Squeezing out

All the Value of Loaded Data:

An Out-Of-Core Graph Processing System with Reduced Disk I/O

Zhiyuan Ai[1], Mingxing Zhang[1], Yongwei Wu[1],
Xuehai Qian[2], Kang Chen[1], Weimin Zheng[1]

Tsinghua University[1], University of Southern California[2]

Graph Processing

Single machine graph processing systems (Disk I/O is bottleneck)

- 2012 — GraphChi [OSDI'12]
- 2013 — X-Stream [SOSP'13]
- 2015 — FlashGraph [FAST'15]
- 2015 — GridGraph [ATC'15]

Wrong trade-off !!

Comparison

CLIP

Trade-off

Bottleneck

Wrong

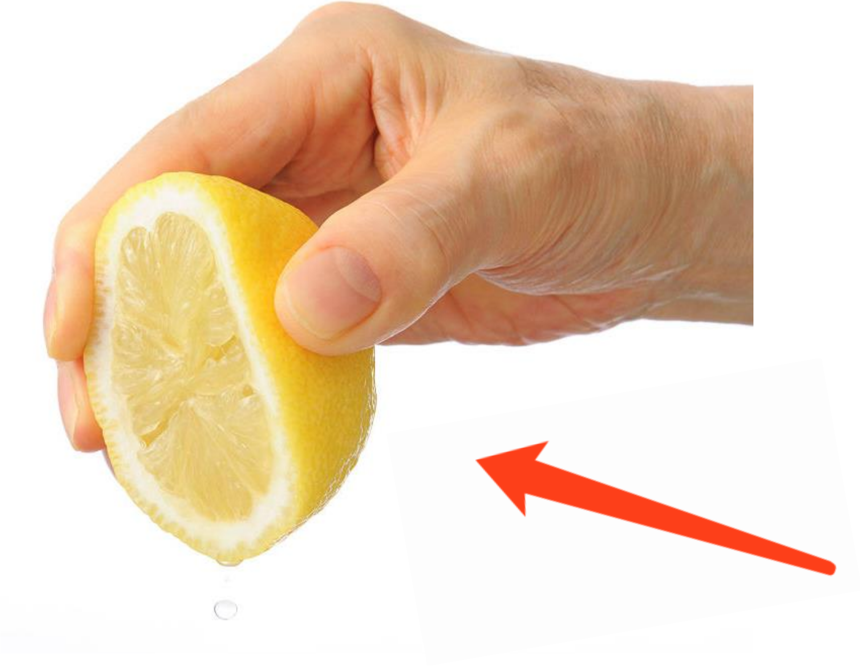
	I/O locality	The amount of Disk I/O
Prior systems	Better	Large
CLIP	Not better	**Little**

CLIP ➡ Squeezing out All the Value of Loaded Data

Up to tens or even thousands of times speedup

Comparison

Prior systems



Squeezing gently

Many iterations

VS

Loaded Data

(In-Memory)

