Fast Minimum Spanning Tree For Large Graphs on the GPU

Vibhav Vineet, Pawan Harish, Suryakant Patidar
and P.J.Narayanan

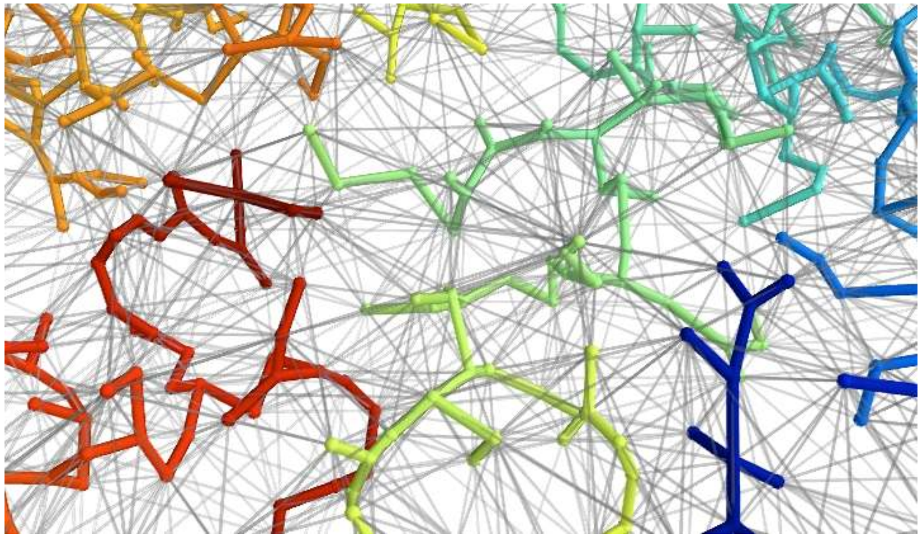
IIIT, Hyderabad

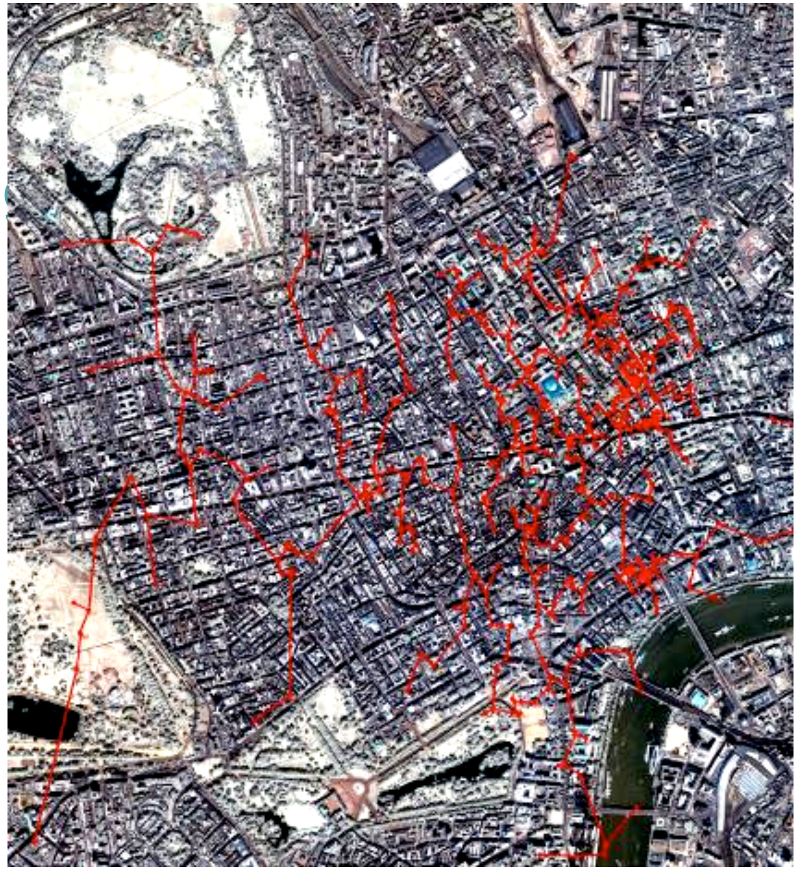
Minimum Spanning Tree

- Given a Graph G(V,W,E) find a tree whose collective weight is minimal and all vertices in the graph are covered by it
- The fastest serial solution takes O(Eα(E,V)) time
- Popular solutions include Prim's, Kruskal's and Sollin's algorithms
- Solution given by Borůvka in 1926 and later discovered by Sollins is generally used in parallel implementations

