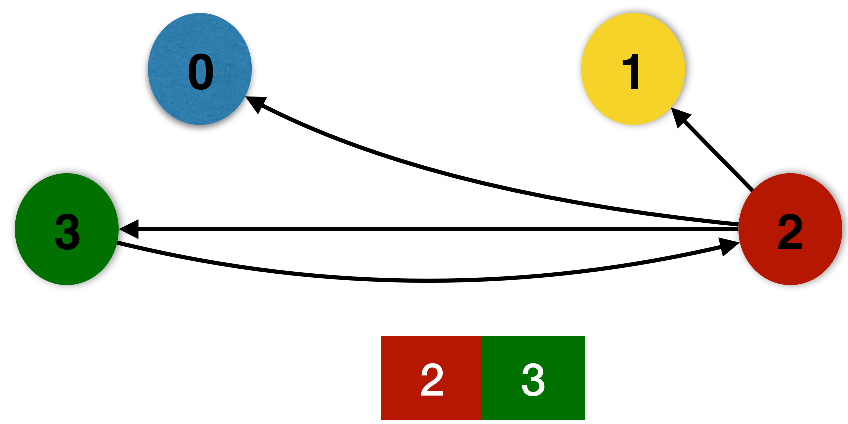
Partitions the original graph into subgraphs that only access a subset of nodes' data

