

# 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

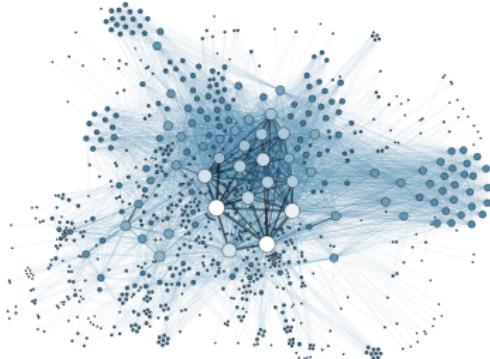
July 2, 2021



# Outline

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

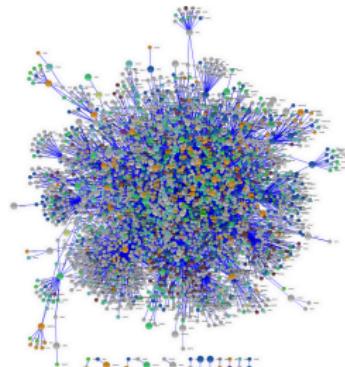
# Graphs are Ubiquitous



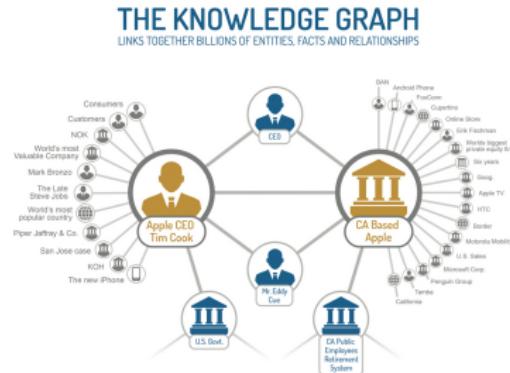
Social Network

Image Source: Google Images

Somesh Singh



Biological Network



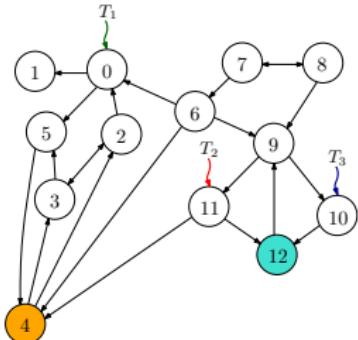
Knowledge Network



Road Network

Efficient Parallel Graph Processing on GPU using Approximate Computing

# Challenges in Parallel Graph Processing



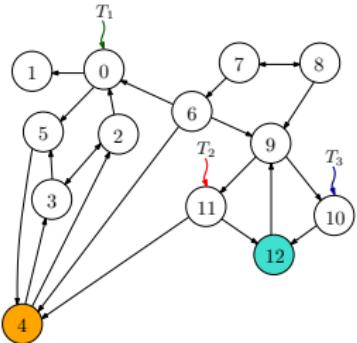
dest	1	5	0	3	2	5	2	3	4	0	4	9	6	8	7	9	10	11	12	4	12	9
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	