CLIP

Squeezing **hard**

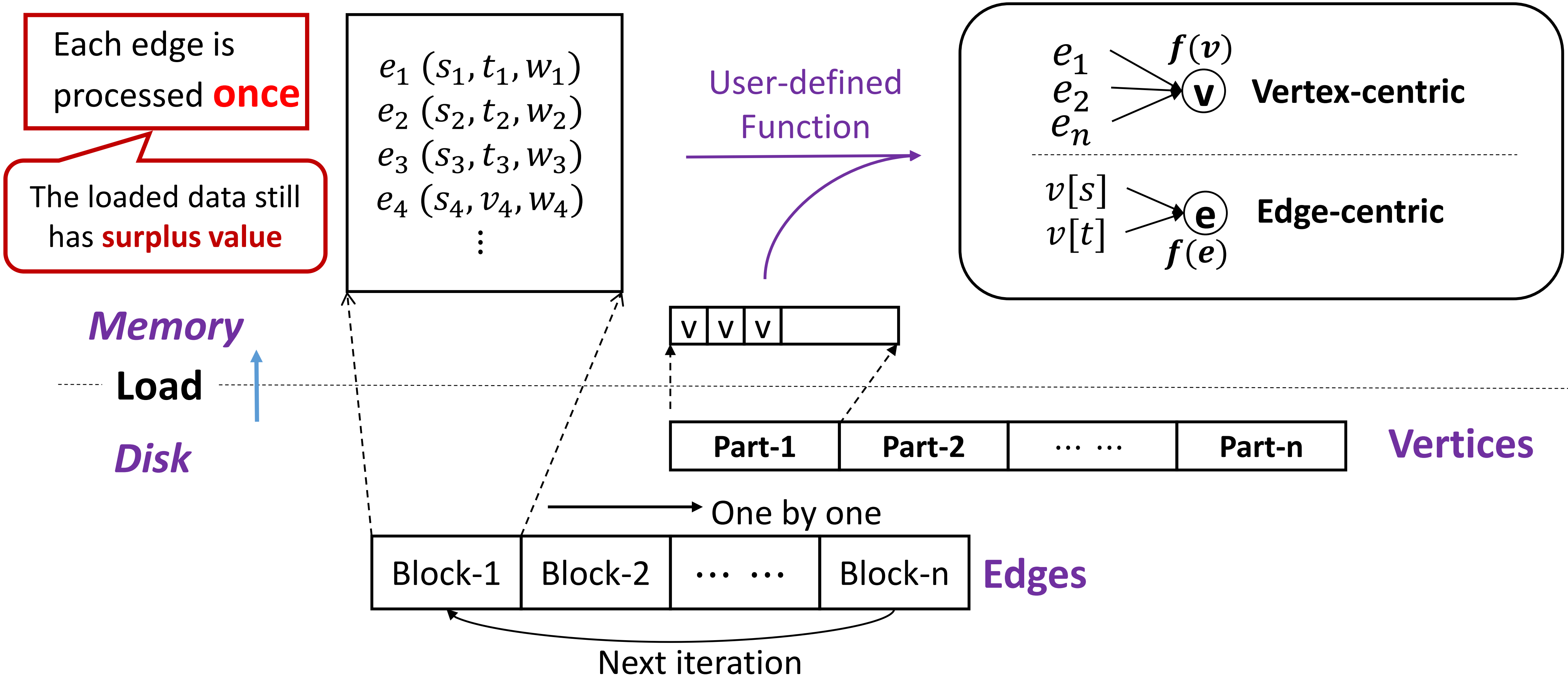
Few iterations

Outline

1. Background
2. **CLIP**
3. Evaluation
4. Conclusion

2.1 Shortcomings of prior works (1)

① Processing order is **not flexible!!**

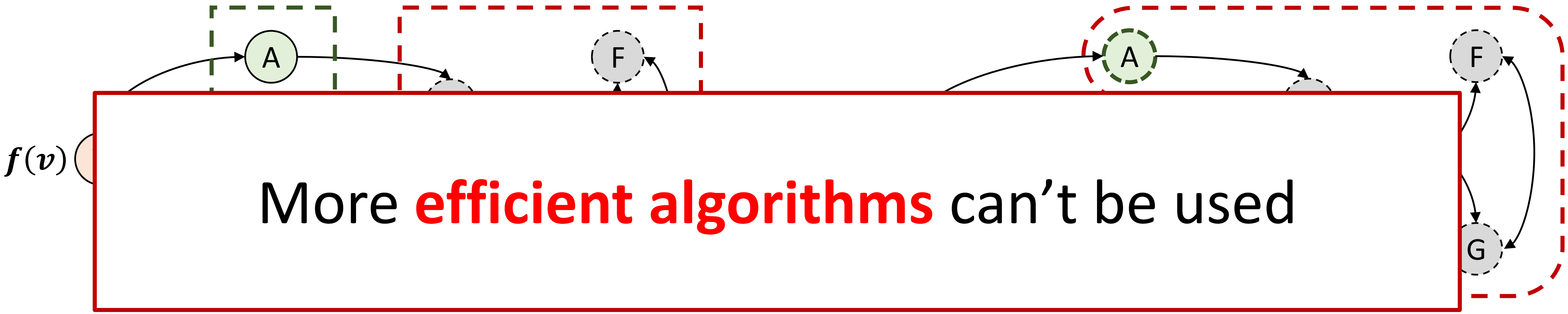


2.1 Shortcomings of prior works (2)

CLIP

② Constrained programming model

Vertex-centric | **Edge-centric**



More **efficient algorithms** can't be used

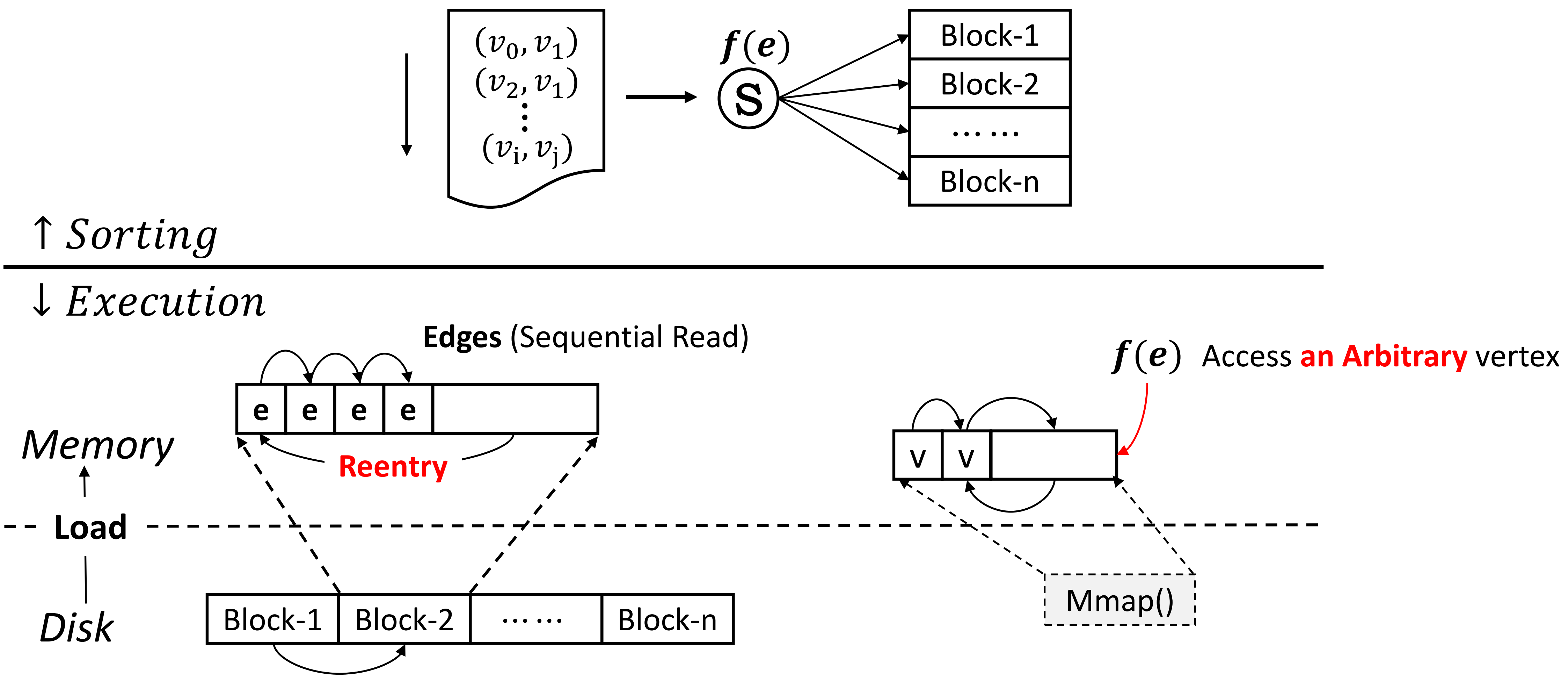
```
Update(vertex) begin
    x[] ← read values of in- and out-edges of vertex ;
    vertex.value ← f(x[]) ;
    foreach edge of vertex do
        edge.value ← g(vertex.value, edge.value);
    end
```

Graphchi [OSDI'12]

```
function STREAMEDGES(F_e, F)
    Sum = 0
    for each active block do          ▷ block with active edges
        for each edge ∈ block do
            if F(edge.source) then
                Sum += F_e(edge)
            end if
```

GridGraph [ATC'15]