Graph Partitioning



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

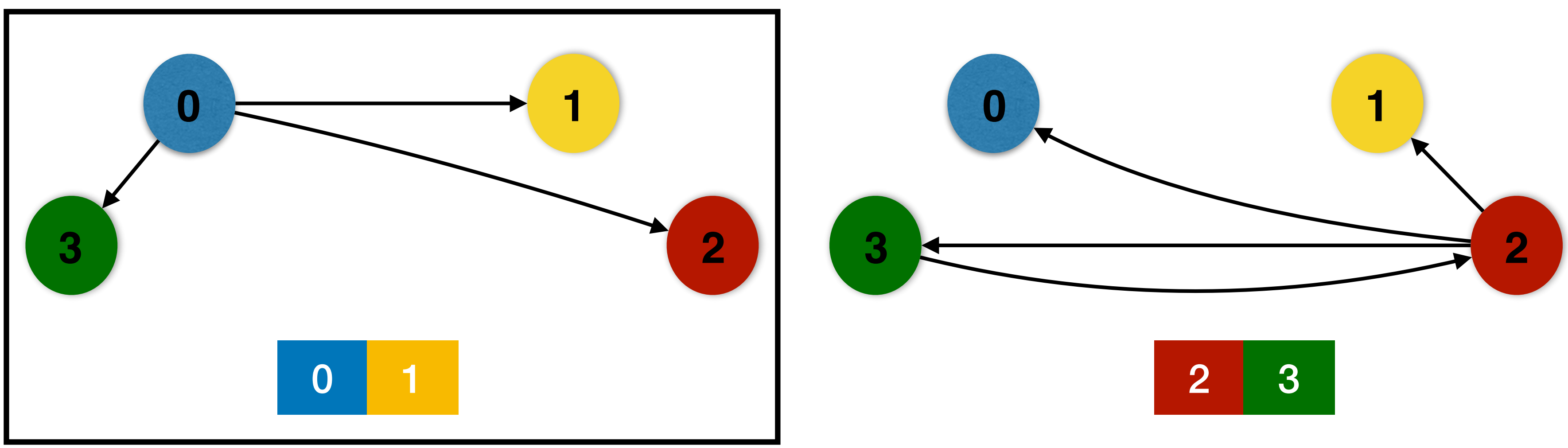




Graph Processing

Cache

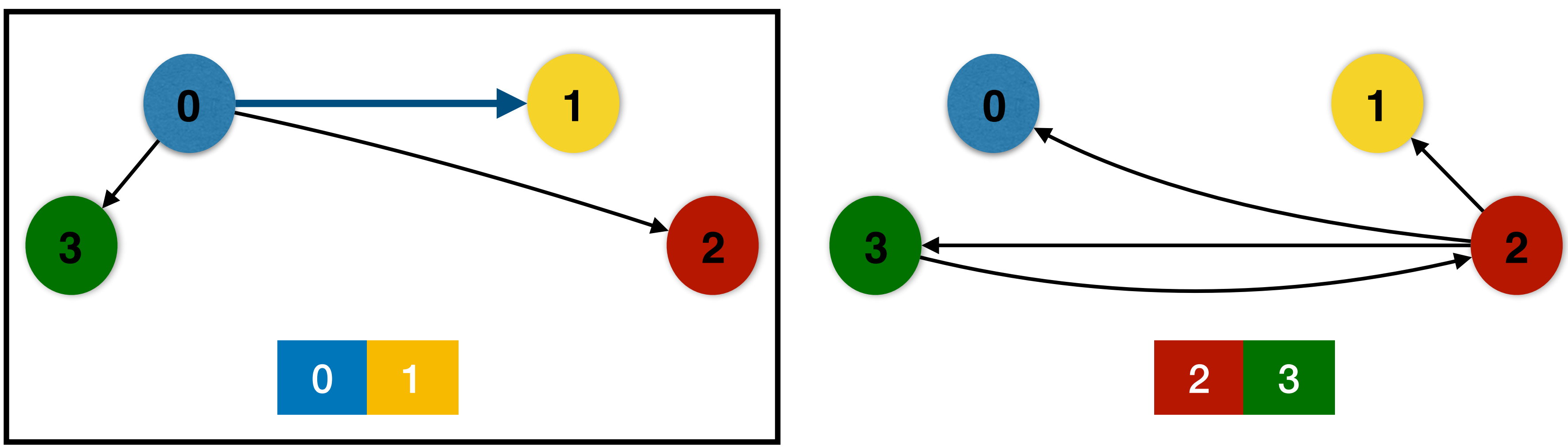
#hits: 0
#misses: 0



Graph Processing

Cache

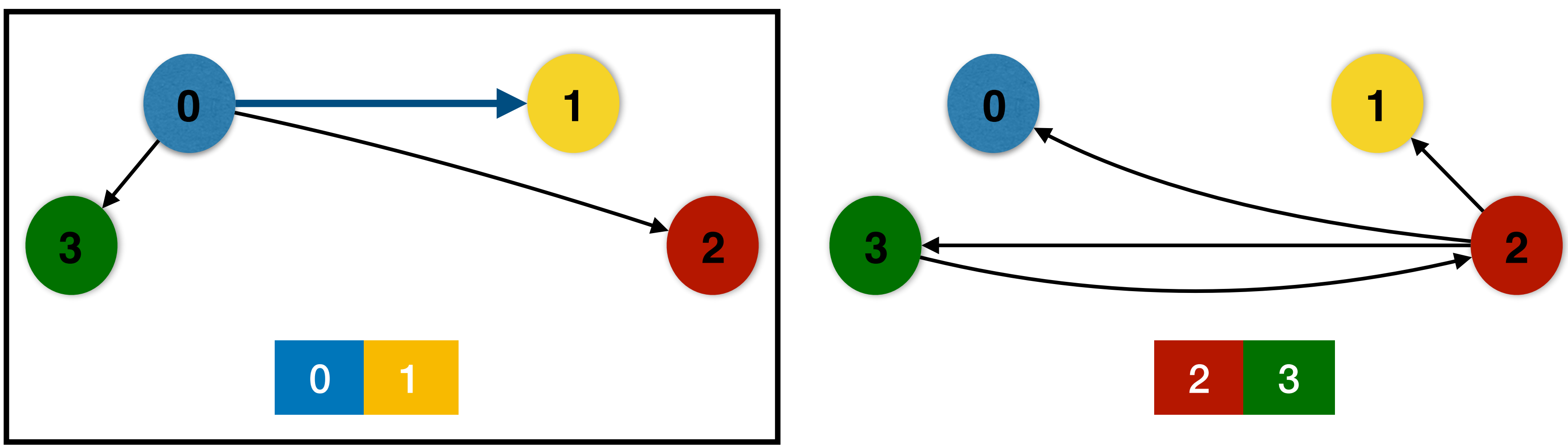
#hits: 0
#misses: 0



Graph Processing

Cache

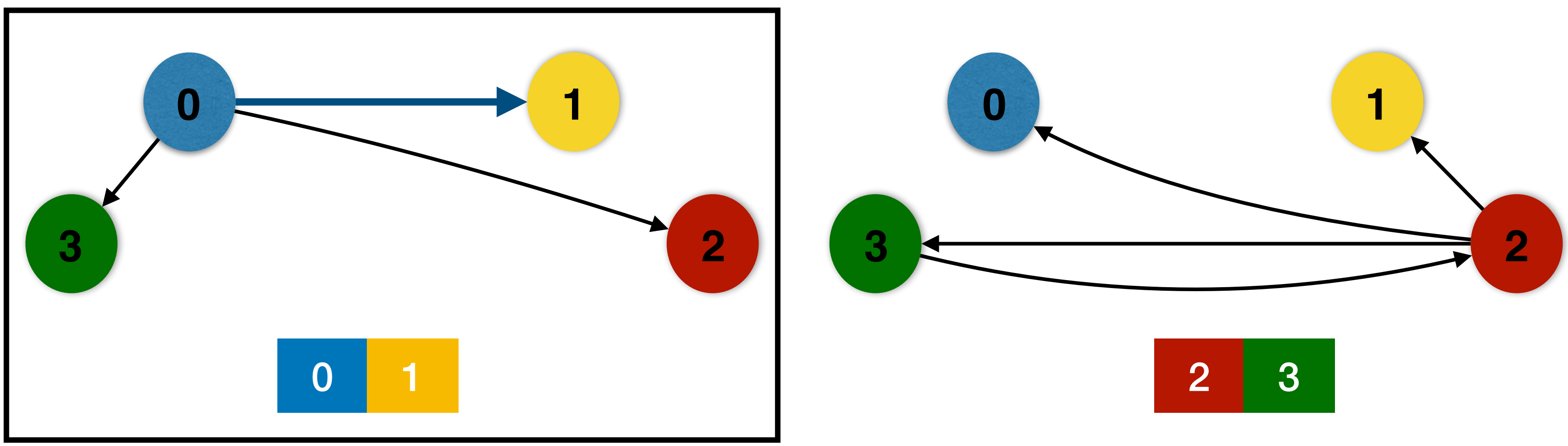
#hits: 0
#misses: 1



Graph Processing

Cache

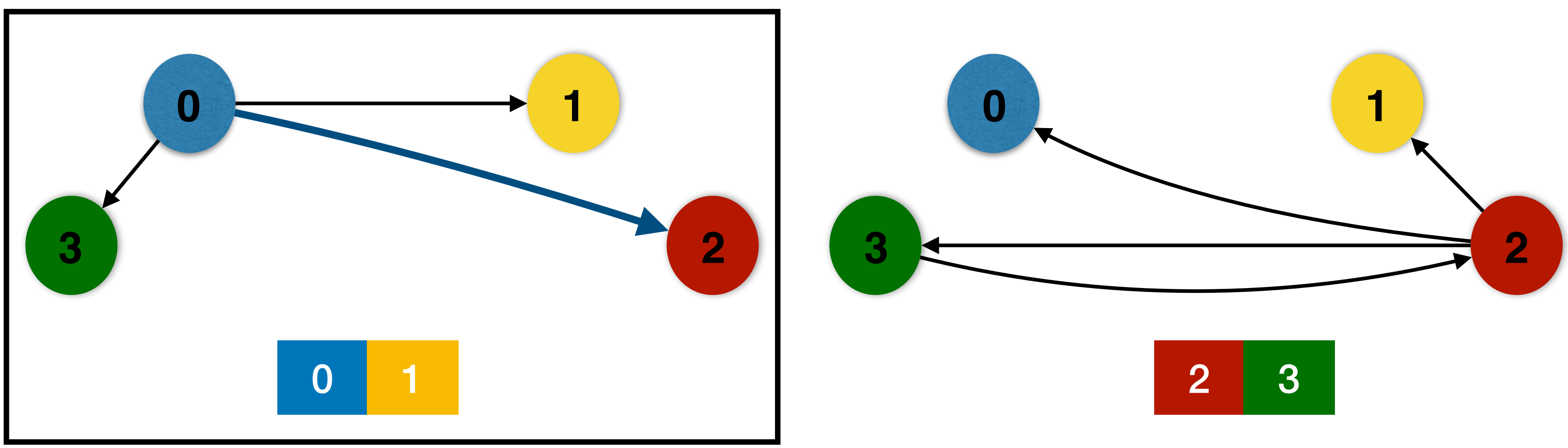
#hits: 0
#misses: 1



Graph Processing

Cache

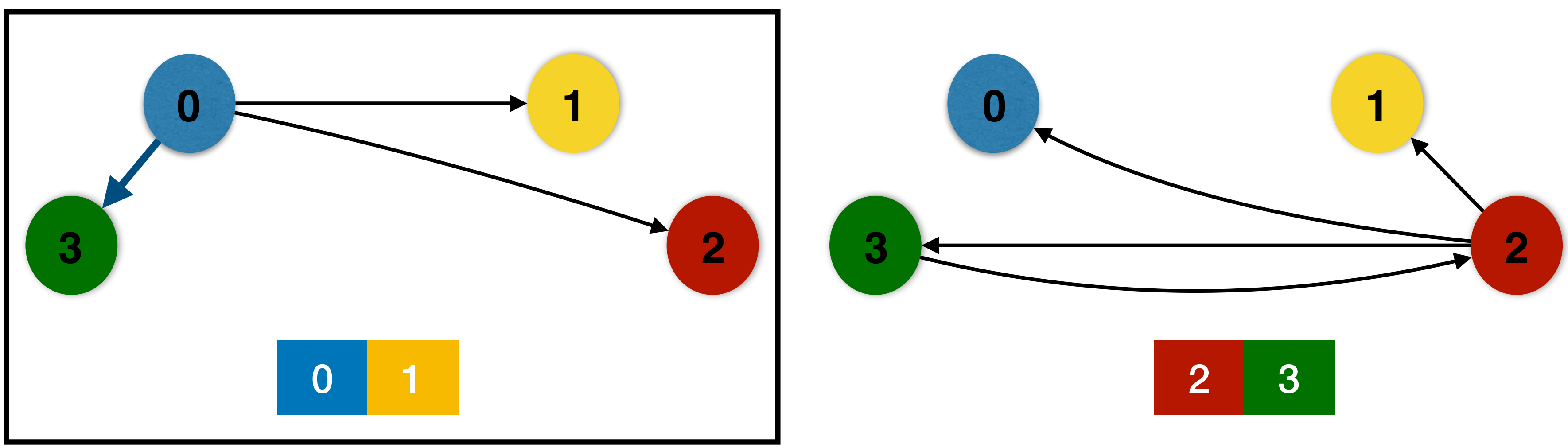
#hits: 1
#misses: 1



Graph Processing

Cache

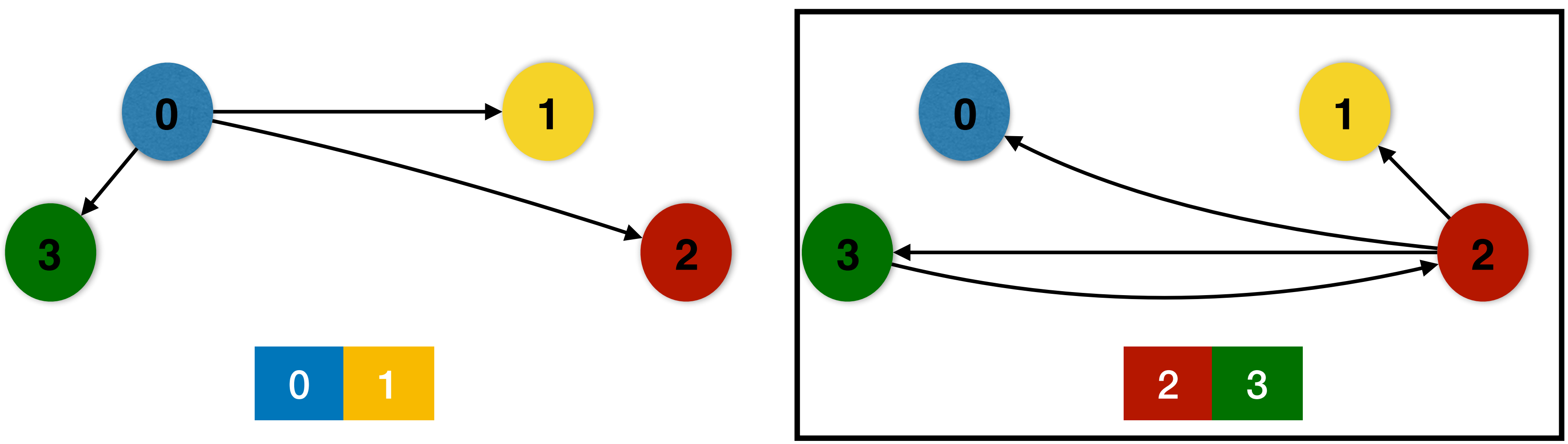
#hits: 2
#misses: 1



Graph Processing

Cache

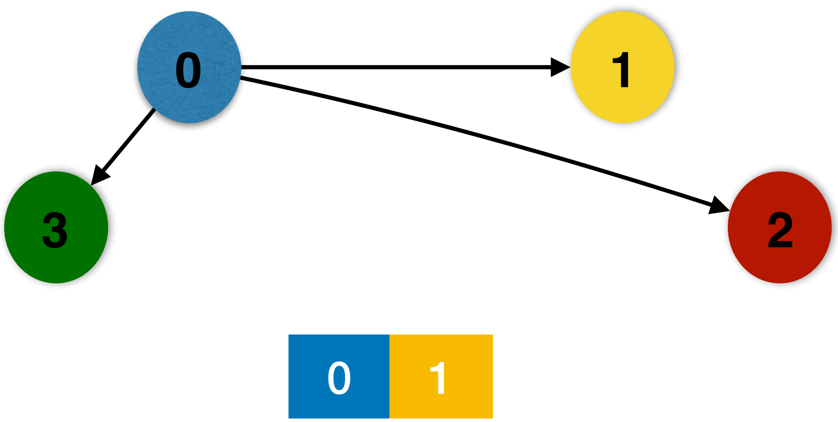
#hits: 2
#misses: 1

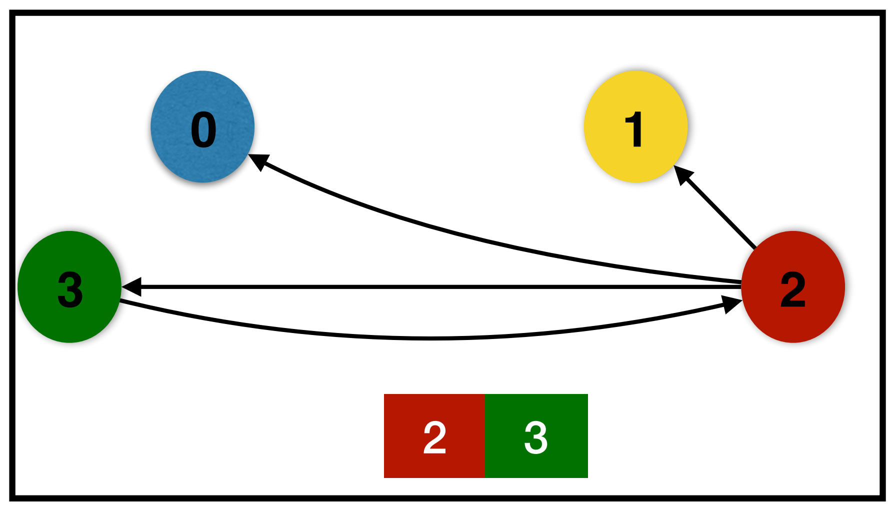


