Efficient Parallel Graph Processing on GPU using Approximate Computing

Somesh Singh

(https://ssomesh.github.io/)

Department of Computer Science and Engineering

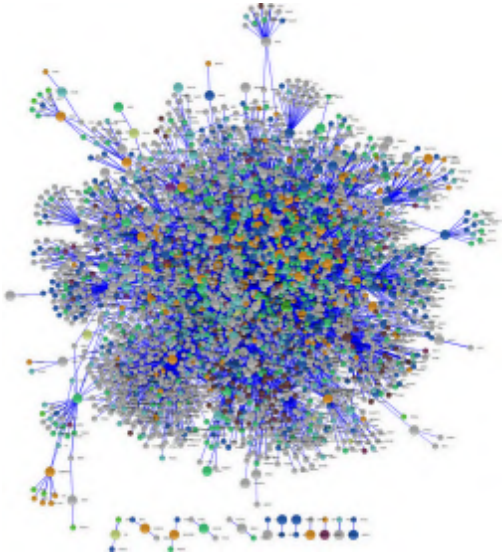
Indian Institute of Technology Madras, India

May 6, 2021

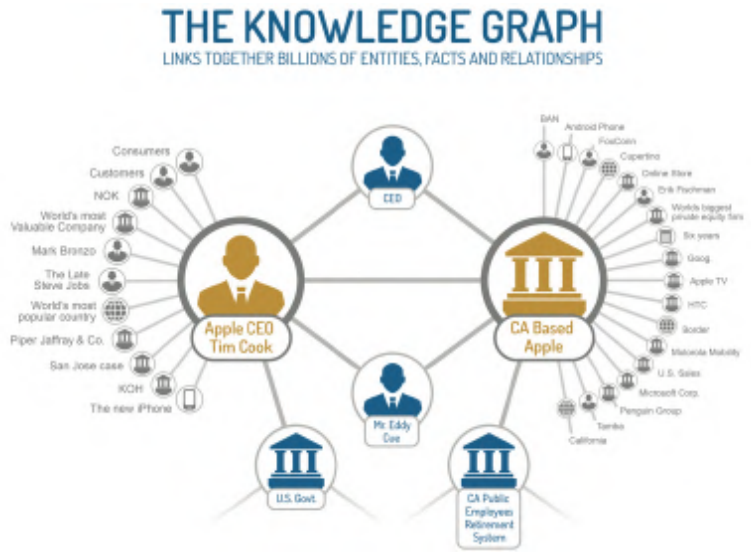
Outline

- `Graffix` : Techniques targeting GPU-specific aspects for parallel approximate graph processing
- `Graprox` : Generalized techniques for parallel approximate graph processing
- Research Interests

Graphs are Ubiquitous



Biological Network



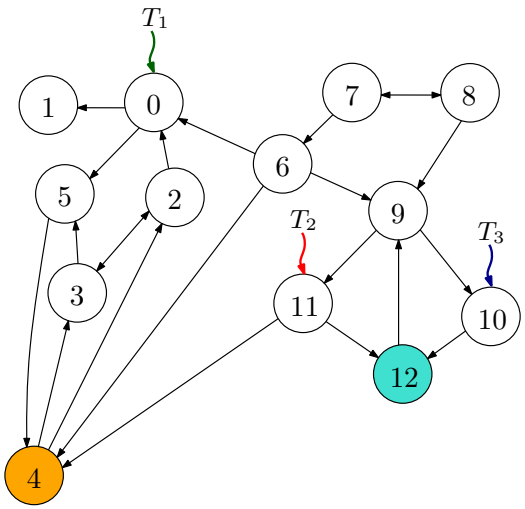
Knowledge Network

Social Network

Road Network

Image Source: Google Images

Challenges in Parallel Graph Processing



dest

1	5	0	3	2	5	2	3	4	0	4	9	6	8	7	9	10	11	12	4	12	9
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

src

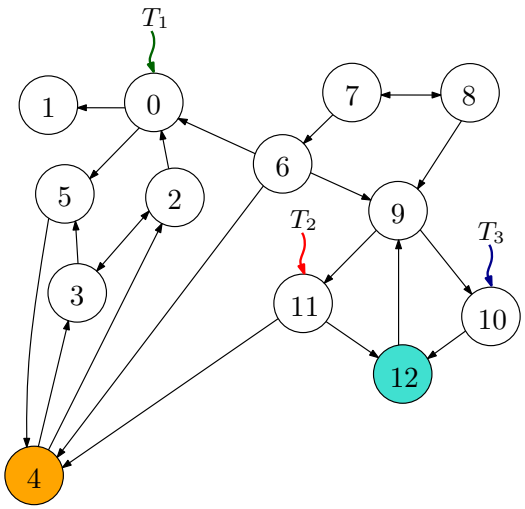
0	2	2	4	6	8	9	12	14	16	18	19	21	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

dist

0	1	2	3	4	5	6	7	8	9	10	11	12

CSR representation

Challenges in Parallel Graph Processing



dest

1	5	0	3	2	5	2	3	4	0	4	9	6	8	7	9	10	11	12	4	12	9
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

src

0	2	2	4	6	8	9	12	14	16	18	19	21	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