2.2 Our solutions



2.2 Our solutions

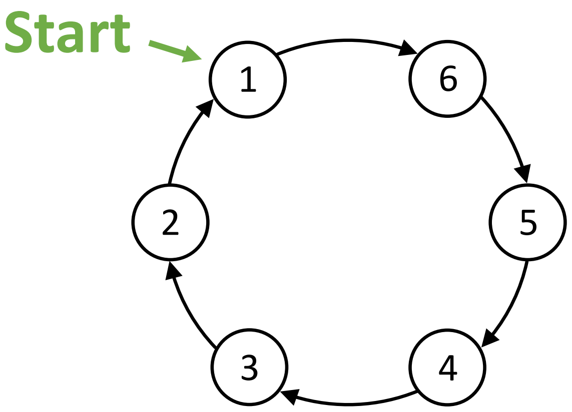
CLIP

CLIP: Squeezing out all the value of loaded data

1. **Reentry of Loaded Data**
   - **Principle**: Processing the loaded data **multiple times** rather than once.
   - **Example**: SSSP (Single Source Shortest Path).
2. **Beyond the Neighborhood**
   - **Principle**: The user-defined function is allowed to update **an arbitrary** vertex's property.
   - **Example**: WCC (Weakly Connected Component).

2.2.1 Reentry of Loaded Data

Example: Calculating single source shortest path (SSSP)



1 ⟶ **6**

Init dist						
	0	∞	∞	∞	∞	∞

	source	*target*	*weight*
e1	1	6	1
e2	2	1	1
e3	3	2	1
e4	4	3	1
e5	5	4	1
e6	6	5	1

Prior system: process once

Disk *Memory*

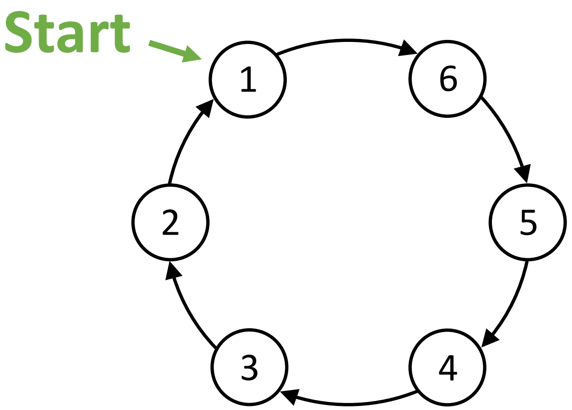
$1 \rightarrow 6$
$2 \rightarrow 1$
$3 \rightarrow 2$
$4 \rightarrow 3$
$5 \rightarrow 4$
$6 \rightarrow 5$

once

	v_1	v_2	v_3	v_4	v_5	v_6
Iteration 1	0	∞	∞	∞	2	1

2.2.1 Reentry of Loaded Data

Example: Calculating single source shortest path (SSSP)



1 ⟶ **6**

Init dist						
	0	∞	∞	∞	∞	∞

	source	*target*	*weight*
e1	1	6	1
e2	2	1	1
e3	3	2	1
e4	4	3	1
e5	5	4	1
e6	6	5	1

Prior system: process once

Disk | *Memory*

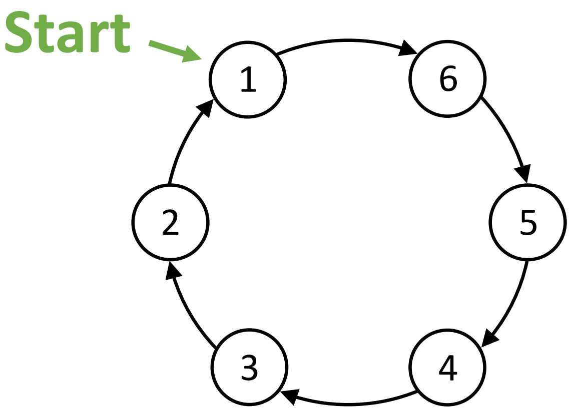
1 → 6 **2 → 1**
3 → 2 **4 → 3**
5 → 4 **6 → 5**

↓ **once**

	v_1	v_2	v_3	v_4	v_5	v_6
Iteration 1	0	∞	∞	∞	**2**	**1**
Iteration 2	0	∞	∞	**3**	**2**	**1**

2.2.1 Reentry of Loaded Data

Example: Calculating single source shortest path (SSSP)



1 ⟶ **6**

Init dist						
	0	∞	∞	∞	∞	∞

	source	*target*	*weight*
e_1	1	6	1
e_2	2	1	1
e_3	3	2	1
e_4	4	3	1
e_5	5	4	1
e_6	6	5	1

Prior system: process once

$1 \rightarrow 6$ $2 \rightarrow 1$
$3 \rightarrow 2$ $4 \rightarrow 3$
$5 \rightarrow 4$ $6 \rightarrow 5$

Disk *Memory*

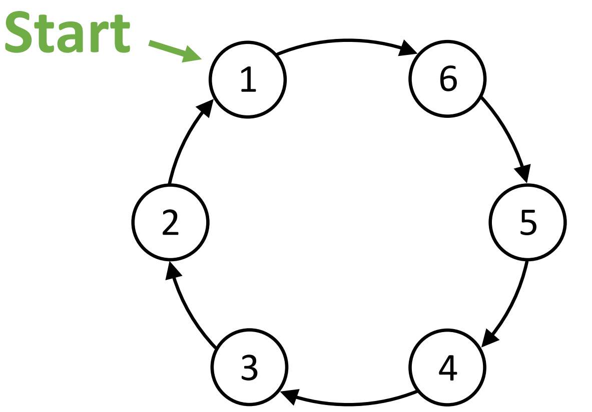
	v_1	v_2	v_3	v_4	v_5	v_6
Iteration 1	0	∞	∞	∞	2	1
Iteration 2	0	∞	∞	3	2	1
Iteration 3	0	∞	4	3	2	1
Iteration 4	0	5	4	3	2	1

$\textit{Total amount of disk I/O}: 4 * \textit{sizeof}(E)$

2.2.1 Reentry of Loaded Data—CLIP

CLIP

Example: Calculating single source shortest path (SSSP)