Graph Processing

Cache

#hits: 2
#misses: 1

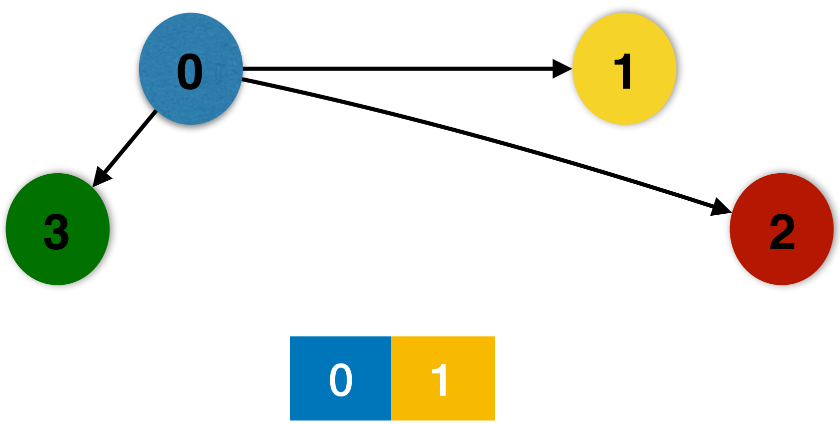


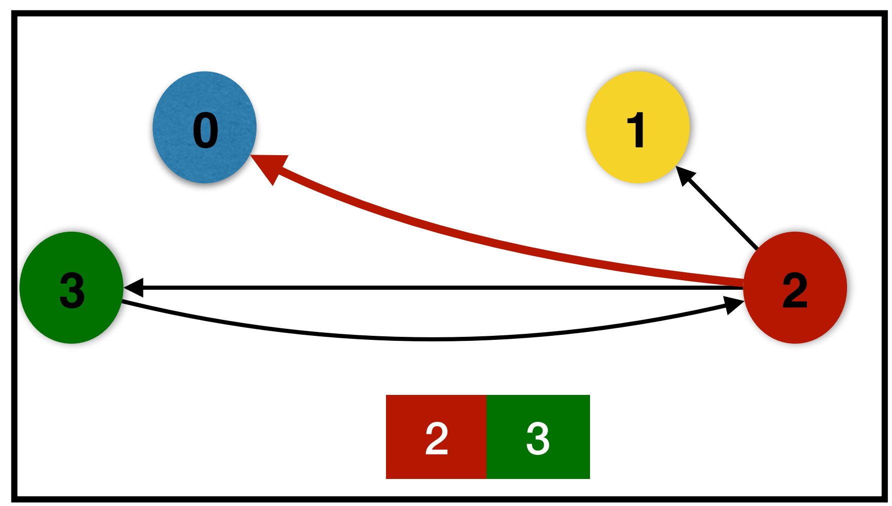


Graph Processing

Cache

#hits: 2
#misses: 1



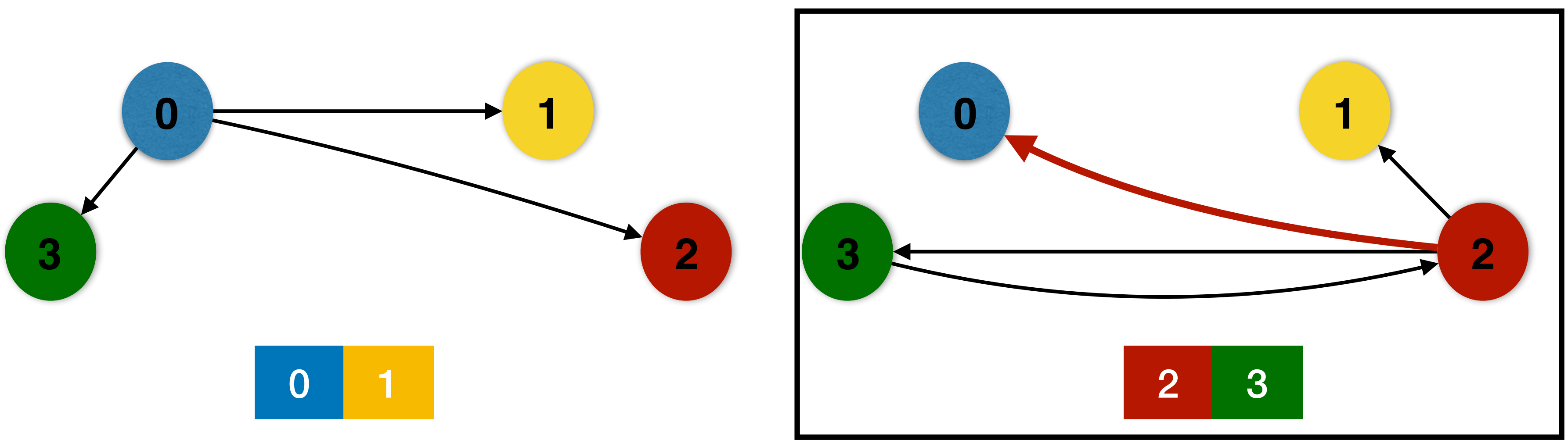


Graph Processing

Cache

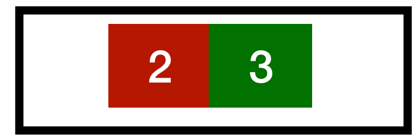
0	1

#hits: 2
#misses: 1

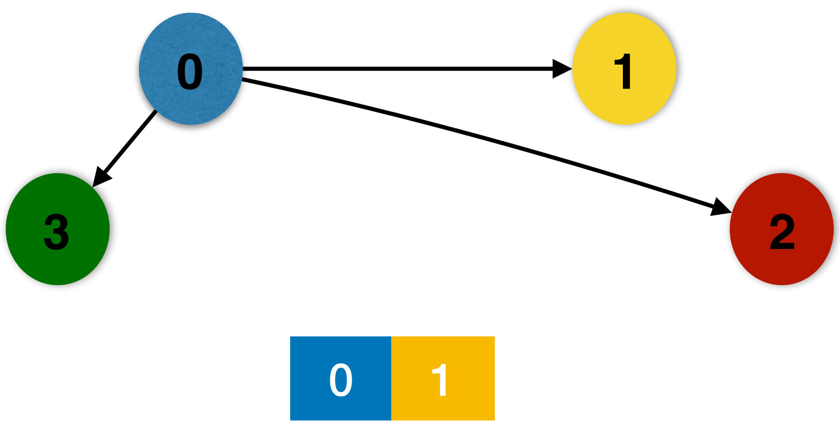


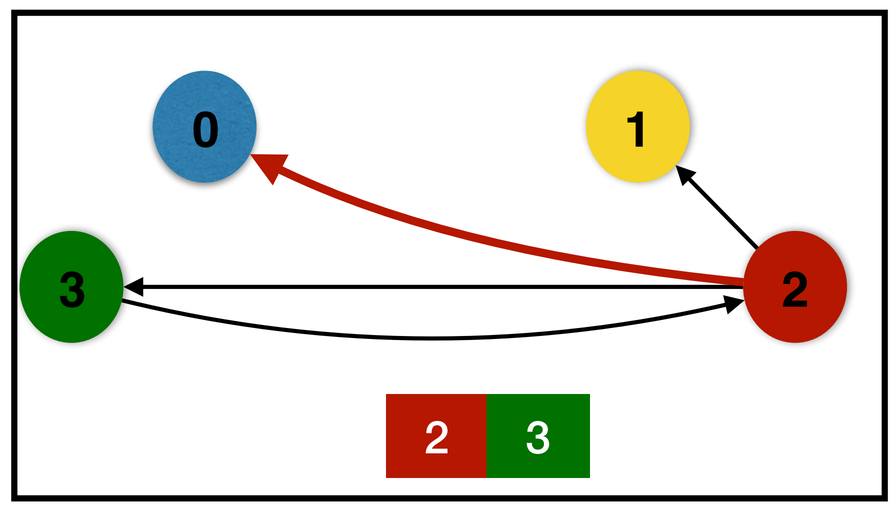
Graph Processing

Cache



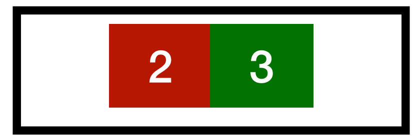
#hits: 2
#misses: 2



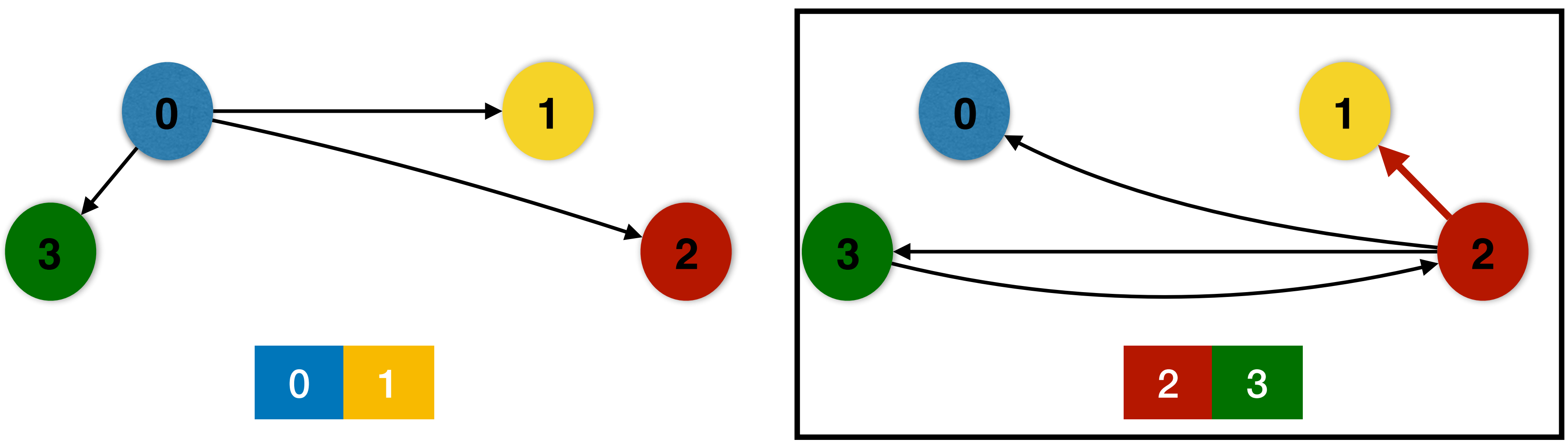


Graph Processing

Cache

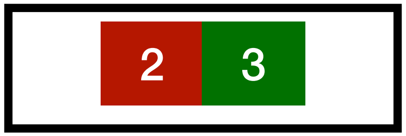


#hits: 3
#misses: 2

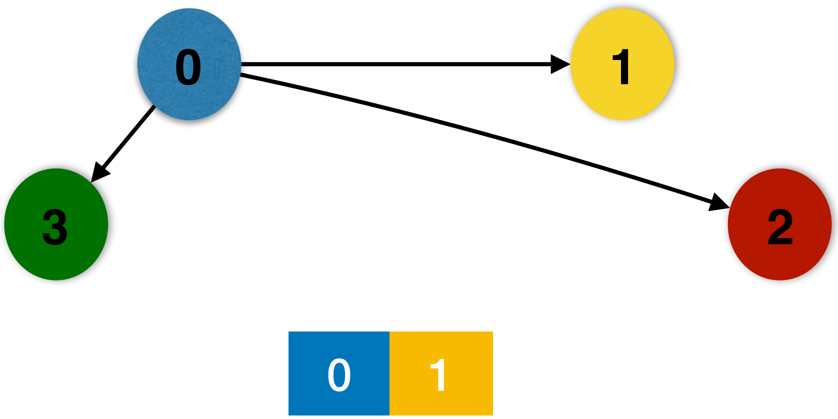


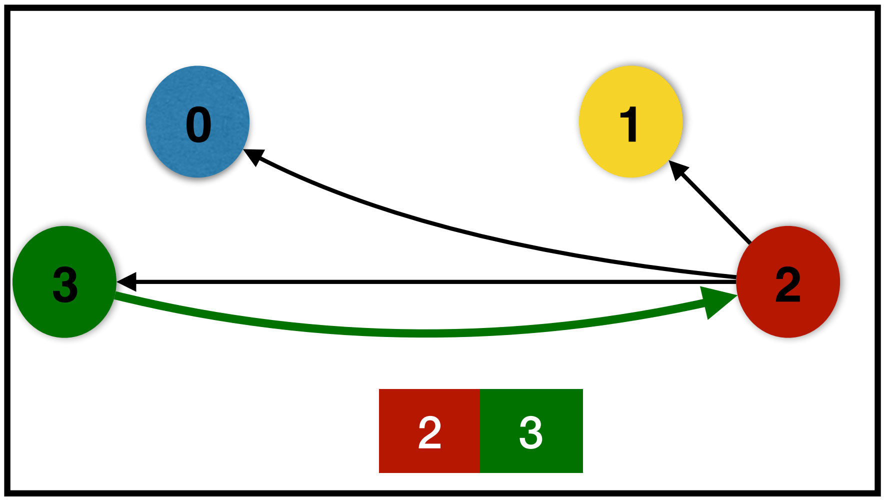
Graph Processing

Cache



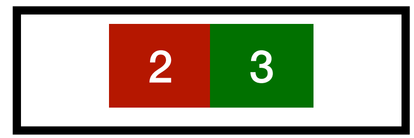