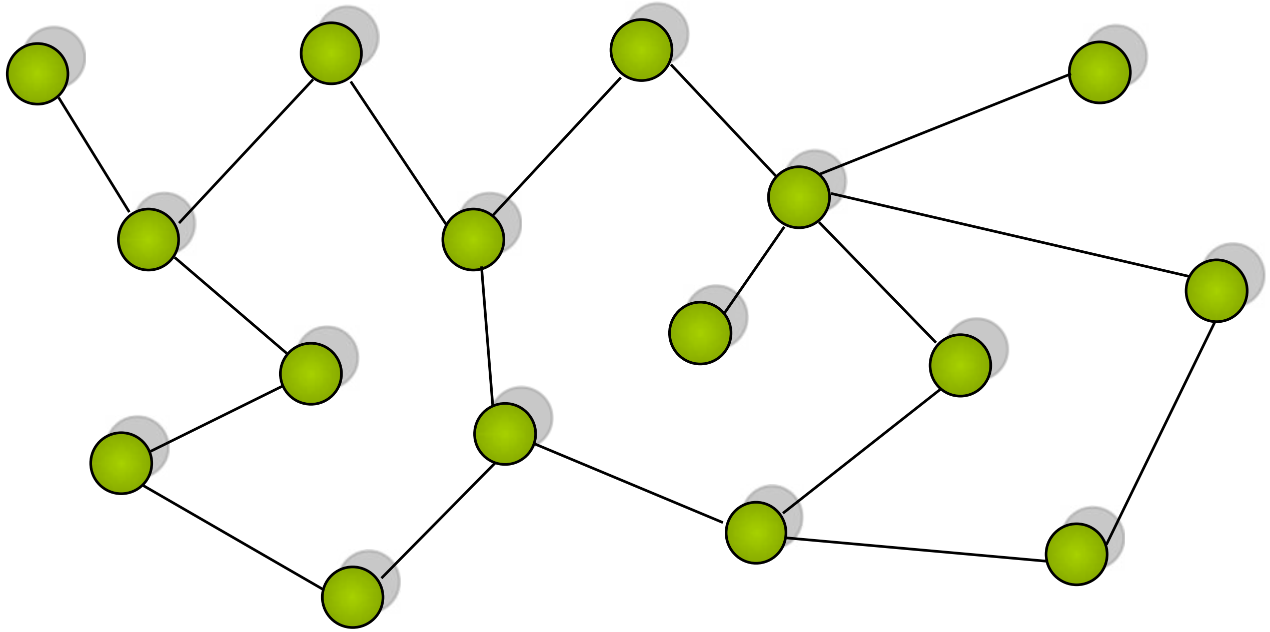
MST - Applications

- Network Design
- Route Finding
- Approximate solution to Traveling Salesman problem