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	13								

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	13							

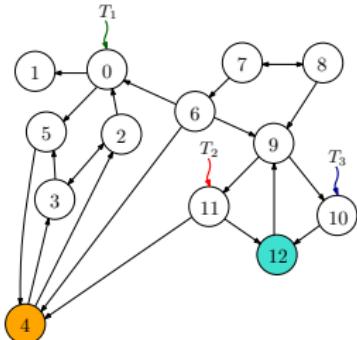
CSR representation

## Assumptions

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

```
1 Graph G(V,E) = read_input();
2 v.dist = infinity ∀ 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] + euv.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            }
17        }
18    }
19 } 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] + euv.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.

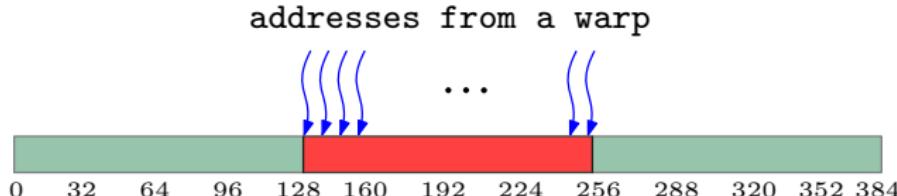
### 3 Reducing Thread Divergence

- normalize degrees across nodes assigned to a warp.

# 1 Improving Memory Coalescing

## About Memory Coalescing

- 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.

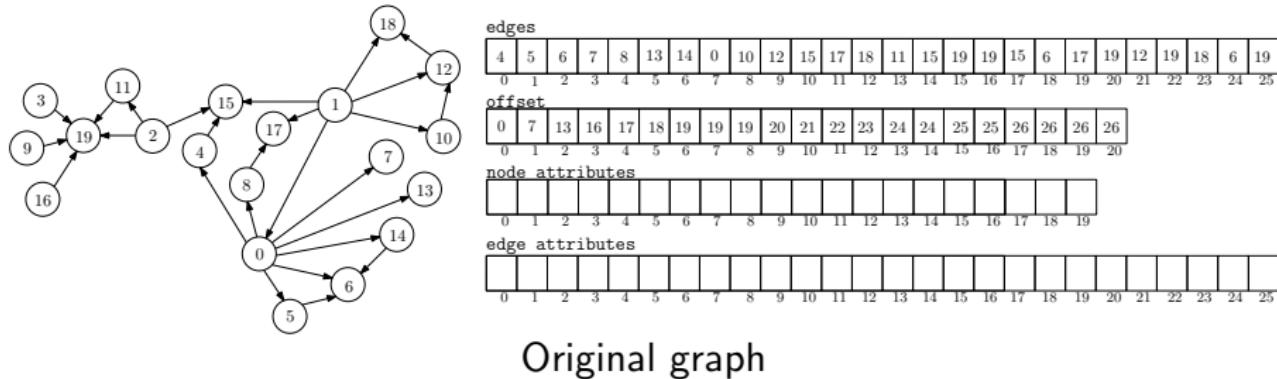
# Improving Memory Coalescing

## 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.

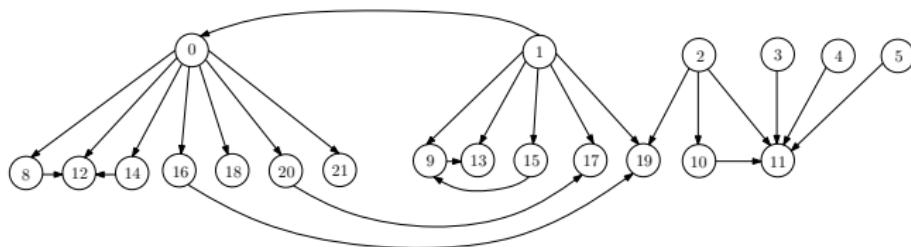


# Improving Memory Coalescing

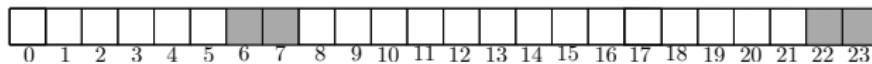
## Approach

- Start a level at a multiple of  $k$  |  $1 \leq k \leq \text{warp-size} \rightarrow \text{creates holes.}$
- 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

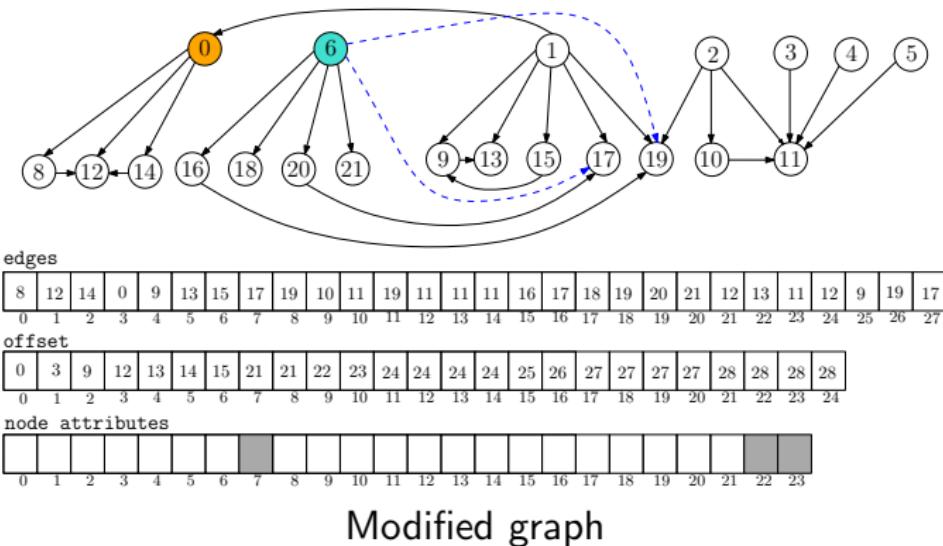
- A node occurs exactly once, so it cannot be nearby all its neighbors even after the renumbering.
- Replication brings such a node *close* to its otherwise *far* neighbors.