dist

0	1	2	3	4	5	6	7	8	9	10	11	12

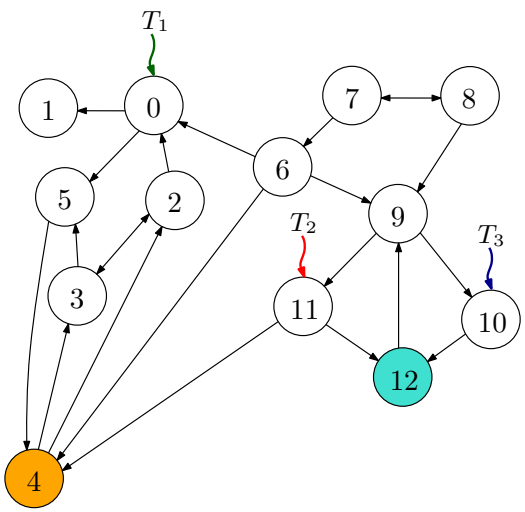
CSR representation

Assumptions

- vertex-centric model of parallelization
- propagation-based graph kernels

```
1  Graph G(V,E) = read_input();
2  v.dist = ∞ ∀ v ∈ V;
3  source.dist = 0;
4  Worklist wl = {source};
5  do {
6   changed = false;
7   forall Node u : wl do {
8    for Node v : G.neighbors(u) do {
9     newVal = dist[u] + e_uv.wt();
10    if(newVal < dist[v]) {
11     oldVal = atomicMin(&dist[v], newVal);
12     if(newVal < oldVal) {
13      wl.push(v);
14      changed = true;
15     } } } }
16 } while(changed);
```

Challenges in Parallel Graph Processing



dest

1	5	0	3	2	5	2	3	4	0	4	9	6	8	7	9	10	11	12	4	12	9
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

src

0	2	2	4	6	8	9	12	14	16	18	19	21	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

dist

0	1	2	3	4	5	6	7	8	9	10	11	12

CSR representation

Assumptions

- vertex-centric model of parallelization
- propagation-based graph kernels

```
1  Graph G(V,E) = read_input();
2  v.dist = ∞ ∀ v ∈ V;
3  source.dist = 0;
4  Worklist wl = {source};
5  do {
6   changed = false;
7   forall Node u : wl do {
8    for Node v : G.neighbors(u) do {
9     newVal = dist[u] + e_uv.wt();
10    if(newVal < dist[v]) {
11     oldVal = atomicMin(&dist[v], newVal);
12     if(newVal < oldVal) {
13      wl.push(v);
14      changed = true;
15     } } } }
16 } while(changed);
```

- Irregular accesses: The indirection "`dist[dest[id]]`".
- Memory-latency bound.
- Load imbalance: Skew in vertex degrees.

Our Approach

- Combine *parallelization* with *approximate computing* to make graph processing more efficient at the expense of accuracy.
- Provide tunable knobs to control the performance-accuracy trade-off.

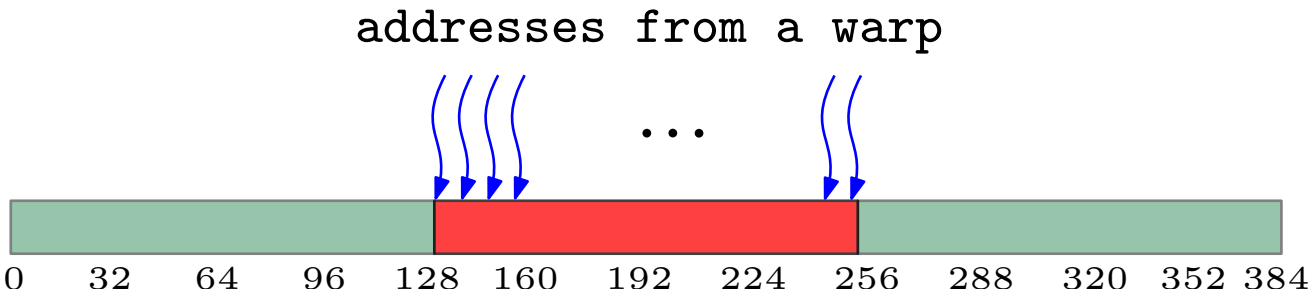
Our Approach

- Combine *parallelization* with *approximate computing* to make graph processing more efficient at the expense of accuracy.
- Provide tunable knobs to control the performance-accuracy trade-off.

Graffix techniques

1. Improving Memory Coalescing
   - make the graph layout more *structured* to improve locality.
   - *renumber* the graph vertices and *replicate* a select set of vertices.
2. Reducing Memory Latency
   - process *well-connected* sub-graphs, iteratively, inside shared memory.
3. Reducing Thread Divergence
   - normalize degrees across nodes assigned to a warp.

- Accesses to global memory by warp-threads are *coalesced* into a single memory transaction if warp-threads access a contiguous block of memory.



- Irregular memory accesses are not coalesced; translate into several load/store transactions.

Image Source: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

Improving Memory Coalescing

Vertex Renumbering

- Assign *nearby* ids to vertices to be accessed by warp-threads.

