ThunderRW: An In-Memory Graph Random Walk Engine

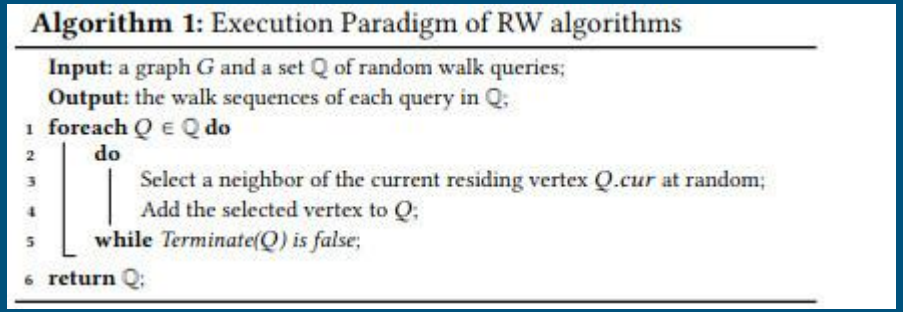
By: Shixuan Sun, Yuhang Chen, Shengliang Lu, Bingsheng He, Yuchen Li

Presentation by Yosef E Mihretie

Summary

- Random walk: processing where the next vertex is randomly chosen among the neighbors of the current vertex
- Observation: upto 75% of CPU slots are stalled due to memory latency
- Observation: As low as 2.3% of memory bandwidth in use
- Intra-query parallelism not possible, look into inter-query parallelism
- ThunderRW: assign multiple queries to each core, a core switches between the queries to hide memory latency
- Result: upto 3333x faster than baseline, 131.7x faster than other frameworks

Introduction: Random Walks



Algorithm 1: Execution Paradigm of RW algorithms

Input: a graph G and a set $\mathbb{Q}$ of random walk queries;
Output: the walk sequences of each query in $\mathbb{Q}$;

```
1 foreach Q ∈ Q do
2     do
3         Select a neighbor of the current residing vertex Q.cur at random;
4         Add the selected vertex to Q;
5     while Terminate(Q) is false;
6 return Q;
```

- e(v, u) selected with probability p(e)
- Unbiased vs Biased RWs: the probability distribution is uniform vs not
- Static vs Dynamic RWs: the probability distribution known before vs after execution
- Some RW algs: Personalized PageRank (PPR), DeepWalk, Node2Vec, MetaPath
- Different sampling methods: NAIVE, ITS, ALIAS, REJ, O-REJ

Previous Work

- Generic graph frameworks don’t take RW workloads into consideration
- **KnightKing**:
  - Distributed, 0-REJ sampling
  - BSP model that moves a step for all queries until done
  - Not generic (doesn’t support MetaPath for instance), suffers from the tail problem
- **C-SAW:**
  - For GPU, ITS sampling
  - BSP model as well
  - Not general (PPR and Node2Vec not support for example), high overhead because of ITS
- **GraphWalker:**
  - External-memory
  - ASP model: assigns queries to each worker and executed independently
  - Limited (only unbiased)

Motivations

- Memory a major bottleneck in RW algorithms

Table 1: Comparison of pipeline slot breakdown and memory bandwidth (the total value of read and write) between traditional graph algorithms and RW algorithms.

Method	Front End	Bad Spec	Core	Memory	Retiring	Memory Bandwidth
BFS	11.6%	9.1%	20.8%	40.6%	18.0%	51.7 GB/s
SSSP	9.1%	12.5%	24.9%	36.9%	16.6%	38.2 GB/s
PPR	0.6%	0.7%	15.8%	**73.1%**	9.7%	**1.4 GB/s**
DeepWalk	1.0%	3.9%	16.7%	**69.7%**	8.7%	**5.6 GB/s**
Node2Vec	11.5%	22.1%	24.3%	28.1%	14.1%	17.1 GB/s
MetaPath	6.2%	7.5%	29.7%	33.9%	22.7%	9.9 GB/s

Motivations

- Cost of computing p(e) is dependent on algorithms while sampling is flexible

Table 2: Comparison of execution time breakdown and the time complexity per step among RW algorithms where $v = Q.cur$ and u is the last vertex of Q.