## Approach

- If a node is well-connected to a chunk, replicate the node in a chunk in the previous BFS level.  
$$\text{connectedness}_{\text{chunk}}^{\text{node}} \triangleq \left( \frac{\# \text{ edges to chunk from a node}}{\# \text{ non-hole nodes in chunk}} \right) \geq \text{threshold}$$
- 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
- connectedness $_{16..23}^0 = \frac{4}{6} = 0.67 \geq 0.5$  (*threshold*)
- Node 0 is replicated in the chunk 0..7; its replica is assigned id 6.



## 2 Reducing Memory Latency using Shared Memory

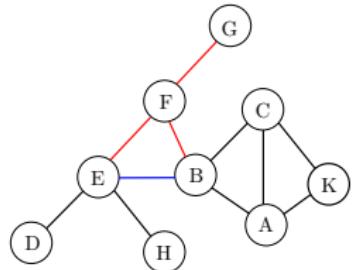
### About Shared Memory

- 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 :

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



$$\begin{aligned}\# \text{ of pairs of } F\text{'s neighbors that are neighbors} &= 1 \\ \# \text{ of pairs of } F\text{'s neighbors} &= \binom{3}{2} = 3 \\ \text{LCC}_F &= \frac{1}{3}\end{aligned}$$

# Reducing Memory Latency using Shared Memory

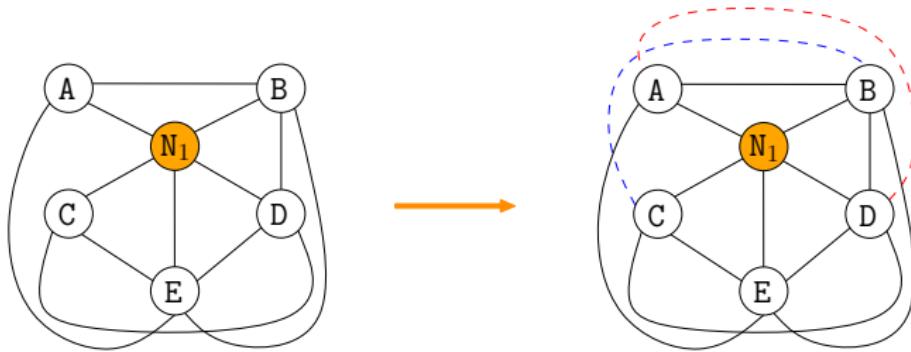
Vertices with *local clustering coefficient* (LCC)  $\geq$  threshold are more frequently accessed in iterative processing.

# Reducing Memory Latency using Shared Memory

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

## Approach

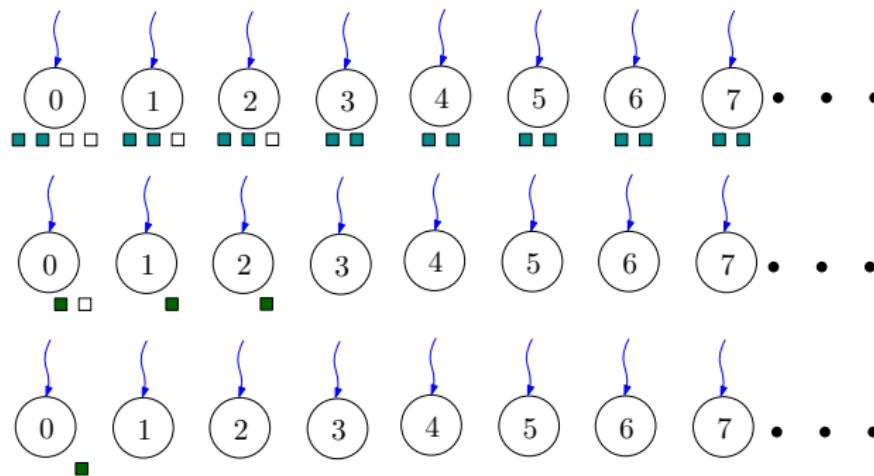
- Increase LCC of node if  $\text{LCC} \leq \text{threshold}$  and  $\text{LCC} \sim \text{threshold}$ .
- Boost LCC of node if  $\text{LCC} \geq \text{threshold}$ .