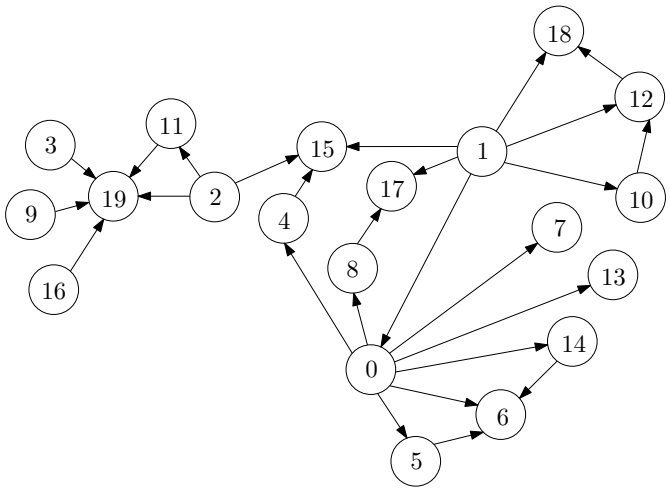
Improving Memory Coalescing

Vertex Renumbering

- Assign *nearby* ids to vertices to be accessed by warp-threads.

Approach

- Perform BFS from a highest outdegree node.
- Assign ids level-by-level; incrementally in a round-robin fashion at a level.



Original graph

edges

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
4	5	6	7	8	13	14	0	10	12	15	17	18	11	15	19	19	15	6	17	19	12	19	18	6	19

offset

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	7	13	16	17	18	19	19	19	20	21	22	23	24	24	25	25	26	26	26	26

node attributes

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

edge attributes

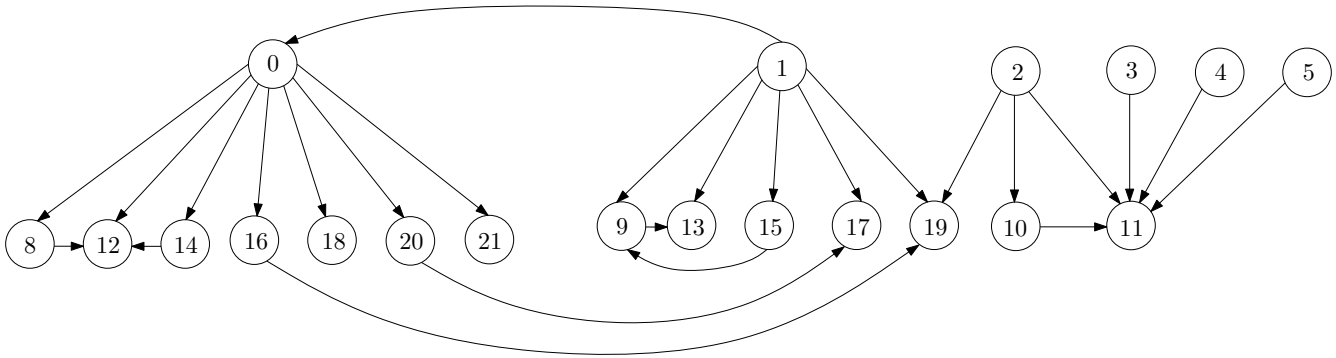
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Improving Memory Coalescing

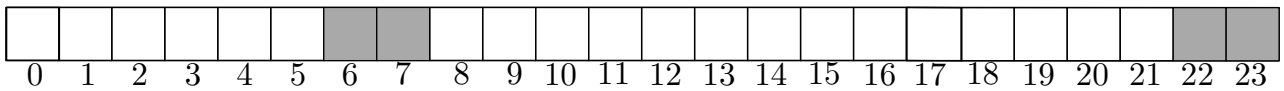
Approach

- Start a level at a multiple of $k \mid 1 \leq k \leq$ warp-size $\rightarrow$ creates *holes*.
- Divide the node array (after renumbering) into chunks of size k.

Renumbered graph



Creation of *holes* after renumbering



$k = 8$

Improving Memory Coalescing

Vertex Replication

- A node occurs exactly once, so it cannot be nearby all its neighbors even after the renumbering.
- Replication brings such a node *close* to its otherwise *far* neighbors.

Improving Memory Coalescing

Vertex Replication

- A node occurs exactly once, so it cannot be nearby all its neighbors even after the renumbering.
- Replication brings such a node *close* to its otherwise *far* neighbors.

