

# GraphChi: Large-Scale Graph Computation on Just a PC

Aapo Kyrölä, Guy Blelloch, Carlos Guestin

CJ QUINES

# GraphChi: Large-Scale Graph Computation on Just a PC

Aapo Kyrölä, Guy Blelloch, Carlos Guestin

CJ QUINES

So are “large-scale graphs”  
in the room with us right now?



So are “large-scale graphs”  
in the room with us right now?



unfortunately...

big graphs are **real**

10 million edges ■

big graphs are **real**

10 million edges ■

LinkedIn (2012) ■■■■■■■■

# big graphs are real

10 million edges ■

LinkedIn (2012) ■ ■ ■ ■ ■ ■ ■

LiveJournal (2006) ■ ■ ■ ■ ■ ■ ■

# big graphs are real

10 million edges



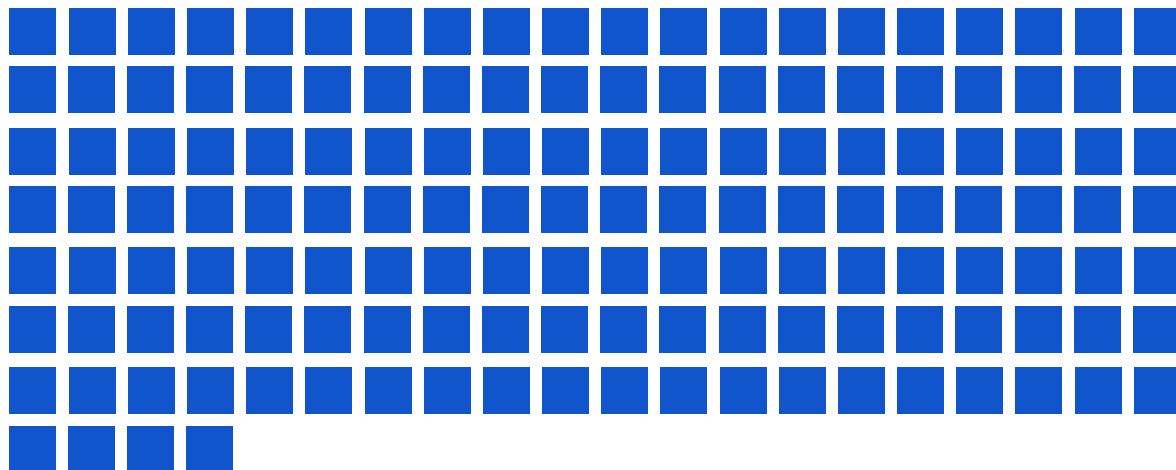
LinkedIn (2012)



LiveJournal (2006)

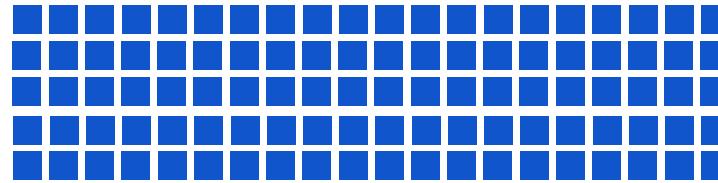


Twitter (2010)

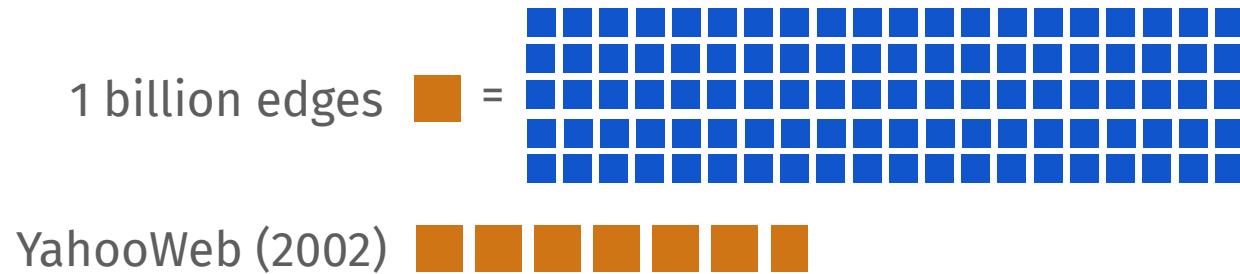


big graphs are real

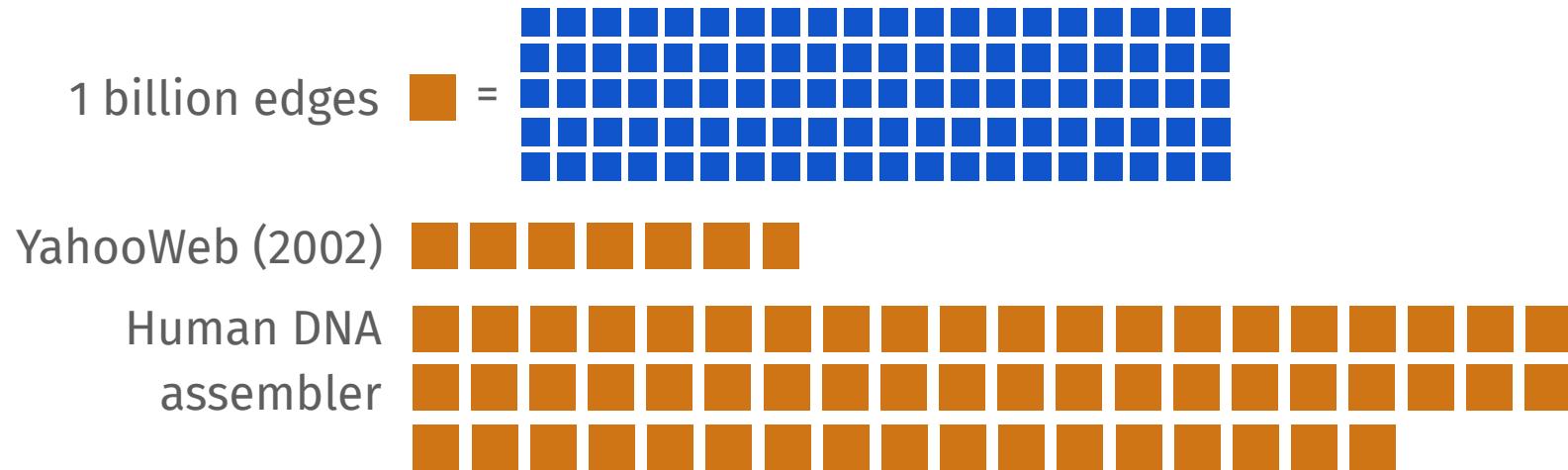
1 billion edges ■ =



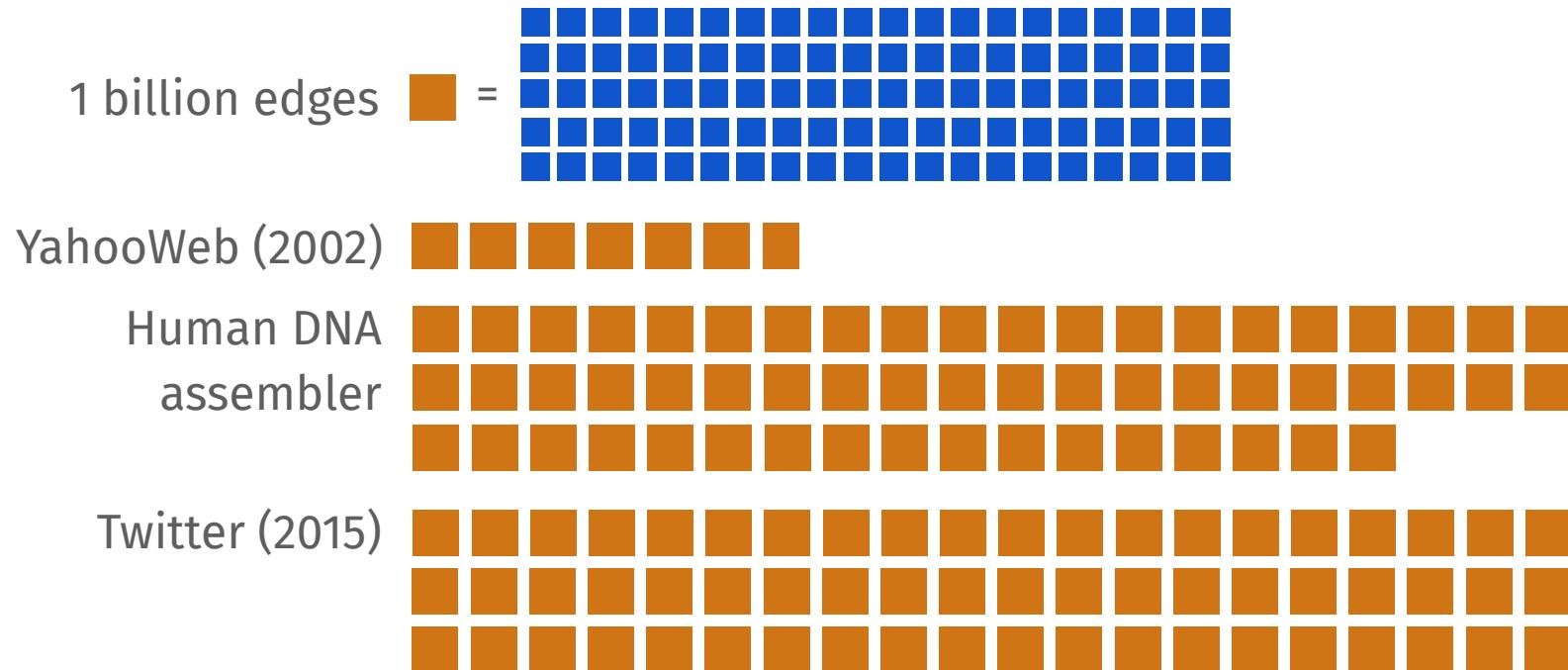
big graphs are real



# big graphs are real



# big graphs are real

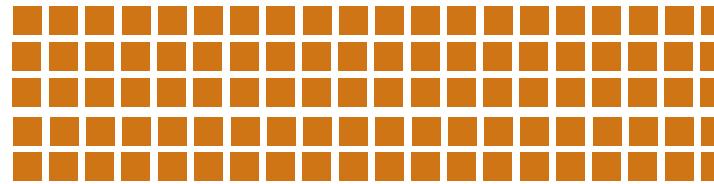


big graphs are real

100 billion edges

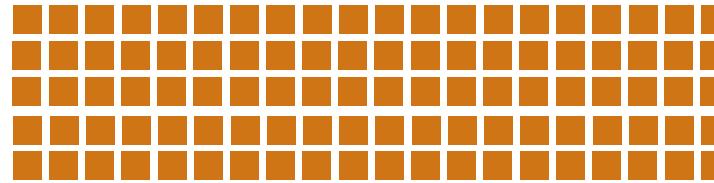


=



# big graphs are real

100 billion edges  =



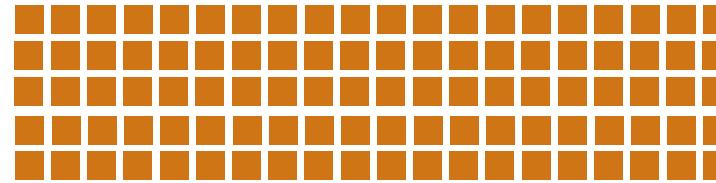
Facebook (2014)    

# big graphs are real

100 billion edges



=



Facebook (2014)

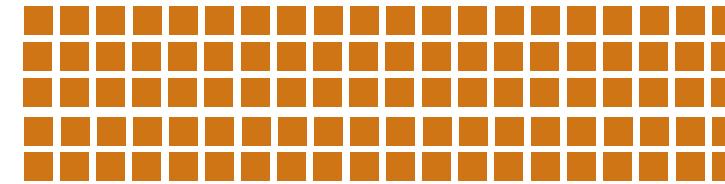


Knowledge Graph (2020)



# big graphs are real

100 billion edges



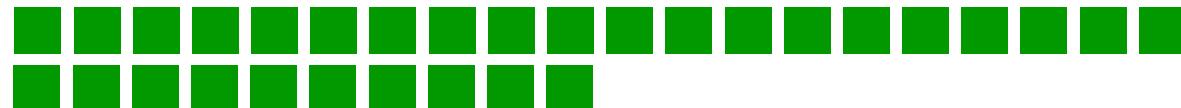
Facebook (2014)



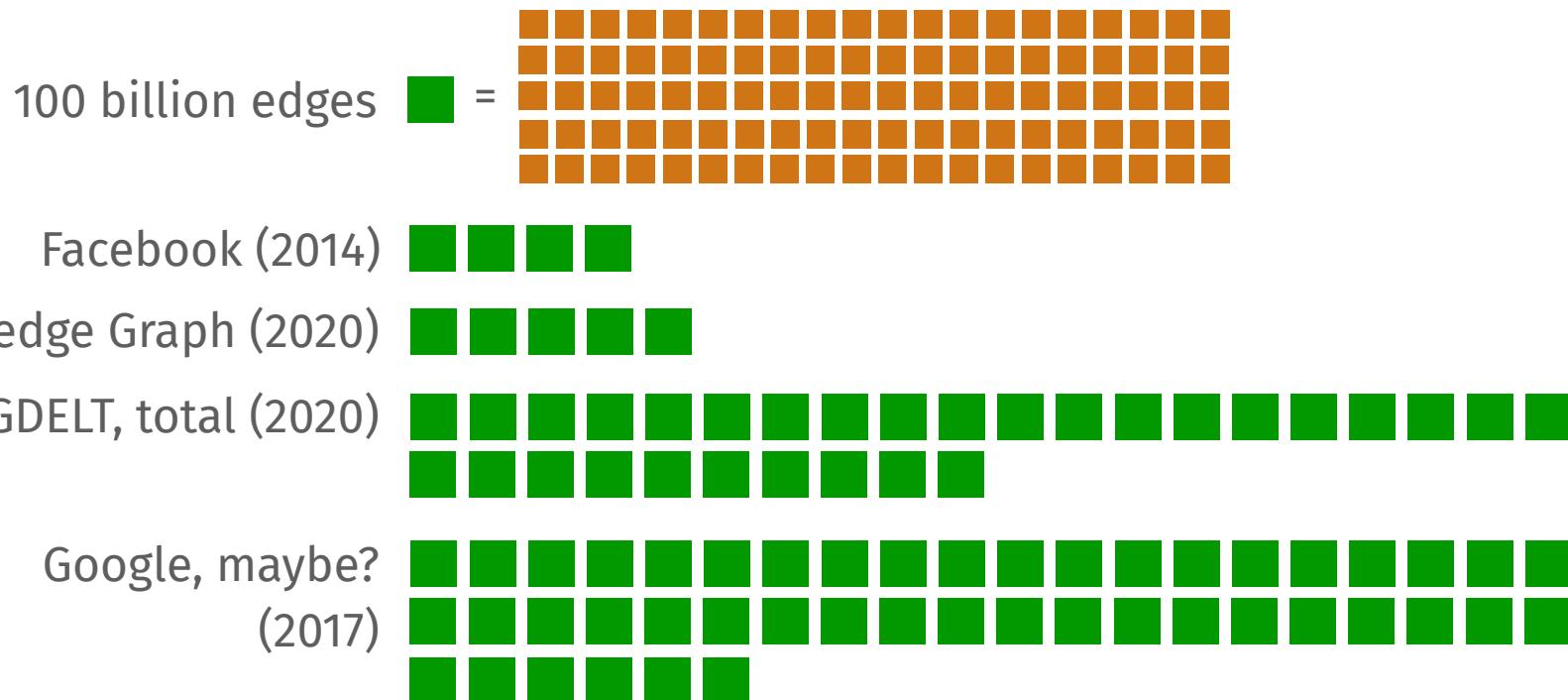
Knowledge Graph (2020)



GDELT, total (2020)



# big graphs are real



# GraphChi: Large-Scale Graph Computation on Just a PC

Aapo Kyrölä, Guy Blelloch, Carlos Guestin

CJ QUINES

# GraphChi: Large-Scale Graph Computation on **Just a PC**

Aapo Kyrölä, Guy Blelloch, Carlos Guestin

CJ QUINES

who would process  first?

 10 million edges

 1 billion edges

 100 billion edges

who would process  first?

100 big computers  
running Hadoop



 10 million edges

 1 billion edges

 100 billion edges

who would process  first?

100 big computers  
running Hadoop



one small boi  
running GraphChi



 10 million edges

 1 billion edges

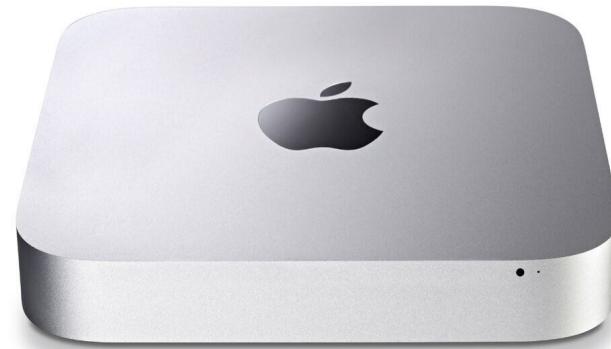
 100 billion edges

who would process  first?

100 big computers  
running Hadoop



one small boi  
running GraphChi



**22 minutes**

 10 million edges

 1 billion edges

 100 billion edges

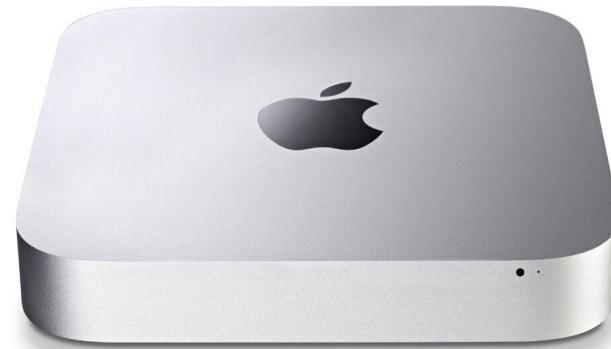
who would process  first?