#hits: 4
#misses: 2



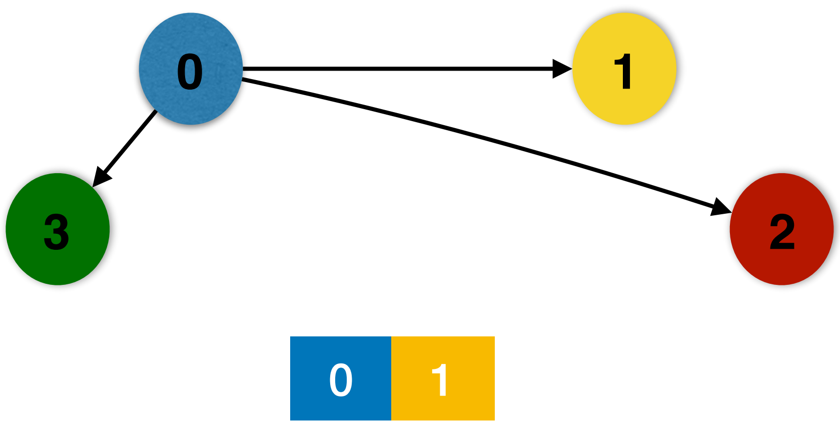


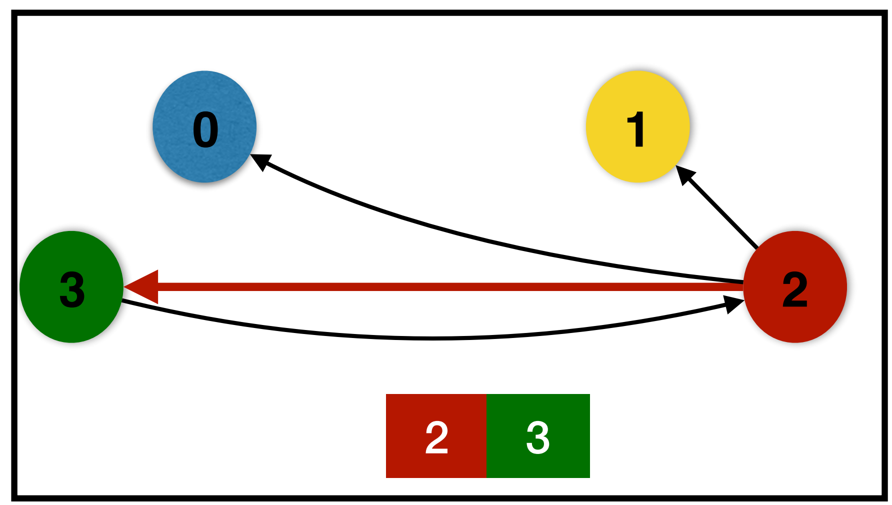
Graph Processing

Cache



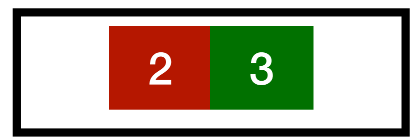
#hits: 5
#misses: 2





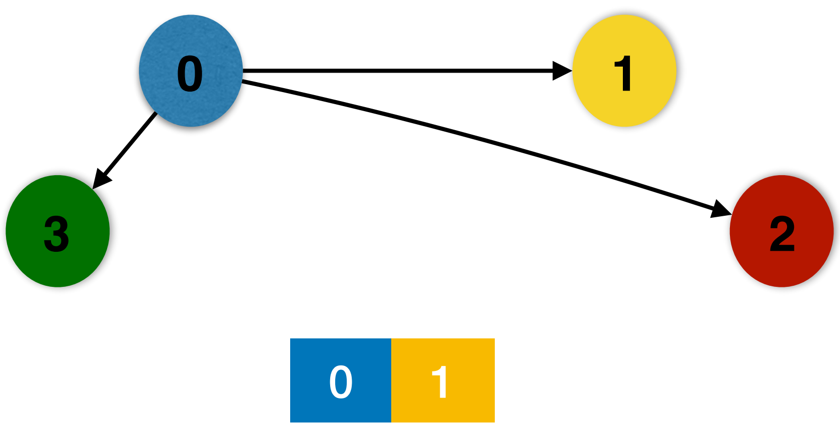
Graph Processing

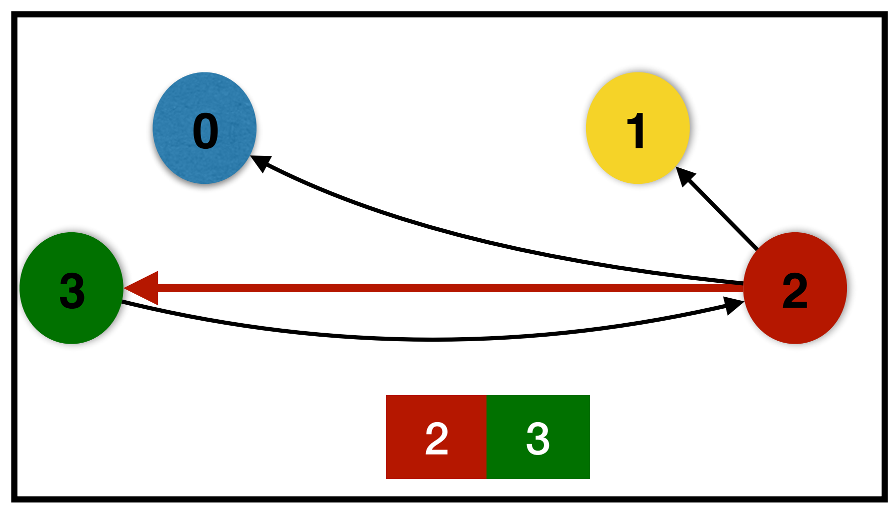
Cache



Only have 2 misses

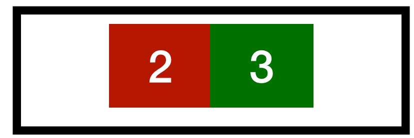
#hits: 5
#misses: 2





Graph Processing

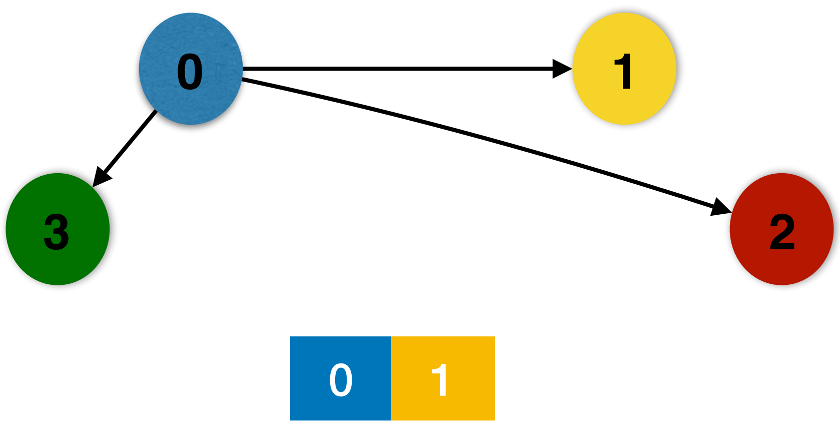
Cache

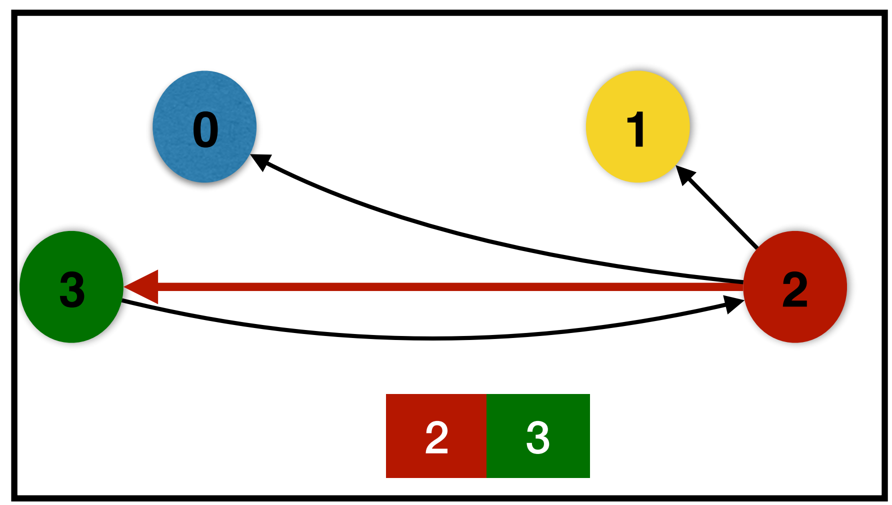


Better than
Frequency based
Reordering

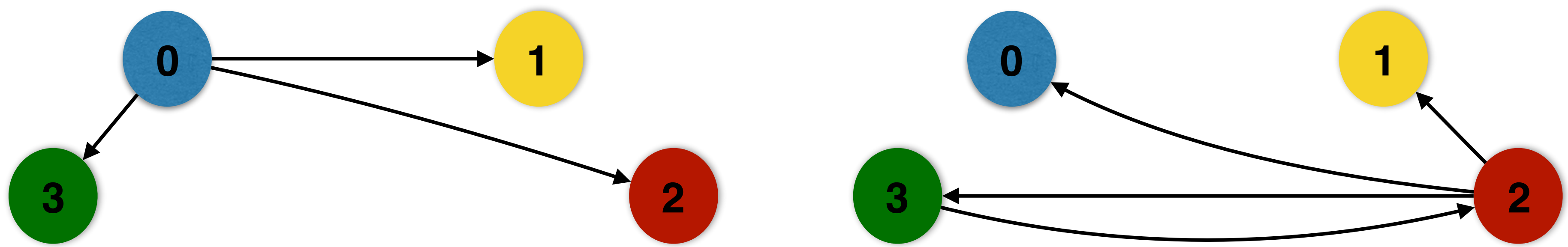
#hits: 4
#misses: 3

#hits: 5
#misses: 2

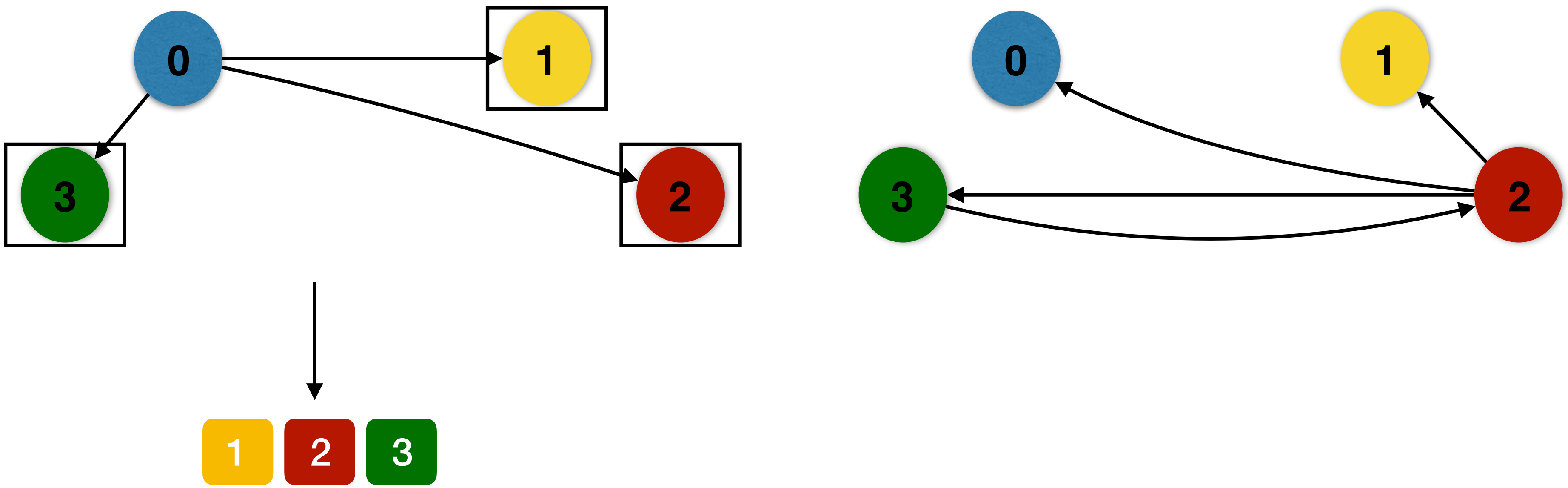




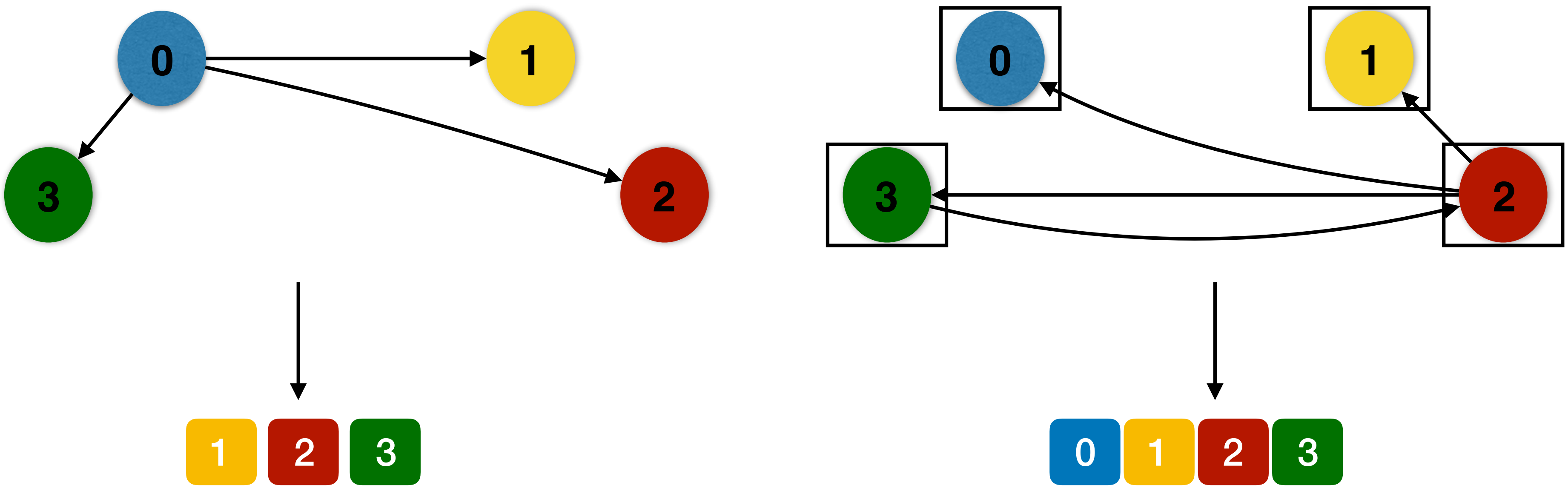
Cache-aware Merge



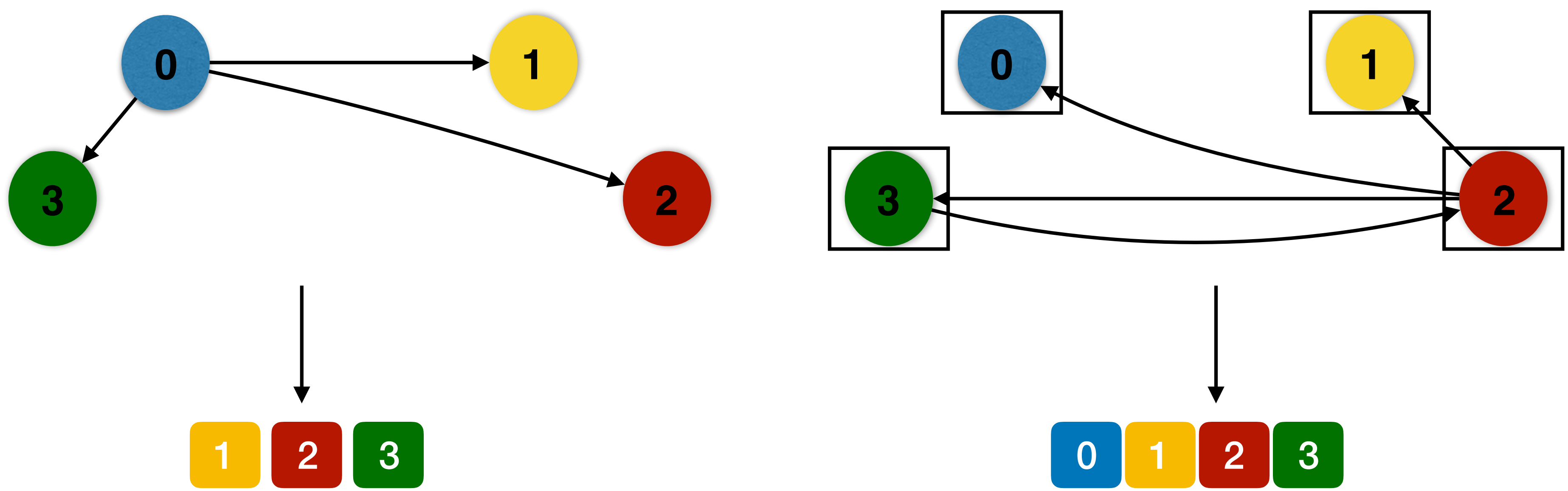