Approach

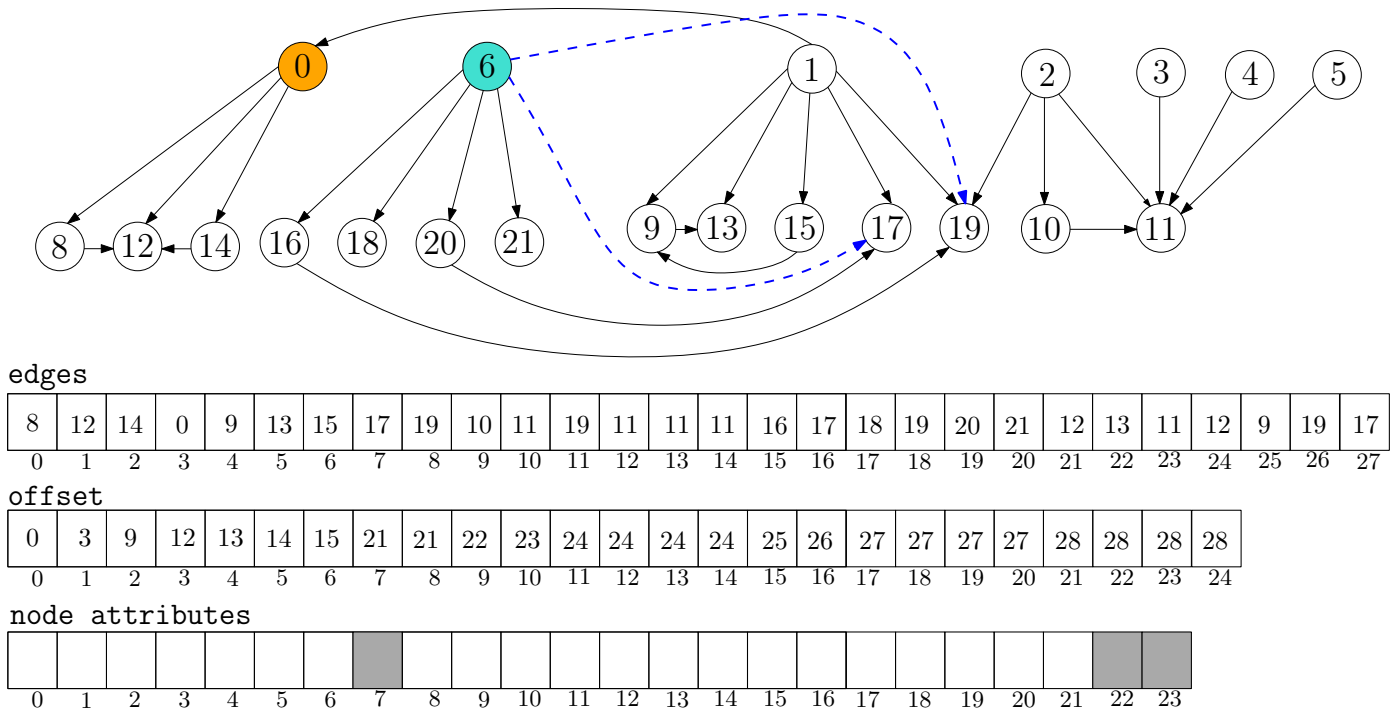
- If a node is well-connected to a chunk, replicate the node in a chunk in the previous BFS level.

$$\text{connectedness}_{\text{chunk}}^{\text{node}} \triangleq \left(\frac{\#\text{ edges to chunk from a node}}{\#\text{ non-hole nodes in chunk}} \right) \geq \textit{threshold}$$

- Distribute the outgoing edges of a node among its copies.
- Add edges from node's replica to its 2-hop neighbors inside the chunk.
- Perform a merge operation on the values of the replicas after each iteration.

Improving Memory Coalescing

- Node 0 is well-connected to the chunk 16..23
- $\text{connectedness}^{0}_{16..23} = \frac{4}{6} = 0.67 \geq 0.5$ (*threshold*)
- Node 0 is replicated in the chunk 0..7; its replica is assigned id 6.



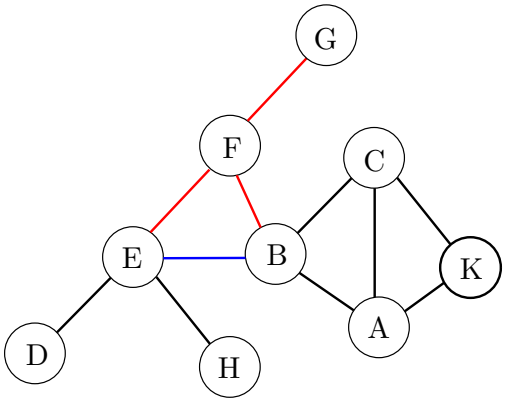
Modified graph

- Shared memory (or scratchpad memory) is a software managed cache.
- Accesses are as fast as registers, if there are no bank conflicts, even for irregular accesses.
- Useful if there is enough *reuse* of the data brought into shared memory.

Reducing Memory Latency using Shared Memory

- Clustering-coefficient measures the degree to which nodes in a graph tend to "cluster".
- Local clustering-coefficient (LCC) of a node, X :

$$\text{LCC}_X = \frac{\#\text{ pairs of X's neighbors that are neighbors}}{\#\text{ pairs of X's neighbors}}$$