Method	Time Breakdown			Complexity per Step		
	Compute	Sampling		Compute	Sampling	
	$p(e)$	Init	Gen	$p(e)$	Init	Gen
PPR	*N/A*	*N/A*	100%	*N/A*	*N/A*	$O(1)$
DeepWalk	*N/A*	*N/A*	100%	*N/A*	*N/A*	$O(1)$
Node2Vec	89.9%	9.9%	0.2%	$O(d_v \times \log d_u)$	$O(d_v)$	$O(1)$
MetaPath	29.0%	69.9%	1.1%	$O(d_v)$	$O(d_v)$	$O(1)$

- ThunderRW targets sampling

Motivations

- Different sampling methods perform differently. Moreover, dynamic RW is generally slower than static and unbiased RW

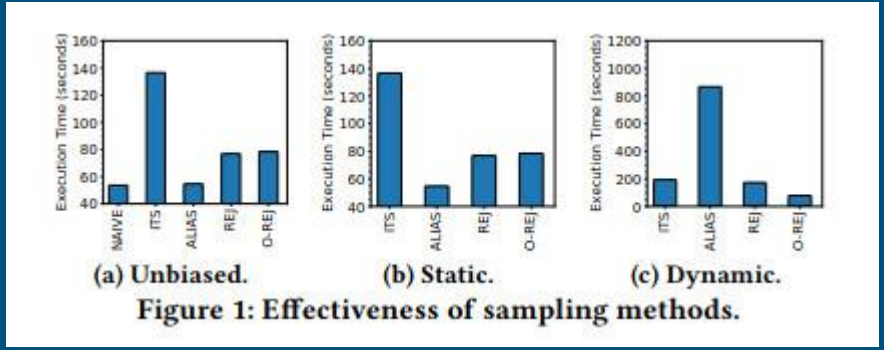
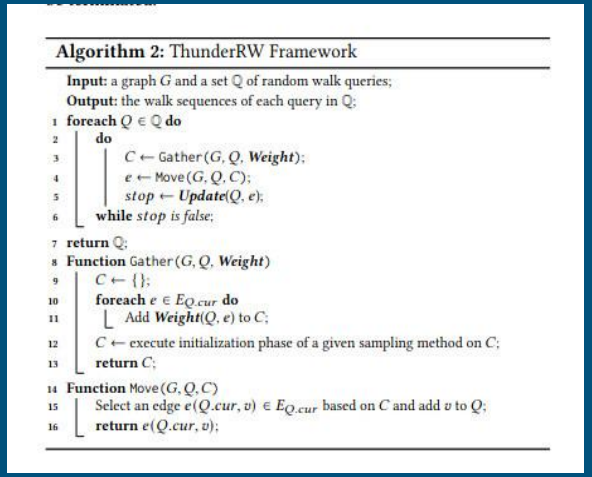


Figure 1: Effectiveness of sampling methods.

- ThunderRW optimizes for static, unbiased and dynamic RWs and it allows multiple different sampling methods

Model and API

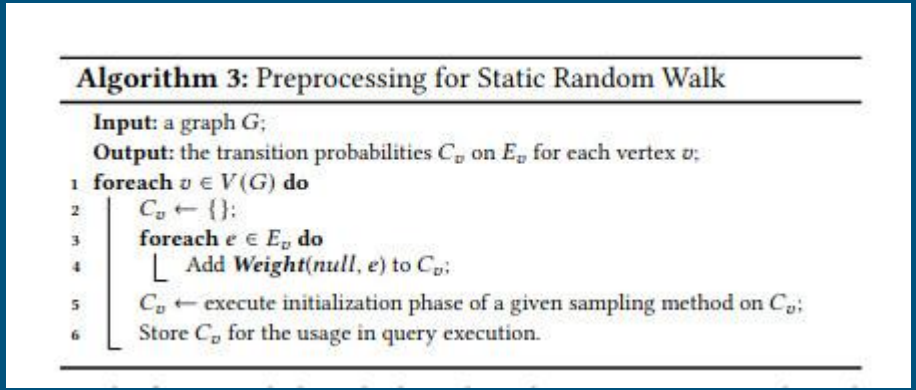
- ThunderRW exploits inter-query parallelism by statically assigning groups of queries to different workers which work independently
- It follows a step centric model where a step is abstracted into the Gather-Move-Update operations