Cache-aware Merge



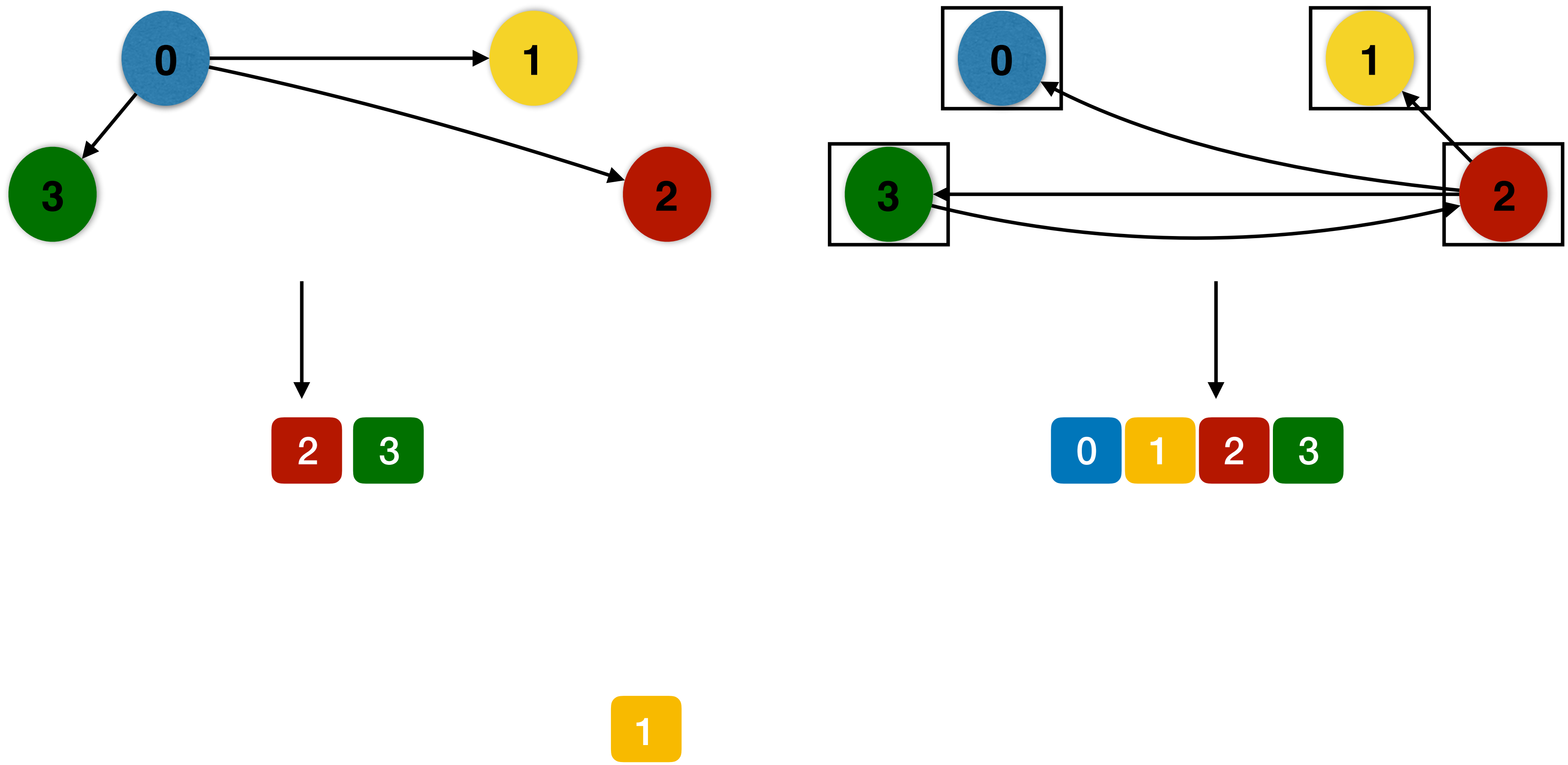
Cache-aware Merge



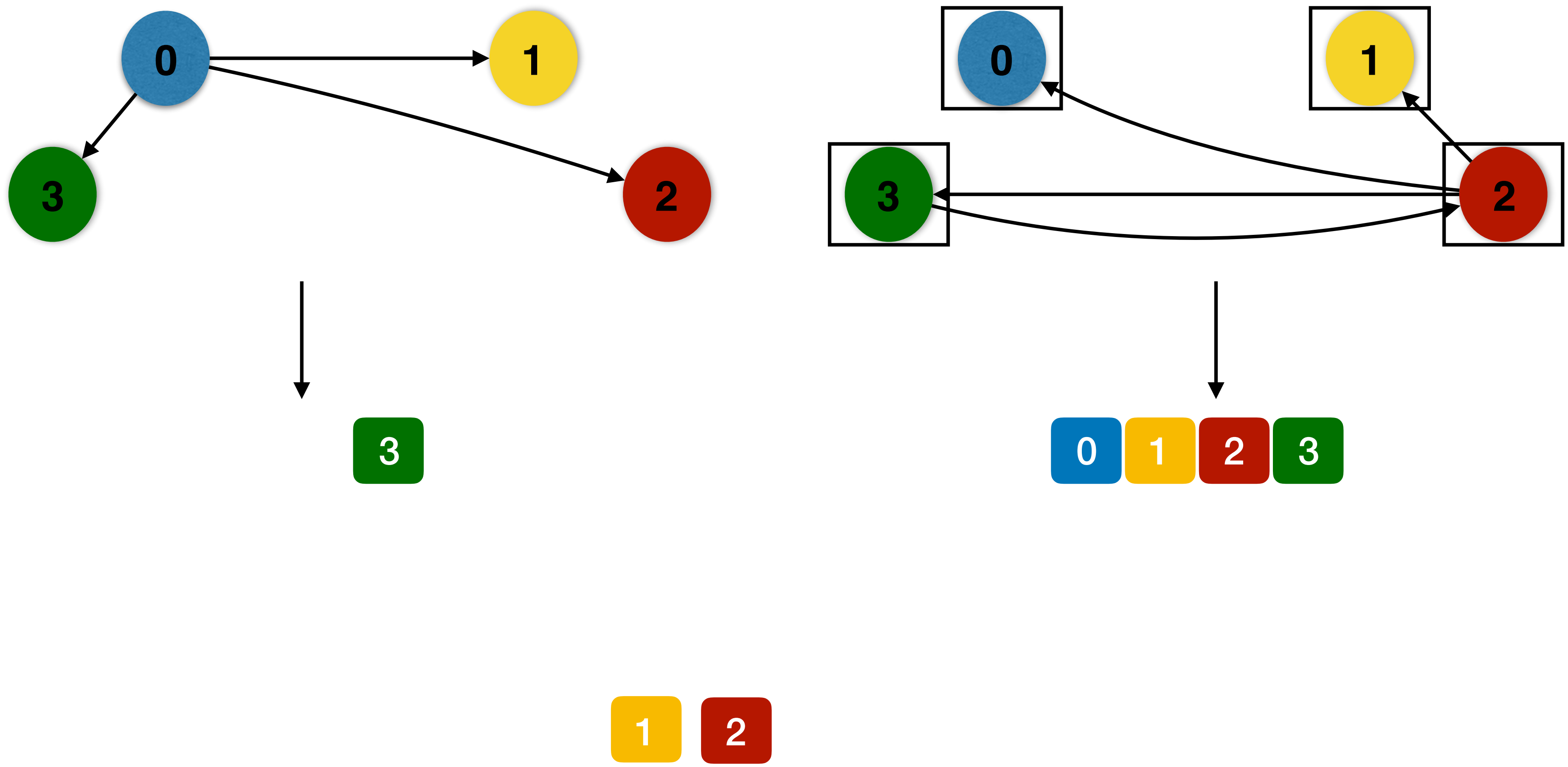
Cache-aware Merge



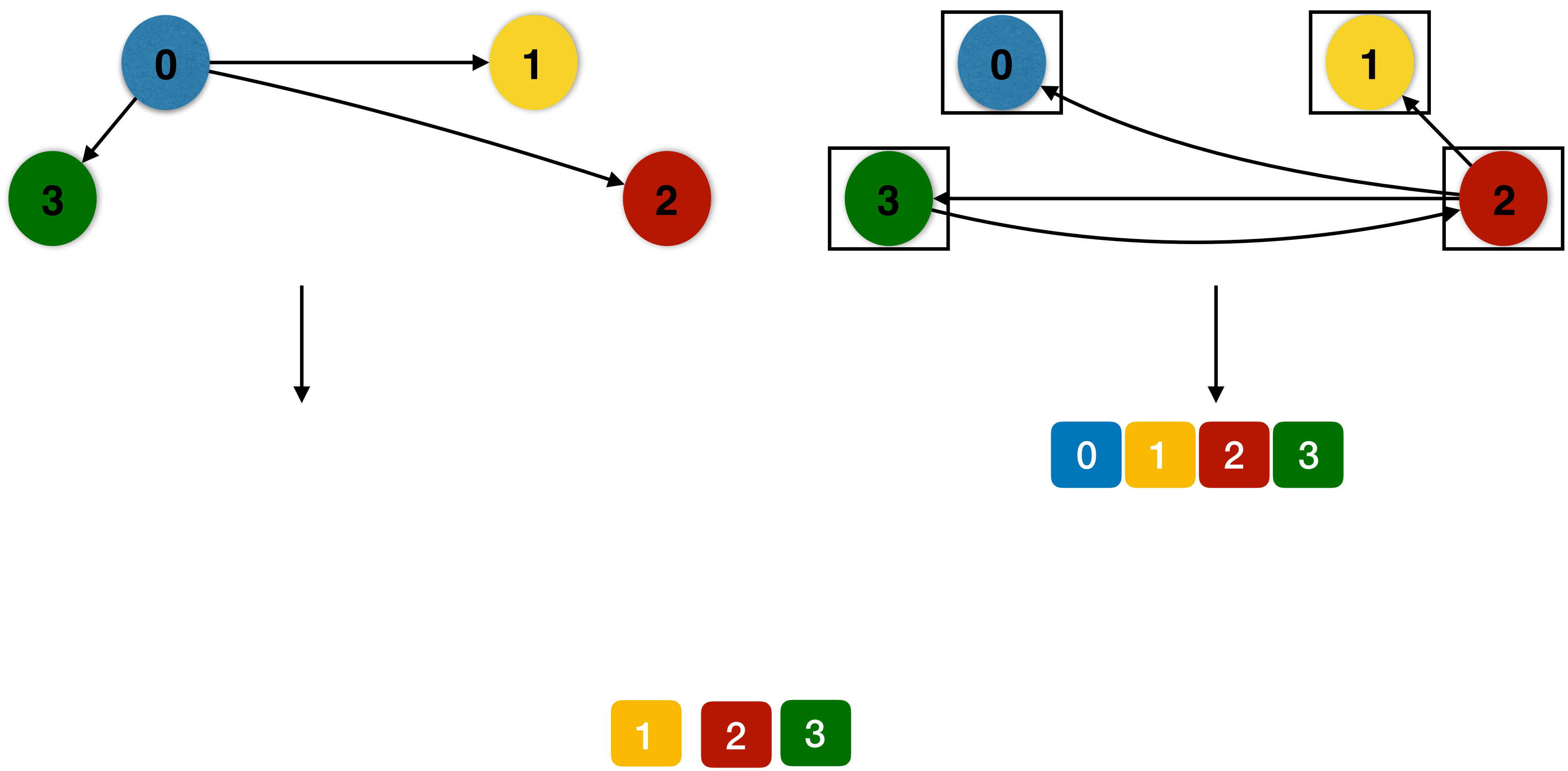
Cache-aware Merge



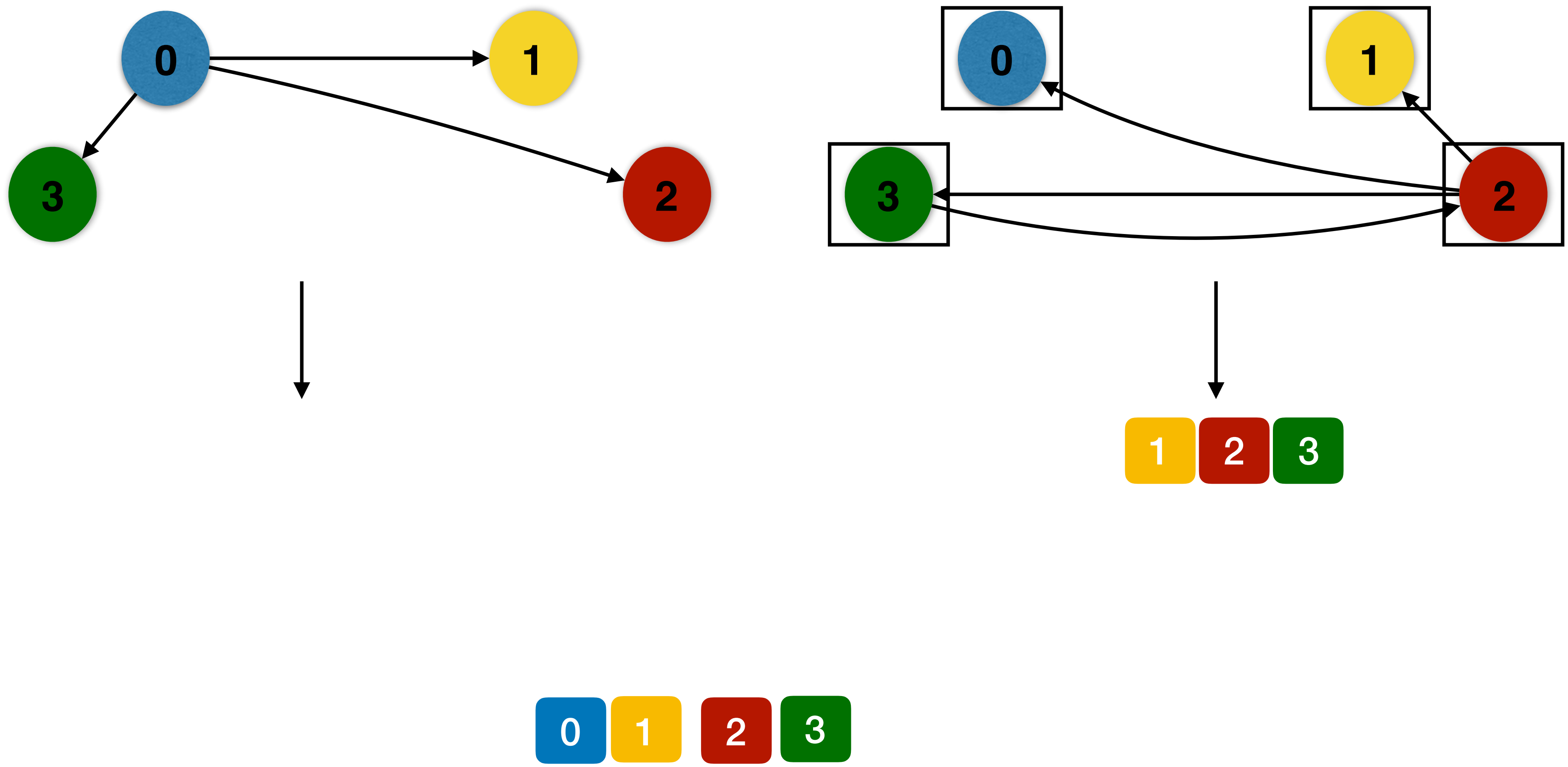
Cache-aware Merge



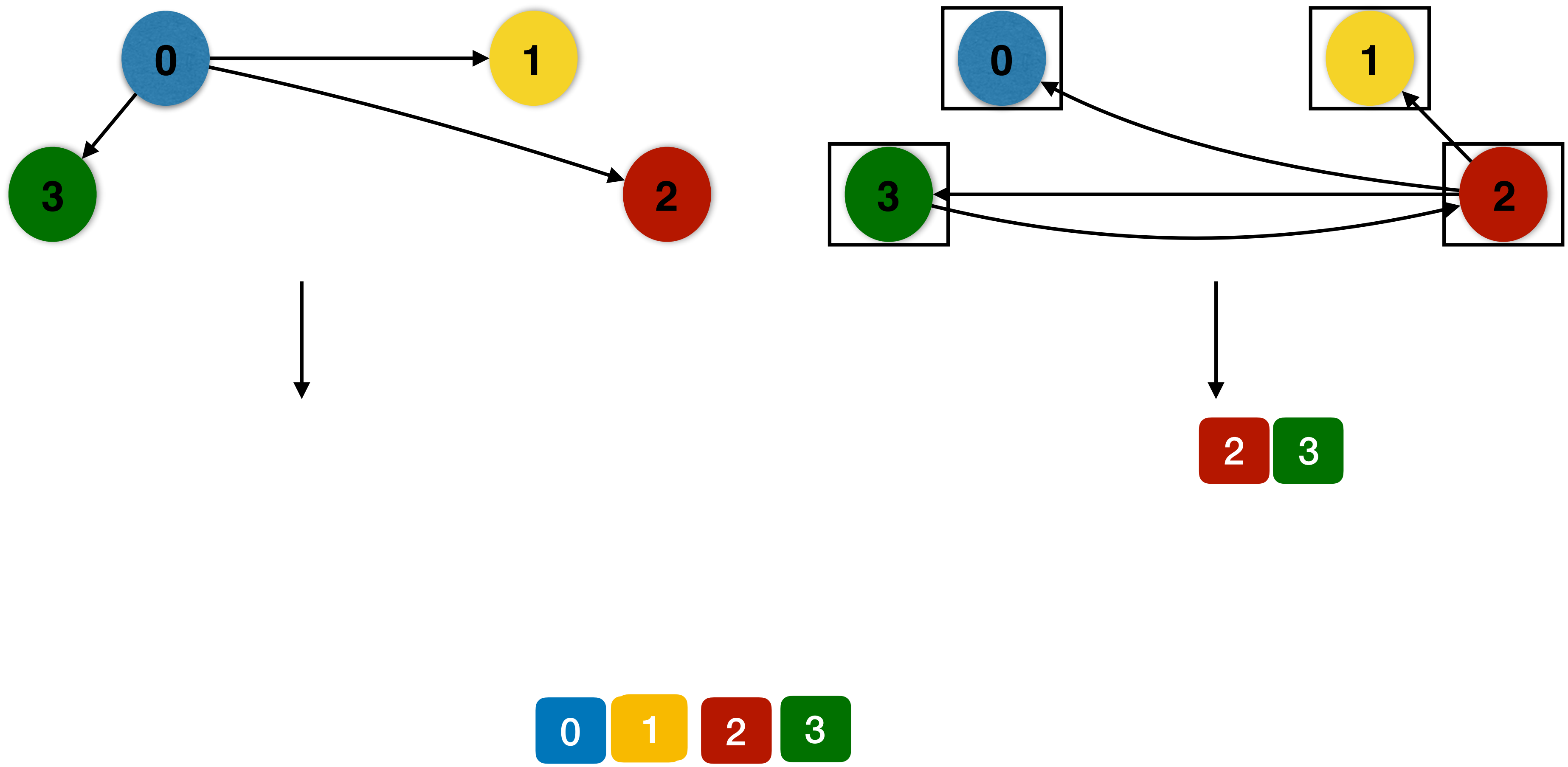
Cache-aware Merge



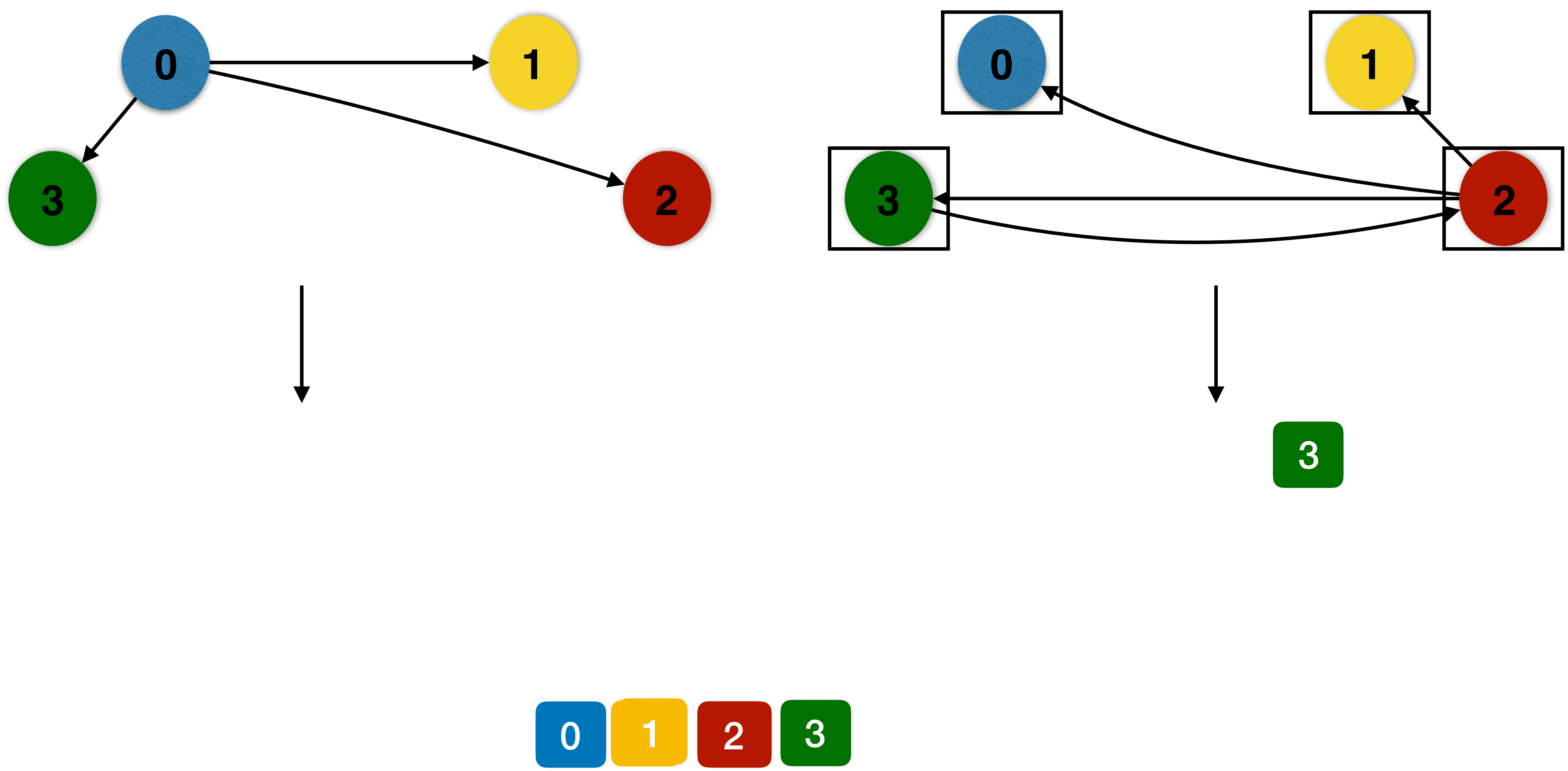
Cache-aware Merge



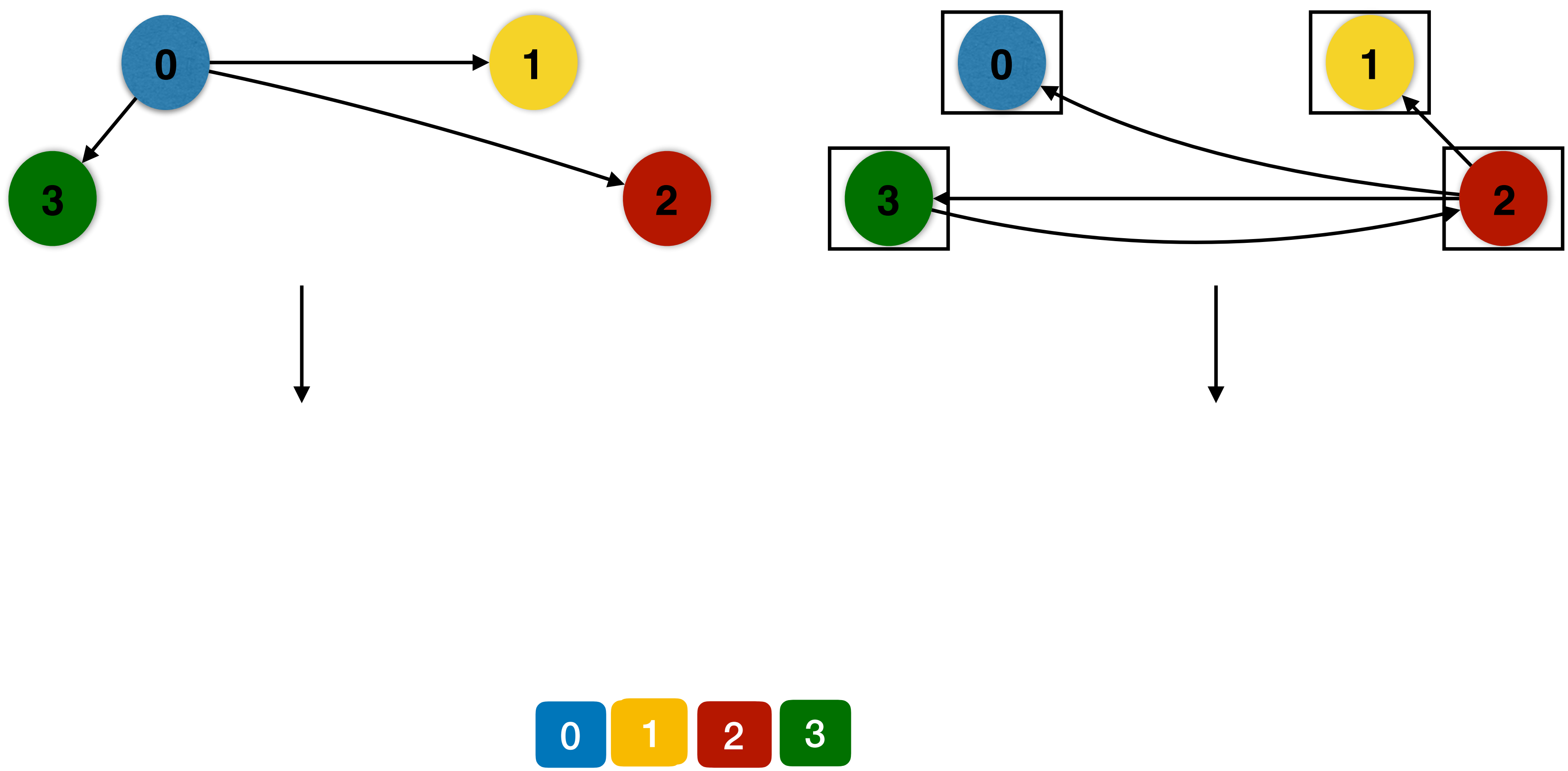
Cache-aware Merge



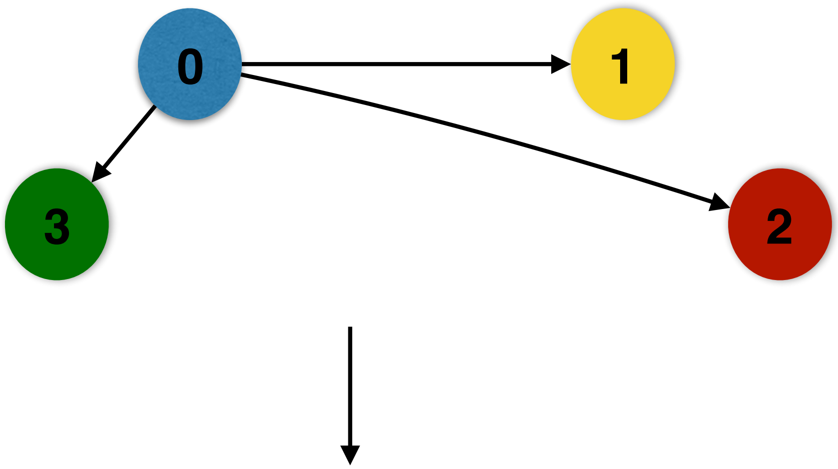
Cache-aware Merge

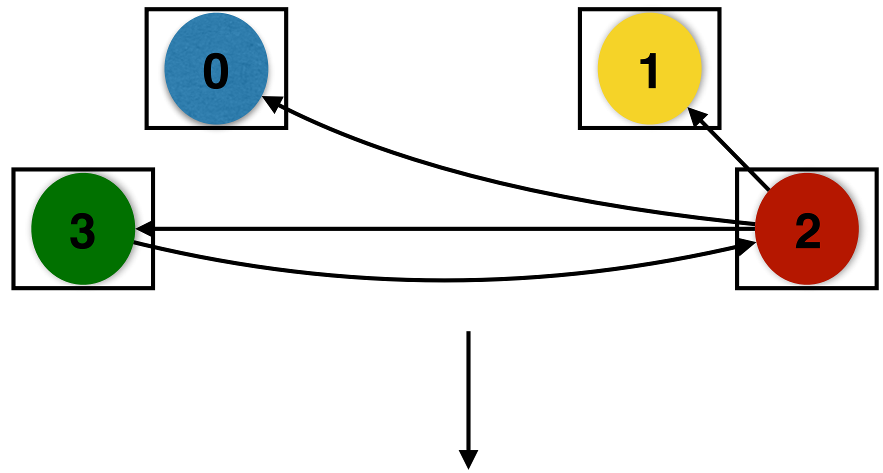


Cache-aware Merge



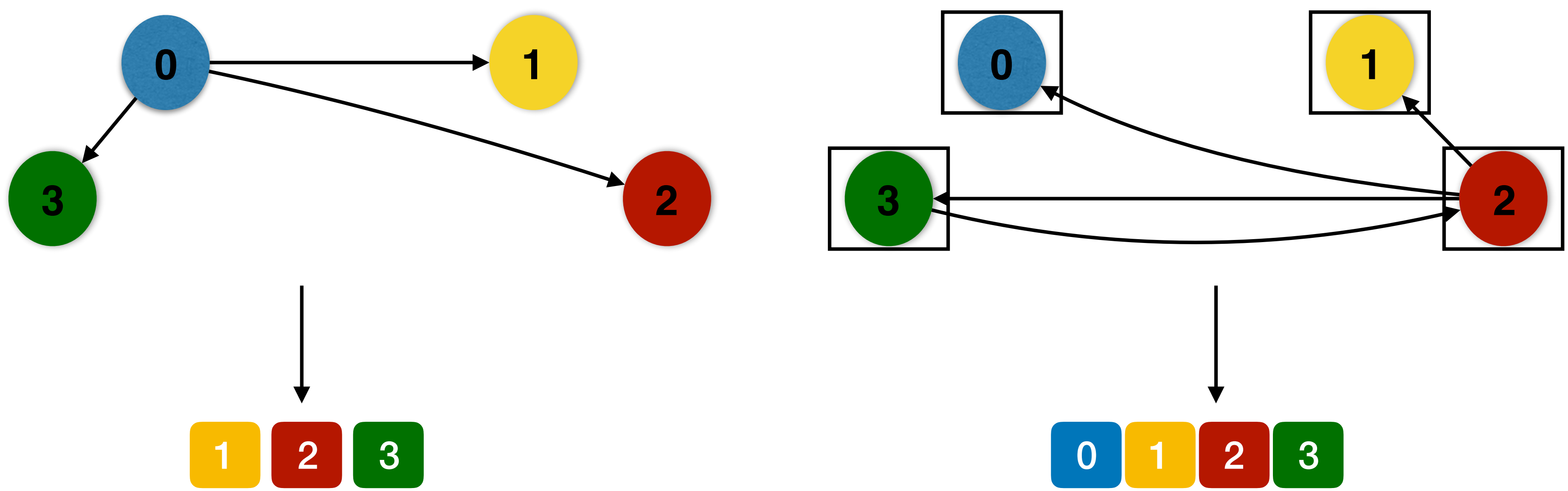
Cache-aware Merge



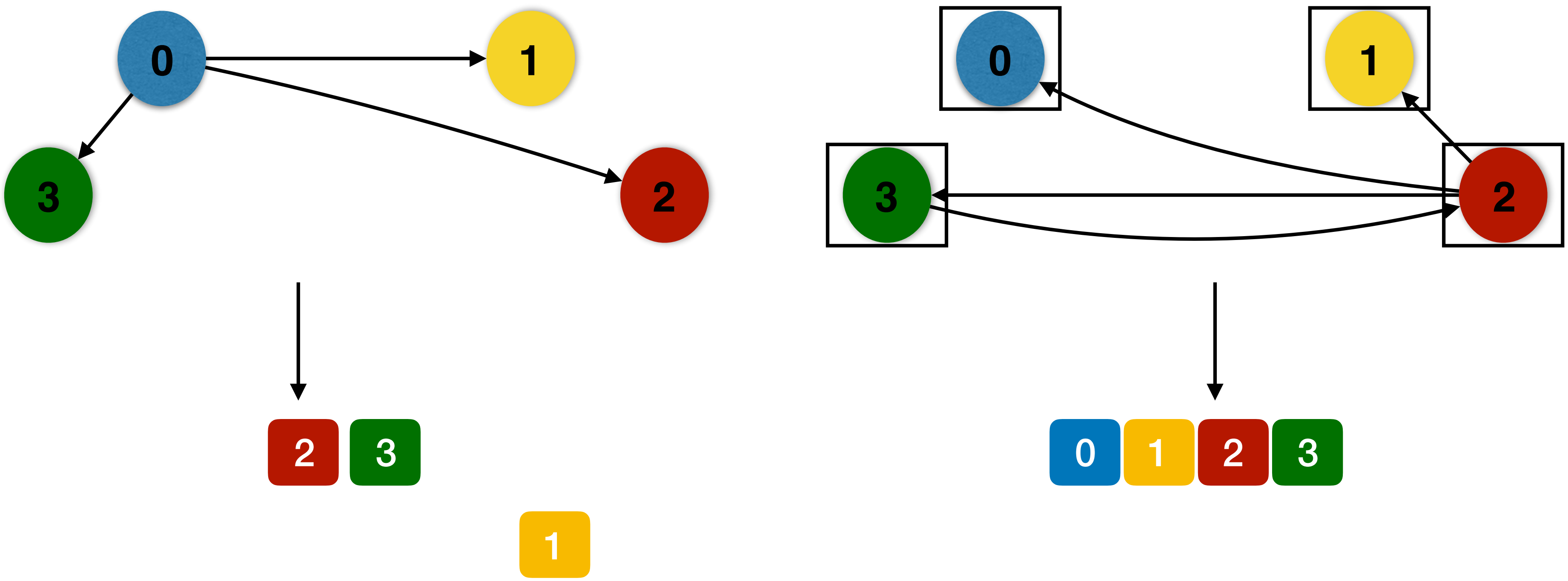


The naive approach incurs random DRAM accesses

