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: up to 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: up to 3333x faster than baseline, 131.7x faster than other frameworks
Introduction: Random Walks

- $e(v, u)$ 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
Motivations

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

- ThunderRW targets sampling
Motivations

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

- 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

```plaintext
Input: a graph G and a set Q of random walk queries;
Output: the walk sequences of each query in Q;
foreach q ∈ Q do
  C ← Gather(G, Q, Weight);
  e ← Move(G, Q, C);
  step ← Update(q, e);
  while step is false;
  return C;
Function Gather(G, Q, Weight)
  C ← {};
  foreach e ∈ Eq, do
    C ← Add Weight(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, e) ∈ E(G), cur based on C and add e to Q;
  return e(Q, cur, e);
```

Algorithm 3: Preprocessing for Static Random Walk

```plaintext
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
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  | \( |V| \) | \( |E| \) | \( d_{av} \) | \( d_{max} \) | 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  |
| wikitdata | 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  |
| friends    | 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>
<thead>
<tr>
<th>Dataset</th>
<th>PPR</th>
<th>DeepWalk</th>
<th>Node2vec</th>
<th>MetaPath</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BL</td>
<td>HG</td>
<td>GW</td>
<td>KK</td>
</tr>
<tr>
<td>am</td>
<td>0.06</td>
<td>0.008</td>
<td>0.42</td>
<td>0.012</td>
</tr>
<tr>
<td>yt</td>
<td>0.33</td>
<td>0.04</td>
<td>1.68</td>
<td>0.05</td>
</tr>
<tr>
<td>up</td>
<td>1.24</td>
<td>0.13</td>
<td>7.19</td>
<td>0.19</td>
</tr>
<tr>
<td>eu</td>
<td>0.16</td>
<td>0.02</td>
<td>0.99</td>
<td>0.03</td>
</tr>
<tr>
<td>ac</td>
<td>4.84</td>
<td>0.51</td>
<td>19.31</td>
<td>0.65</td>
</tr>
<tr>
<td>ab</td>
<td>8.86</td>
<td>0.94</td>
<td>26.74</td>
<td>1.09</td>
</tr>
<tr>
<td>dj</td>
<td>1.69</td>
<td>0.19</td>
<td>7.90</td>
<td>0.23</td>
</tr>
<tr>
<td>ot</td>
<td>1.49</td>
<td>0.16</td>
<td>5.25</td>
<td>0.19</td>
</tr>
<tr>
<td>wk</td>
<td>21.86</td>
<td>2.21</td>
<td>47.05</td>
<td>3.07</td>
</tr>
<tr>
<td>uk</td>
<td>6.47</td>
<td>0.69</td>
<td>27.72</td>
<td>0.90</td>
</tr>
<tr>
<td>tw</td>
<td>26.42</td>
<td>2.73</td>
<td>77.12</td>
<td>3.61</td>
</tr>
<tr>
<td>fs</td>
<td>79.14</td>
<td>8.20</td>
<td>223.81</td>
<td>10.72</td>
</tr>
</tbody>
</table>

Table 6: Overall performance comparison (seconds).
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
Experiments: Evaluating Step Interleaving

- 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

- 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

- 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.