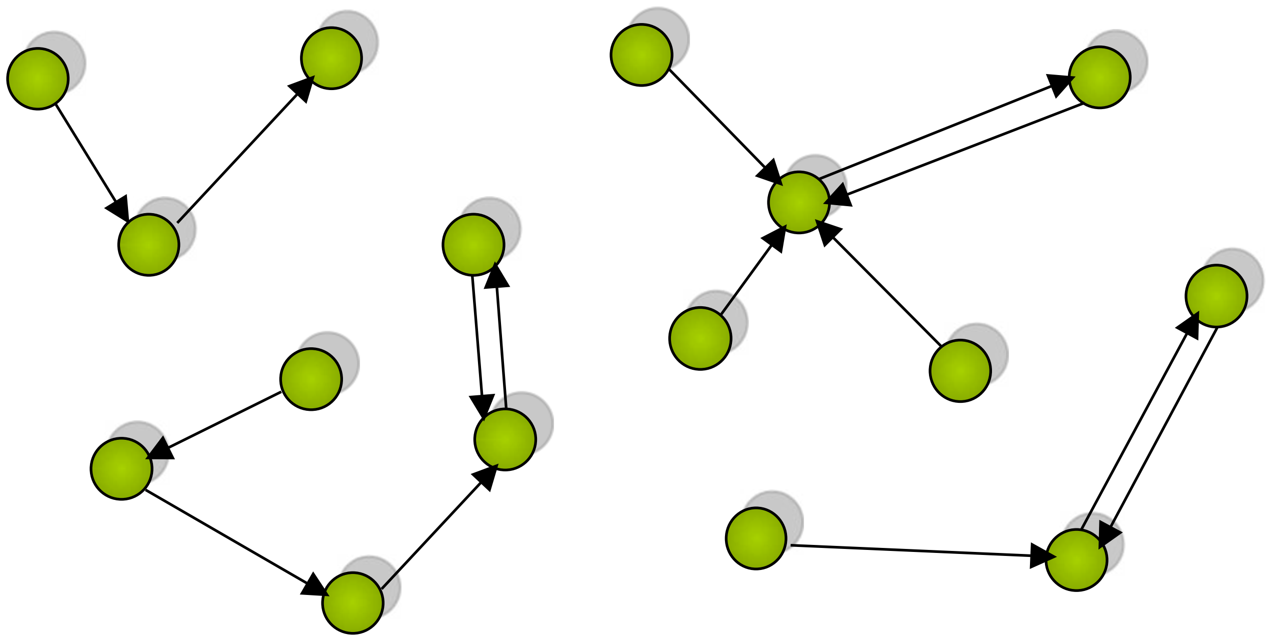


Borůvka's Solution to MST



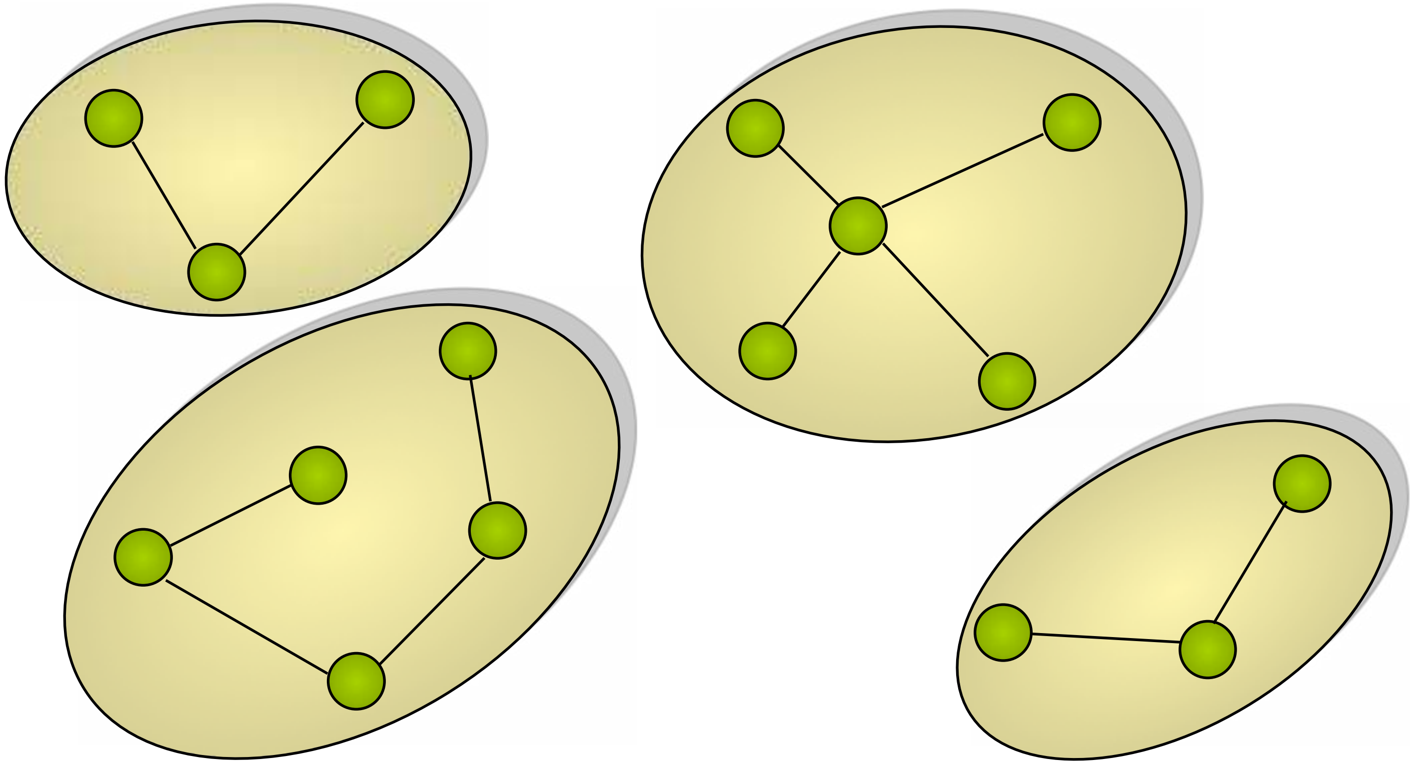
Works for undirected graphs only