1 → 6

Init dist						
	0	∞	∞	∞	∞	∞

	source	*target*	*weight*
e_1	1	6	1
e_2	2	1	1
e_3	3	2	1
e_4	4	3	1
e_5	5	4	1
e_6	6	5	1

CLIP: Process multiple times

Disk *Memory*

$1 \to 6$
$2 \to 1$

$3 \to 2$
$4 \to 3$

$5 \to 4$
$6 \to 5$

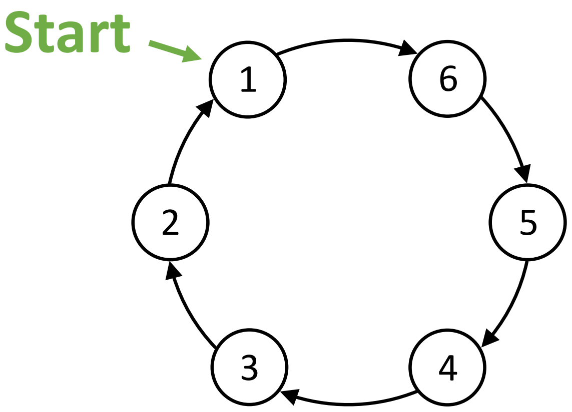
Pass two

	v_1	v_2	v_3	v_4	v_5	v_6
Iteration 1	0	∞	∞	**3**	**2**	**1**

2.2.1 Reentry of Loaded Data—CLIP

CLIP

Example: Calculating single source shortest path (SSSP)



1 ⟶ **6**

Init dist					
0	∞	∞	∞	∞	∞

	source	*target*	*weight*
e1	1	6	1
e2	2	1	1
e3	3	2	1
e4	4	3	1
e5	5	4	1
e6	6	5	1

CLIP: Process multiple times

Disk | *Memory*

$1 \to 6$
$2 \to 1$

$3 \to 2$
$4 \to 3$

$5 \to 4$
$6 \to 5$

Pass two

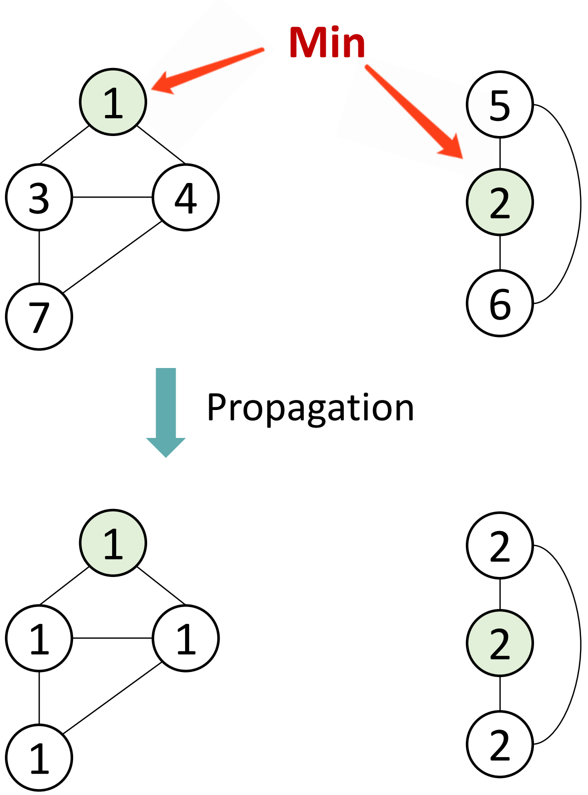
	v_1	v_2	v_3	v_4	v_5	v_6
Iteration 1	0	∞	∞	3	2	1
Iteration 2	0	5	4	3	2	1

Total amount of disk I/O: $2 * sizeof(E)$

2.2.2 Beyond the Neighborhood

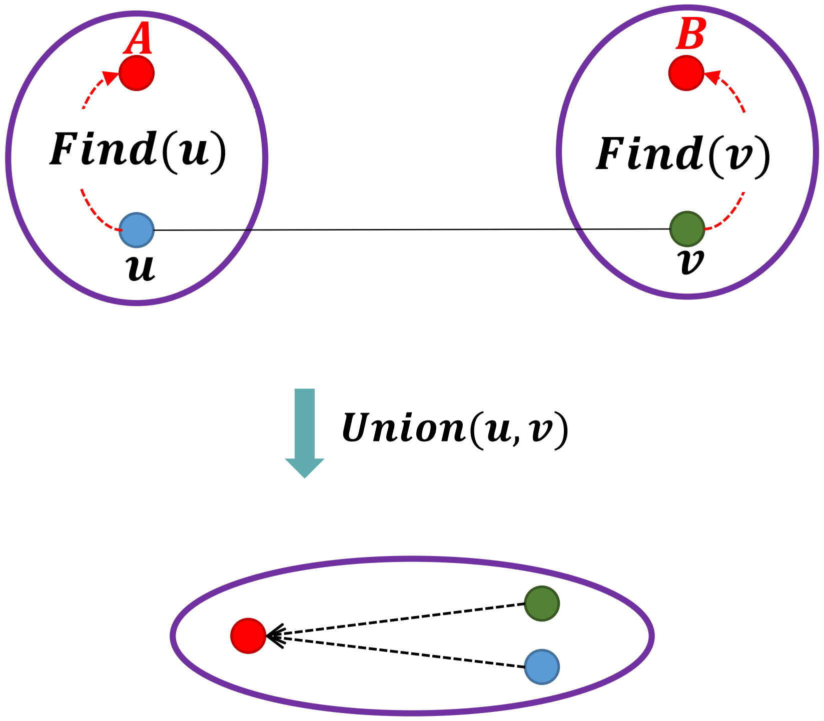
Example: Weakly Connected Component (WCC)