$\#$ of pairs of F's neighbors that are neighbors $= 1$

$\#$ of pairs of F's neighbors $= \binom{3}{2} = 3$

$\text{LCC}_F = \frac{1}{3}$

Reducing Memory Latency using Shared Memory

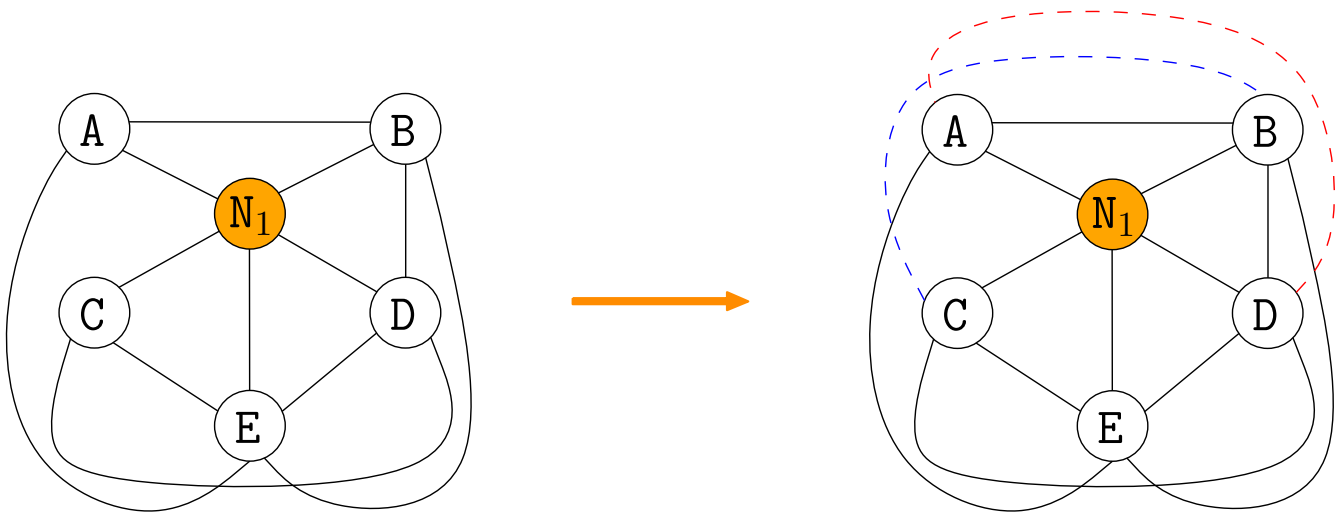
Vertices with *local clustering coefficient* (LCC) $\geq$ *threshold* are more frequently accessed in iterative processing.

Reducing Memory Latency using Shared Memory

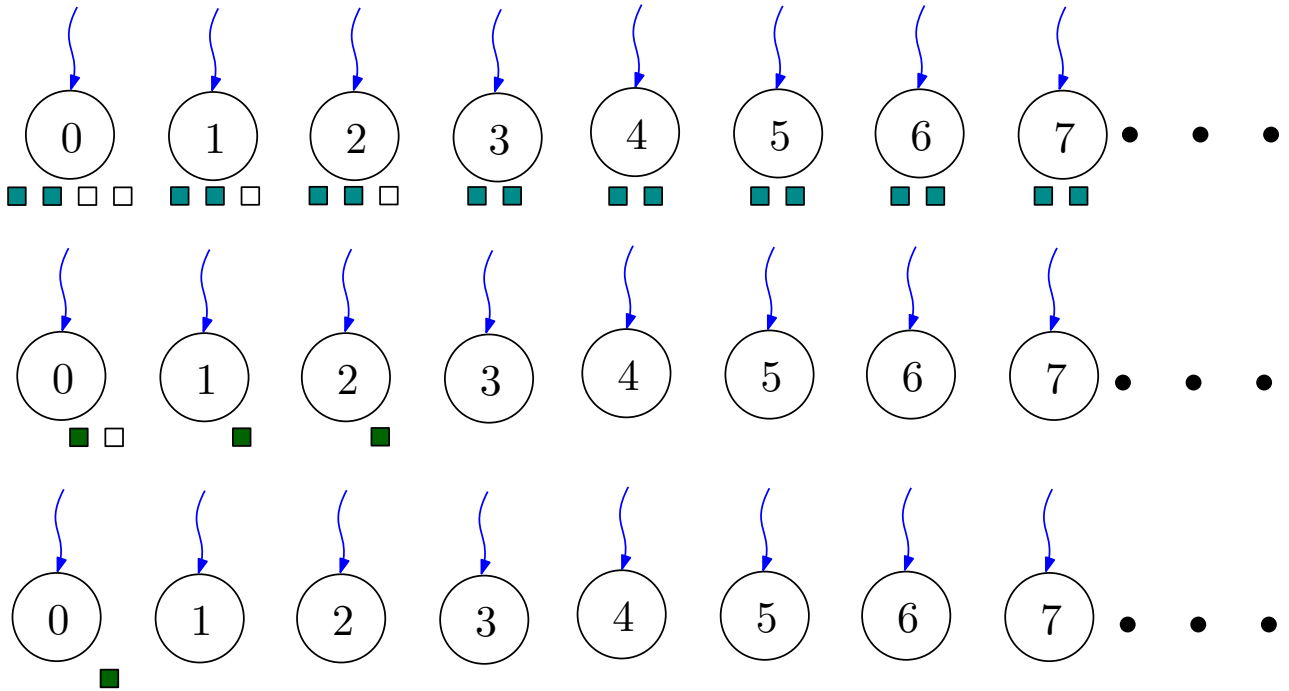
Vertices with *local clustering coefficient* (LCC) $\geq$ *threshold* are more frequently accessed in iterative processing.

Approach

- Increase LCC of node if LCC $\leq$ *threshold* and LCC $\sim$ *threshold*.
- Boost LCC of node if LCC $\geq$ *threshold*.
- Cap on the total number of additional edges added in the graph.



- All threads of a warp execute in lockstep.
- When there is load-imbalance among warp threads, other threads have to wait for the slowest thread.