Borůvka's Solution to MST



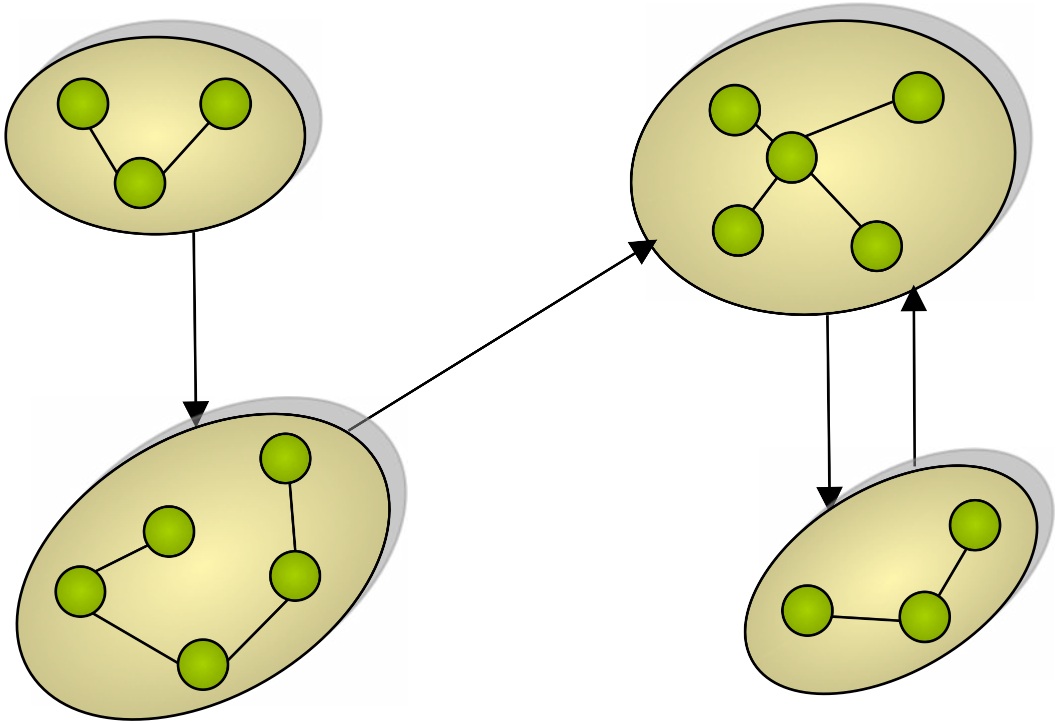
Each vertex finds the minimum weighted edge to minimum outgoing vertex. Cycles are removed explicitly