- Cap on the total number of additional edges added in the graph.



### 3 Reducing Thread Divergence

#### About Thread Divergence

- 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

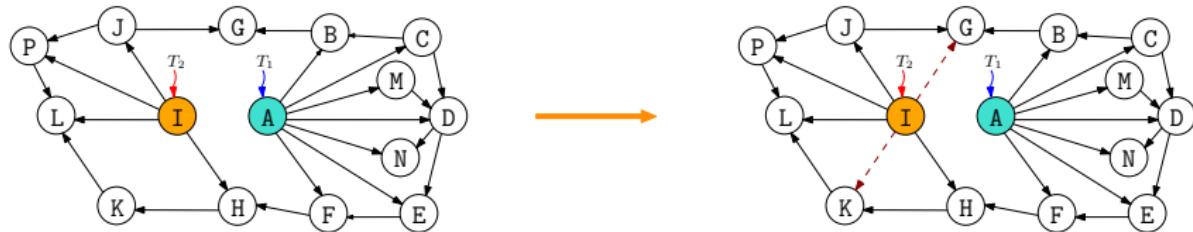
Make node degrees *nearly* uniform within each warp.

# 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  $\alpha\%$  of max. degree (e.g., 85%);  $\alpha$  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](#)<sup>†</sup>.

---

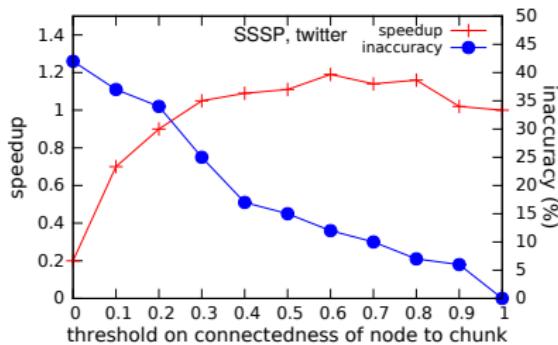
\*<https://github.com/AutomataLab/Tigr>

†<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.)

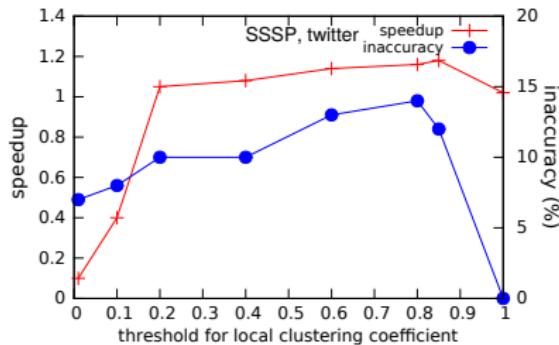
## Takeaway:

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.

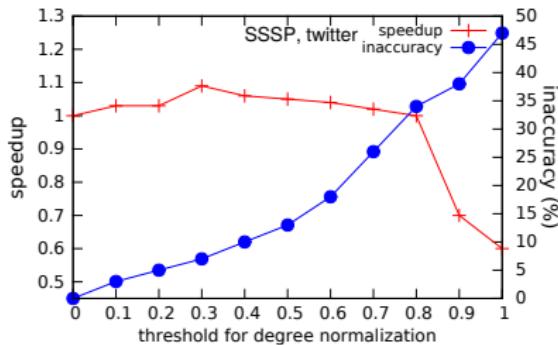
## Takeaway:

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:

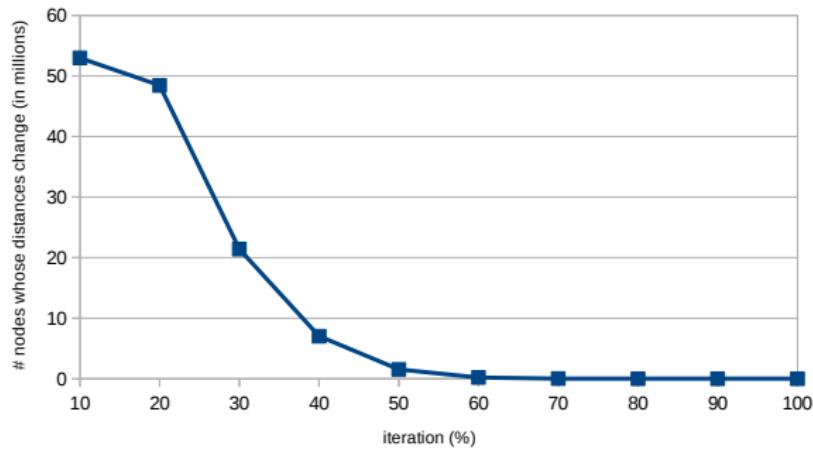
Small speedup with low inaccuracy can be achieved using a low threshold for degree normalization.

# 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

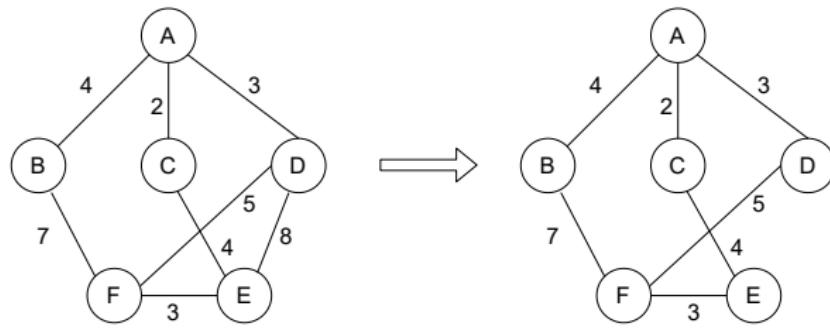
- Cut-short the number of outerloop iterations based on online stopping criteria.
- Helpful when majority of work gets done in the initial iterations.



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.



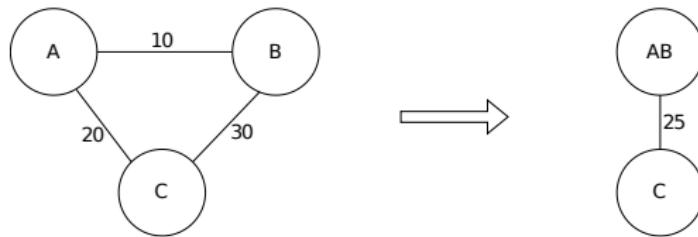
For SSSP computation

# 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:



- The compressed graph is fed as input to the exact parallel implementation of the algorithm.