Thread divergence due to load-imbalance

Reducing Thread Divergence

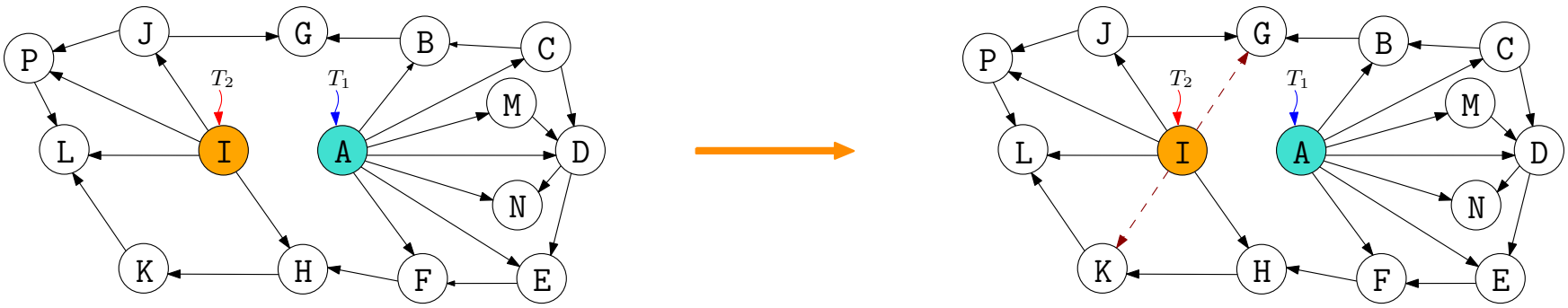
Make node degrees *nearly* uniform within each warp.

Reducing Thread Divergence

Make node degrees *nearly* uniform within each warp.

Approach

- Add edges to the nodes that are deficient in their connectivity.
- Add edges between 2-hop neighbors for faster convergence.
- Increase the degree of the candidate nodes to be close to α% of max. degree (e.g., 85%); α is tunable.



Experimental Setup

CPU	Intel Xeon E5-2650 v2 (32 cores, 2.6 GHz, 96 GB RAM).
GPU	Nvidia Pascal P100 (56 SMXs, 3584 cores, 16 GB global memory with bandwidth of 732 GB/s).
Software	CentOS 6.5, gcc 4.8.2, CUDA 8.0

Machine Configuration

Graph	$\|V\|$ $\times 10^6$	$\|E\|$ $\times 10^6$	Graph type
USA-road	23.9	57.7	Road network, large diameter
LiveJournal	4.8	68.9	Social network, small diameter
rmat26	67.1	1073.7	Synthetic scale-free graph
random26	67.1	1073.7	Synthetic random graph
twitter	41.6	1468.3	Twitter graph 2010 snapshot

Input Graphs

Experimental Setup

Graph Algorithms

- Single Source Shortest Path (Distance) computation (SSSP)
- PageRank computation (PR)
- Strongly Connected Component computation (SCC)
- Minimum Spanning Tree Weight computation (MST)
- Betweenness Centrality computation (BC)

Baselines

- Baseline I: Our exact parallel versions of SSSP, PR, SCC, MST, BC.
- Baseline II: SSSP, PR, BC from Tigr*.
- Baseline III: SSSP, PR, BC from Gunrock†.

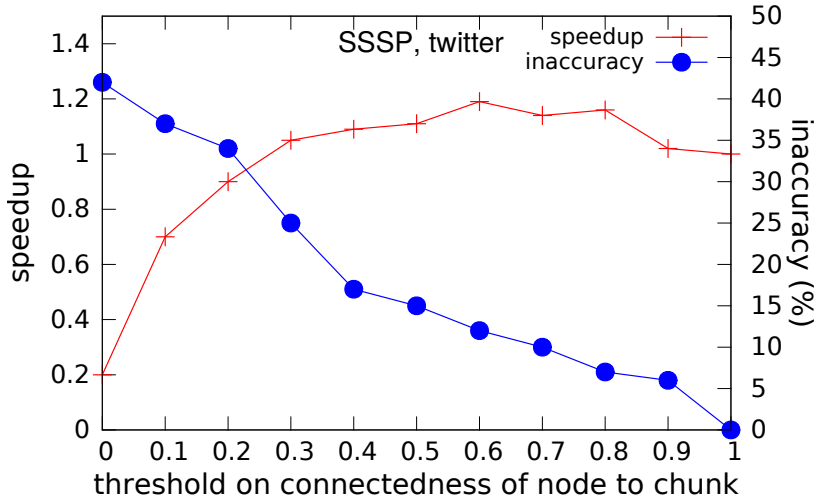
*https://github.com/AutomataLab/Tigr

†https://github.com/gunrock/gunrock

Results

Improving Memory Coalescing

	Baseline I	Baseline II	Baseline III
Mean Speedup	1.16×	1.10×	1.14×
Mean Inaccuracy	10%	9%	9%



Effect of varying the threshold for node replication on memory coalescing. (Chunk size is set to 16.)