100 big computers  
running Hadoop



**22 minutes**

one small boi  
running GraphChi



**27 minutes**

 10 million edges

 1 billion edges

 100 billion edges

who would process  first?

100 big computers  
running Hadoop



**22 minutes**

one small boi  
running GraphChi



**27 minutes**

 10 million edges

 1 billion edges

 100 billion edges



*i'm faster.....*

*but at what cost?*



# you're faster, but at what cost?

Application & Graph	Iter.	Comparative result	GraphChi (Mac Mini)
Pagerank & 	3	GraphLab[30] on AMD server (8 CPUs) <b>87 s</b>	<b>132 s</b>
Pagerank & 	5	Spark [45] with 50 nodes (100 CPUs): <b>486.6 s</b>	<b>790 s</b>
Pagerank & 	100	Stanford GPS, 30 EC2 nodes (60 virt. cores), <b>144 min</b>	approx. <b>581 min</b>
Pagerank & 	1	Piccolo, 100 EC2 instances (200 cores) <b>70 s</b>	approx. <b>26 min</b>
Webgraph-BP & 	1	Pegasus (Hadoop) on 100 machines: <b>22 min</b>	<b>27 min</b>
ALS & 	10	GraphLab on AMD server: <b>4.7 min</b>	<b>9.8 min</b> (in-mem) <b>40 min</b> (edge-repl.)
Triangle-count & 	-	Hadoop, 1636 nodes: <b>423 min</b>	<b>60 min</b>
Pagerank & 	1	PowerGraph, 64 x 8 cores: <b>3.6 s</b>	<b>158 s</b>
Triangle-count & 	-	PowerGraph, 64 x 8 cores: <b>1.5 min</b>	<b>60 min</b>

 10 million edges

 1 billion edges

 100 billion edges

# GraphChi: Large-Scale Graph Computation on **Just a PC**

Aapo Kyrölä, Guy Blelloch, Carlos Guestin

CJ QUINES

# GraphChi: Large-Scale Graph Computation on Just a PC

Aapo Kyrölä, Guy Blelloch, Carlos Guestin

CJ QUINES

# obligatory outline slide

■ 10 million edges

■ 1 billion edges

■ 100 billion edges

# obligatory outline slide

- size limitations
  - can  fit in a Mac Mini?

 10 million edges

 1 billion edges

 100 billion edges

# obligatory outline slide

- size limitations
  - can  fit in a Mac Mini?
- access pattern speed
  - “put it on disk and call it a day” doesn’t work



10 million edges



1 billion edges



100 billion edges

# obligatory outline slide

- size limitations
  - can  fit in a Mac Mini?
- access pattern speed
  - “put it on disk and call it a day” doesn’t work
- parallel sliding window
  - you could’ve built GraphChi (maybe)



10 million edges



1 billion edges



100 billion edges

# ACT I

*size limitations*

*access pattern speed*

*parallel sliding window*

can  fit in a Mac Mini?

 10 million edges

 1 billion edges

 100 billion edges

can  fit in a Mac Mini?

- how big is an edge?

 10 million edges

 1 billion edges

 100 billion edges

can  fit in a Mac Mini?

- how big is an edge?
  - no compression; we're operating on edges in memory

 10 million edges

 1 billion edges

 100 billion edges

can  fit in a Mac Mini?

- how big is an edge?
  - no compression; we're operating on edges in memory
  - let's say it's 64 bits = 8 bytes

 10 million edges

 1 billion edges

 100 billion edges

can  fit in a Mac Mini?

- how big is an edge?
  - no compression; we're operating on edges in memory
  - let's say it's 64 bits = 8 bytes
- that means  =  $10^7$  edges =  $80^7$  bytes = 80 MB

 10 million edges

 1 billion edges

 100 billion edges

can ■■■■■■■ fit in a Mac Mini?

- how big is an edge?
  - no compression; we're operating on edges in memory
  - let's say it's 64 bits = 8 bytes
- that means ■ =  $10^7$  edges =  $8 \times 10^7$  bytes = 80 MB
- so ■ = 100 ■ = 8 GB

■ 10 million edges

■ 1 billion edges

■ 100 billion edges

can  fit in a Mac Mini?

- how big is an edge?
  - no compression; we're operating on edges in memory
  - let's say it's 64 bits = 8 bytes
- that means  =  $10^7$  edges =  $8 \times 10^7$  bytes = 80 MB
- so  = 100  = 8 GB
- and  = 100  = 800 GB

 10 million edges

 1 billion edges

 100 billion edges

does our math work out?

 10 million edges = 80 MB

 1 billion edges = 8 GB

 100 billion edges = 800 GB

# does our math work out?

- the Mac Mini used in the paper has 8 GB of RAM

 10 million edges = 80 MB

 1 billion edges = 8 GB

 100 billion edges = 800 GB

# does our math work out?

- the Mac Mini used in the paper has 8 GB of RAM
- according to our math, ■ barely fits

■ 10 million edges = 80 MB

■ 1 billion edges = 8 GB

■ 100 billion edges = 800 GB

# does our math work out?

- the Mac Mini used in the paper has 8 GB of RAM
- according to our math, ■ barely fits
- according to the paper, it's around 

■ 10 million edges = 80 MB

■ 1 billion edges = 8 GB

■ 100 billion edges = 800 GB

# does our math work out?

- the Mac Mini used in the paper has 8 GB of RAM
- according to our math,  barely fits
- according to the paper, it's around 
- so we underestimated by an order of magnitude, why?

 10 million edges = 80 MB

 1 billion edges = 8 GB

 100 billion edges = 800 GB

# does our math work out?

- the Mac Mini used in the paper has 8 GB of RAM
- according to our math,  barely fits
- according to the paper, it's around 
- so we underestimated by an order of magnitude, why?
- answer: graphchi limits edges to 1 GB. why?

 10 million edges = 80 MB

 1 billion edges = 8 GB

 100 billion edges = 800 GB

# does our math work out?

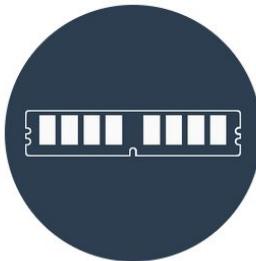
- the Mac Mini used in the paper has 8 GB of RAM
- according to our math,  barely fits
- according to the paper, it's around 
- so we underestimated by an order of magnitude, why?
- answer: graphchi limits edges to 1 GB. why?
  - why won't more RAM help?

 10 million edges = 80 MB

 1 billion edges = 8 GB

 100 billion edges = 800 GB

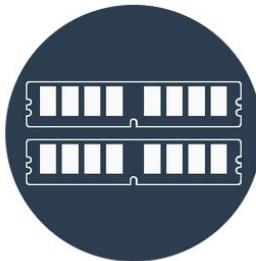
# “just add more RAM”