# 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
- Parallel computing
- High-performance graph analytics

## Problems of Interest

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

# Research Interests

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

## Problems of Interest

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

**Thank You**

# Backup Slides

# Results

## Graffix

SSSP	Graphs	Speedup	Inaccuracy	Graphs	Speedup	Inaccuracy	Graphs	Speedup	Inaccuracy
	rmat26	1.22 ×	12%	rmat26	1.26 ×	12%	rmat26	1.06 ×	8%
MST	random26	1.13 ×	10%	random26	1.08 ×	17%	random26	1.03 ×	9%
	LiveJournal	1.18 ×	11%	LiveJournal	1.22 ×	13%	LiveJournal	1.07 ×	8%
SCC	USA-road	1.15 ×	9%	USA-road	1.30 ×	13%	USA-road	1.12 ×	7%
	twitter	1.17 ×	12%	twitter	1.18 ×	12%	twitter	1.09 ×	6%
PR	rmat26	1.18 ×	13%	rmat26	1.22 ×	16%	rmat26	1.05 ×	10%
	random26	1.13 ×	15%	random26	1.10 ×	18%	random26	1.02 ×	11%
BC	LiveJournal	1.14 ×	12%	LiveJournal	1.18 ×	16%	LiveJournal	1.07 ×	8%
	USA-road	1.23 ×	11%	USA-road	1.20 ×	19%	USA-road	1.09 ×	10%
BC	twitter	1.17 ×	13%	twitter	1.16 ×	15%	twitter	1.05 ×	9%
	rmat26	1.14 ×	8%	rmat26	1.20 ×	12%	rmat26	1.04 ×	9%
SCC	random26	1.08 ×	14%	random26	1.10 ×	16%	random26	1.00 ×	7%
	LiveJournal	1.13 ×	7%	LiveJournal	1.22 ×	13%	LiveJournal	1.04 ×	6%
PR	USA-road	1.16 ×	11%	USA-road	1.20 ×	12%	USA-road	1.05 ×	9%
	twitter	1.15 ×	12%	twitter	1.18 ×	13%	twitter	1.06 ×	8%
BC	rmat26	1.20 ×	5%	rmat26	1.32 ×	7%	rmat26	1.10 ×	4%