Algorithm 2: ThunderRW Framework

Input: a graph G and a set $\mathbb{Q}$ of random walk queries;
Output: the walk sequences of each query in $\mathbb{Q}$;

```
foreach Q ∈ ℚ do
    do
        C ← Gather(G, Q, Weight);
        e ← Move(G, Q, C);
        stop ← Update(Q, e);
    while stop is false;
return ℚ;
Function Gather(G, Q, Weight)
    C ← {};
    foreach e ∈ E_{Q.cur} do
        Add Weight(Q, e) to C;
    C ← execute initialization phase of a given sampling method on C;
    return C;
Function Move(G, Q, C)
    Select an edge e(Q.cur, v) ∈ E_{Q.cur} based on C and add v to Q;
    return e(Q.cur, v);
```



Algorithm 3: Preprocessing for Static Random Walk

Input: a graph G;
Output: the transition probabilities C_v on E_v for each vertex v;

```
foreach v ∈ V(G) do
    C_v ← {};
    foreach e ∈ E_v do
        Add Weight(null, e) to C_v;
    C_v ← execute initialization phase of a given sampling method on C_v;
    Store C_v for the usage in query execution.
```

Step Interleaving

- ThunderRW targets move operations for memory latency hiding
- Decomposes a move into multiple steps where each step consumes data fetched by previous steps and fetches data for subsequent steps
- Workers hide memory latency by interleaving steps from different queries

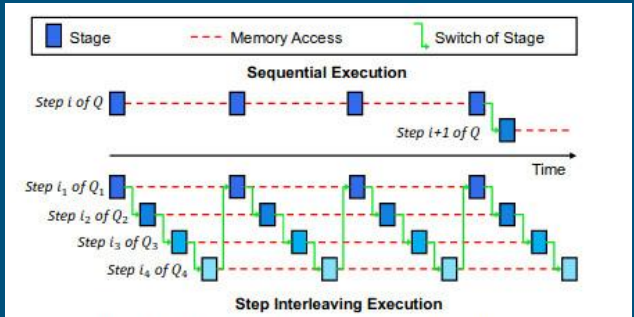


Figure 2: Sequential versus step interleaving.

Step Interleaving

- Efficient interleaving requires an efficient switch mechanism and enough workload for hiding memory latency
- ThunderRW builds and uses a stage dependency graph to make this easy

Table 4: Stages of Move with ALIAS and REJ ($v = Q.cur$).

Stage	ALIAS
S_0	O_0: Load d_v.
S_1	O_1: Generate an int random num x in $[0, d_v)$.
	O_2: Generate a real random num y in $[0, 1)$.
	O_3: Load $C[x] = (H[x], A[x])$.
S_2	O_4: If $y < H[x]$, $v' = A[x].first$; Else $v' = A[x].second$.
	O_5: Add v' to Q and return $e(v, v')$.

Stage	REJ
S_0	O_0: Load d_v.
S_1	O_1: Load the maximum value p_v^*.
S_2	O_2: Generate an int random num x in $[0, d_v)$.
	O_3: Generate a real random num y in $[0, p_v^*)$.
	O_4: Load $C[x] = p$.
S_3	O_5: If $y > C[x]$, jump to O_2; Else jump to O_6.
S_4	O_6: Load $e(v, v') = E_v[x]$.
S_5	O_7: Add v' to Q and return $e(v, v')$.

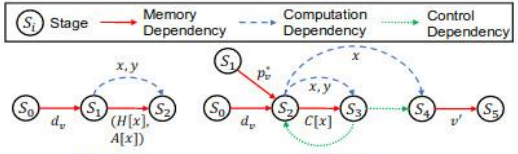


Figure 3: Stage dependency graph.

Experiments: Setup

- Intel Xeon W-2155 CPU with 220GB RAM, 10 physical cores w hyperthreading disabled for consistency, L1 = 32KB, L2 = 1MB and L3 = 13.75MB
- They run PPR, DeepWalk, Node2Vec and MetaPath algorithms on a variety of datasets

Table 5: Properties of real-world datasets.