1. Label propagation based algorithm (Prior systems)



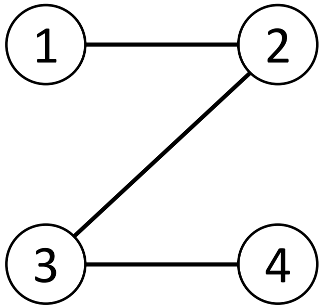
VS

2. Disjoint set algorithm (CLIP)



2.2.2 Beyond the Neighborhood

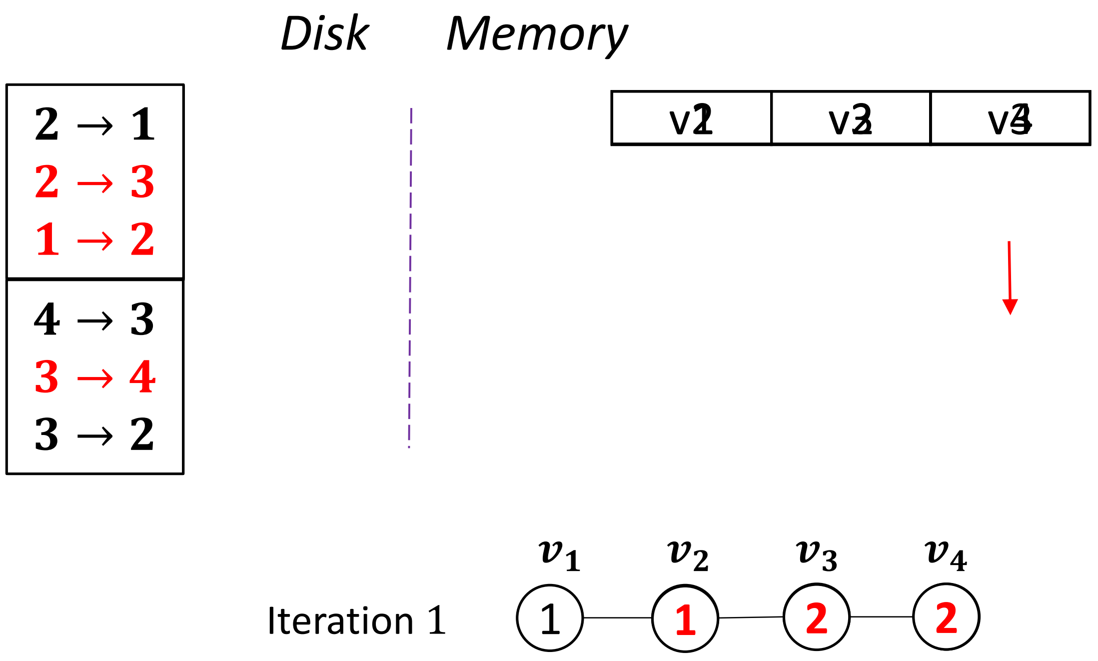
Example: Calculating weakly connected component (WCC)



Undirected graph

	source	*target*
e1	2	3
e2	2	1
e3	1	2
e4	4	3
e5	3	4
e6	3	2

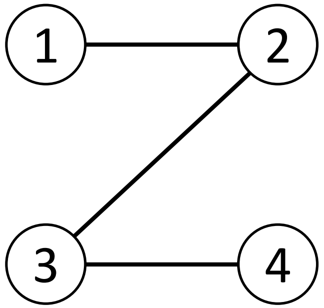
Prior system: label propagation based



2.2.2 Beyond the Neighborhood

CLIP

Example: Calculating weakly connected component (WCC)



Undirected graph

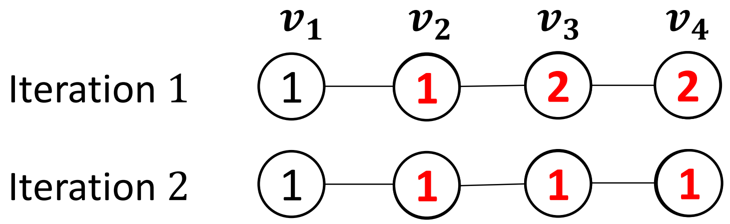
	source	*target*
e_1	2	3
e_2	2	1
e_3	1	2
e_4	4	3
e_5	3	4
e_6	3	2

Prior system: label propagation based

Disk *Memory*

$2 \rightarrow 1$
$2 \rightarrow 3$
$1 \rightarrow 2$

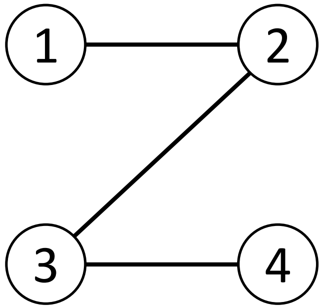
$4 \rightarrow 3$
$3 \rightarrow 4$
$3 \rightarrow 2$



$$Total\ amount\ of\ disk\ I/O: 2 * sizeof(E)$$

2.2.2 Beyond the Neighborhood—CLIP

Example: Calculating weakly connected component (WCC)



Undirected graph

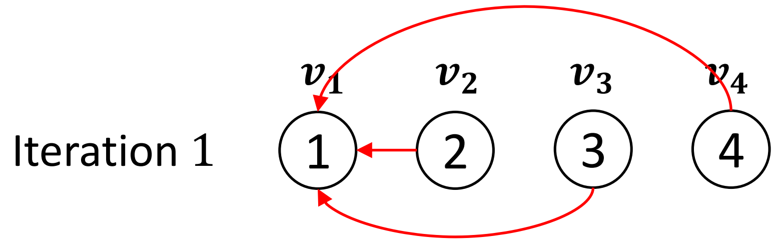
	$source$	$target$
e_1	2	3
e_2	2	1
e_3	1	2
e_4	4	3
e_5	3	4
e_6	3	2

Clip: disjoint set (Can access an arbitrary vertex)

Disk *Memory*

$2 \to 1$
$2 \to 3$
$1 \to 2$
$4 \to 3$
$3 \to 4$
$3 \to 2$

v1	v2	v3	v4



$Total\ amount\ of\ disk\ I/O: 1 * sizeof(E)$

2.2.3 APIs

CLIP

Programming model of CLIP

Sort($\mathcal{F}_s$)	—	$\mathcal{F}_s$:= **double** function(*Edge &e*)
Exec($\mathcal{F}_e$)	—	$\mathcal{F}_e$:= **void** function(*Vertexes &v_list, Edge &e*)
VMap($\mathcal{F}_v$)	—	$\mathcal{F}_v$:= **void** function(*Vertexes &v_list, VertexID &vid*)

All vertices

Algorithm 2 SSSP Algorithm in CLIP.