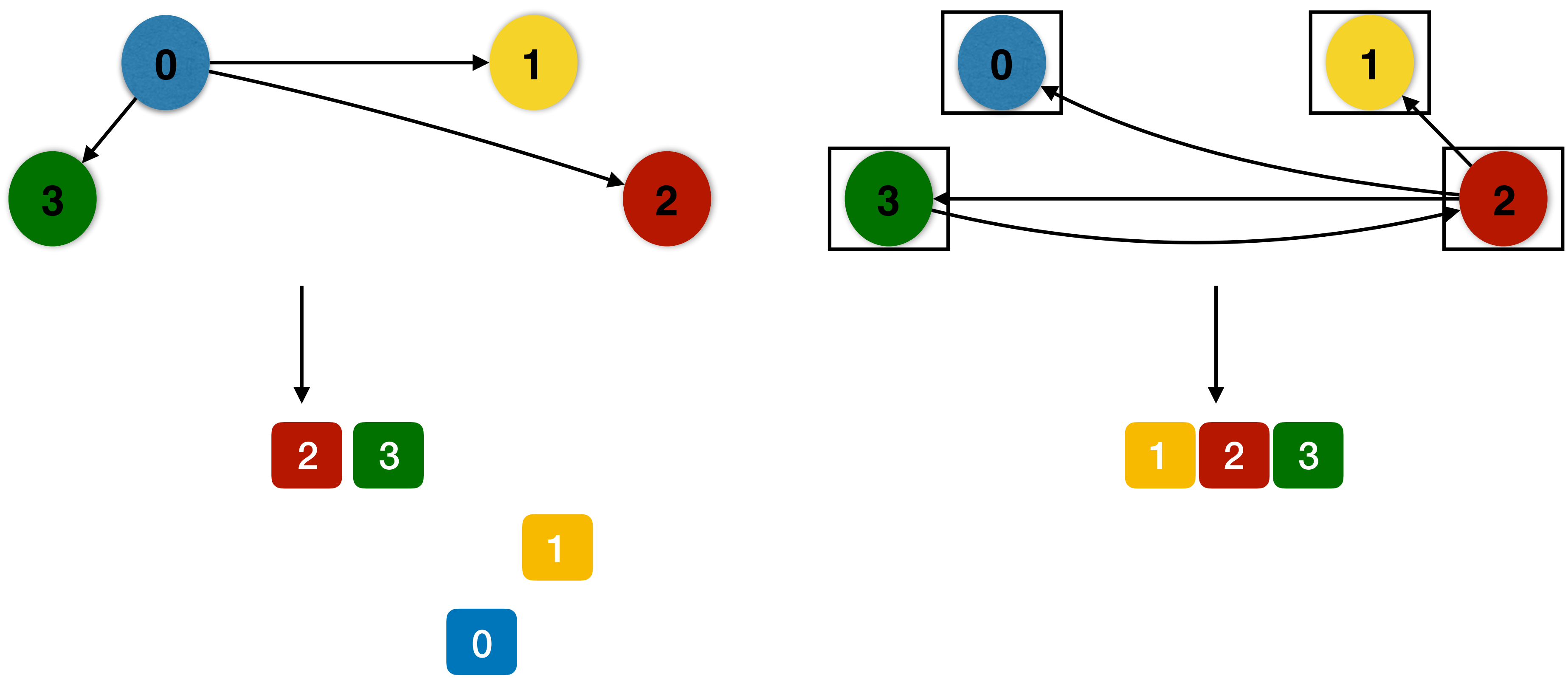
Cache-aware Merge



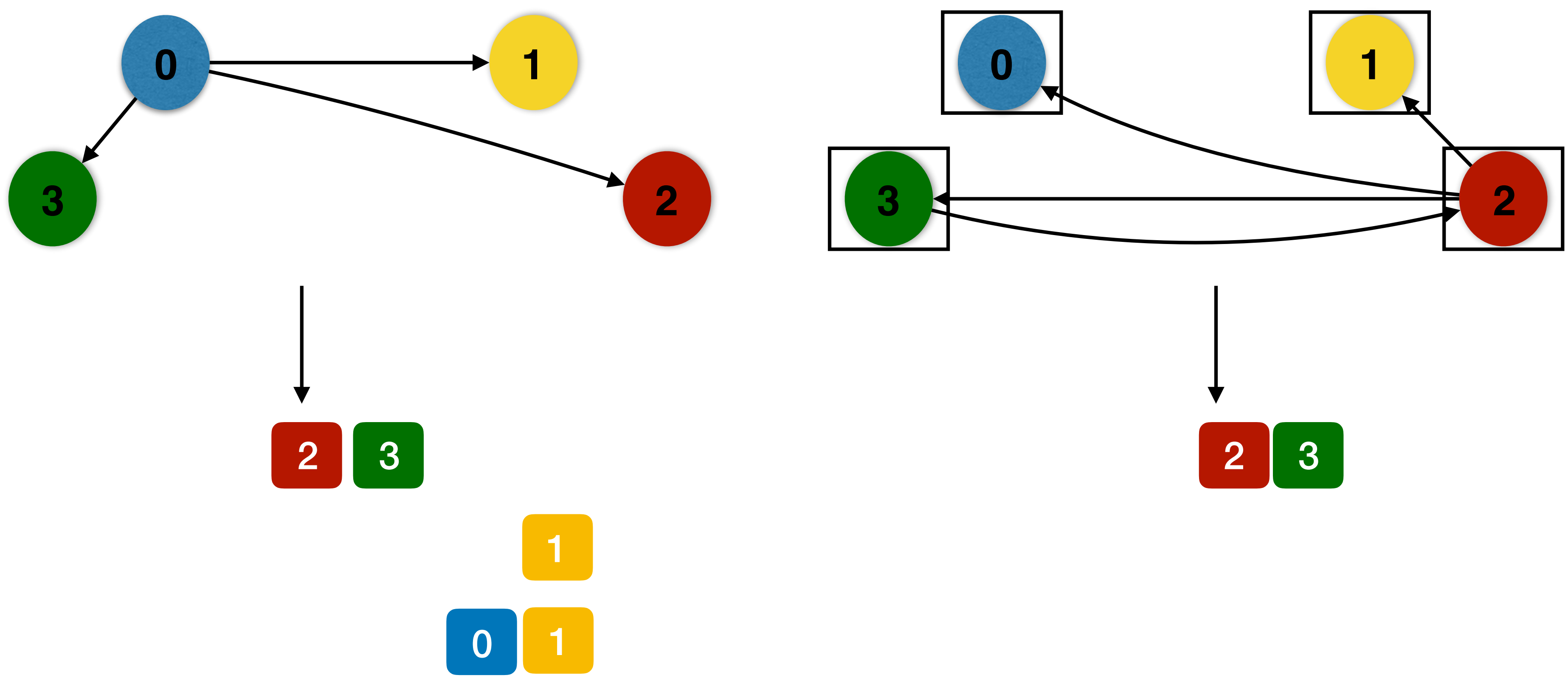
Cache-aware Merge



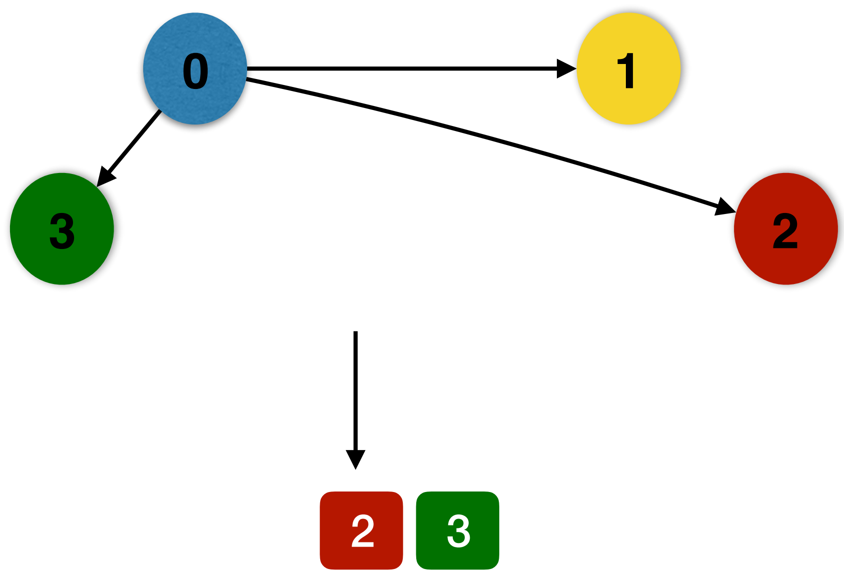
Cache-aware Merge

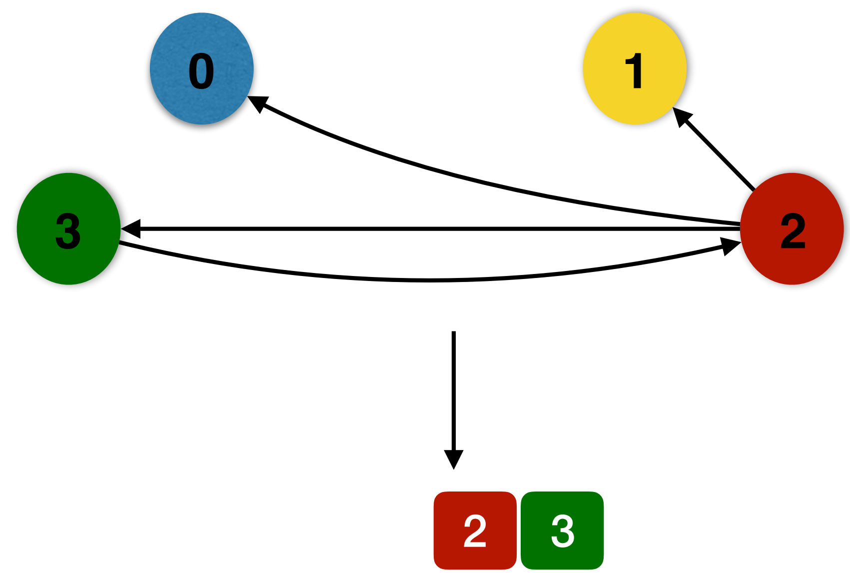


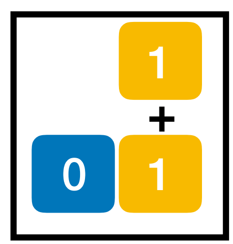
Cache-aware Merge



Cache-aware Merge

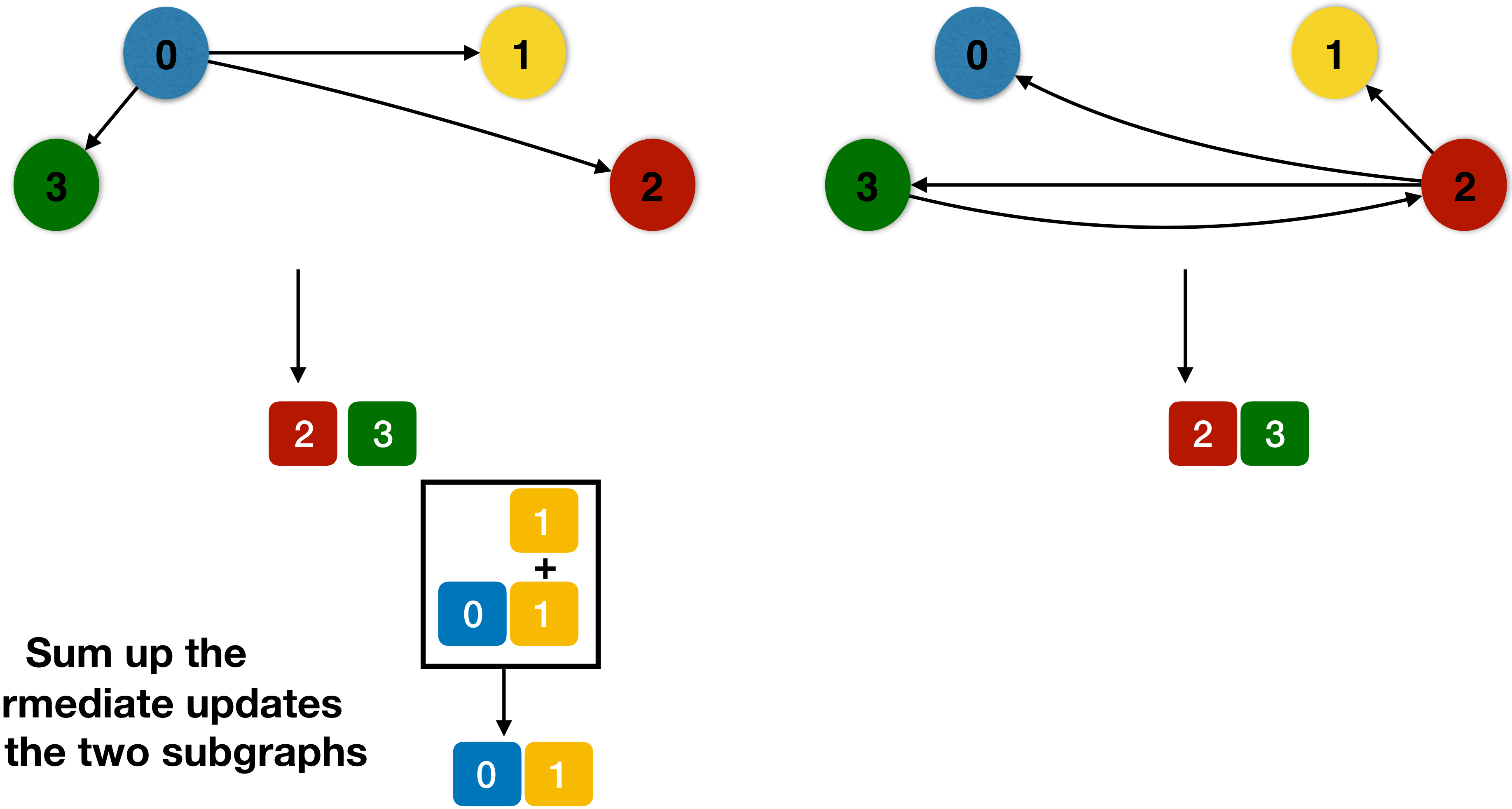






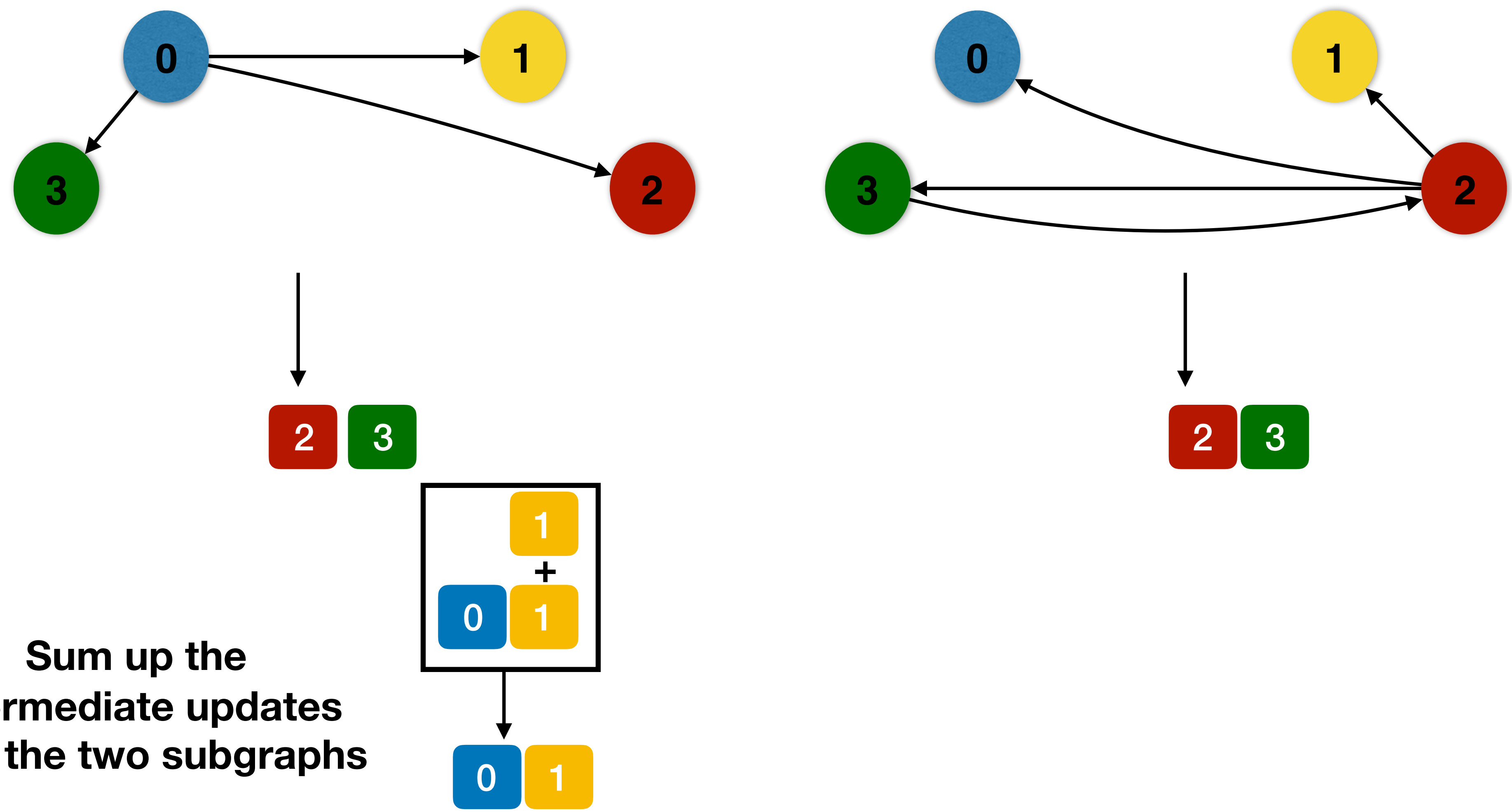
Break down into chunks that fit in cache

Cache-aware Merge



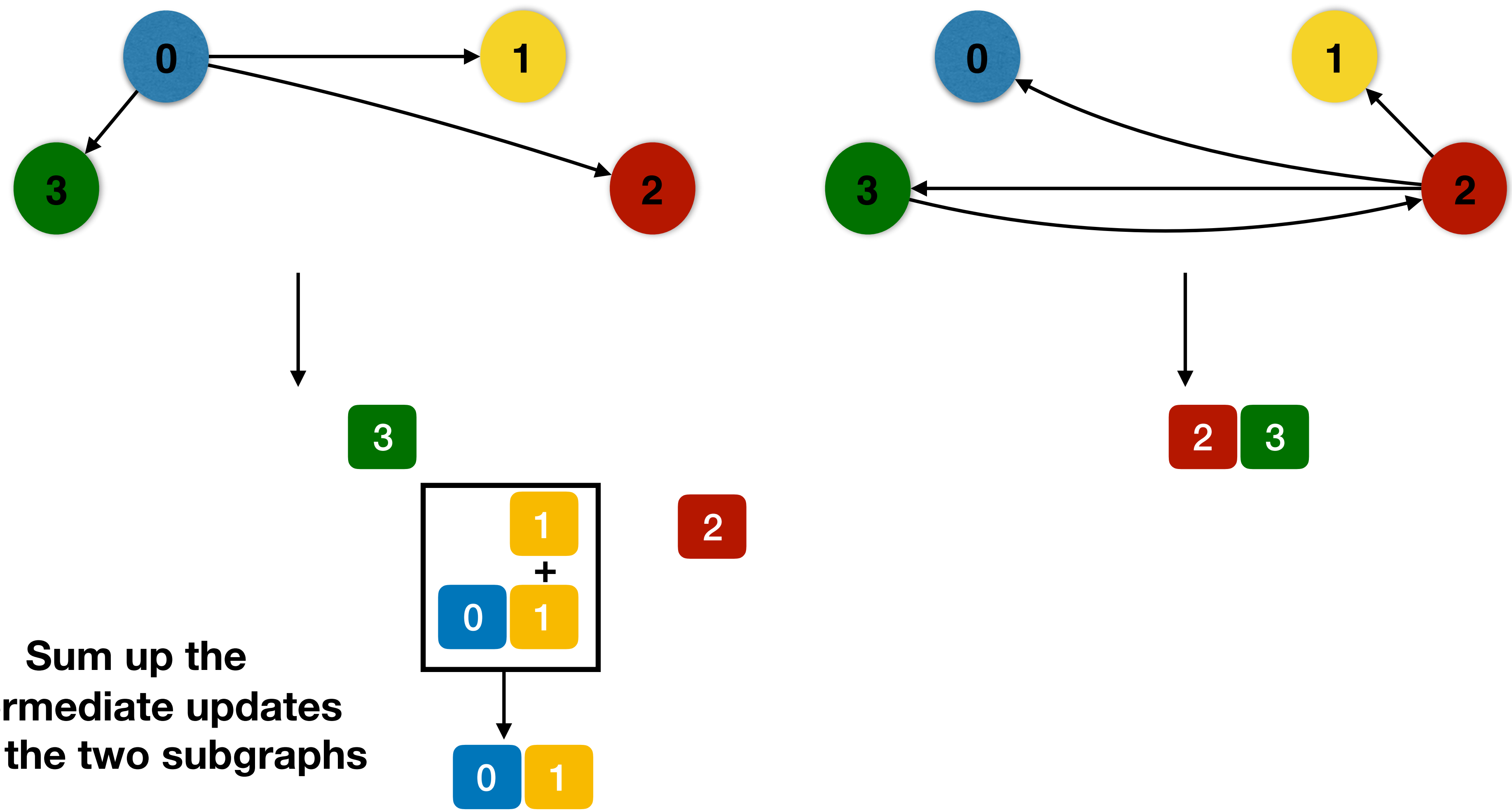
Sum up the intermediate updates from the two subgraphs

Cache-aware Merge



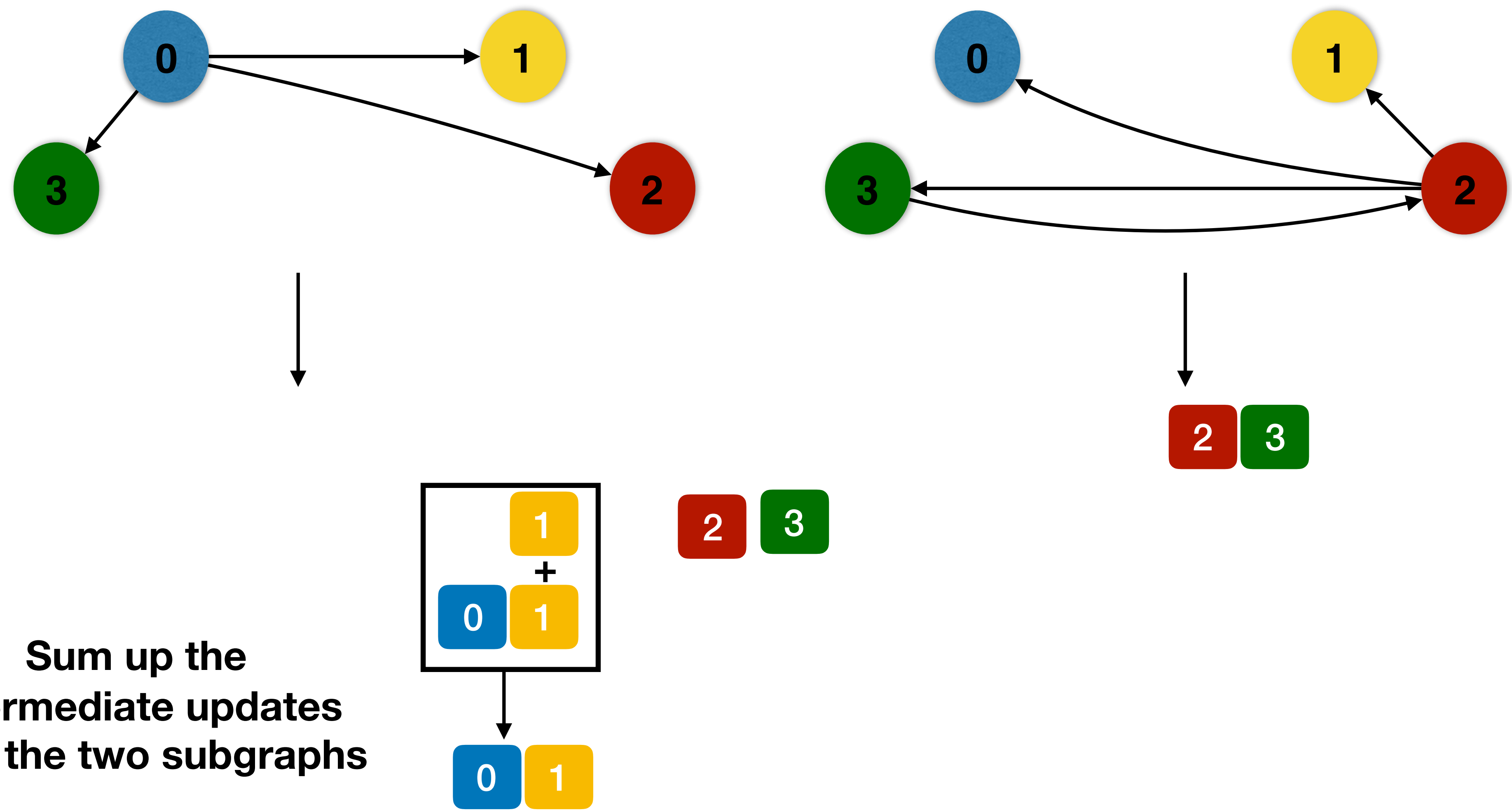
Sum up the intermediate updates from the two subgraphs

Cache-aware Merge



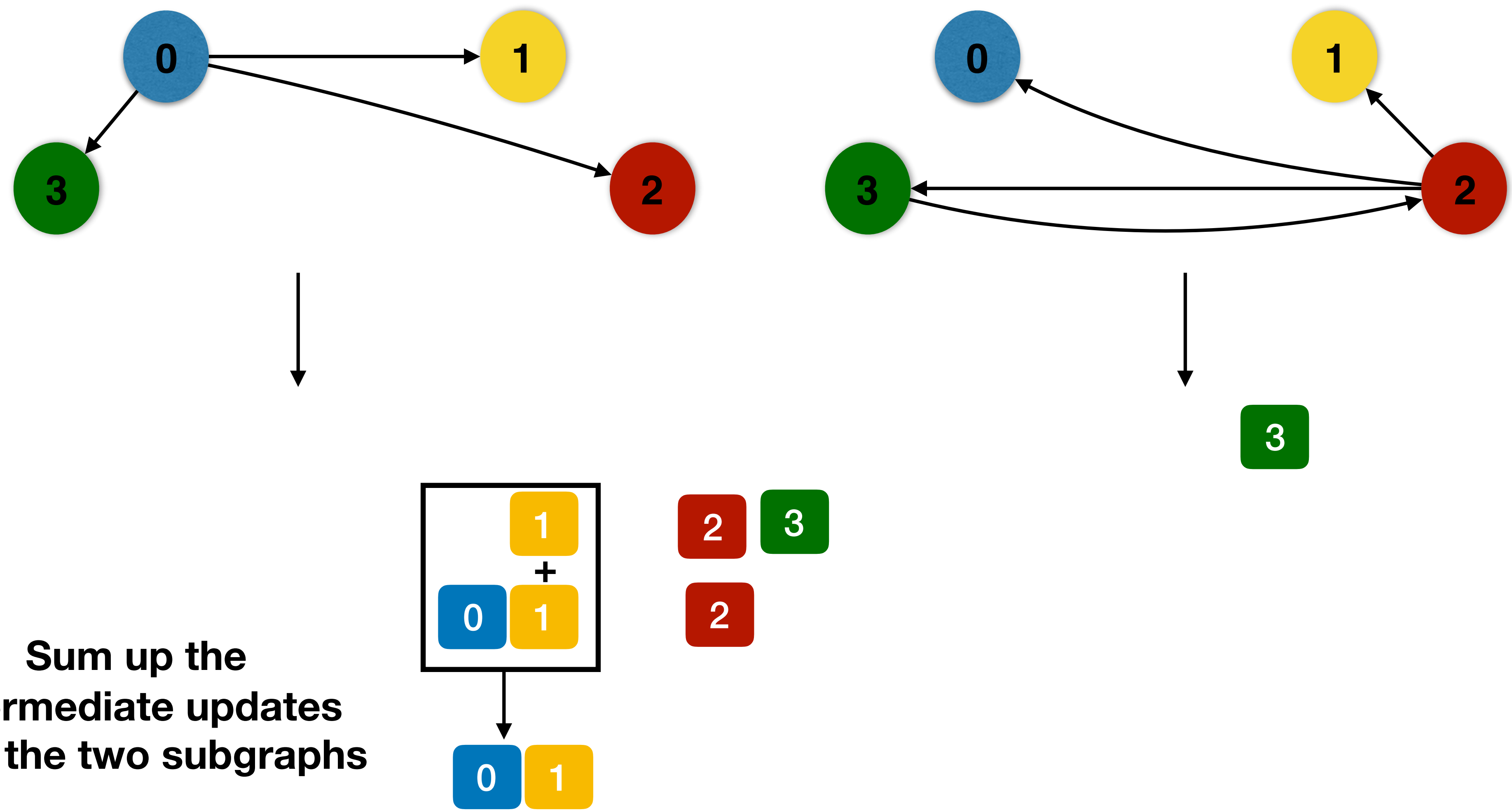
Sum up the intermediate updates from the two subgraphs