Borůvka's Solution to MST



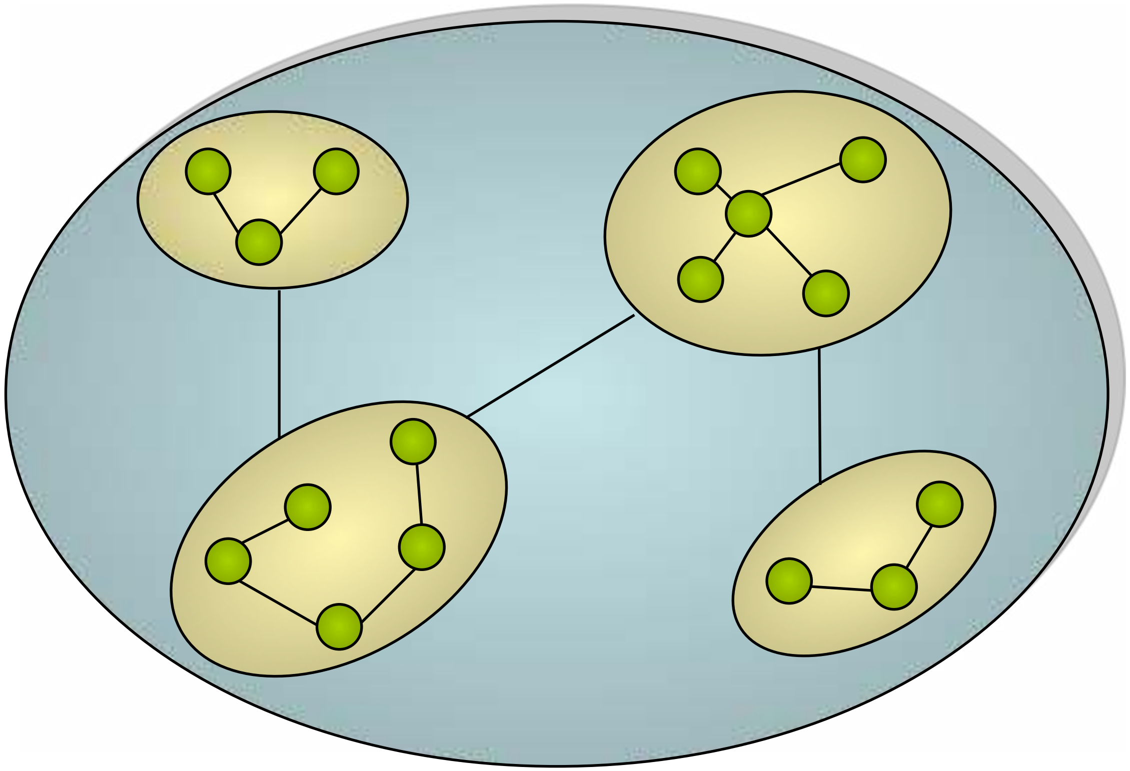
Vertices are merged together into disjoint components called Supervertices.

Borůvka's Solution to MST



Supervertices are treated as vertices for next level of recursion

Borůvka's Solution to MST



The process continues until one supervertex remains

Parallelizing Borůvka's Solution

Borůvka's approach is a greedy solution. It has two basic steps:

- **Step1:** Each vertex finds the minimum outgoing edge to another vertex. Can be seen as
  - Running a loop over edges and finding the min; writing to a common location using atomics. This is an O(V) operation.
  - Segmented min scan over |E| elements.
- **Step2:** Merger of vertices into supervertex. This can be implemented as:
  - Writing to a common location using atomics, O(V) operation.
  - Splitting on |V| elements with supervetex id as the key

Related Work

[Bader And Cong] David Bader and G. Cong. 2005. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). J. Parallel Distrib. Comput.

[Bader and Madduri] David Bader and Kamesh Madduri, 2006. GTgraph: A synthetic graph generator suite,

[Blelloch] G. E. Blelloch, 1989. Scans as Primitive Parallel Operations. IEEE Trans. Computers

[Boruvka] O. Boruvka,1926. O Jistém Problému Minimálním (About a Certain Minimal Problem) Práce Mor. Prírodoved.

[Chazelle] B. Chazelle, 2000. A minimum spanning tree algorithm with inverse-Ackermann type complexity. J. ACM

[Johnson And Metaxas] Donald Johnson and Panagiotis Metaxas. 1992. A parallel algorithm for computing minimum spanning trees. SPAA'92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures

[HVN] Pawan Harish, Vibhav Vineet and P.J. Narayanan, 2009. Large Graph Algorithms for Massively Multithreaded Architectures. Tech. Rep. IIIT/TR/2009/74.