Functions:
$\mathcal{F}_v(v\_list, vid)$:— {
  if $vid == start$ **do**
    $v\_list[vid].dist \leftarrow 0$; $v\_list.setActive(vid, true)$;
  else $v\_list[vid].dist \leftarrow INF$; $v\_list.setActive(vid, false)$; }
$\mathcal{F}_e(v\_list, e)$:— {
  if $v\_list[e.dst].dist > v\_list[e.src].dist + e.weight$ **do**
    $v\_list[e.dst].dist \leftarrow v\_list[e.src].dist + e.weight$;
    $v\_list.setActive(e.dst, true)$;
  else $v\_list.setActive(e.dst, false)$; }

Computation:
  VMap($\mathcal{F}_v$);
  Until convergence:
    Exec($\mathcal{F}_e$);

Simple

Algorithm 3 WCC Algorithm in CLIP.

Functions:
$\mathcal{F}_{find}(v\_list, vid)$:— {
  if $v\_list[vid].pa == vid$ **do return** vid;
  else return $v\_list[vid].pa = \mathcal{F}_{find}(v\_list, v\_list[vid].pa)$; }
$\mathcal{F}_{union}(v\_list, src, dst)$:— {
  $s \leftarrow \mathcal{F}_{find}(v\_list, src)$;
  $d \leftarrow \mathcal{F}_{find}(v\_list, dst)$;
  if $s < d$ **do** $v\_list[d].pa \leftarrow v\_list[s].pa$;
  else if $s > d$ **do** $v\_list[s].pa \leftarrow v\_list[d].pa$; }
$\mathcal{F}_e(v\_list, e)$:— { $\mathcal{F}_{union}(v\_list, e.src, e.dst)$; }
$\mathcal{F}_v(v\_list, vid)$:— {
  $v\_list[vid].pa \leftarrow vid$; $v\_list.setActive(vid, true)$; }

Computation:
  VMap($\mathcal{F}_v$);
  Exec($\mathcal{F}_e$);

2.2.4 Applications

①

- **Reentry of loaded data (Asynchronous applications)**
  - SSSP (Single Source Shortest Path)
  - BFS (Breadth-first Search)

Delta-based PageRank, Diameter Approximation, Transitive Closures, Betweenness Centrality ……

②

- **Beyond-neighborhood applications**
  - WCC (Weakly connected component)
  - MIS (Maximal Independent Set)

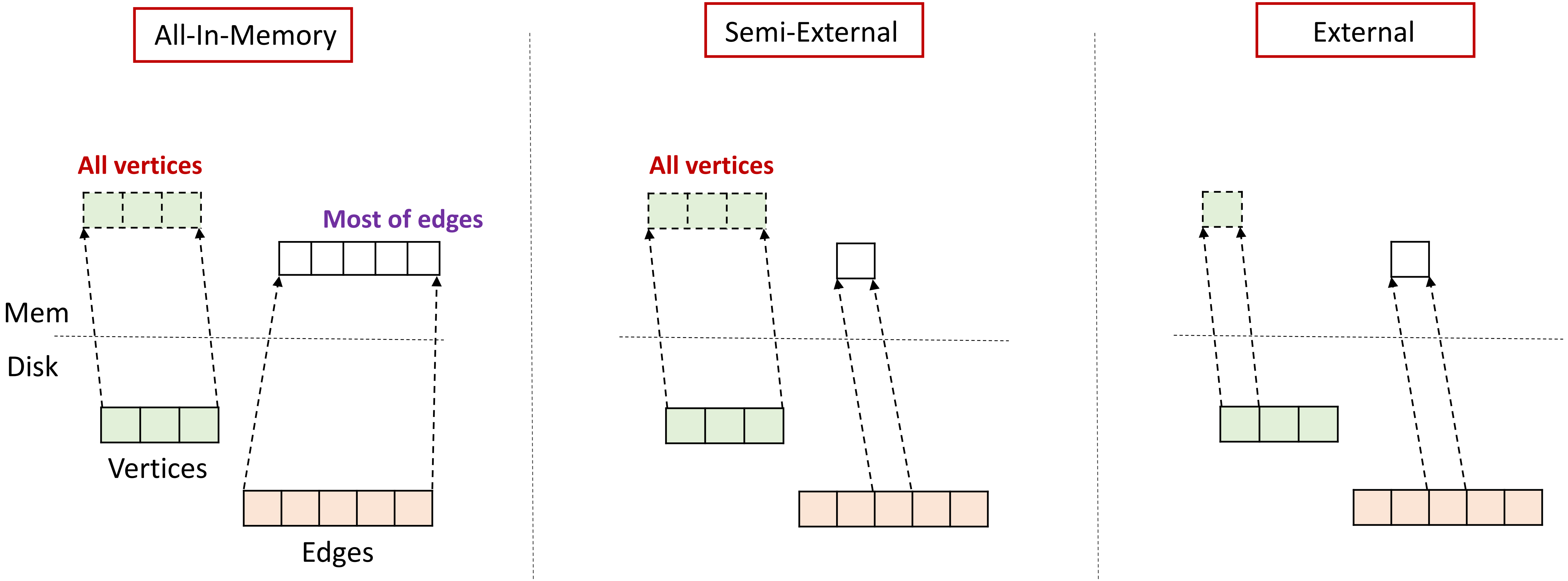
Graph Stream Algorithms, SCC, Coloring, MCST, Triangle counting ……

Outline

1. Background
2. CLIP
3. **Evaluation**
4. Conclusion

3.1 Setup

Testing three different scenarios:



(**Limit is 32GB**)

3.1 Setup

The real-world graph datasets

	Vertexes	Edges	Threshold	
			External	Semi
LiveJournal	4.85M	69.0M	16MB	256MB
Dimacs	23.9M	58.3M	64MB	256MB
Twitter	41.7M	1.47B	128MB	4GB
Friendster	65.6M	1.8B	128MB	4GB
Yahoo	1.4B	6.64B	4GB	8GB

Tips: Dimacs and Yahoo have a large diameter

Test environment

- **CPU:** Two Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz (each has 8-cores)
- **DRAM:** 32GB, L3 cache 20MB
- **Disk:** Standard 1TB SSD drive. The average throughput is about 450MB/s for sequential read.