4 GB

DDR4-2400  
10-12-10-27  
1.65V

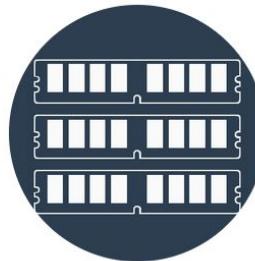
SELECT PLAN



8 GB

DDR4-2400  
10-12-10-27  
1.65V

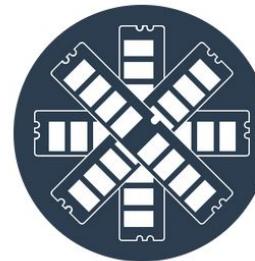
SELECT PLAN



16 GB

DDR4-2400  
10-12-10-27  
1.65V

SELECT PLAN



32 GB

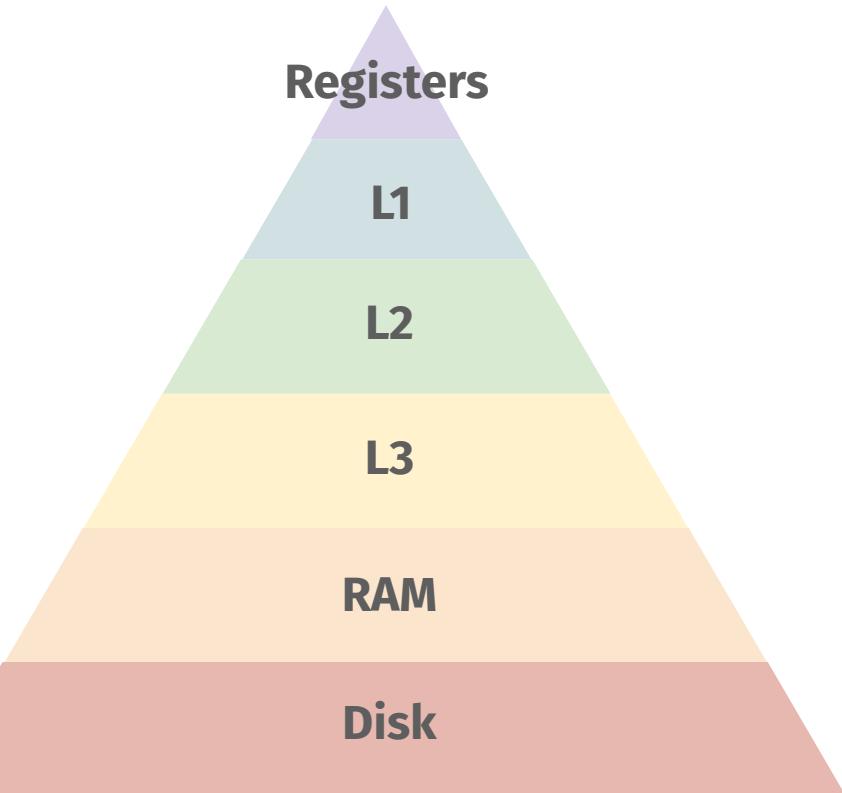
DDR4-2400  
10-12-10-27  
1.65V

SELECT PLAN

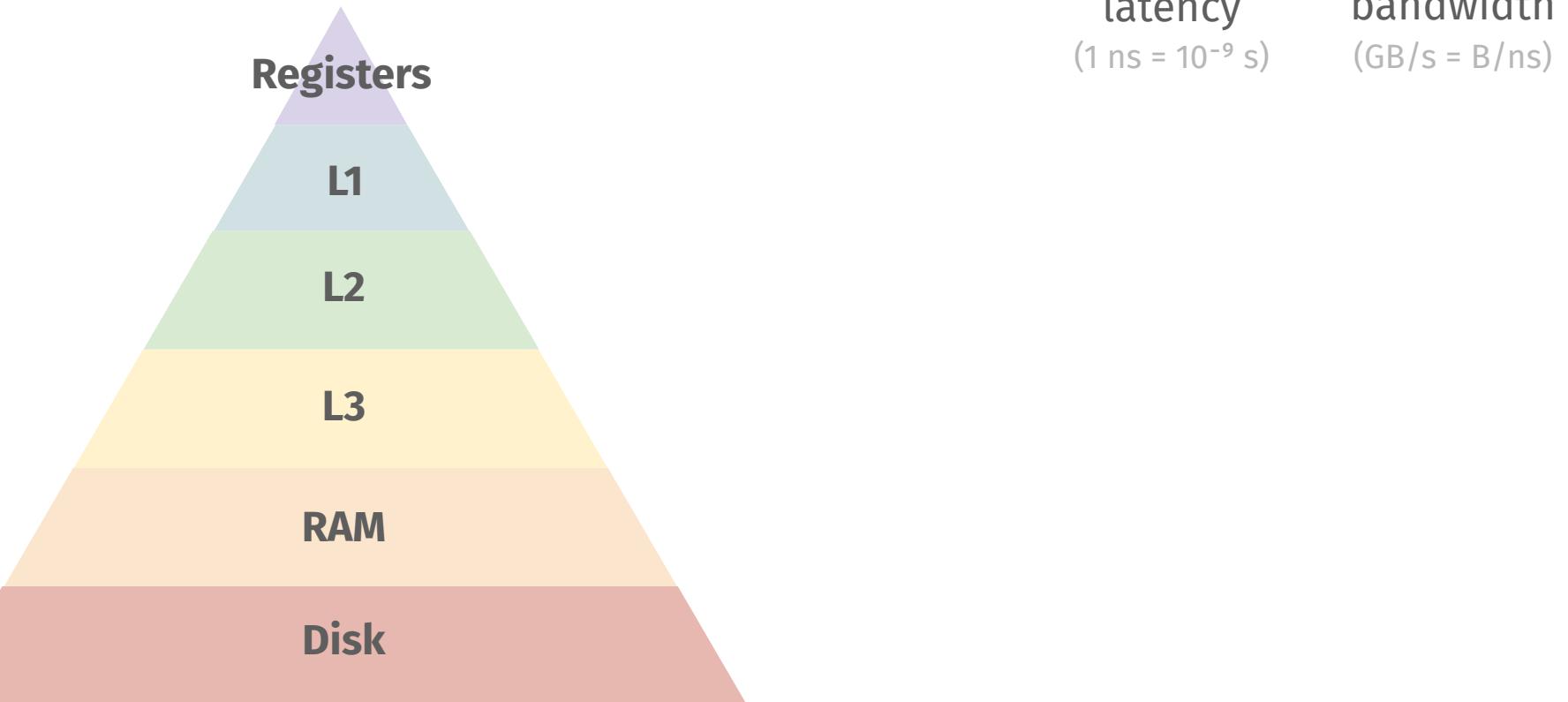
<https://downloadmoreram.com>

# the memory hierarchy

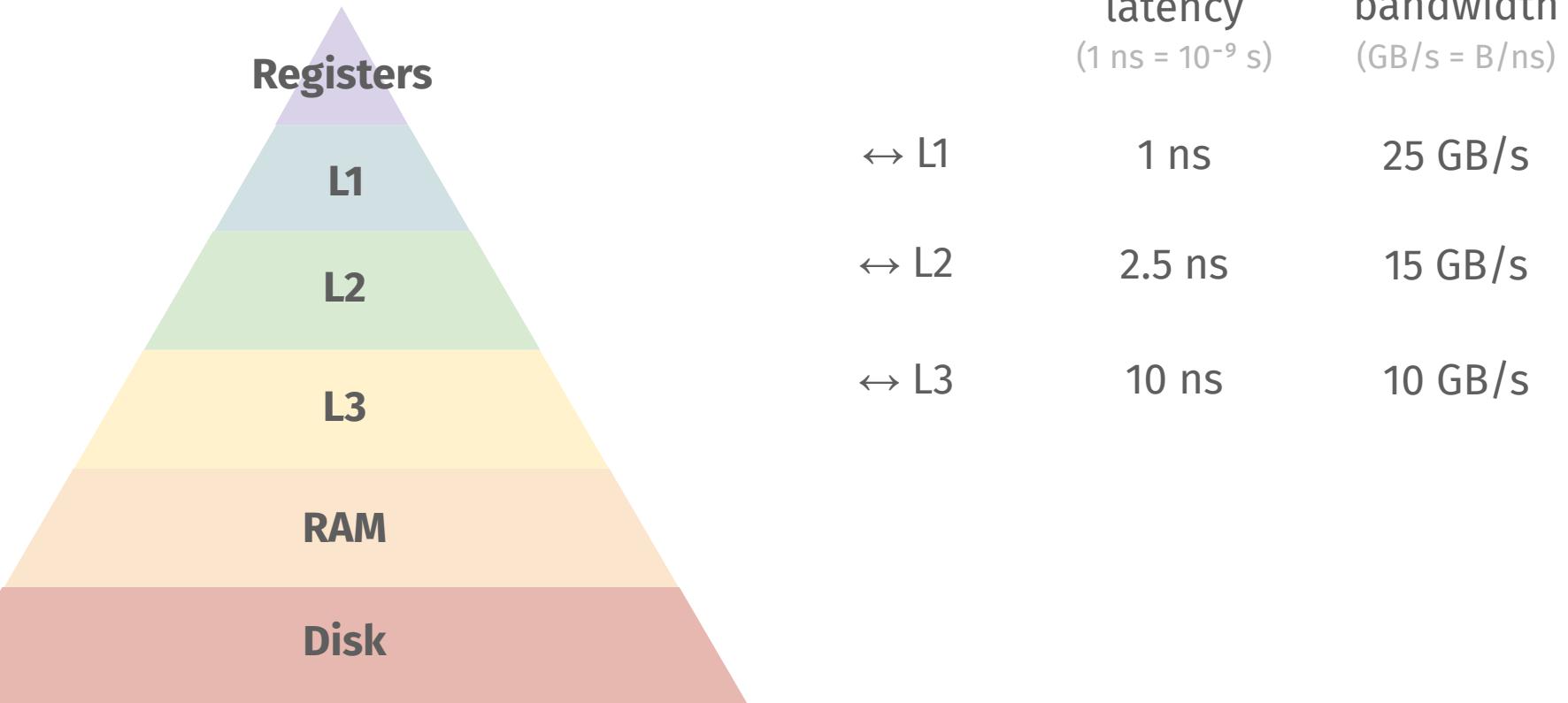
# the memory hierarchy



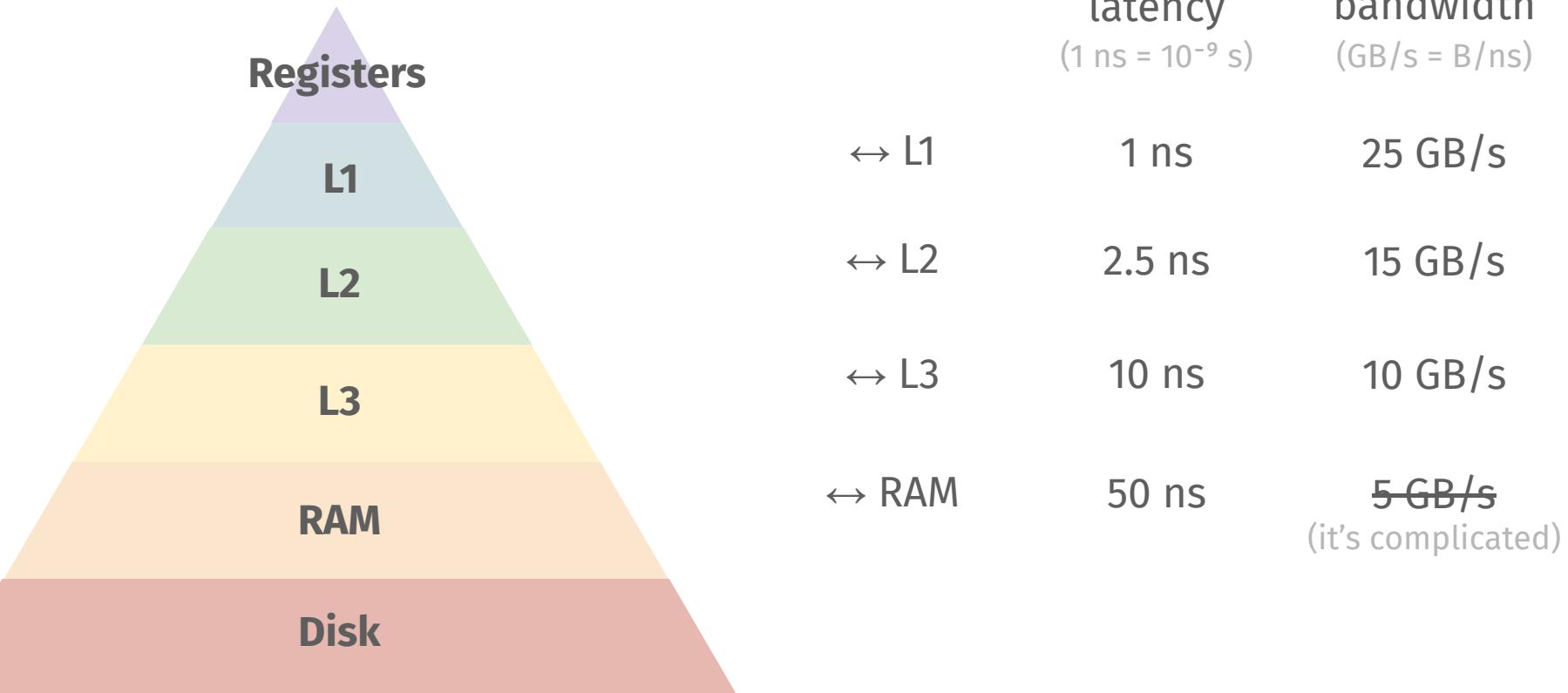
# the memory hierarchy



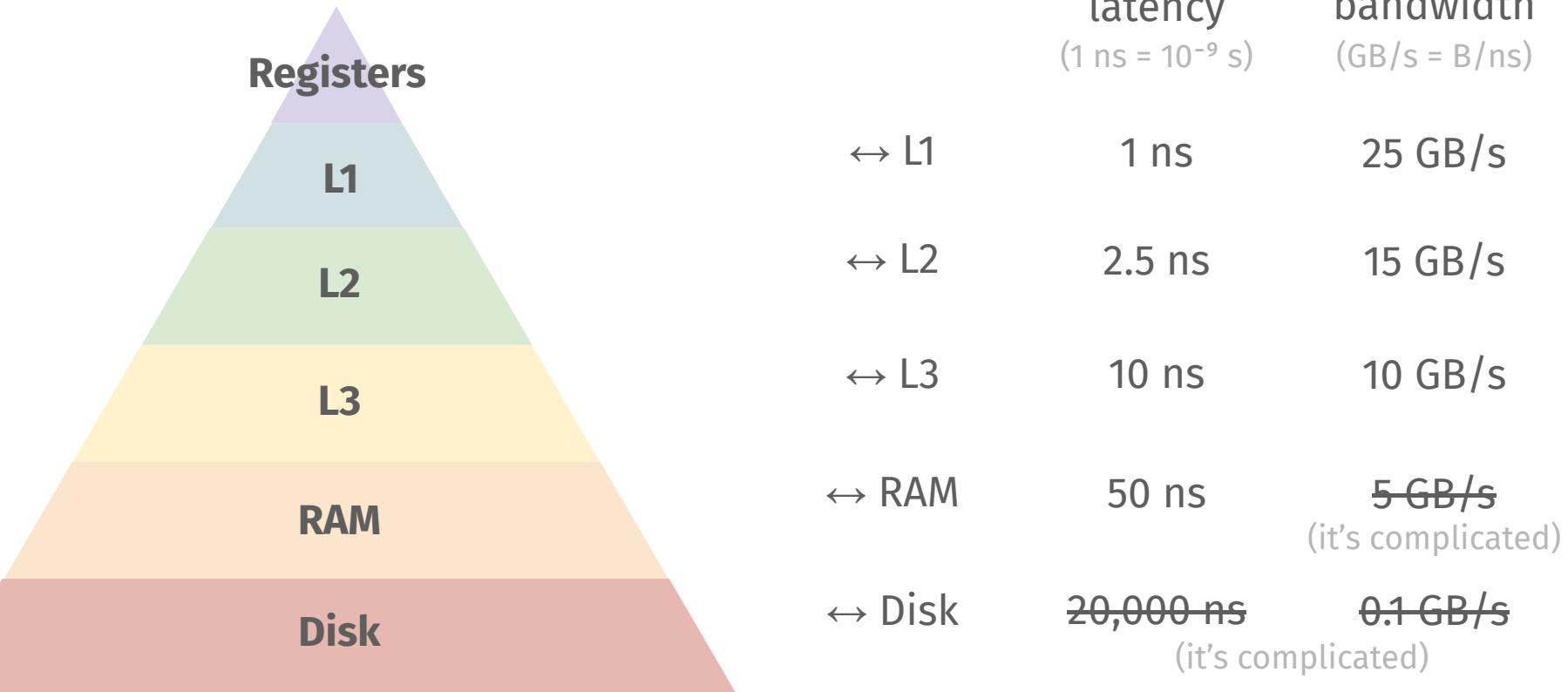
# the memory hierarchy



# the memory hierarchy



# the memory hierarchy



bandwidth is a limit too

 10 million edges = 80 MB

 1 billion edges = 8 GB

 100 billion edges = 800 GB

# bandwidth is a limit too

- more memory runs into diminishing returns

 10 million edges = 80 MB

 1 billion edges = 8 GB

 100 billion edges = 800 GB

# bandwidth is a limit too

- more memory runs into diminishing returns
- eventually, the bottleneck is bandwidth

 10 million edges = 80 MB

 1 billion edges = 8 GB

 100 billion edges = 800 GB

# bandwidth is a limit too

- more memory runs into diminishing returns
- eventually, the bottleneck is bandwidth
  - especially if we're compute-light:

 10 million edges = 80 MB

 1 billion edges = 8 GB

 100 billion edges = 800 GB

# bandwidth is a limit too

- more memory runs into diminishing returns
- eventually, the bottleneck is bandwidth
  - especially if we're compute-light:

Application	SSD	In-mem	Ratio
Connected components	45 s	18 s	2.5x
Community detection	110 s	46 s	2.4x
Matrix fact. (D=5, 5 iter)	114 s	65 s	1.8x
Matrix fact. (D=20, 5 iter.)	560 s	500 s	1.1x

■ 10 million edges = 80 MB

■ 1 billion edges = 8 GB

■ 100 billion edges = 800 GB

# bandwidth is a limit too

- more memory runs into diminishing returns
- eventually, the bottleneck is bandwidth
  - especially if we're compute-light:

Application	SSD	In-mem	Ratio
Connected components	45 s	18 s	2.5x
Community detection	110 s	46 s	2.4x
Matrix fact. (D=5, 5 iter)	114 s	65 s	1.8x
Matrix fact. (D=20, 5 iter.)	560 s	500 s	1.1x

- so we can dump things on disk and call it a day right?

■ 10 million edges = 80 MB

■ 1 billion edges = 8 GB

■ 100 billion edges = 800 GB

# ACT I

*size limitations*

*access pattern speed*

*parallel sliding window*

## ACT II

*size limitations*

*access pattern speed*

*parallel sliding window*

numbers i made up

- remember this?

# numbers i made up

- remember this?

	latency (1 ns = $10^{-9}$ s)	bandwidth (GB/s = B/ns)
↔ RAM	50 ns	<del>5 GB/s</del> (it's complicated)
↔ Disk	<del>20,000 ns</del>	<del>0.1 GB/s</del> (it's complicated)

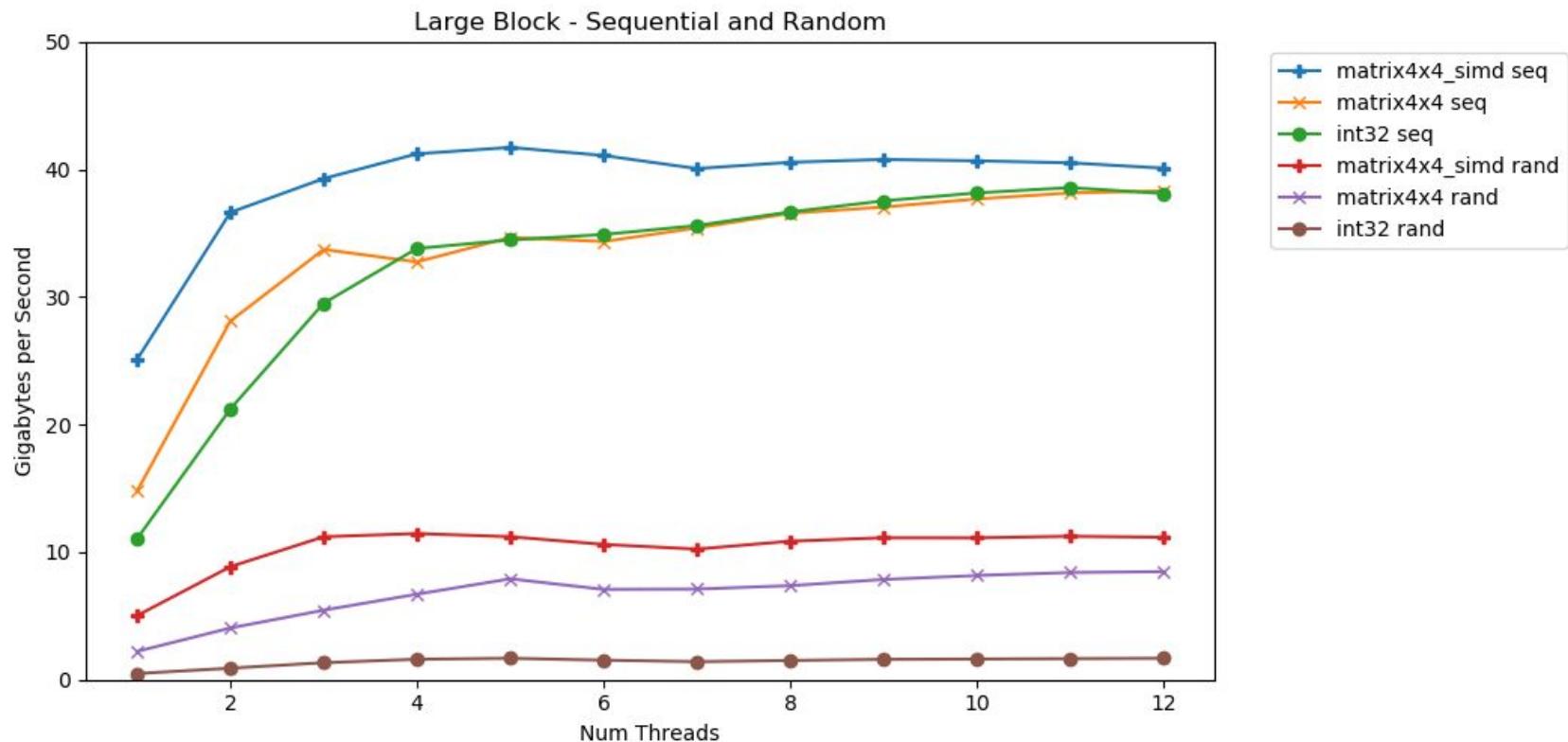
# numbers i made up

- remember this?

	latency (1 ns = $10^{-9}$ s)	bandwidth (GB/s = B/ns)
↔ RAM	50 ns	<del>5 GB/s</del> (it's complicated)
↔ Disk	<del>20,000 ns</del>	<del>0.1 GB/s</del> (it's complicated)

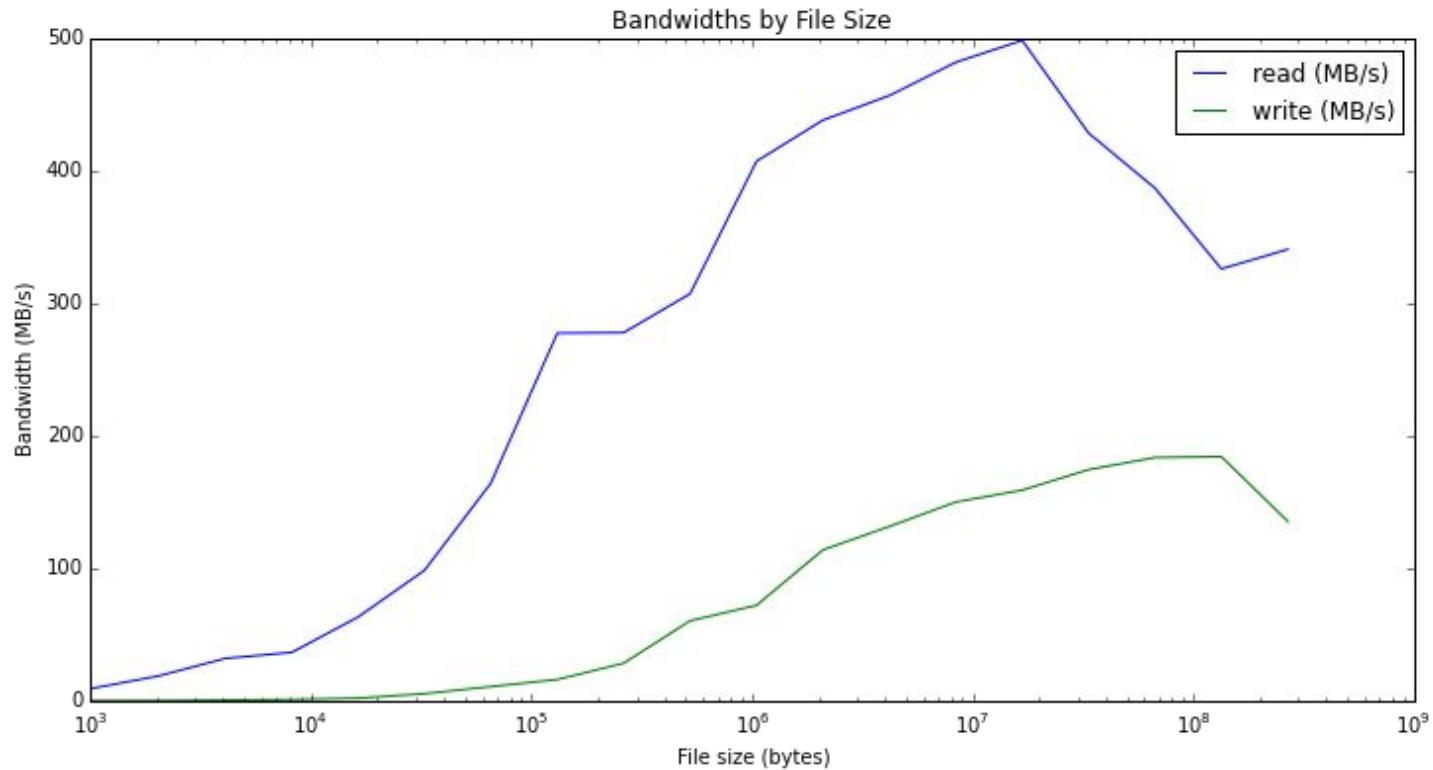
- it's complicated because of random vs. sequential access

RAM, read/write ints, look at  and 



# SSD, read/write multiple files

random regime  
(many small blocks)



sequential regime  
(less large blocks)

numbers i mostly made up

numbers i mostly made up

	random	sequential
↔ RAM	latency (1 ns = $10^{-9}$ s)	bandwidth (GB/s = B/ns)
↔ SSD		
↔ HDD		

numbers i mostly made up

	random	sequential
	latency (1 ns = $10^{-9}$ s)	bandwidth (GB/s = B/ns)
↔ RAM	50 ns	0.5 GB/s
↔ SSD	20,000 ns	0.005 GB/s
↔ HDD	2,000,000 ns	0.001 GB/s

numbers i mostly made up

	random		sequential	
	latency (1 ns = $10^{-9}$ s)	bandwidth (GB/s = B/ns)	latency (1 ns = $10^{-9}$ s)	bandwidth (GB/s = B/ns)
↔ RAM	50 ns	0.5 GB/s	50 ns	10 GB/s
↔ SSD	20,000 ns	0.005 GB/s	20,000 ns	0.5 GB/s
↔ HDD	2,000,000 ns	0.001 GB/s	500,000 ns	0.25 GB/s

random access problem

# random access problem

- why can't we “put it on disk and call it a day”?