Cache-aware Merge



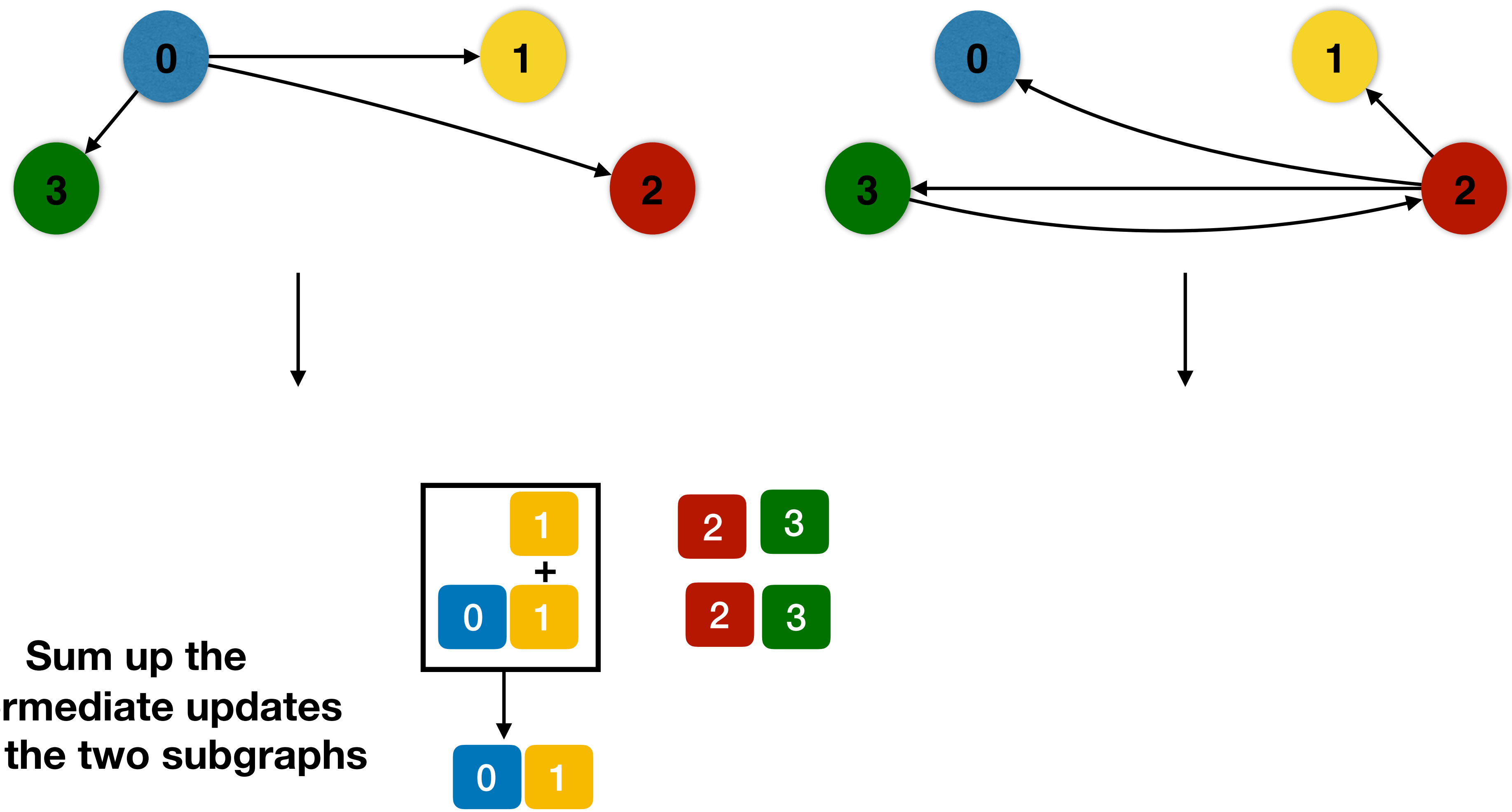
Sum up the intermediate updates from the two subgraphs

Cache-aware Merge



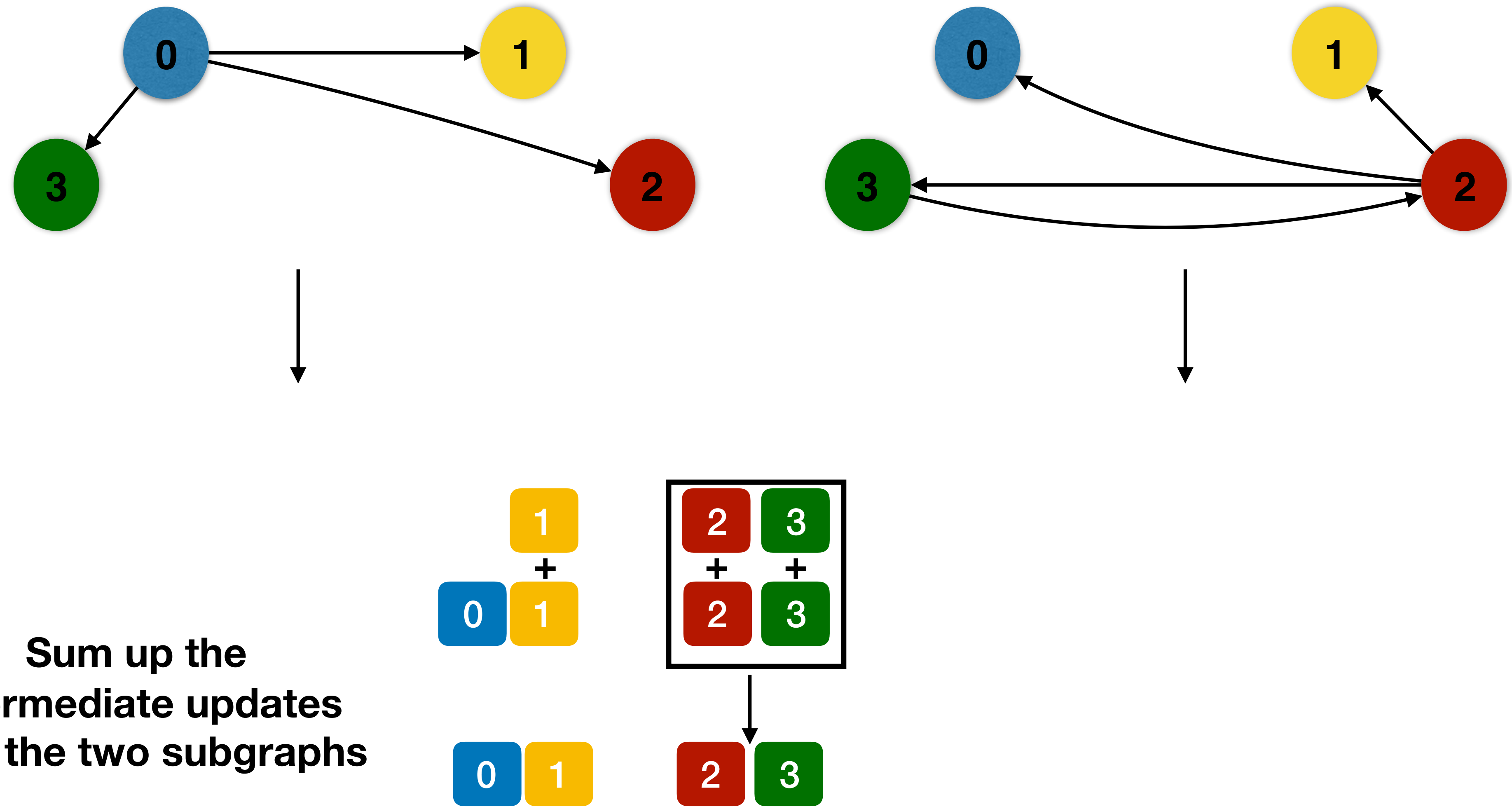
Sum up the intermediate updates from the two subgraphs

Cache-aware Merge



Sum up the intermediate updates from the two subgraphs

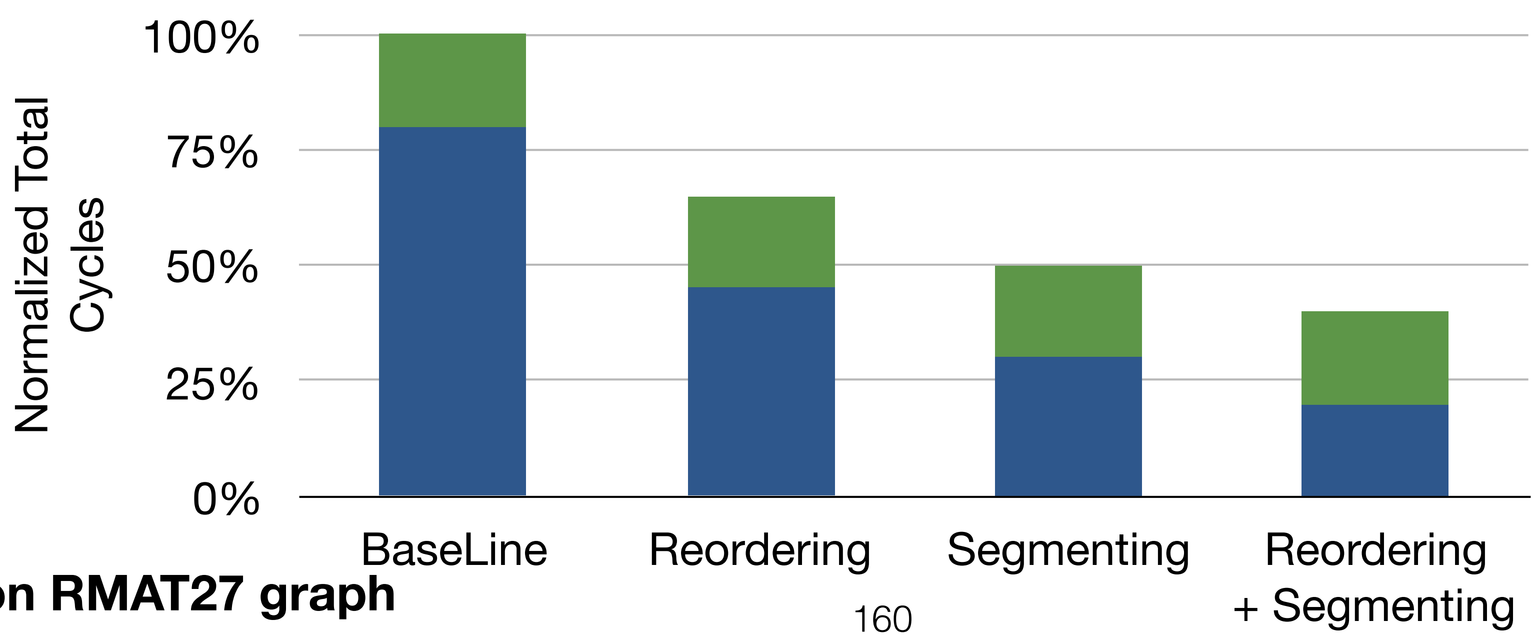
Cache-aware Merge



Sum up the intermediate updates from the two subgraphs

PageRank

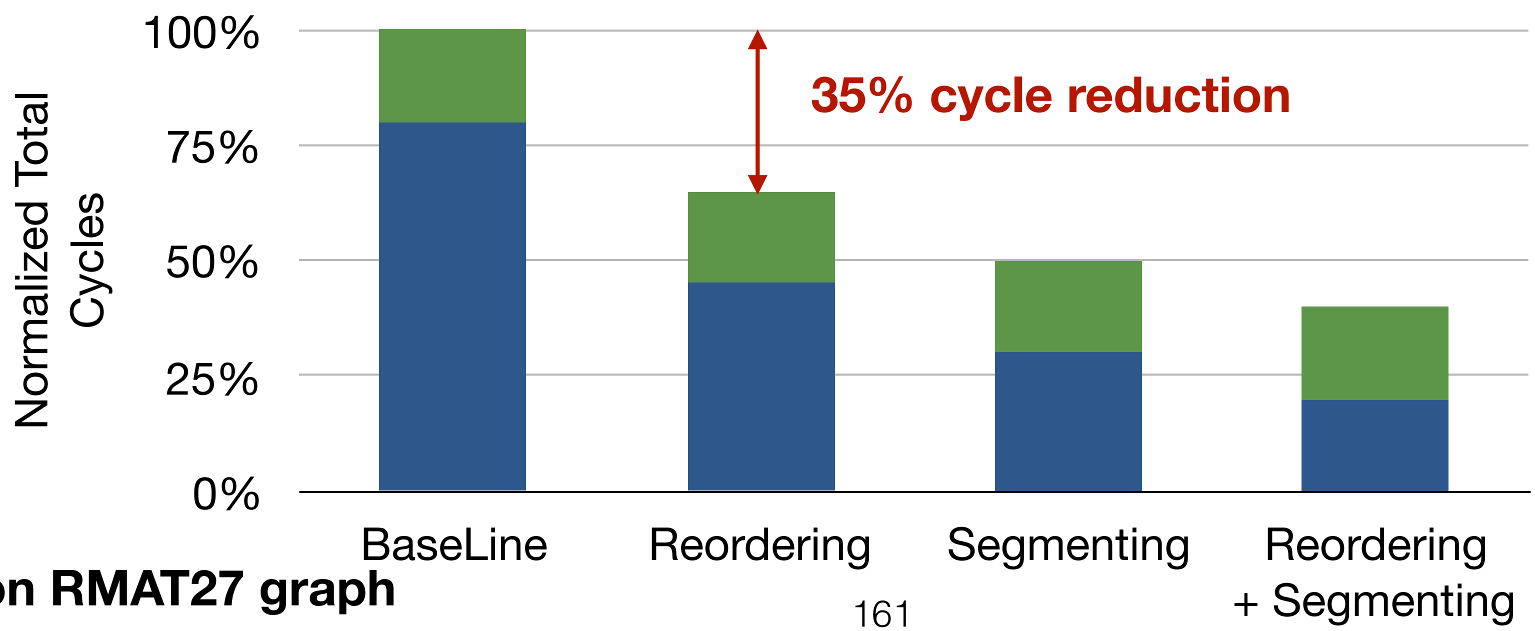
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



on RMAT27 graph

PageRank

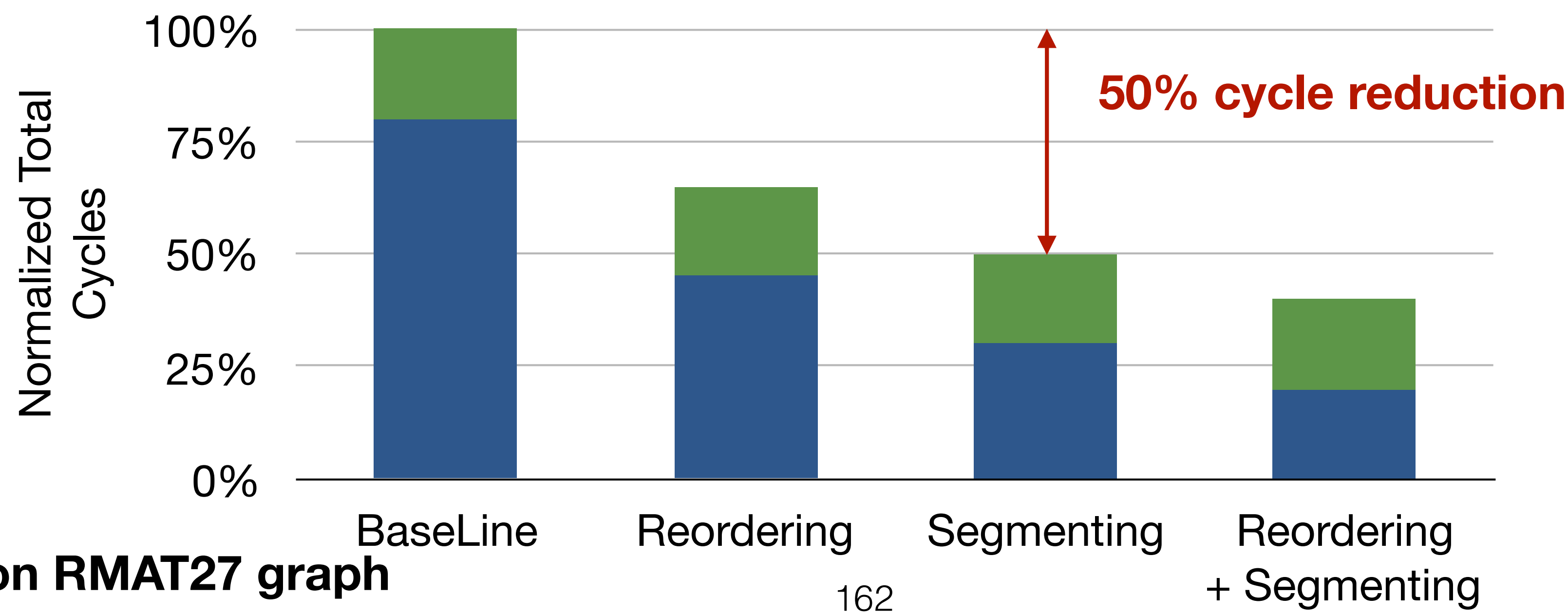
```
while …
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];
  swap ranks and newRanks
```



on RMAT27 graph

PageRank

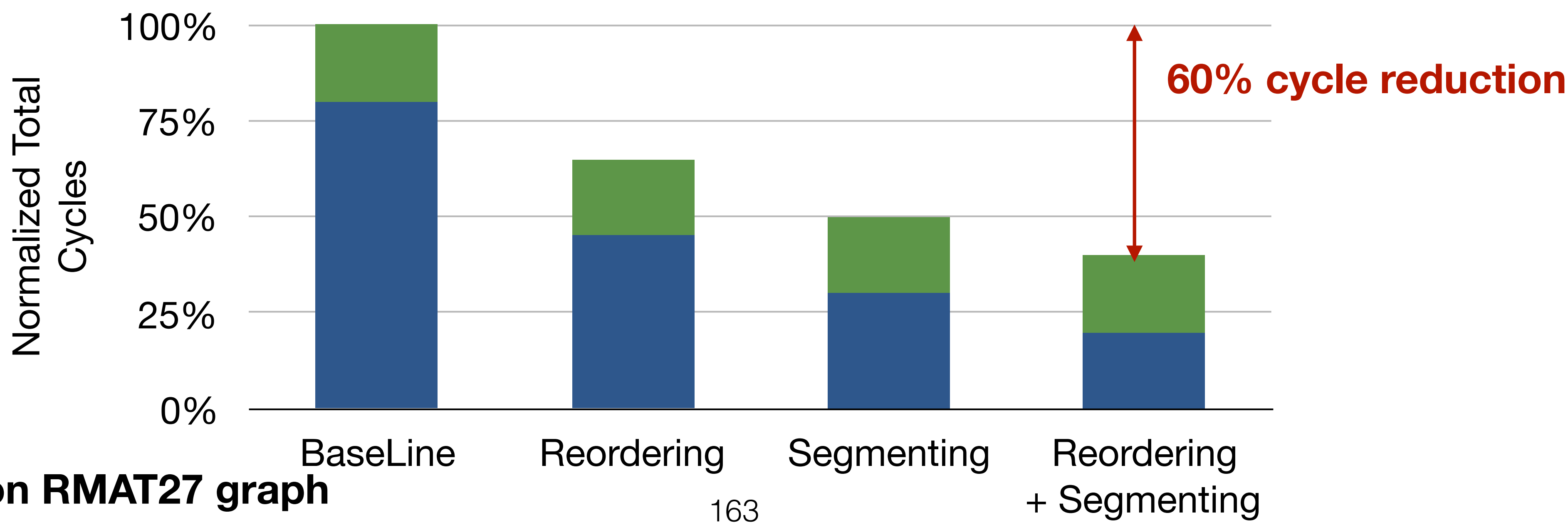
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



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

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, XStream)