Dataset	Name	$\|V\|$	$\|E\|$	d_{avg}	d_{max}	Memory
amazon	*am*	0.55M	1.85M	3.38	549	0.01GB
youtube	*yt*	1.14M	2.99M	5.24	28754	0.03GB
us patents	*up*	3.78M	16.52M	8.74	793	0.17GB
eu-2005	*eu*	0.86M	19.24M	44.74	68963	0.15GB
amazon-clothing	*ac*	15.16M	63.33M	4.18	12845	0.35GB
amazon-book	*ab*	18.29M	102.12M	5.58	58147	0.52GB
livejournal	*lj*	4.85M	68.99M	28.45	20333	0.54GB
com-orkut	*ot*	3.07M	117.19M	76.34	33313	0.89GB
wikidata	*wk*	40.96M	265.20M	6.47	8085513	1.29GB
uk-2002	*uk*	18.52M	298.11M	32.19	194955	2.30GB
twitter	*tw*	41.66M	1.21B	58.08	2997487	9.27GB
friendster	*fs*	65.61M	1.81B	55.17	5214	13.71GB

Experiments: ThunderRW vs Everyone

- Compared ThunderRW to
  - BL: a baseline implementation for in-memory random walks
  - HG: BL + suitable sampling technique selection + each query parallelized in OpenMP
  - GW: GraphWalker executed in-memory
  - KK: KnightKing run on a single machine

Table 6: Overall performance comparison (seconds).

	PPR					DeepWalk				Node2vec				MetaPath		
Dataset	***BL***	***HG***	***GW***	***KK***	***TRW***	***BL***	***HG***	***KK***	***TRW***	***BL***	***HG***	***KK***	***TRW***	***BL***	***HG***	***TRW***
am	0.06	0.008	0.42	0.012	**0.007**	2.16	0.21	0.44	**0.07**	9.97	0.26	2.08	**0.14**	0.22	0.018	**0.012**
yt	0.33	0.04	1.68	0.05	**0.015**	9.78	0.98	1.93	**0.26**	853.13	1.30	5.94	**1.03**	6.18	**0.23**	0.24
up	1.24	0.13	7.19	0.19	**0.07**	45.44	4.33	8.41	**0.95**	369.00	6.20	16.92	**4.01**	4.88	0.40	**0.24**
eu	0.16	0.02	0.99	0.03	**0.011**	8.16	0.82	1.56	**0.20**	2731.07	1.47	4.43	**1.14**	90.55	**3.18**	3.55
ac	4.84	0.51	19.31	0.65	**0.19**	173.66	17.86	31.88	**3.31**	6951.12	24.54	87.86	**6.26**	45.01	2.01	**1.69**
ab	8.86	0.94	26.74	1.09	**0.26**	212.80	22.24	40.07	**4.01**	26231.45	32.04	100.78	**7.87**	128.35	5.06	**4.47**
lj	1.69	0.19	7.90	0.23	**0.06**	55.63	5.44	10.67	**1.19**	2951.33	9.09	24.95	**6.20**	18.08	0.94	**0.75**
ot	1.49	0.16	5.25	0.19	**0.04**	38.54	3.70	7.97	**0.80**	5891.28	7.28	15.16	**4.82**	40.77	1.72	**1.57**
wk	21.86	2.21	47.05	3.07	**0.59**	502.27	49.67	95.17	**9.26**	*OOT*	68.43	216.24	**27.68**	5.98	**0.54**	0.55
uk	6.47	0.69	27.72	0.90	**0.24**	203.86	20.42	21.40	**4.56**	12630.01	34.36	94.69	**28.68**	322.66	12.84	**12.56**
tw	26.42	2.73	77.12	3.61	**1.16**	575.43	61.18	115.92	**11.13**	*OOT*	130.72	232.41	**91.00**	*OOT*	12300.32	**9780.20**
fs	79.14	8.20	223.81	10.72	**4.10**	1043.93	108.23	208.45	**17.67**	*OOT*	178.15	364.51	**120.16**	683.05	28.69	**25.01**