# random access problem

- why can't we “put it on disk and call it a day”?
- the vertex-centric model has lots of random access patterns

# analyzing memory access patterns

```
def update(v):
    for e in v.outedges:
        e.value = v.value
    v.value = sum(
        e.value for e in v.inedges
    )
```

# analyzing memory access patterns

```
def update(v):
    for e in v.outedges:
        e.value = v.value
    v.value = sum(
        e.value for e in v.inedges
    )
```



# analyzing memory access patterns

```
read  
def update(v):  
    for e in v.outedges:  
        e.value = v.value  
    v.value = sum(  
        e.value for e in v.inedges  
    )
```



?



miss



hit

# analyzing memory access patterns

memory layout

read



```
def update(v):  
    for e in v.outedges:  
        e.value = v.value  
  
    v.value = sum(  
        e.value for e in v.inedges  
    )
```

...

v

...



?



miss



hit

# analyzing memory access patterns

memory layout

```
read      read
def update(v):
    for e in v.outedges:
        e.value = v.value
    v.value = sum(
        e.value for e in v.inedges
    )
```

...  
v  
...



?



miss

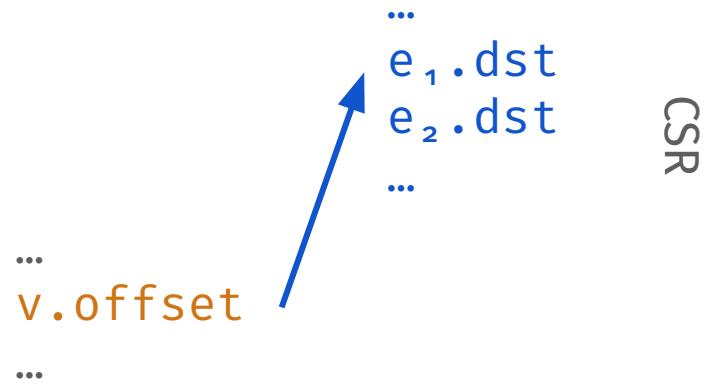


hit

# analyzing memory access patterns

memory layout

```
read      read
def update(v):
    for e in v.outedges:
        e.value = v.value
        v.value = sum(
            e.value for e in v.inedges
        )
```



?



miss



hit

# analyzing memory access patterns

memory layout

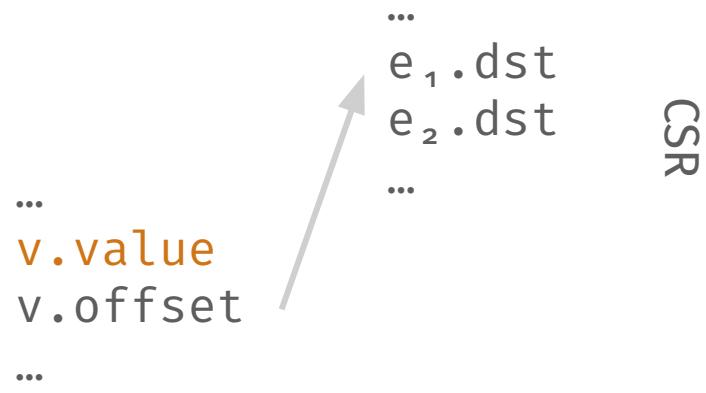
```
def update(v):
    for e in v.outedges:
        e.value = v.value
    v.value = sum(
        e.value for e in v.inedges
    )
```



# analyzing memory access patterns

memory layout

```
def update(v):
    for e in v.outedges:
        e.value = v.value
    v.value = sum(
        e.value for e in v.inedges
    )
```



?



miss



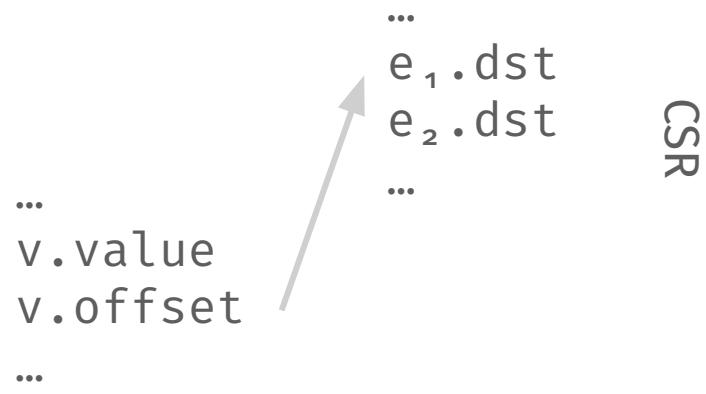
hit

# analyzing memory access patterns

memory layout

```
def update(v):
    for e in v.outedges:
        e.value = v.value
        v.value = sum(
            e.value for e in v.inedges
        )
```

The code shows a loop where each edge's value is updated based on its vertex's value. The memory layout diagram on the right shows vertices (v) and edges (e). The code involves multiple reads (blue arrows) and one write (green arrow) per iteration.



?



miss



hit

# analyzing memory access patterns

memory layout

```
def update(v):
    for e in v.outedges:
        e.value = v.value
        v.value = sum(
            e.value for e in v.inedges
        )
```

write  
(looped)

read

read

read

...

v.value

v.offset

...

CSR

...

e<sub>1</sub>.dst

e<sub>1</sub>.value

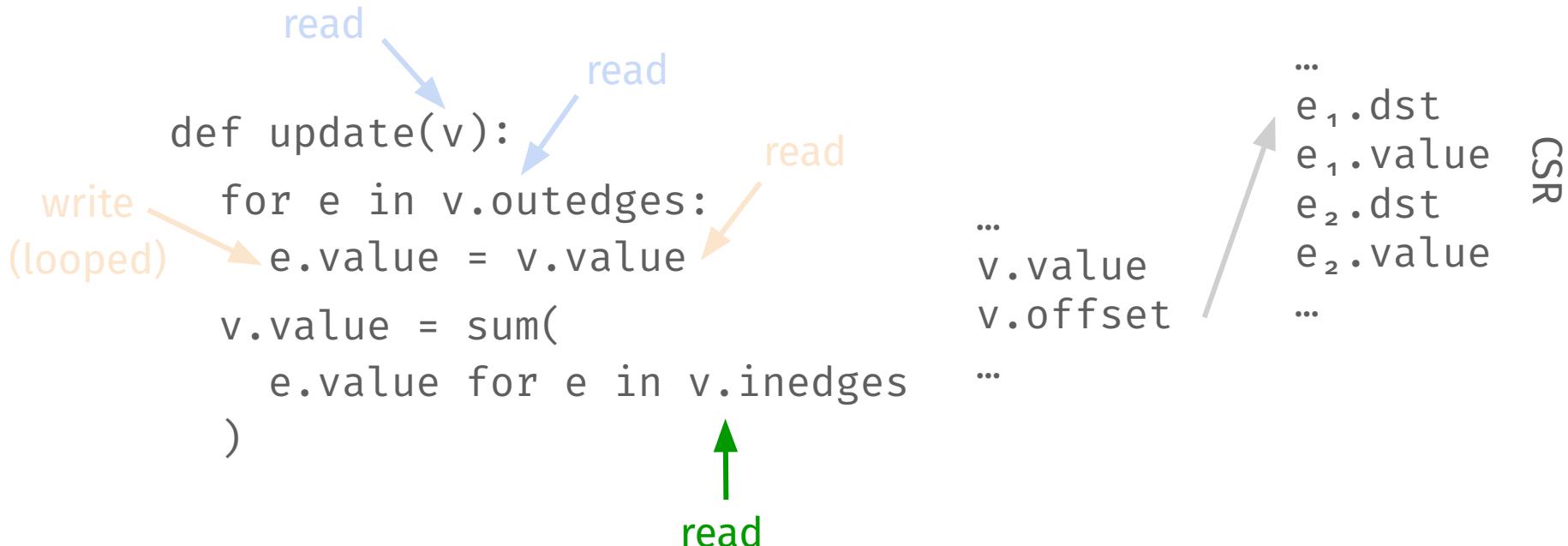
e<sub>2</sub>.dst

e<sub>2</sub>.value

...



# analyzing memory access patterns



?



miss



hit

# analyzing memory access patterns

```
def update(v):
    for e in v.outedges:
        e.value = v.value
        v.value = sum(
            e.value for e in v.inedges
        )
```

Diagram illustrating memory access patterns for the `update` function:

- Reads (blue arrows):
  - From memory to `v.value` (top), `e.value` (middle), and `v.value` (bottom).
- Hits (orange arrows):
  - From memory to `e.value` (middle) in the loop body.
- Writes (orange arrow):
  - From memory to `v.value` (middle) in the loop body, labeled "write (looped)".
- Reads (blue arrow):
  - From `v.value` (bottom) back to memory.

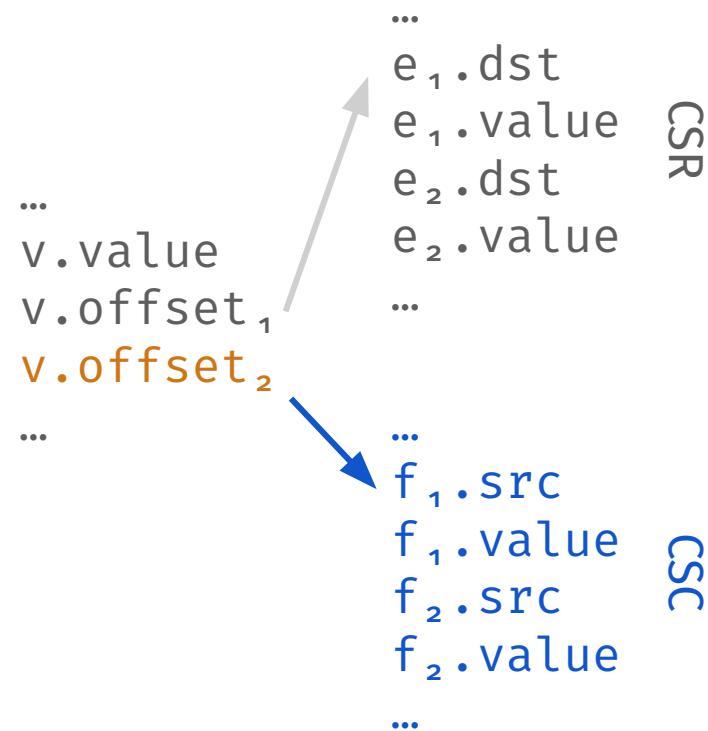


miss



hit

## memory layout



# analyzing memory access patterns

```
def update(v):
    for e in v.outedges:
        e.value = v.value
        v.value = sum(
            e.value for e in v.inedges
        )
```

The diagram illustrates the memory access patterns for the `update` function. It shows the flow of data from memory into the function and back out. Blue arrows labeled "read" point from memory into the function at various stages. Orange arrows labeled "hit" point from memory into the function, indicating a write operation that has been read back. Green arrows labeled "read (looped)" point from the function back out to memory, indicating a read miss. The code itself is as follows:

```
def update(v):
    for e in v.outedges:
        e.value = v.value
        v.value = sum(
            e.value for e in v.inedges
        )
```

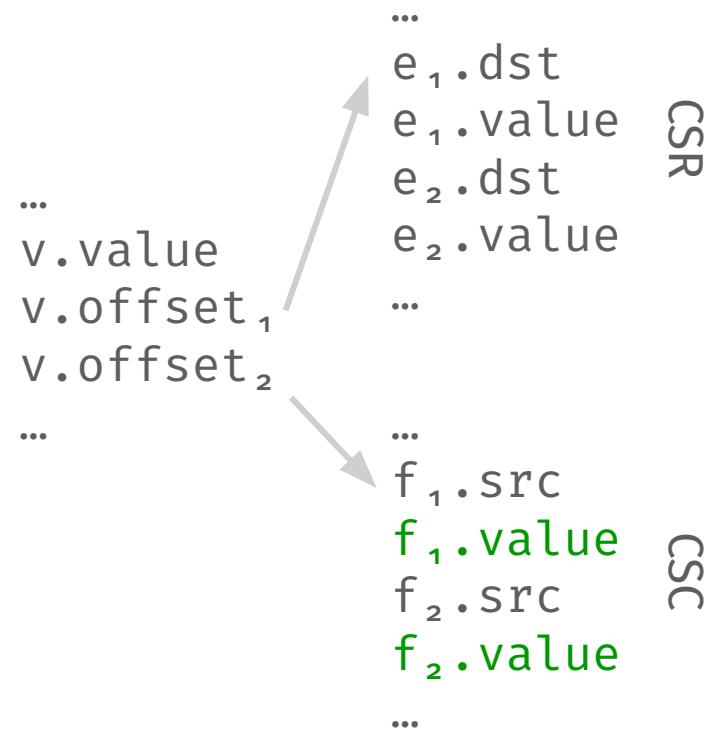


miss



hit

## memory layout



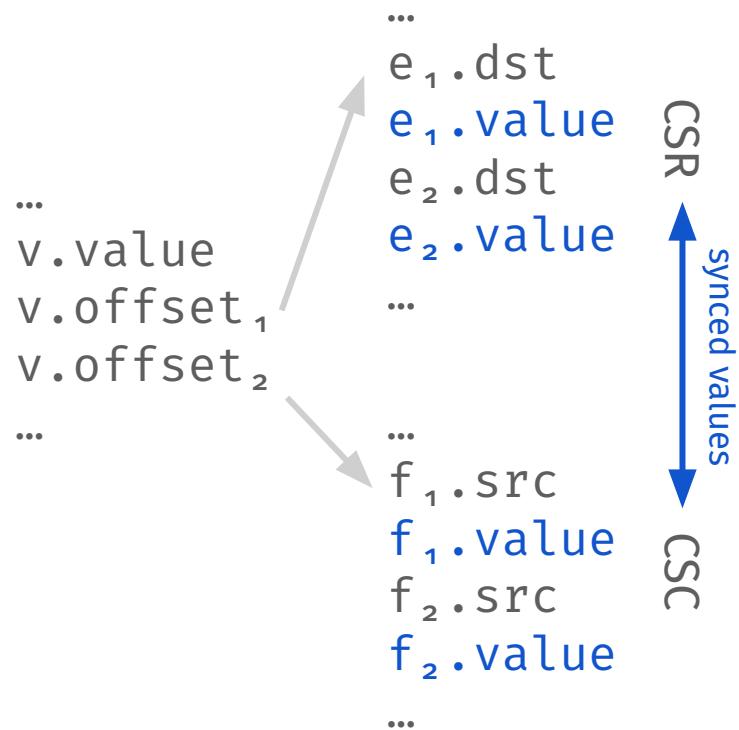
# analyzing memory access patterns

```
def update(v):
    for e in v.outedges:
        e.value = v.value
        v.value = sum(
            e.value for e in v.inedges
        )
    read (looped)
    miss due to CSR/CSC sync
```

Diagram illustrating memory access patterns for the `update` function:

- Reads (blue arrows):**
  - Initial `v.value` read.
  - Looped reads for `e.value` from `v.outedges`.
  - Final `v.value` read after the loop.
- Writes (orange arrows):**
  - Looped writes to `e.value` for each edge `e` in `v.outedges`.
  - Final write to `v.value` as the sum of edge values.
- Miss due to CSR/CSC sync:** A blue arrow pointing to the final `v.value` read, indicating a miss due to CSR/CSC synchronization.

## memory layout



# analyzing memory access patterns

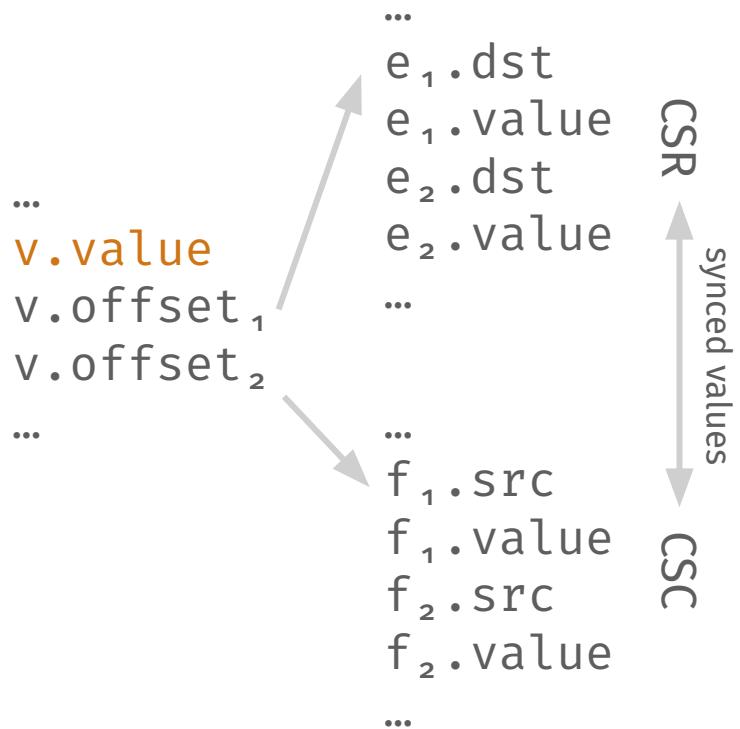
```
def update(v):
    for e in v.outedges:
        e.value = v.value
    v.value = sum(
        e.value for e in v.inedges
    )
    read (looped)
    miss due to CSR/CSC sync
```

The diagram illustrates the memory access patterns for the `update` function. It shows the flow of data from the CSR (left) to the CSC (right) through various memory locations.

- Reads:** Indicated by blue arrows pointing to memory locations. These include:
  - A read from `v.value` at the start of the loop.
  - Reads from `e.value` for all outgoing edges `e`.
  - A read from `v.value` at the end of the loop.
- Writes (hit):** Indicated by orange arrows pointing to memory locations. These include:
  - Writes to `e.value` for all outgoing edges `e`.
  - A write to `v.value` at the end of the loop.
- Write from CSR to CSC:** Indicated by a red arrow pointing from the CSR column to the CSC column.