Outline

- Motivation
- 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

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

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

Absolute Running Times on 24 core Intel Xeon E5 servers

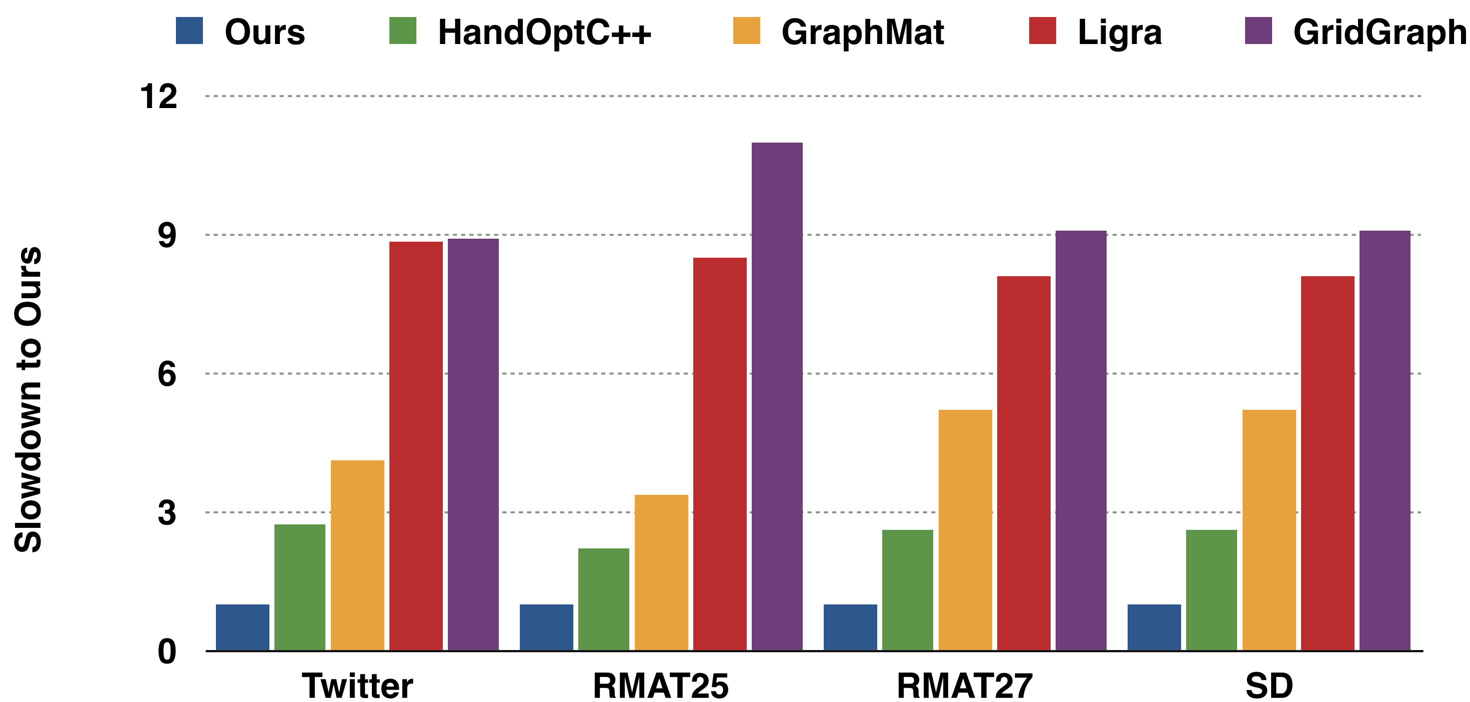
Evaluation

	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)

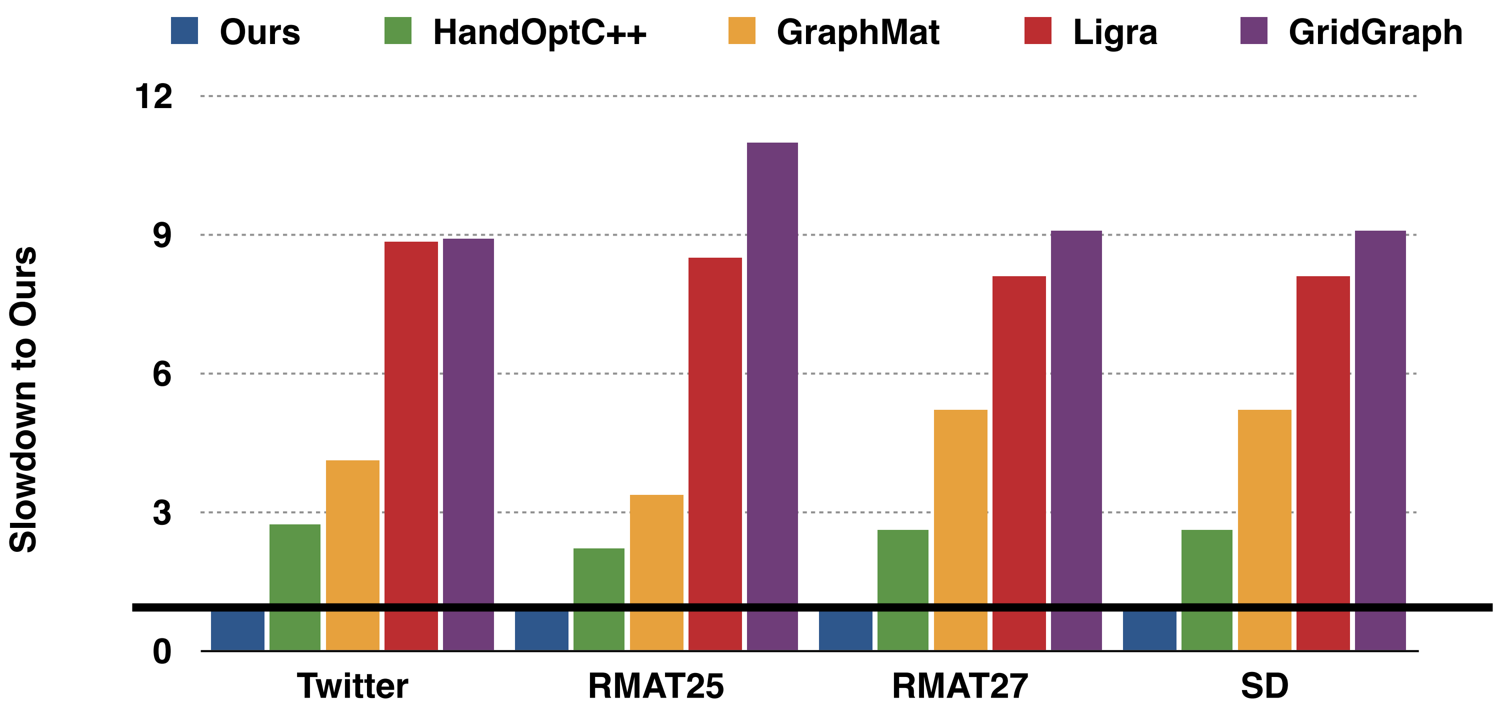
Absolute Running Times on 24 core Intel Xeon E5 servers

PageRank



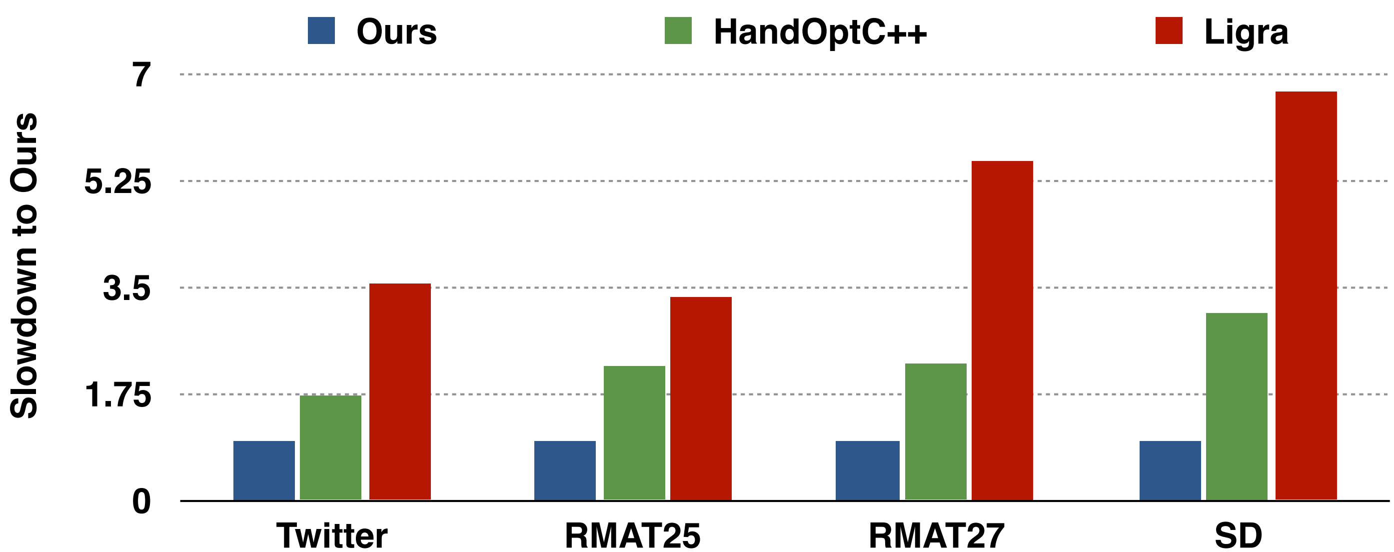
Ours
HandOptC++
GraphMat
Ligra
GridGraph
12
9
6
3
0
Slowdown to Ours
Twitter
RMAT25
RMAT27
SD

PageRank

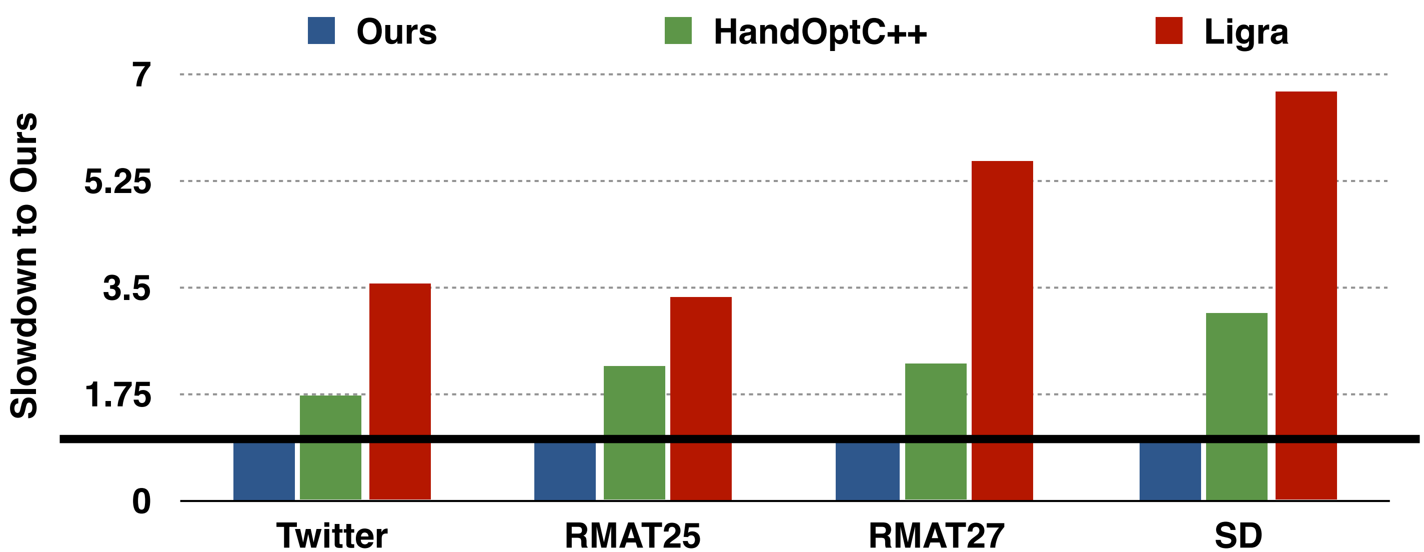


Intel expert hand optimized version and state-of-the art graph frameworks are 2.2-11x slower than our version

Label Propagation

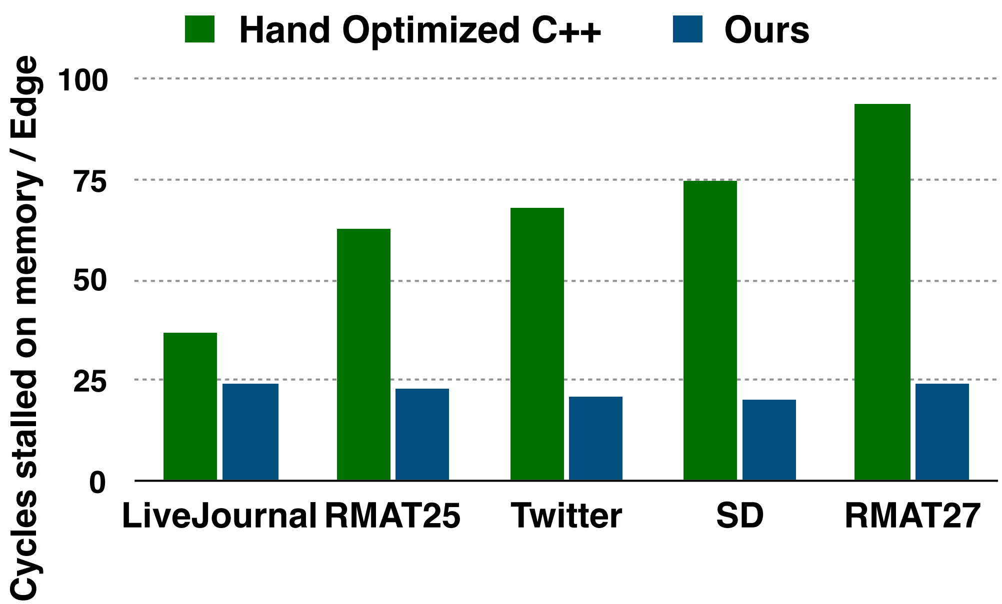


Label Propagation



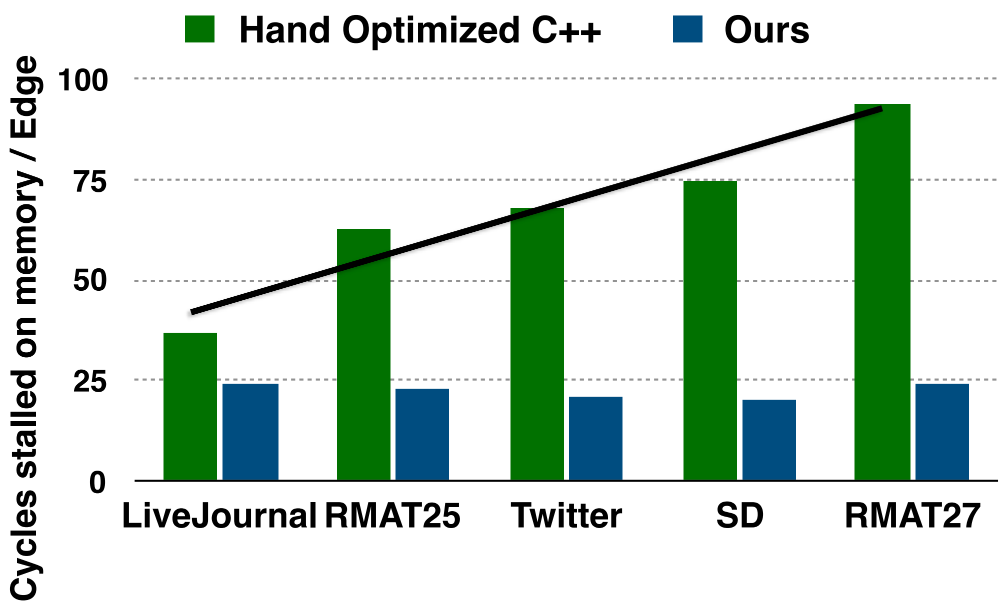
Intel expert hand optimized version and state-of-the art graph frameworks are 1.7-6.7x slower than our version

Evaluation



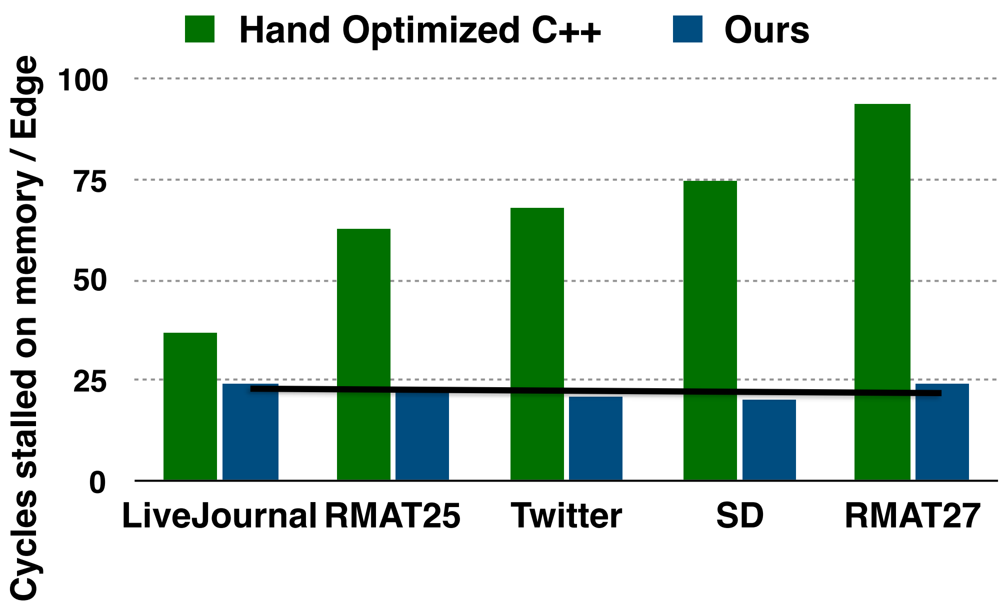
Hand Optimized C++
Ours
Cycles stalled on memory / Edge
100
75
50
25
0
LiveJournal
RMAT25
Twitter
SD
RMAT27

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

- Performance Analysis for Graph Applications
- Milk / Propagation Blocking
- Frequency based Clustering
- CSR Segmenting
- Summary

Improving Cache Performance for Graph Computations

- Reordering the Graph
- Partitioning the Graph for Locality
- Runtime Reordering the Memory Accesses

Improving Cache Performance for Graph Computations

- Reordering the Graph
- Partitioning the Graph for Locality
- Runtime Reordering the Memory Accesses

What are the tradeoffs ?

Improving Cache Performance for Graph Computations

- Reordering the Graph
  - Small preprocessing cost, modest performance improvement, dependent on graph structure
- Partitioning the Graph for Locality
  - Bigger preprocessing cost, small runtime overhead, bigger performance gains, suitable for applications with lots of random accesses.
- Runtime Reordering the Memory Accesses
  - No preprocessing cost, bigger runtime overhead

Outside of Graph Computing?

- Sparse Linear Algebra
  - Matrix Reordering, Preconditioning (Graph Reordering)
  - Cache Blocking (CSR segmenting)
  - Inspector-Executor (Runtime Access Reordering)
- These are Fundamental Communication Reductions Techniques , used in many other domains (sparse linear algebra, join optimization in databases)

Performance Engineering

- Understand your applications’ performance characteristics
    - Many papers worked on different ways to abandon cache and improve MLP with a large number of threads
- Understand the tradeoff space of the optimizations
    - Pick the technique that best suit your hardware, application and data