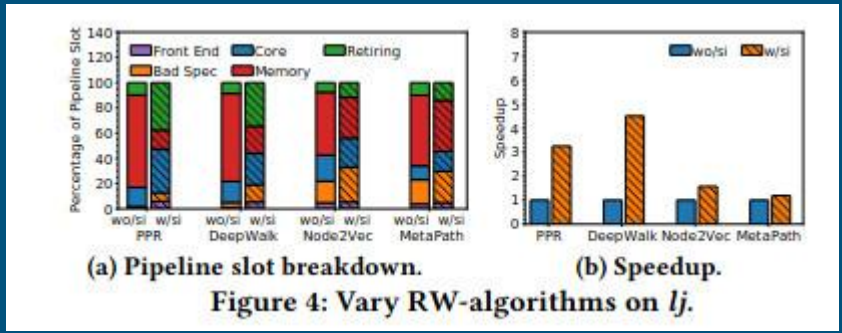
Experiments: ThunderRW vs Everyone

- TRW runs 54.6-131.7x faster than GW and 1.7-14.6x faster than KK
- TRW runs 8.6-3333x faster than BL and 6.1x faster than HG
- For MetaPath, the gather stage has more weight and sometimes HG runs faster

Table 6: Overall performance comparison (seconds).

	PPR					DeepWalk				Node2vec				MetaPath		
Dataset	*BL*	*HG*	*GW*	*KK*	*TRW*	*BL*	*HG*	*KK*	*TRW*	*BL*	*HG*	*KK*	*TRW*	*BL*	*HG*	*TRW*
am	0.06	0.008	0.42	0.012	**0.007**	2.16	0.21	0.44	**0.07**	9.97	0.26	2.08	**0.14**	0.22	0.018	**0.012**
yt	0.33	0.04	1.68	0.05	**0.015**	9.78	0.98	1.93	**0.26**	853.13	1.30	5.94	**1.03**	6.18	**0.23**	0.24
up	1.24	0.13	7.19	0.19	**0.07**	45.44	4.33	8.41	**0.95**	369.00	6.20	16.92	**4.01**	4.88	0.40	**0.24**
eu	0.16	0.02	0.99	0.03	**0.011**	8.16	0.82	1.56	**0.20**	2731.07	1.47	4.43	**1.14**	90.55	**3.18**	3.55
ac	4.84	0.51	19.31	0.65	**0.19**	173.66	17.86	31.88	**3.31**	6951.12	24.54	87.86	**6.26**	45.01	2.01	**1.69**
ab	8.86	0.94	26.74	1.09	**0.26**	212.80	22.24	40.07	**4.01**	26231.45	32.04	100.78	**7.87**	128.35	5.06	**4.47**
lj	1.69	0.19	7.90	0.23	**0.06**	55.63	5.44	10.67	**1.19**	2951.33	9.09	24.95	**6.20**	18.08	0.94	**0.75**
ot	1.49	0.16	5.25	0.19	**0.04**	38.54	3.70	7.97	**0.80**	5891.28	7.28	15.16	**4.82**	40.77	1.72	**1.57**
wk	21.86	2.21	47.05	3.07	**0.59**	502.27	49.67	95.17	**9.26**	*OOT*	68.43	216.24	**27.68**	5.98	**0.54**	0.55
uk	6.47	0.69	27.72	0.90	**0.24**	203.86	20.42	21.40	**4.56**	12630.01	34.36	94.69	**28.68**	322.66	12.84	**12.56**
tw	26.42	2.73	77.12	3.61	**1.16**	575.43	61.18	115.92	**11.13**	*OOT*	130.72	232.41	**91.00**	*OOT*	12300.32	**9780.20**
fs	79.14	8.20	223.81	10.72	**4.10**	1043.93	108.23	208.45	**17.67**	*OOT*	178.15	364.51	**120.16**	683.05	28.69	**25.01**

Experiments: Evaluating Step Interleaving

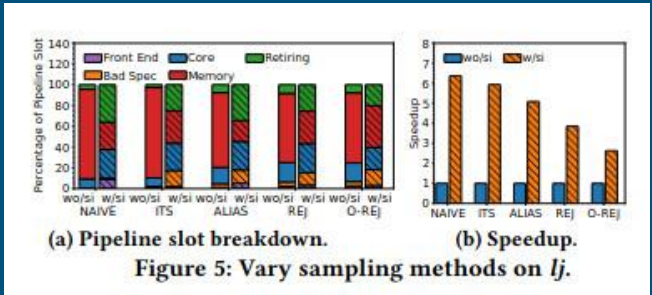


(a) Pipeline slot breakdown. (b) Speedup.