## memory layout



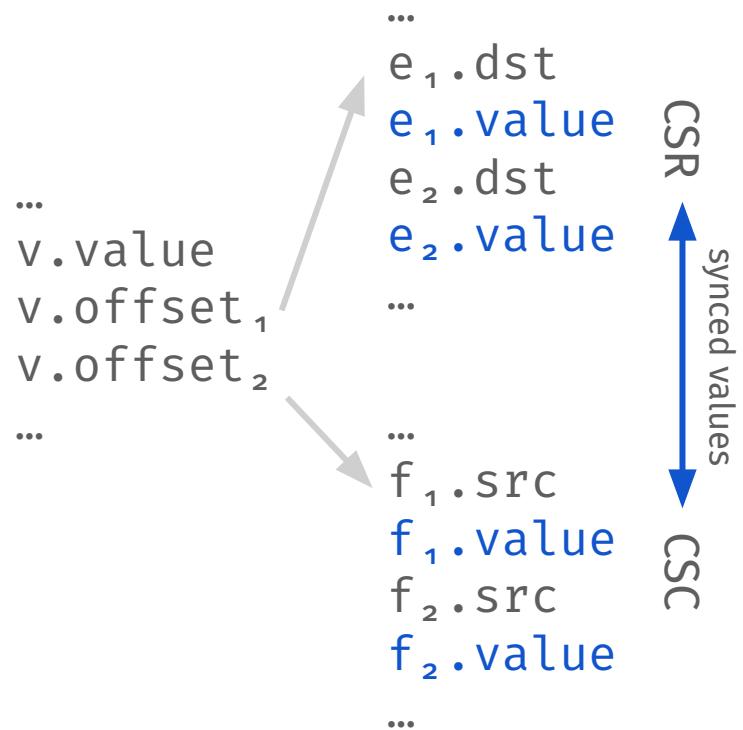
# analyzing memory access patterns

```
def update(v):
    for e in v.outedges:
        e.value = v.value
    v.value = sum(
        e.value for e in v.inedges
    )
    write (looped)
```

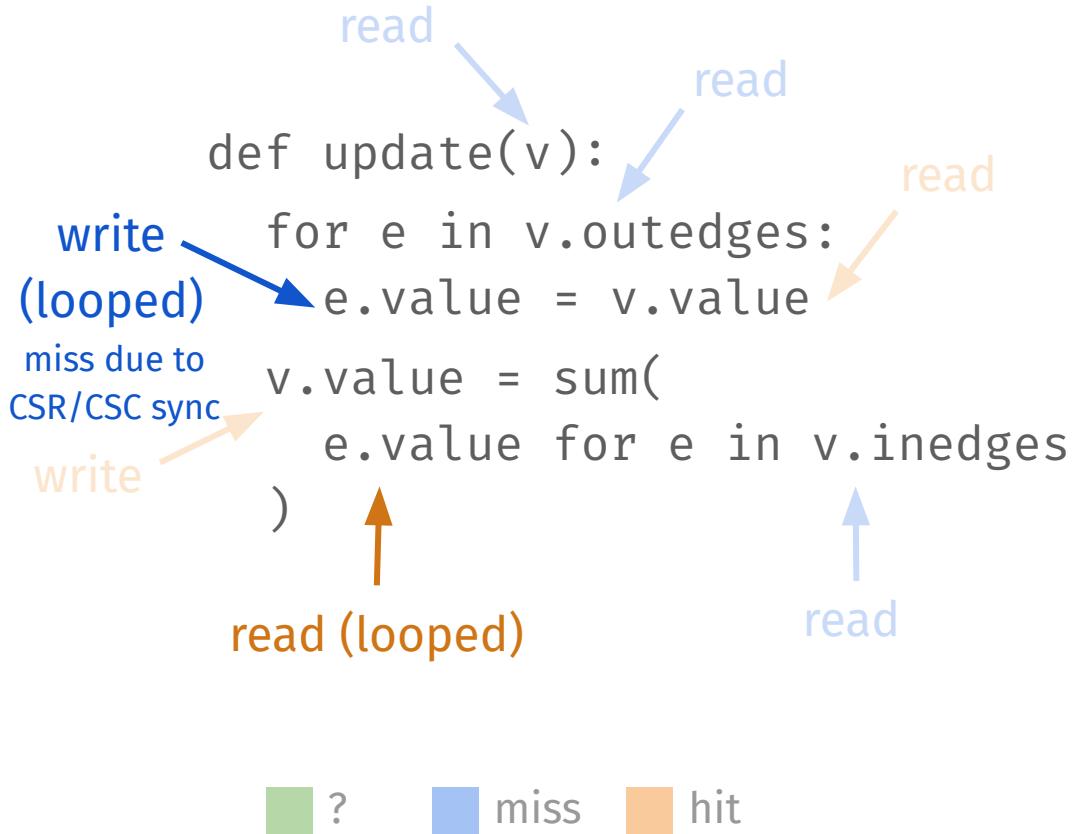
read  
read  
read  
write (looped)  
read (looped)  
miss due to CSR/CSC sync



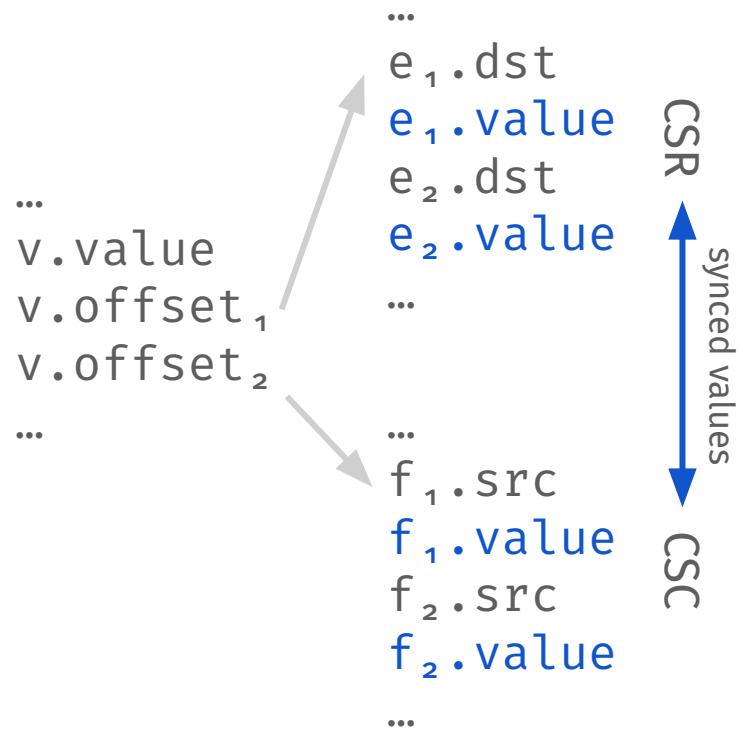
## memory layout



# analyzing memory access patterns



## memory layout



# random access problem

- why can't we “put it on disk and call it a day”?
- the vertex-centric model has lots of random access patterns

# random access problem

- why can't we “put it on disk and call it a day”?
- the vertex-centric model has lots of random access patterns
  - particularly, accessing out/inedges, as values need to be synced

# random access problem

- why can't we “put it on disk and call it a day”?
- the vertex-centric model has lots of random access patterns
  - particularly, accessing out/inedges, as values need to be synced
  - we can put outedges together, but no guarantee for inedges

# random access problem

- why can't we “put it on disk and call it a day”?
- the vertex-centric model has lots of random access patterns
  - particularly, accessing out/inedges, as values need to be synced
  - we can put outedges together, but no guarantee for inedges
  - (or vice versa: inedges together, no guarantee for outedges)

# random access problem

- why can't we “put it on disk and call it a day”?
- the vertex-centric model has lots of random access patterns
  - particularly, accessing out/inedges, as values need to be synced
  - we can put outedges together, but no guarantee for inedges
  - (or vice versa: inedges together, no guarantee for outedges)
- the authors call this the **random access problem**

# random access problem

- why can't we “put it on disk and call it a day”?
- the vertex-centric model has lots of random access patterns
  - particularly, accessing out/inedges, as values need to be synced
  - we can put outedges together, but no guarantee for inedges
  - (or vice versa: inedges together, no guarantee for outedges)
- the authors call this the **random access problem**
  - not a problem for e.g. GraphLab, because everything's in memory

# random access problem

- why can't we “put it on disk and call it a day”?
- the vertex-centric model has lots of random access patterns
  - particularly, accessing out/inedges, as values need to be synced
  - we can put outedges together, but no guarantee for inedges
  - (or vice versa: inedges together, no guarantee for outedges)
- the authors call this the **random access problem**
  - not a problem for e.g. GraphLab, because everything's in memory
  - (recall: SSD random access is **100× slower** than RAM random access)

## ACT II

*size limitations*

*access pattern speed*

*parallel sliding window*

## ACT III

*size limitations*

*access pattern speed*

*parallel sliding window*

motivation

# motivation

- assume we keep inedges sequentially

# motivation

- assume we keep inedges sequentially
- we have to access outedges randomly

# motivation

- assume we keep inedges sequentially
- we have to access outedges randomly
  - which is a problem if they're all on disk

# motivation

- assume we keep inedges sequentially
- we have to access outedges randomly
  - which is a problem if they're all on disk
- but random access isn't a problem if it's in memory!

# motivation

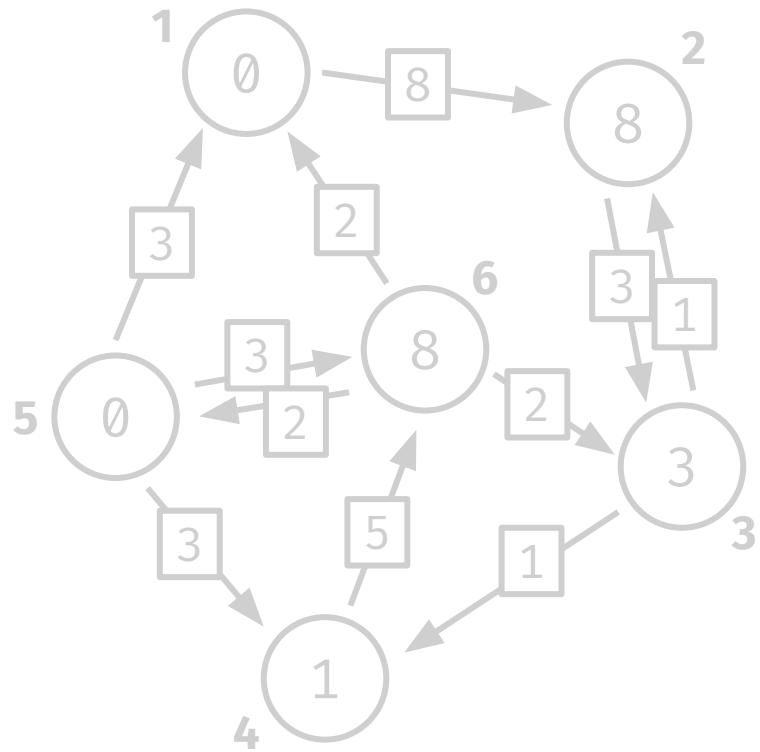
- assume we keep inedges sequentially
- we have to access outedges randomly
  - which is a problem if they're all on disk
- but random access isn't a problem if it's in memory!
- so move all the outedges we need to memory first

# motivation

- assume we keep inedges sequentially
- we have to access outedges randomly
  - which is a problem if they're all on disk
- but random access isn't a problem if it's in memory!
- so move all the outedges we need to memory first
  - it can't all fit, so do it one subgraph at a time

parallel sliding window

# parallel sliding window

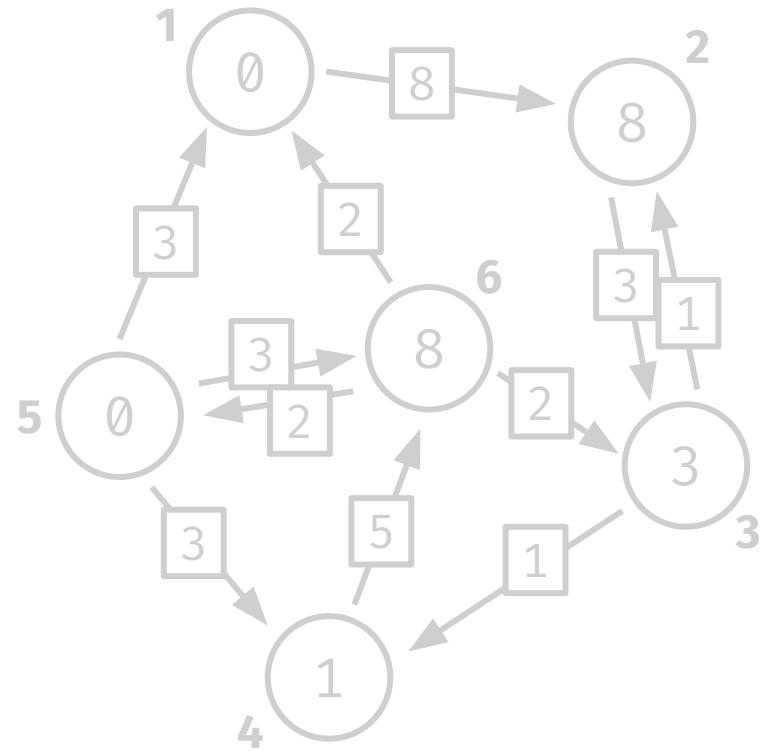


# parallel sliding window

vertices 1–2

vertices 3–4

vertices 5–6



# parallel sliding window

vertices 1-2

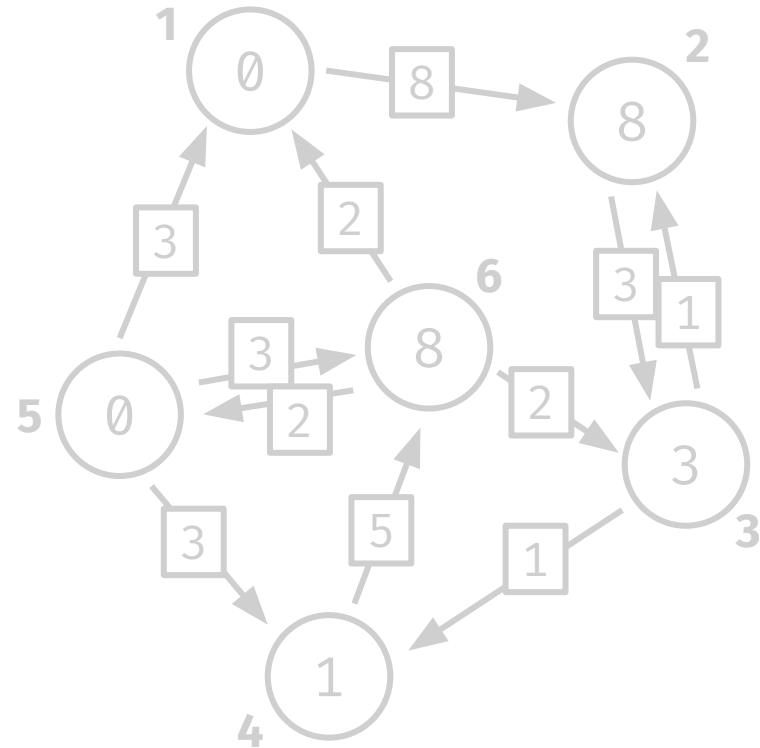
src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2



# parallel sliding window

vertices 1-2

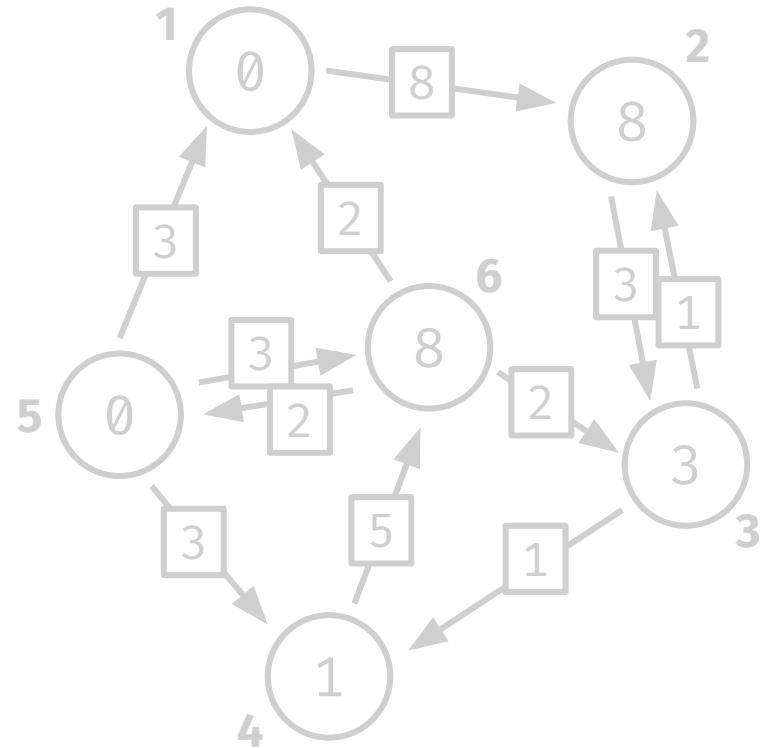
src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