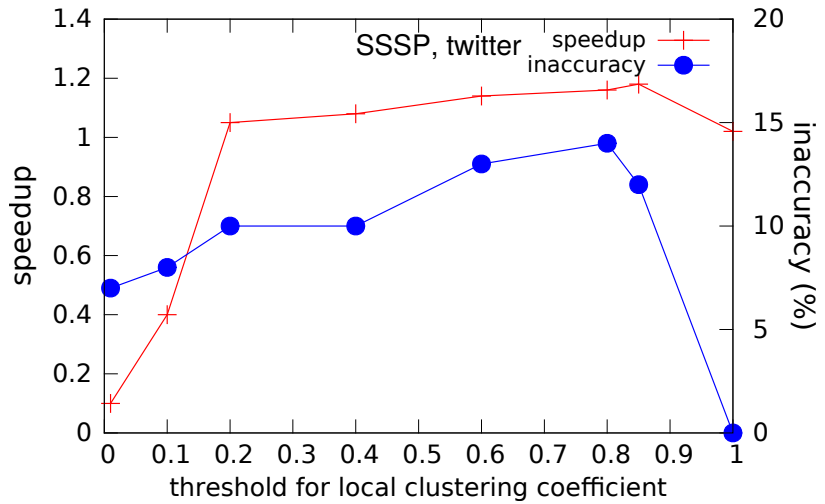
Takeaway:

Desired accuracy and performance for an algorithm – input graph pair can be achieved by tuning the chunk size and the threshold for node replication.

Results

Reducing Memory Latency

	Baseline I	Baseline II	Baseline III
Mean Speedup	1.20×	1.19×	1.19×
Mean Inaccuracy	13%	12%	12%



Effect of varying the LCC threshold on memory latency.

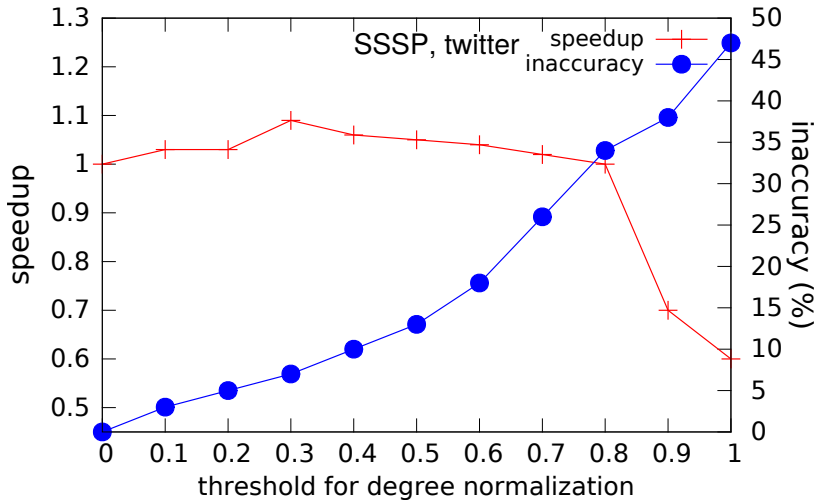
Takeaway:

Appreciable speedup, with low inaccuracy, can be achieved by processing well-connected subgraphs inside shared memory.

Results

Reducing Thread Divergence

	Baseline I	Baseline II	Baseline III
Mean Speedup	1.07×	1.03×	1.07×
Mean Inaccuracy	8%	8%	8%



Effect of varying the threshold for degree normalization.

Takeaway:

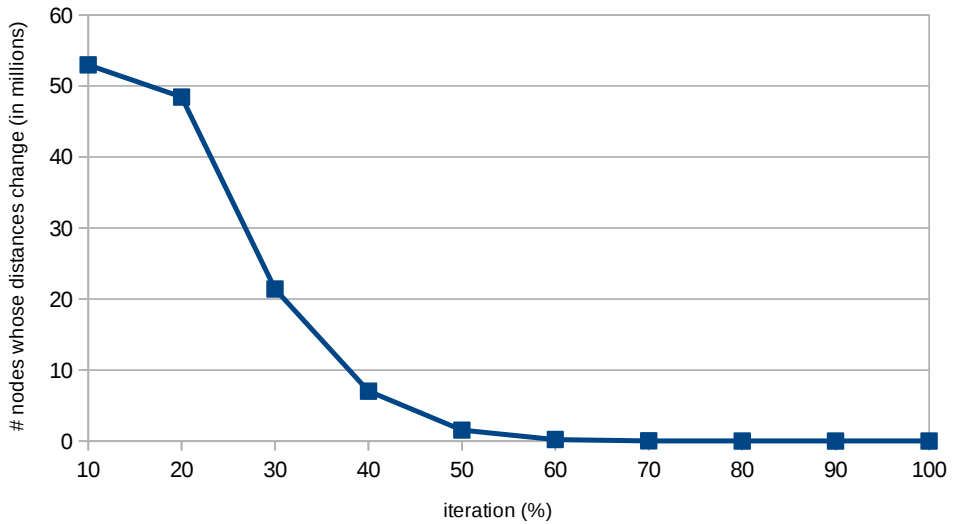
Small speedup with low inaccuracy can be achieved using a low threshold for degree normalization.

Graprox : Techniques for Parallel Approximate Graph Processing

1. Reduced execution
   - cut-short the number of outerloop iterations.
2. Partial graph processing
   - process only a subset of the edges in each outerloop iteration.
3. Approximate graph representation
   - merge nodes with overlapping neighbors, based on Jaccard's similarity.