- Our previous implementation similar to the algorithm given in [Johnson And Metaxas]

Motivation for using primitives

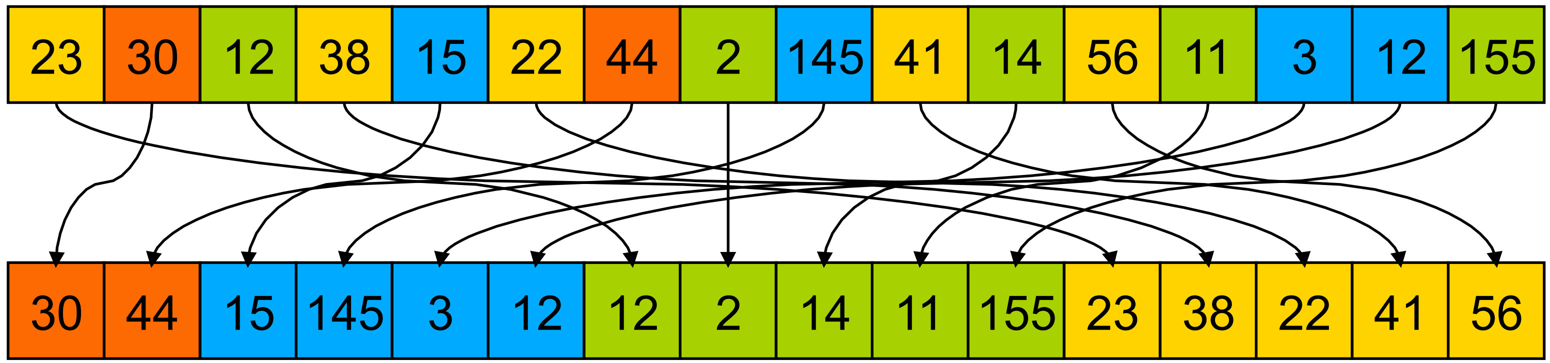
- **Primitives** are efficient
  - Non-expert programmer needs to know hardware details to code efficiently
    - Shared Memory usage, optimizations at grid.
    - Memory Coalescing, bank conflicts, load balancing
  - Primitives can port irregular steps of an algorithm to data-parallel steps transparently
- Borůvka’s approach seen as primitive operations
  - Min finding can be ported to a scan primitive
  - Merger can be seen as a split on supervertex ids.

Primitives used for MST

- Scan (CUDPP implementation):
  - Used to allot ids to supervertices after merging of vertices into a supervertex
- Segmented Scan (CUDPP implementation):
  - Used to find the minimum outgoing edge to minimum outgoing vertex for each vertex
- Split (Our implementation):
  - Used to bring together vertices belonging to same supervertex
  - Reducing the edge-list by eliminating duplicate edges