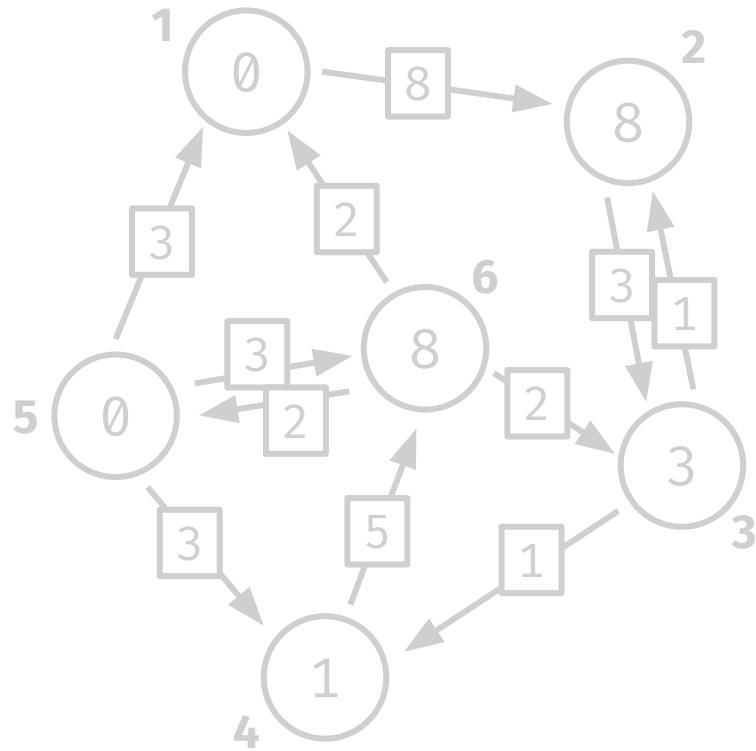
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

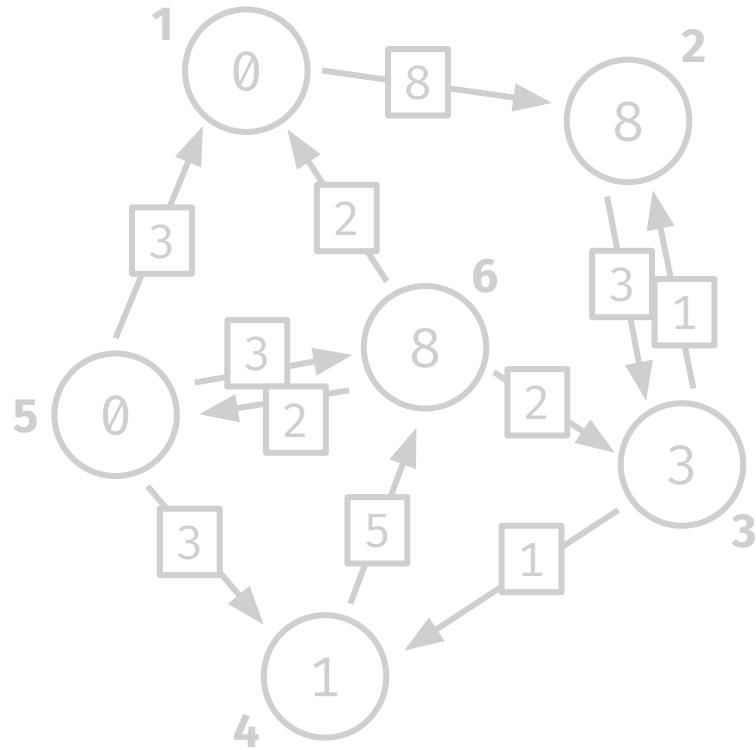
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

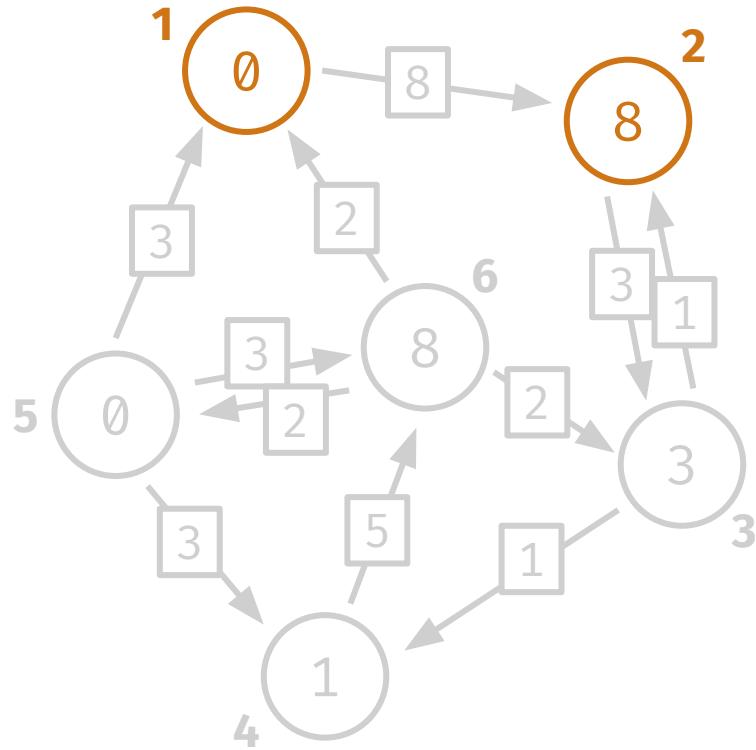
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

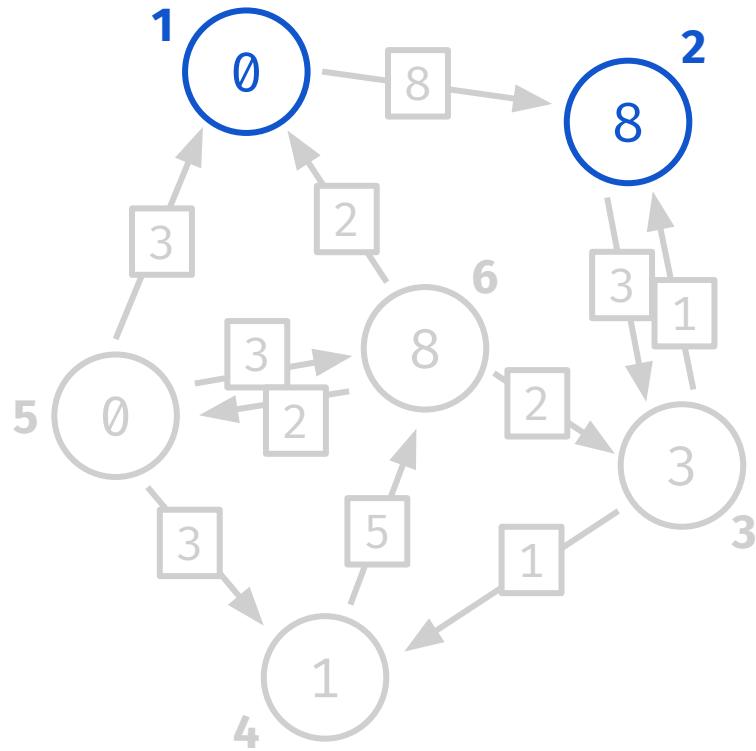
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

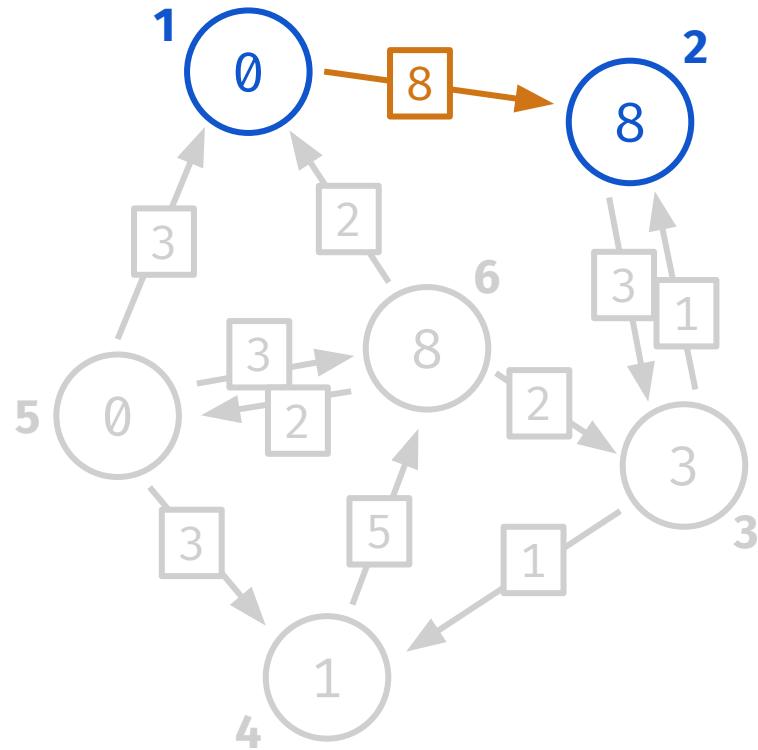
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

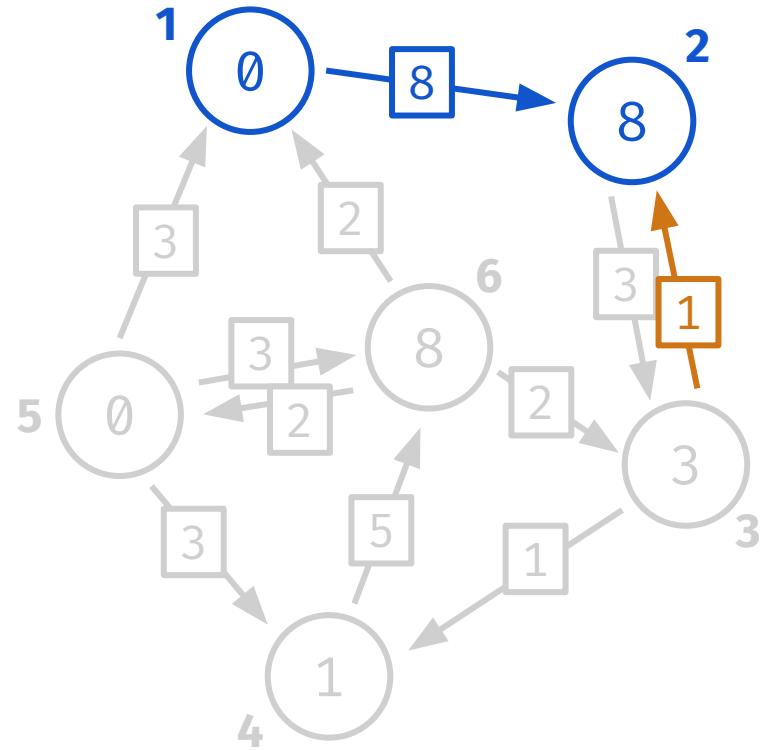
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

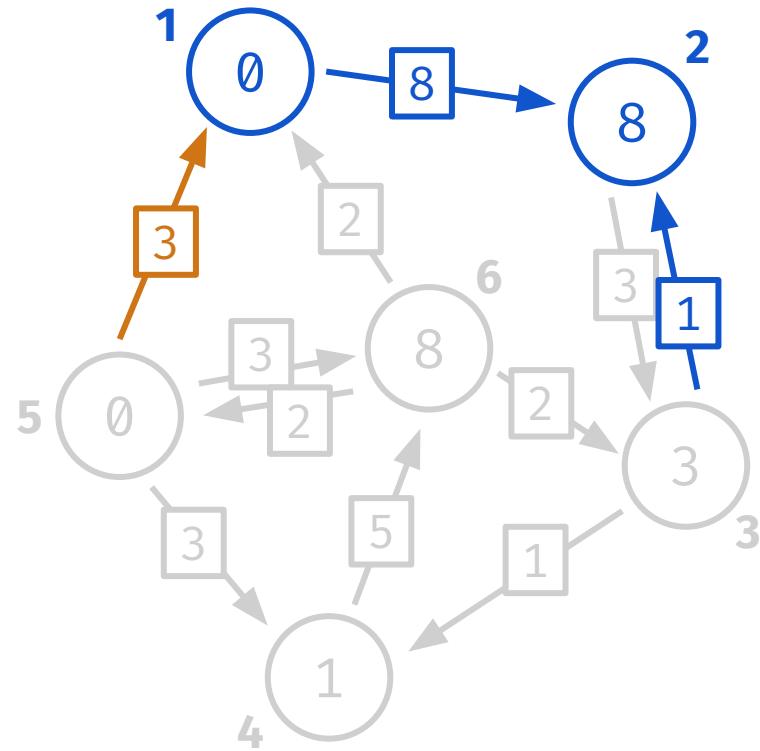
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

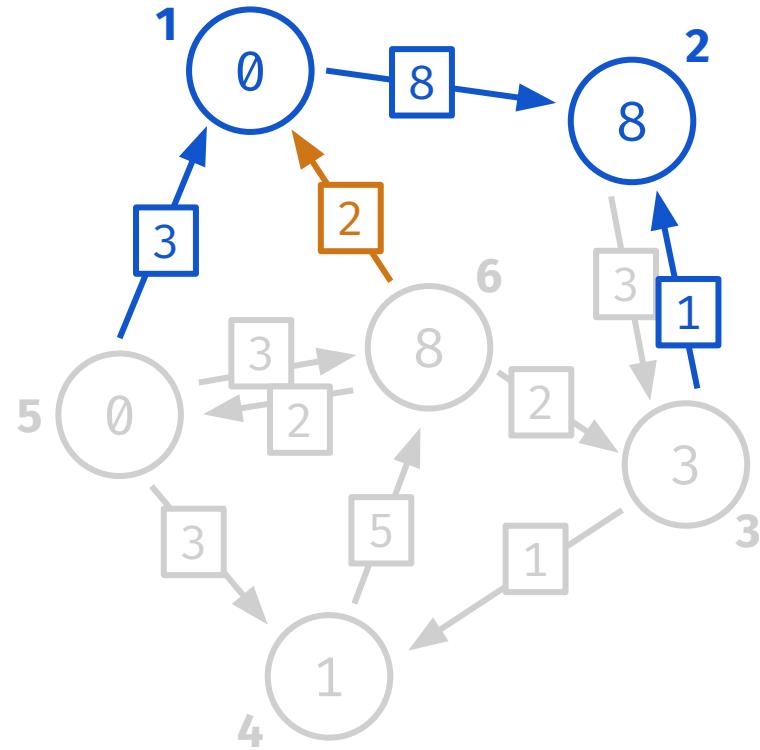
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

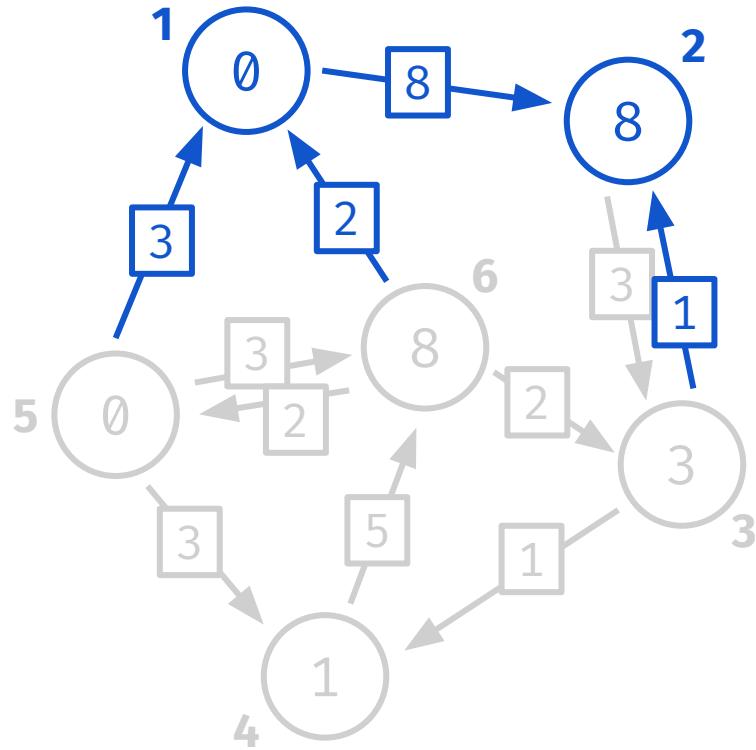
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

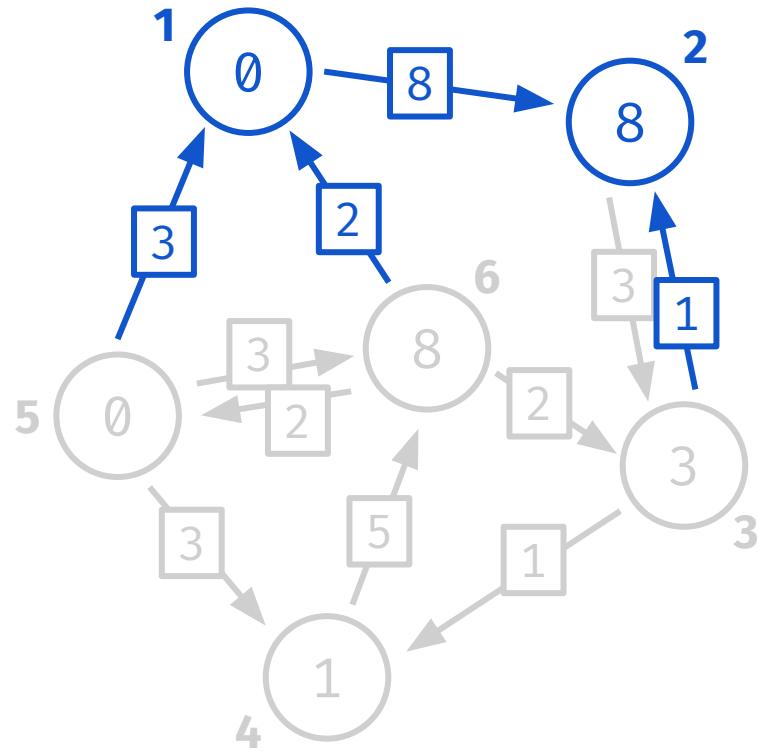
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