3.2 Evaluation—Loaded data reentry

CLIP

Execution time (in seconds, external / semi-external / all-in-memory)

		LiveJournal	Dimacs	Friendster	Twitter	Yahoo
SSSP	X-Stream	357.9 / 118.4 / 8.45	77212/ 22647/ 853.2	6352 / 3346 / -	4065 / 2255 / -	∞ / ∞ / -
	GridGraph	66.42 / 48.1 / 6.97	14618/ 13480/ 889.9	1086 / 784.6 / 85.31	1639 / 1083 / 83.51	77298/ 17432/ -
	CLIP	30.14 / 11.23 / 5.09	3202 / 1981 / 316.1	176.2 / 55.79 / 55.85	1353 / 600.6 / 91.82	18160/ 6932 / -
BFS	X-Stream	91.50 / 22.94 / 4.06	8934 / 6538 / 114.9	2526 / 1084 / -	1421 / 627.4 / -	∞ / ∞ / -
	GridGraph	13.20 / 15.4 / 2.49	5199 / 5239 / 406.2	499.6 / 493.7 / 61.54	220.5 / 209.6 / 32.16	35572/ 7403 / -
	CLIP	10.01 / 5.46 / 2.53	1768 / 1059 / 96.12	98.87 / 38.55 / 38.72	141.2 / 110.4 / 44.7	10533/ 3297 / -

The number of iterations

		LiveJornal	Dimacs	Friendster	Twitter	Yahoo
SSSP	X-Stream	45	10790	50	39	>500
	GridGraph	37	10788	27	33	5824
	CLIP	8	934	1	19	484
BFS	X-Stream	16	6263	30	14	>720
	GridGraph	15	6262	29	13	4375
	CLIP	4	574	1	5	243