Reduced Execution

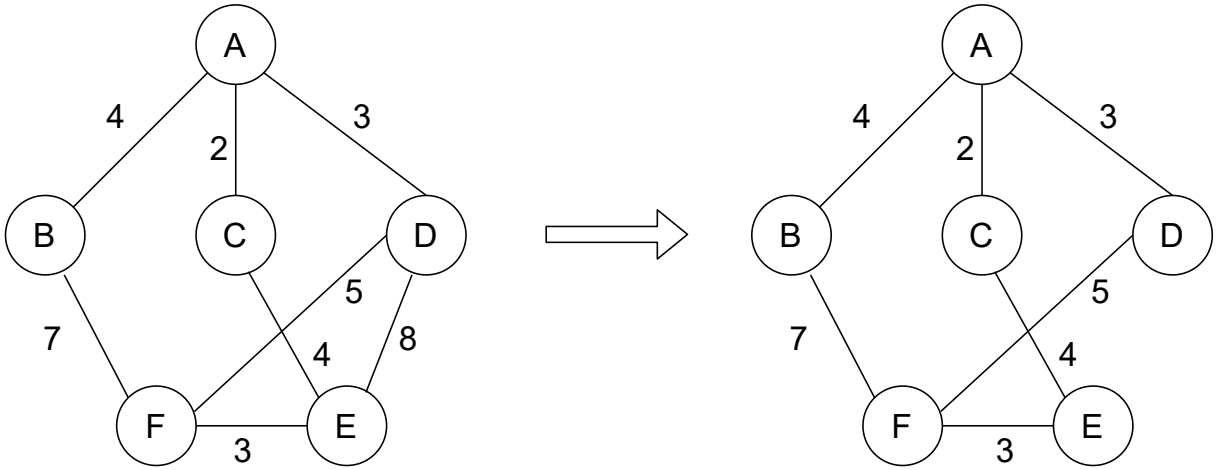
- Cut-short the number of outerloop iterations based on online stopping criteria.
- Helpful when majority of work gets done in the initial iterations.



SSSP computation on `rmat26` graph

Partial Graph Processing

- Process only a subset of the edges in each outerloop iteration.
- At each node, select the edges to be processed; ignore others.
- Helps improve performance since the work done per iteration (measured as number of edges traversed) is less.



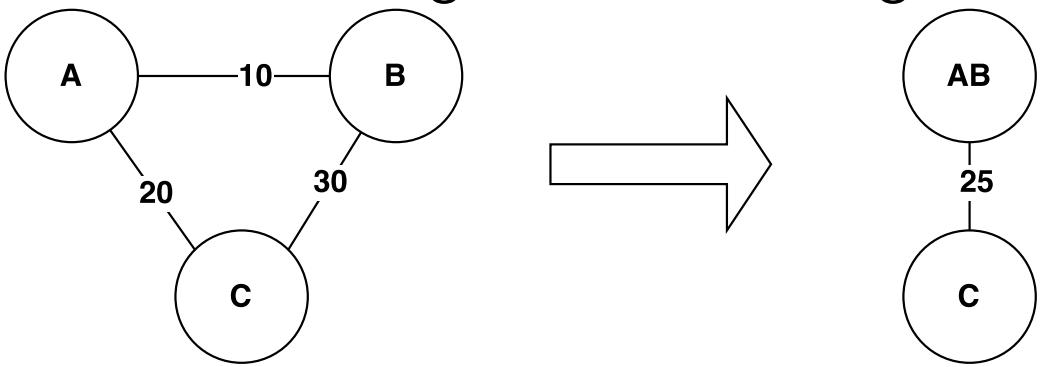
For SSSP computation

Approximate Graph Representation

- Lossy graph compression by merging nodes with overlapping neighbors.
- Jaccard's coefficient J_{ij}, for vertices v_i and v_j with sets of neighbors $N(v_i)$ and $N(v_j)$ respectively, is:

$$J_{ij} = \frac{|N(v_i) \cap N(v_j)|}{|N(v_i) \cup N(v_j)|}$$

- If there is a triangle a-b-c and a-b get merged:



Results

Graprox

	Technique	Mean Speedup	Mean Inaccuracy
SSSP	Outer-loop iterations	1.34×	6.07%
	Partial processing of graph	1.38×	16.19%
	Approx. graph representation	1.22×	13.87%
MST	Outer-loop iterations	1.18×	16.05%
	Partial processing of graph	1.65×	17.44%
	Approx. graph representation	1.44×	15.17%
SCC	Outer-loop iterations	1.25×	18.26%
	Partial processing of graph	1.32×	19.61%
	Approx. graph representation	1.45×	20.11%
PR	Outer-loop iterations	2.03×	2.75%
	Partial processing of graph	1.43×	15.74%
	Approx. graph representation	1.37×	13.70%

Takeaway:

Approximate computing techniques are consistently helpful in improving the execution performance of graph analytics in exchange for inaccuracy.

In Summary

1. Parallel graph processing is challenging due to *irregularity* in the data-access, control-flow, and communication patterns.
2. We proposed techniques for making graphs more amenable to processing on GPU.
3. Our techniques provide *tunable knobs* to control the performance-accuracy trade-off in graph applications.
4. The techniques are generally applicable to a large class of parallel graph algorithms and input graphs of varying characteristics.
5. Approximate computing combined with parallelization promises to make heavy-weight graph computation practical, as well as, scalable.

Research Interests

- High-performance computing
- High-performance graph analytics
- Parallel computing

Problems of Interest

- Scalable graph mining
- Optimizing parallel sparse matrix computations
- Parallel approximate processing on dynamic graphs

Research Interests

- High-performance computing
- High-performance graph analytics
- Parallel computing

Problems of Interest

- Scalable graph mining
- Optimizing parallel sparse matrix computations
- Parallel approximate processing on dynamic graphs

Thank You