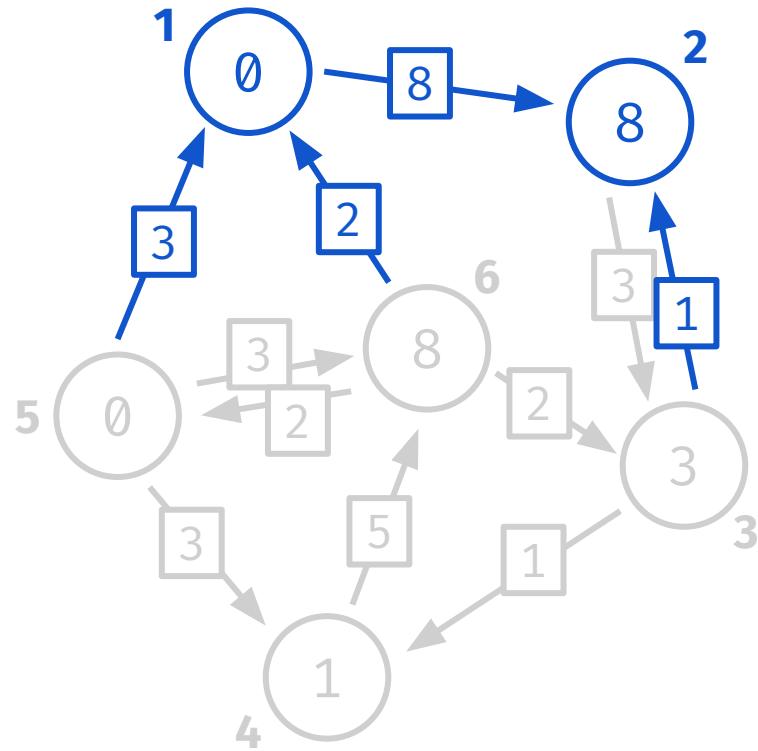
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

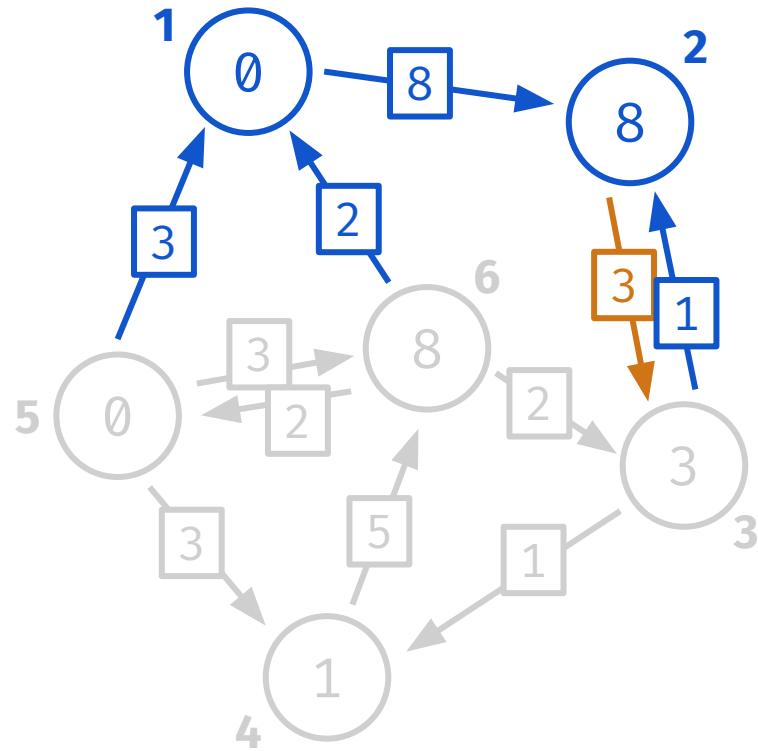
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

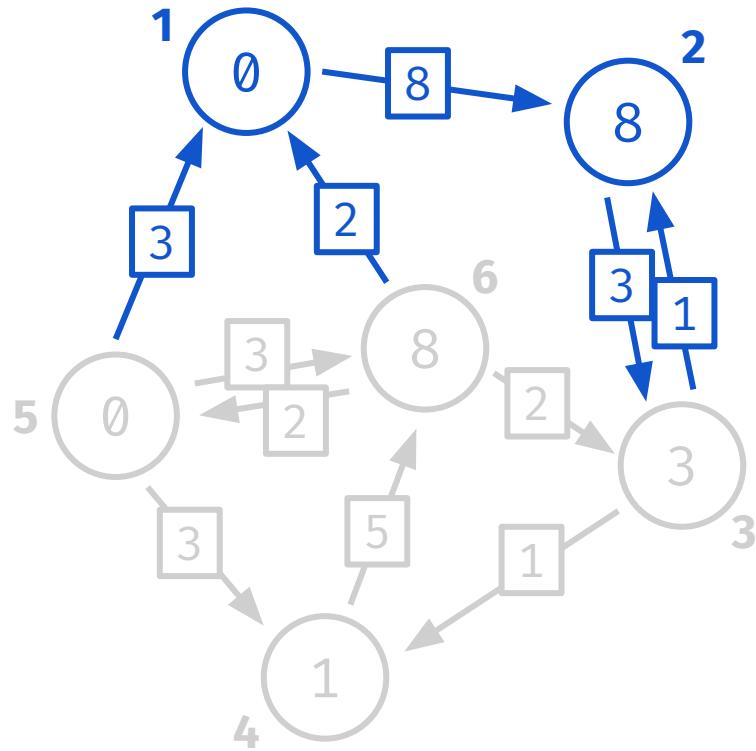
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

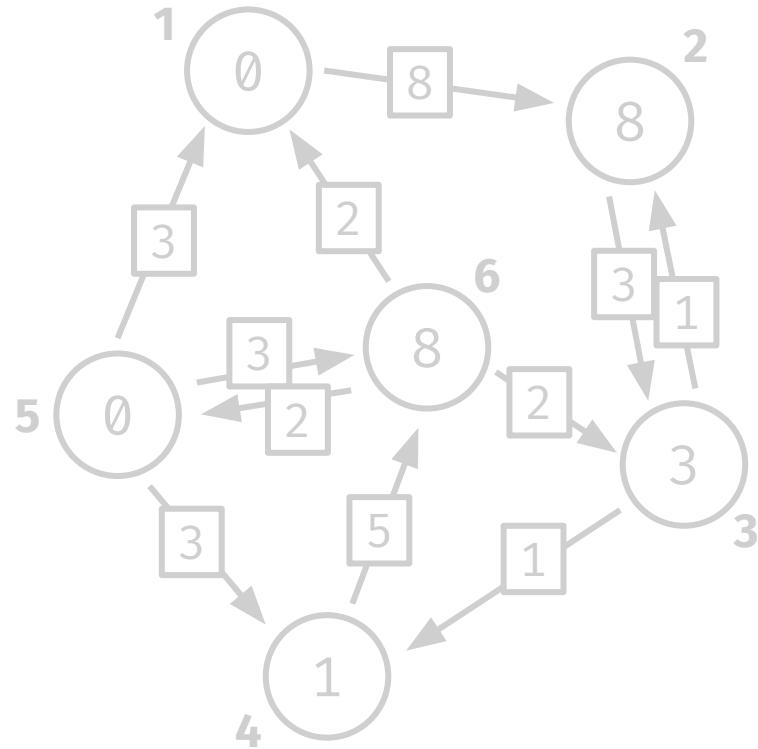
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

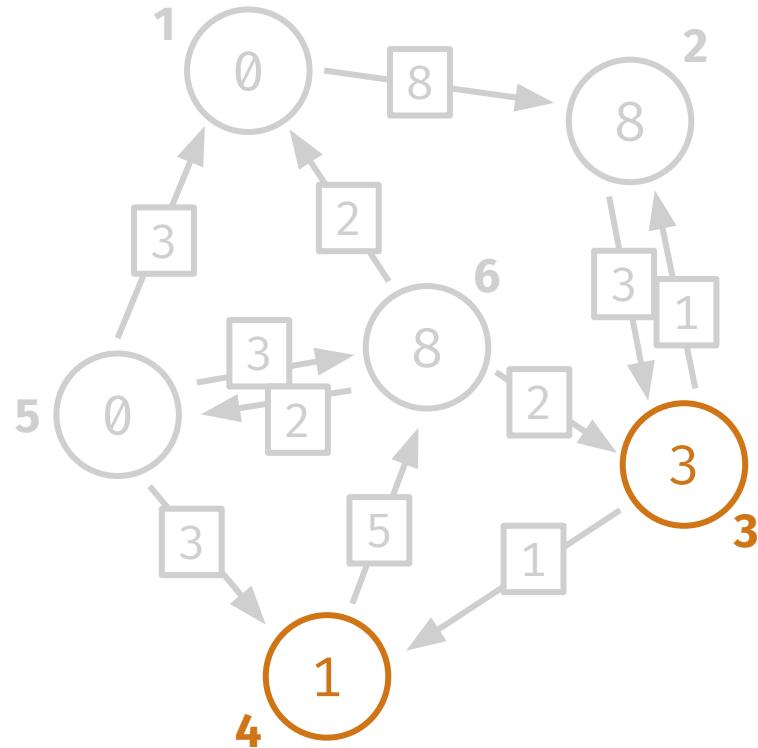
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

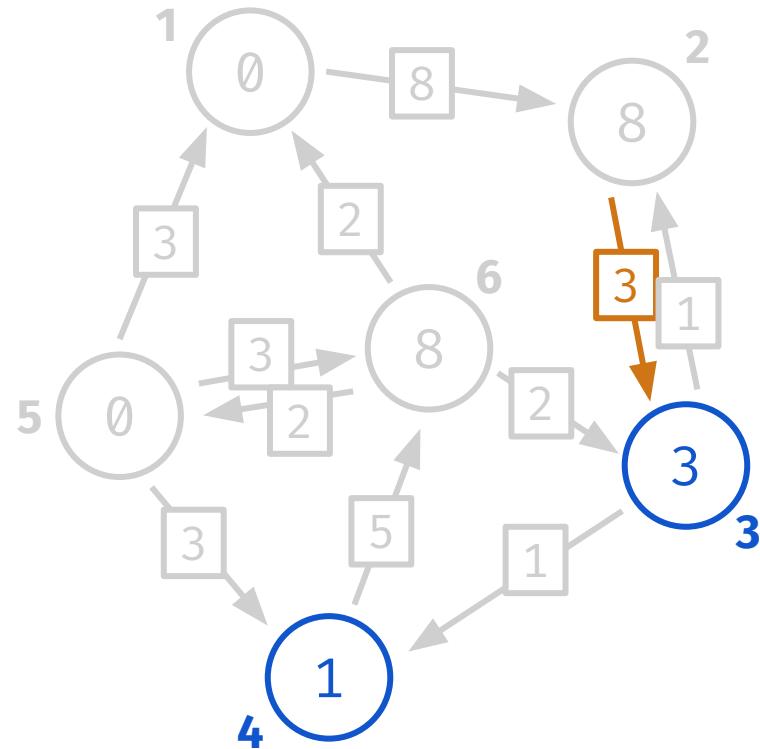
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

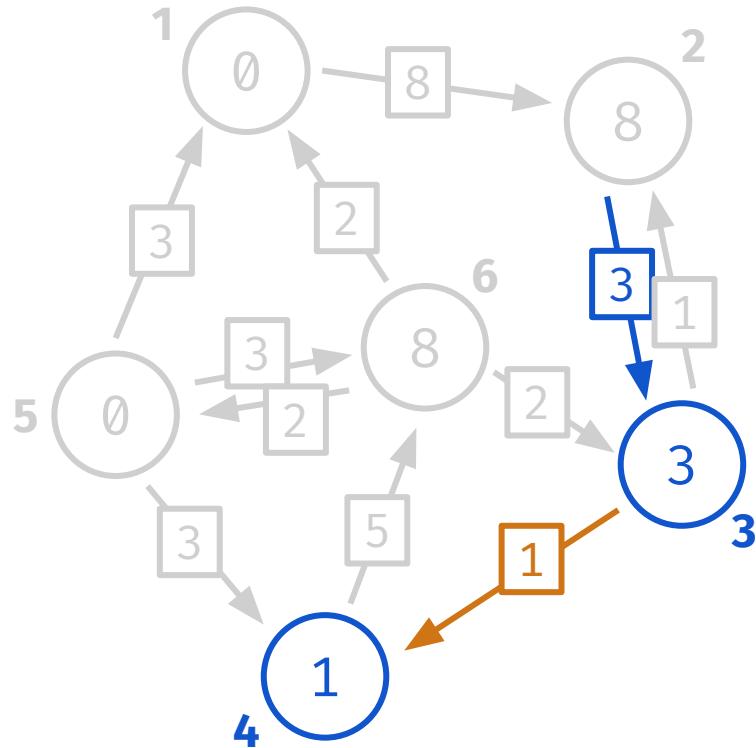
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

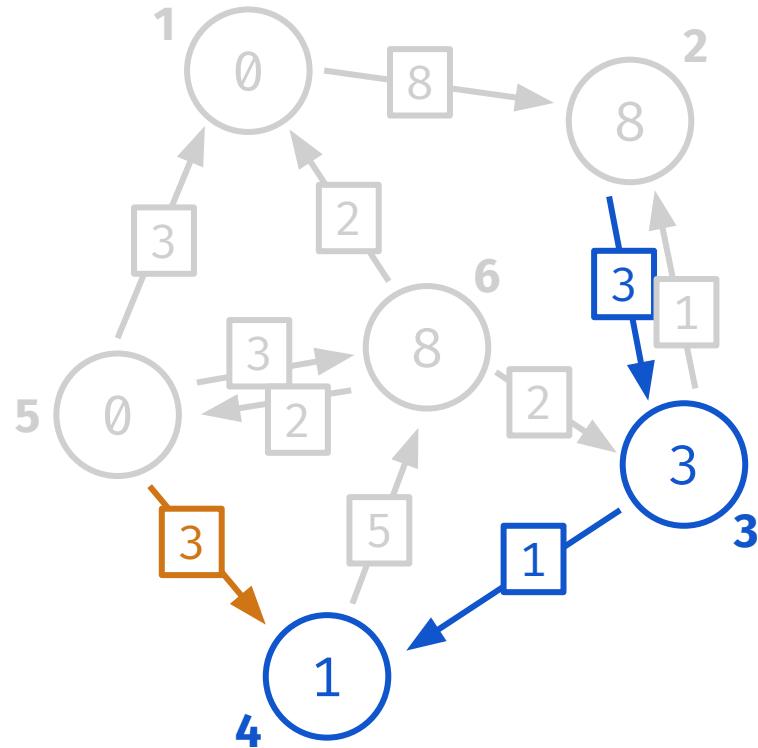
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

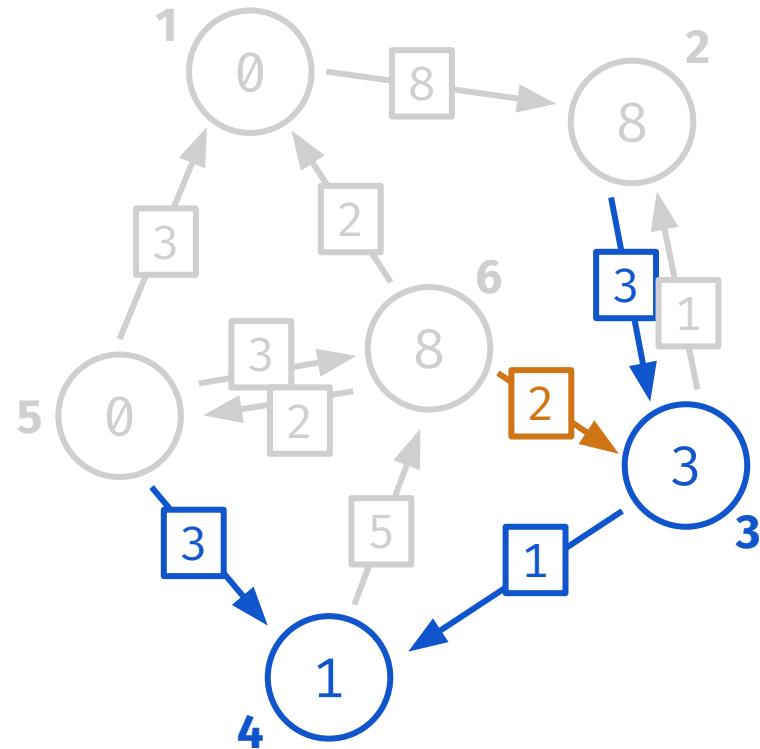
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

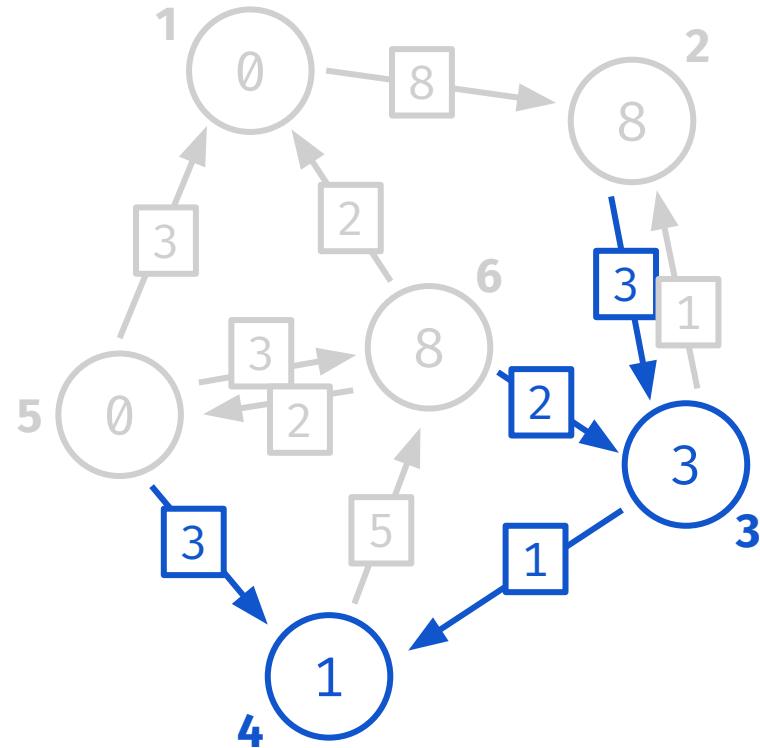
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