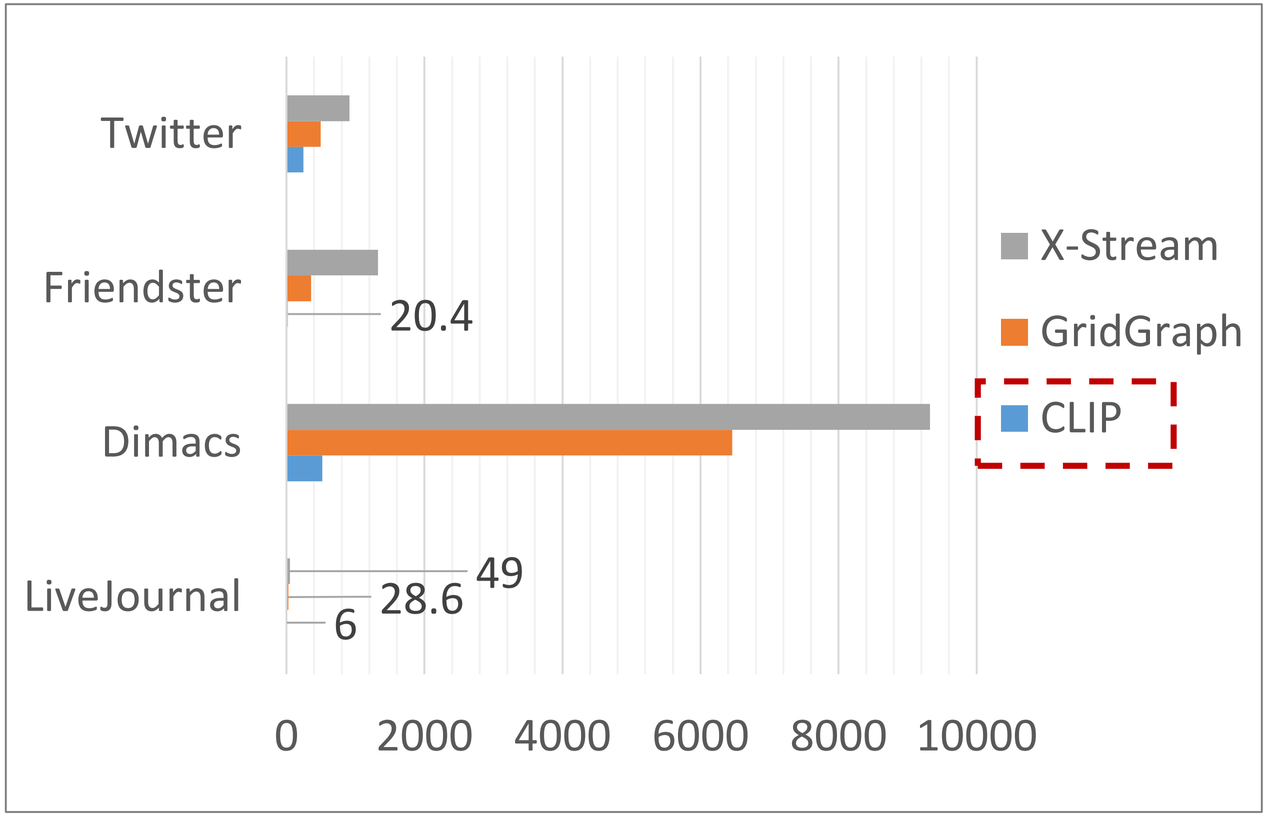
SSSP speedup: 1.8x-14.06x

BFS speedup: 1.9x-12.08x

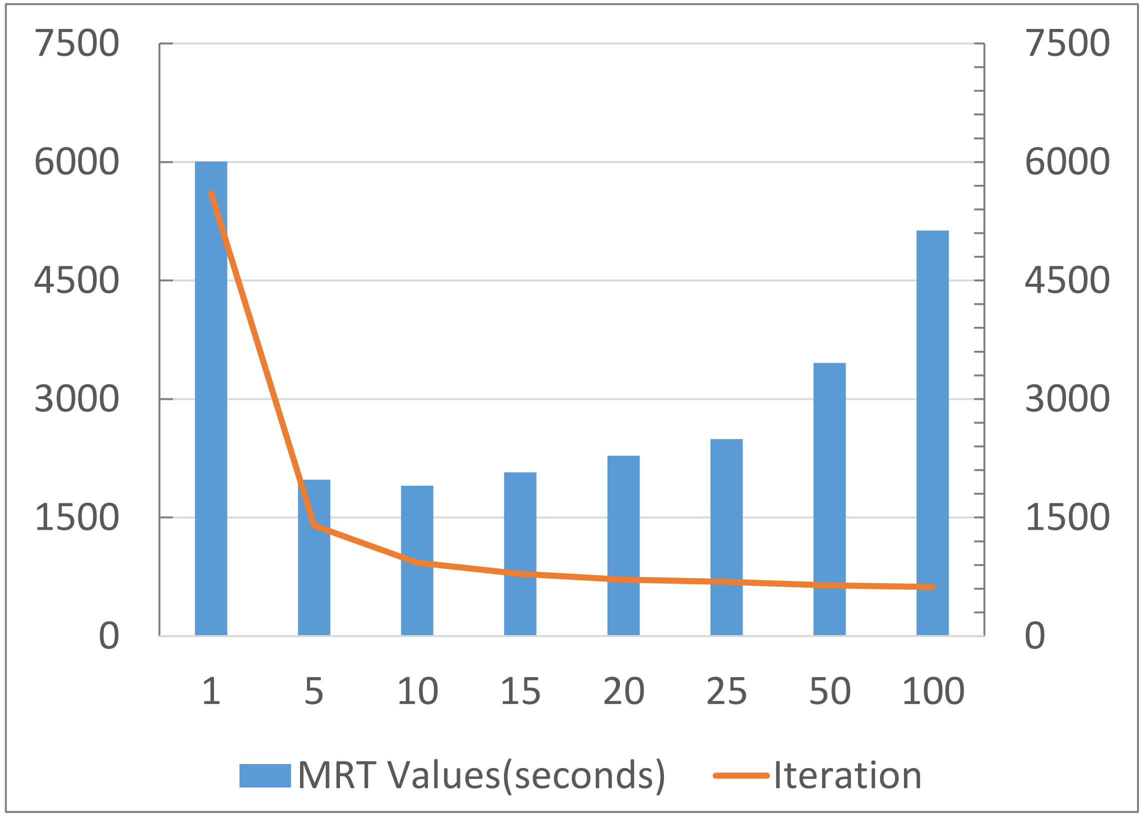
3.2 Evaluation—Loaded data reentry

CLIP

Total amount of disk I/O for SSSP (GB)



SSSP on Dimacs graph



- All the values we reported is measured at **"MRT (Maximum Reentry Times) = 5"**
- Heuristically setting MRT to **5-10** is usually enough (less than 4% difference)

3.3 Evaluation—Beyond-neighborhood

CLIP

Execution time (in seconds, external / semi-external / all-in-memory)

		LiveJournal	Dimacs	Friendster	Twitter	Yahoo
WCC	X-Stream	179.5 / 57.77 / 10.25	16633/ 6751 / 185.3	4521 / 2341 / -	1904 / 1194 / -	∞ / ∞ / -
	GridGraph	22.32 / 13.8 / 3.57	6547 / 5757 / 422.5	967.5 / 466.6 / 82.95	431.5 / 272.3 / 62.3	19445/ 2916 / -
	CLIP	3.73 / 2.40 / 2.43	2.61 / 1.35 / 1.33	186 / 65.48 / 64.56	132.7 / 49.03 / 48.85	310.6 / 220.9 / -
MIS	X-Stream	422.1 / 152.6 / 13.06	103.4 / 41.42 / 5.95	9880 / 4867 / -	5513 / 3042 / -	∞ / ∞ / -
	GridGraph	166.6 / 122.1 / 2.98	46.32 / 39.19 / 14.46	3945 / 3777 / 253.7	2510 / 2473 / 156.1	∞ / ∞ / -
	CLIP	6.7 / 2.57 / 2.58	1.6 / 1.17 / 1.21	188.8 / 62.49 / 62.18	90.44 / 49.08 / 49.13	321.5 / 220.2 / -

The number of iterations

		LiveJornal	Dimacs	Friendster	Twitter	Yahoo
WCC	X-Stream	13	6263	24	16	>360
	GridGraph	37	6261	12	10	1368
	CLIP	1	1	1	1	1
MIS	X-Stream	54	31	65	53	>300
	GridGraph	53	31	64	52	>400
	CLIP	1	1	1	1	1

CLIP

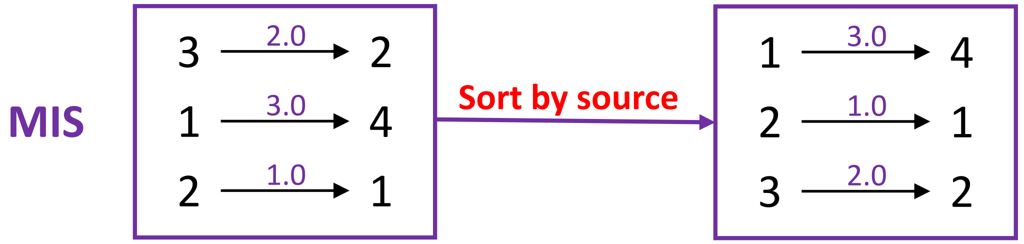
- Sequential implementation
- But only need **one** iteration

WCC speedup: 5.6x-4264x
MIS speedup: 34x-60x

3.4 Evaluation—Preprocessing

Preprocessing (Sorting) Evaluation

- Sorting is required by some algorithms and can be **amortized by reusing**



Preprocessing time (seconds)

	GridGraph	CLIP (by source ID)	CLIP (by weight)
LiveJornal	4.57	5.06	9.06
Dimacs	4.09	5.12	6.81
Friendster	185.5	145.3	228.9
Twitter	160.3	126.2	188.1
Yahoo	1616	1410	1563

Total time on Friendster (seconds)

		Proc.	Exec.	Total
MIS	X-Stream	0	4867	4867
	GridGraph	185.5	3777	3962.5
	Clip	**145.3**	**62.49**	**207.79**

MIS 19.07x faster than GridGraph
MIS 23.42x faster than X-Stream

4 Conclusion

CLIP: A single-machine graph processing system

Focus on squeezing out all the value of loaded data

Two techniques:

- **Reentry of Loaded Data**
  - Processing the loaded data **multiple times** rather than once → Reducing iterations → Reducing the total amount of disk I/O → Reducing execution time
- **Beyond the Neighborhood**
  - The user-defined function is allowed to update **an arbitrary** vertex's property

Up to **tens or even thousands of** times speedup

Thank You!!