	random26	1.15 ×	7%	random26	1.16 ×	11%	random26	1.04 ×	9%
PR	LiveJournal	1.21 ×	7%	LiveJournal	1.26 ×	7%	LiveJournal	1.08 ×	5%
	USA-road	1.19 ×	6%	USA-road	1.30 ×	5%	USA-road	1.06 ×	8%
BC	twitter	1.22 ×	7%	twitter	1.22 ×	9%	twitter	1.09 ×	8%
	Mean	1.16 ×	10%	Mean	1.20 ×	13%	Mean	1.07 ×	8%

(a) Effect of memory coalescing

(b) Effect of memory latency

(c) Effect of thread divergence

# Execution time of exact version

Graph	Exact Time (sec)				
	SSSP	MST	SCC	PR	BC
rmat26	37	8996	21	12	15223
random26	29	10087	23	16	13127
LiveJournal	2	3424	7	1	1711
USA-road	152	82	12	1	2043
twitter	231	10943	37	18	21462

Table 1: Baseline-I: Execution time for exact versions

Graph	Exact Time (sec)		
	SSSP	PR	BC
rmat26	6	0.914	587
random26	4	1.180	498
LiveJournal	0.046	0.452	66
USA-road	12	0.130	38
twitter	17	3.000	827

Table 2: Baseline-II: Execution time for Tigr

Graph	Exact Time (sec)		
	SSSP	PR	BC
rmat26	19	1.070	872
random26	8	1.500	740
LiveJournal	0.142	0.530	98
USA-road	25.139	0.181	56
twitter	53	4.000	1227

Table 3: Baseline-III: Execution time for Gunrock

# Preprocessing overhead

Technique	Graph	Preprocessing overhead	
		Time (sec)	Additional space
Improving coalescing	rmat26	76	9%
	random26	59	11%
	LiveJournal	8	6%
	USA-road	304	8%
	twitter	463	7%
Reducing latency	rmat26	155	5%
	random26	107	8%
	LiveJournal	21	5%
	USA-road	348	4%
	twitter	532	7%
Reducing thread-divergence	rmat26	42	2%
	random26	46	3%
	LiveJournal	5	2%
	USA-road	38	1.5%
	twitter	157	4%

Table 4: Preprocessing overhead