Figure 4: Vary RW-algorithms on *lj*.

- DeepWalk and PPR benefit the most from interleaving. PPR less so because all queries in PPR issue from a single vertex (are just more optimal to begin with)
- Node2Vec uses binary search in Move, thus makes more random accesses. MetaPath has the gather step dominating cost. Thus the two see smaller benefits.

Experiments: Evaluating Step Interleaving



(a) Pipeline slot breakdown. (b) Speedup.

Figure 5: Vary sampling methods on *lj*.

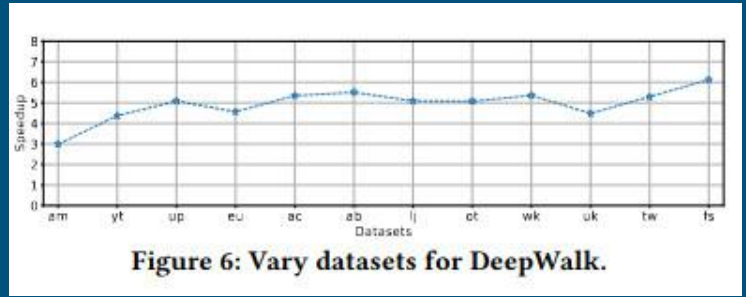


Figure 6: Vary datasets for DeepWalk.

- Step Interleaving effective across multiple sampling methods and datasets
- am can fit in the LLC while yt is twice the LLC. eu and uk are dense and thus have good memory locality. Thus these see lower speedups. Sparse and large graphs see higher speedups.

Experiments: Scalability Evaluation

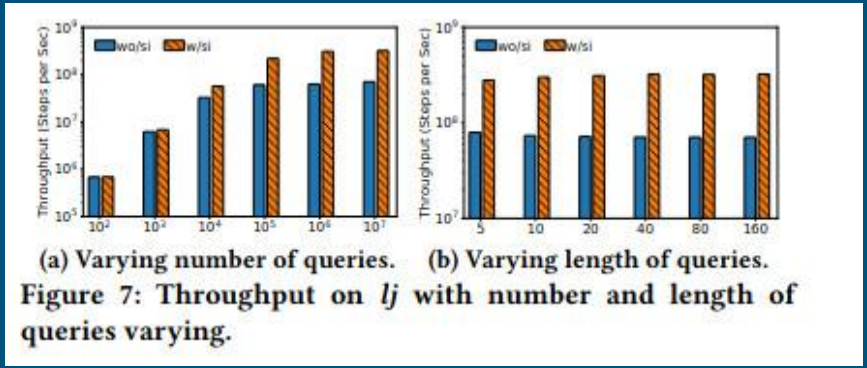


Figure 7: Throughput on *lj* with number and length of queries varying.

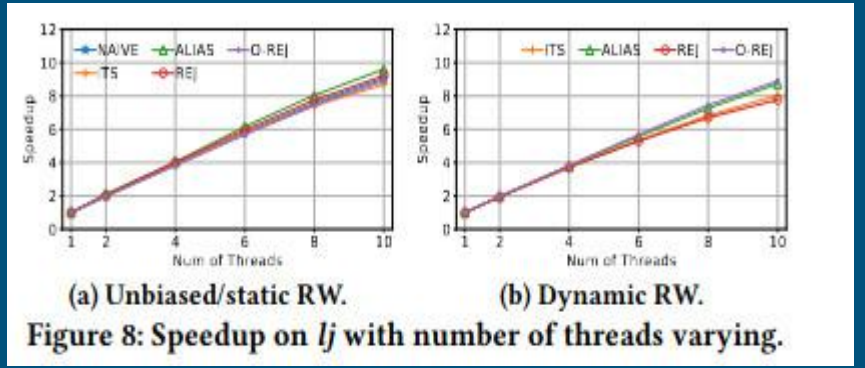


Figure 8: Speedup on *lj* with number of threads varying.

- Highly scalable with number & length of queries
- Scales almost linearly with number of threads

Summary

- ThunderRW is an in-memory RW engine that leverages inter-query parallelism by first assigning different queries to different cores and then interleaving the steps of queries assigned to the same core to hide memory latency
- Possible future work: dynamic load adjustment