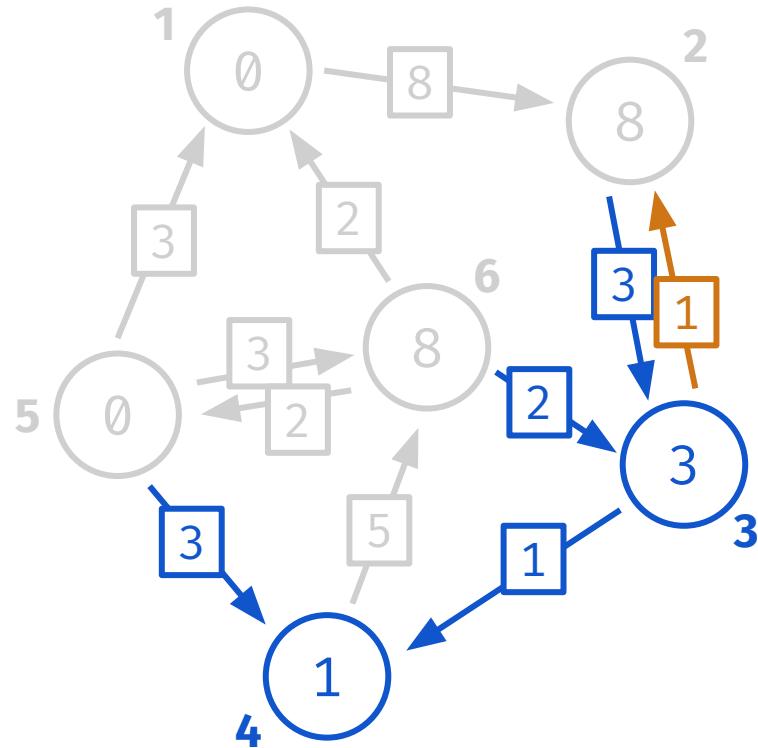
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

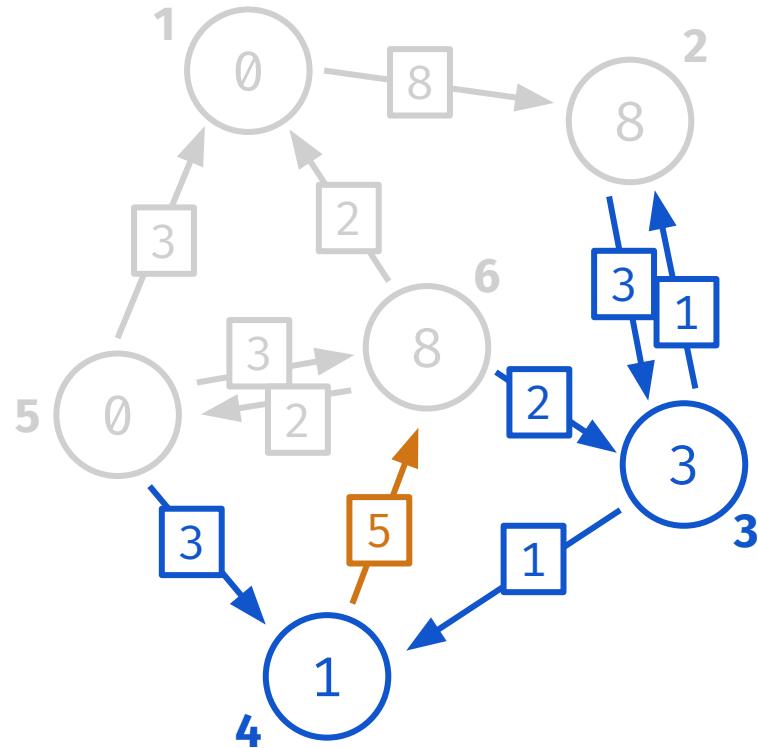
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

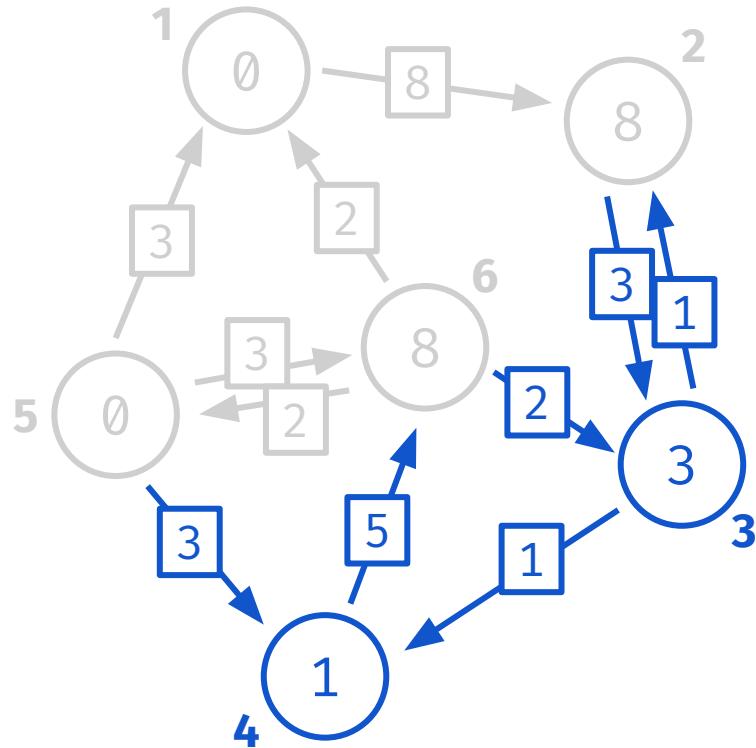
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

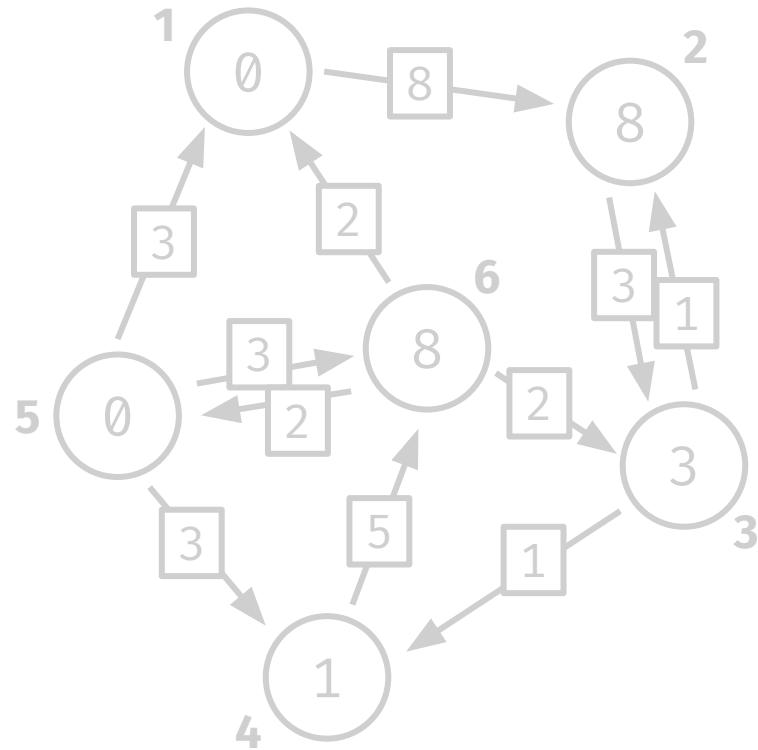
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

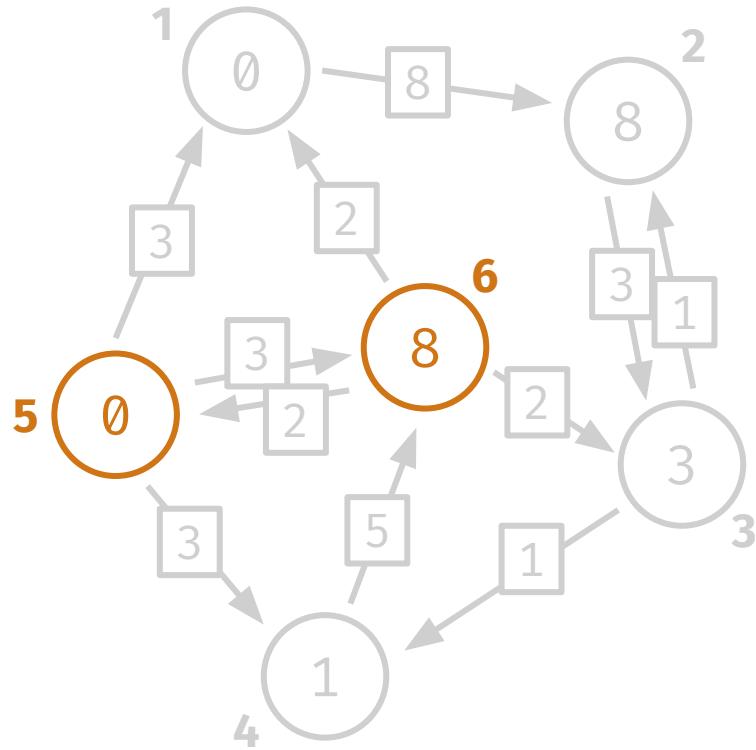
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

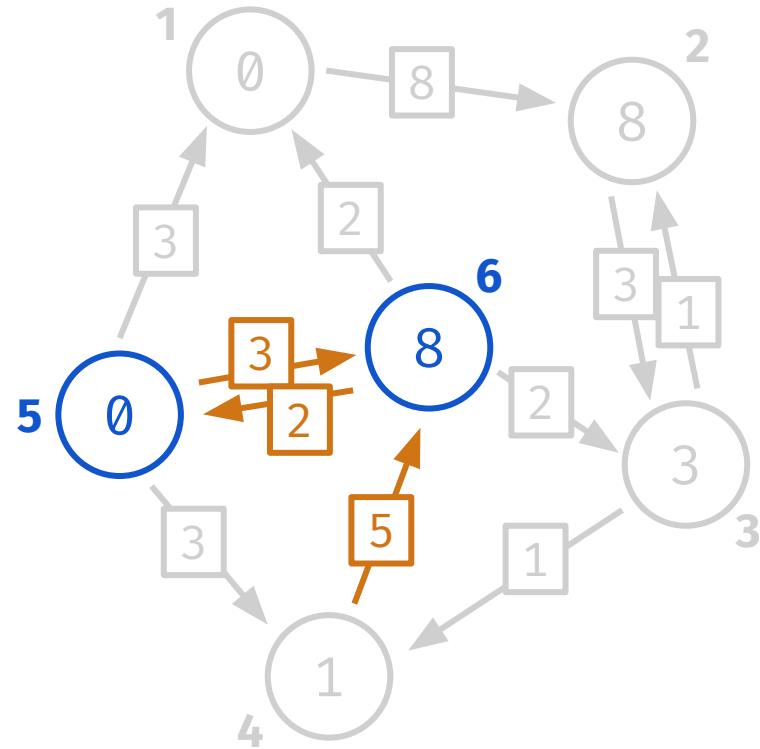
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
5	1	3
6	1	2

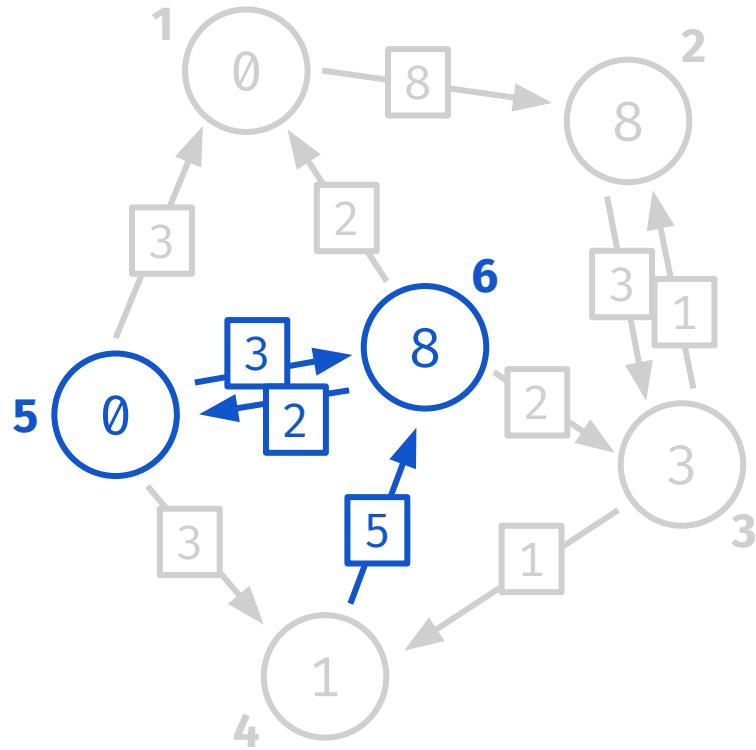
vertices 3-4

src	dst	val
2	3	3
3	4	1
5	4	3
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
<b>5</b>	<b>1</b>	<b>3</b>
6	1	2

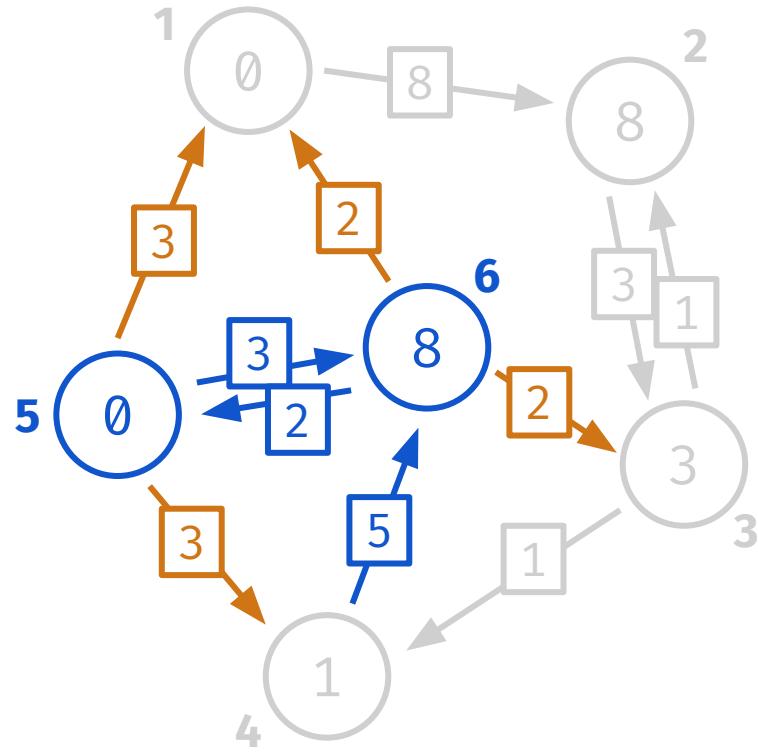
vertices 3-4

src	dst	val
2	3	3
3	4	1
<b>5</b>	<b>4</b>	<b>3</b>
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
6	5	2

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



# parallel sliding window

vertices 1-2

src	dst	val
1	2	8
3	2	1
<b>5</b>	<b>1</b>	<b>3</b>
6	1	2

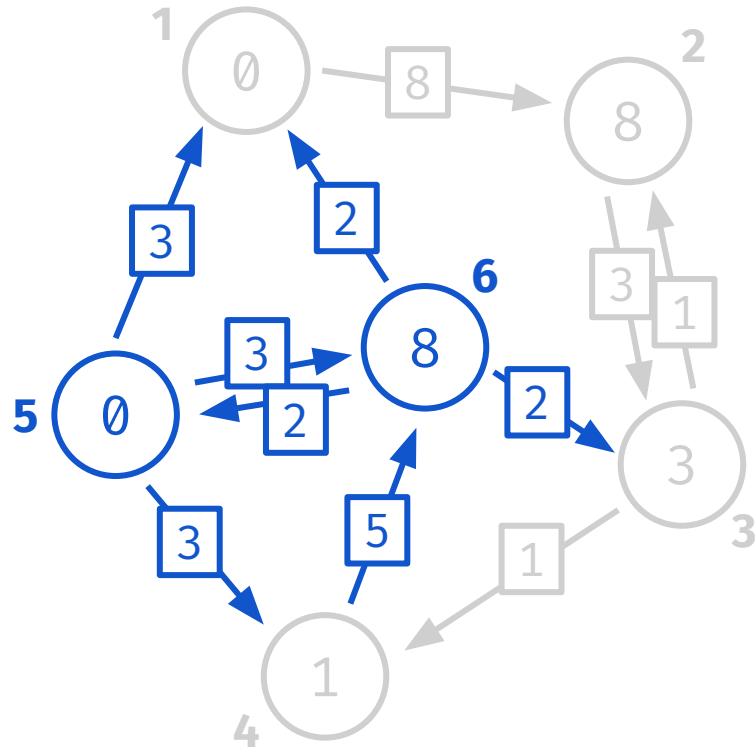
vertices 3-4

src	dst	val
2	3	3
3	4	1
<b>5</b>	<b>4</b>	<b>3</b>
6	3	2

vertices 5-6

src	dst	val
4	6	4
5	6	3
<b>6</b>	<b>5</b>	<b>2</b>

1. load vertices
2. load inedges
3. slide windows
4. load outedges
5. update values
6. write subgraph



## ACT III

*size limitations*

*access pattern speed*

*parallel sliding window*

other things i didn't have time to talk about

- the IO cost of PSW
- how PSW can handle adding and removing edges
- PSW is asynchronous and visits vertices in a specific order
  - despite these limitations, still has good applications

# GraphChi: Large-Scale Graph Computation on Just a PC

Aapo Kyrölä, Guy Blelloch, Carlos Guestin

CJ QUINES