The Split Primitive

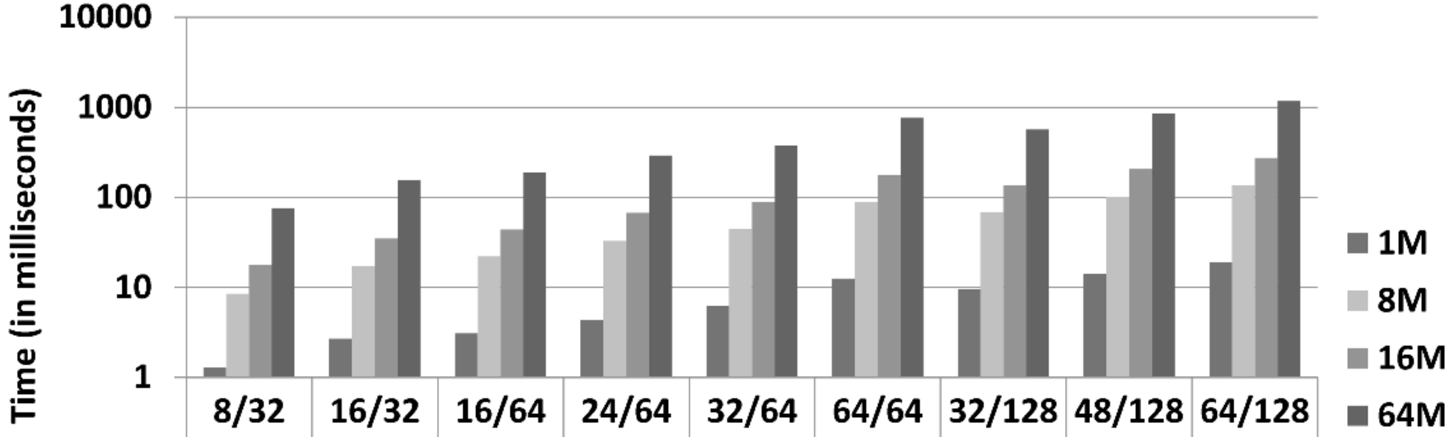
Input to Split



Output of Split

The Split primitive is used to bring together all elements in an array based on a key

The Split Primitive - Performance

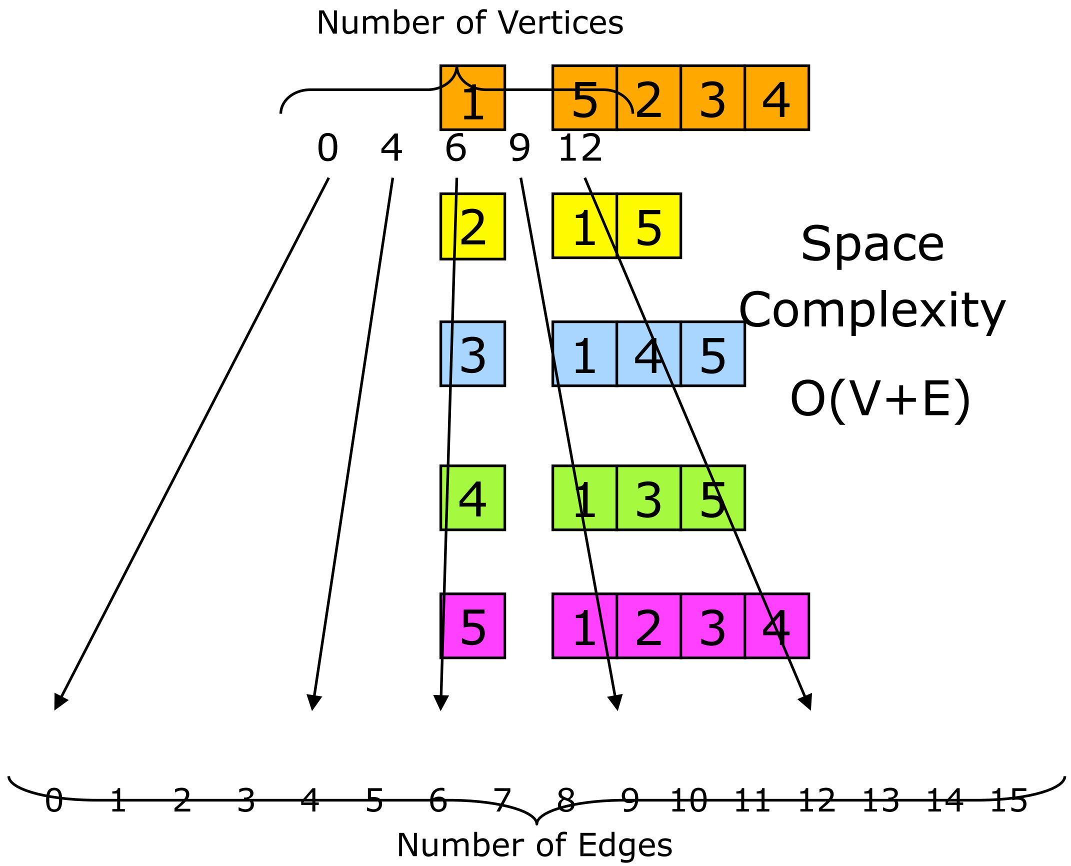


	8/32	16/32	16/64	24/64	32/64	64/64	32/128	48/128	64/128
1M	1.3	2.7	3.1	4.4	6.2	12.5	9.6	14.2	19.1
8M	9	17	22	33	44	89	69	102	136
16M	18	35	44	67	89	177	137	207	273
64M	76	155	189	291	376	767	568	858	1167

X-axis represents combinations of key-size/record size. Times on GTX 280

Code available from http://cvit.iiit.ac.in

Graph Representation



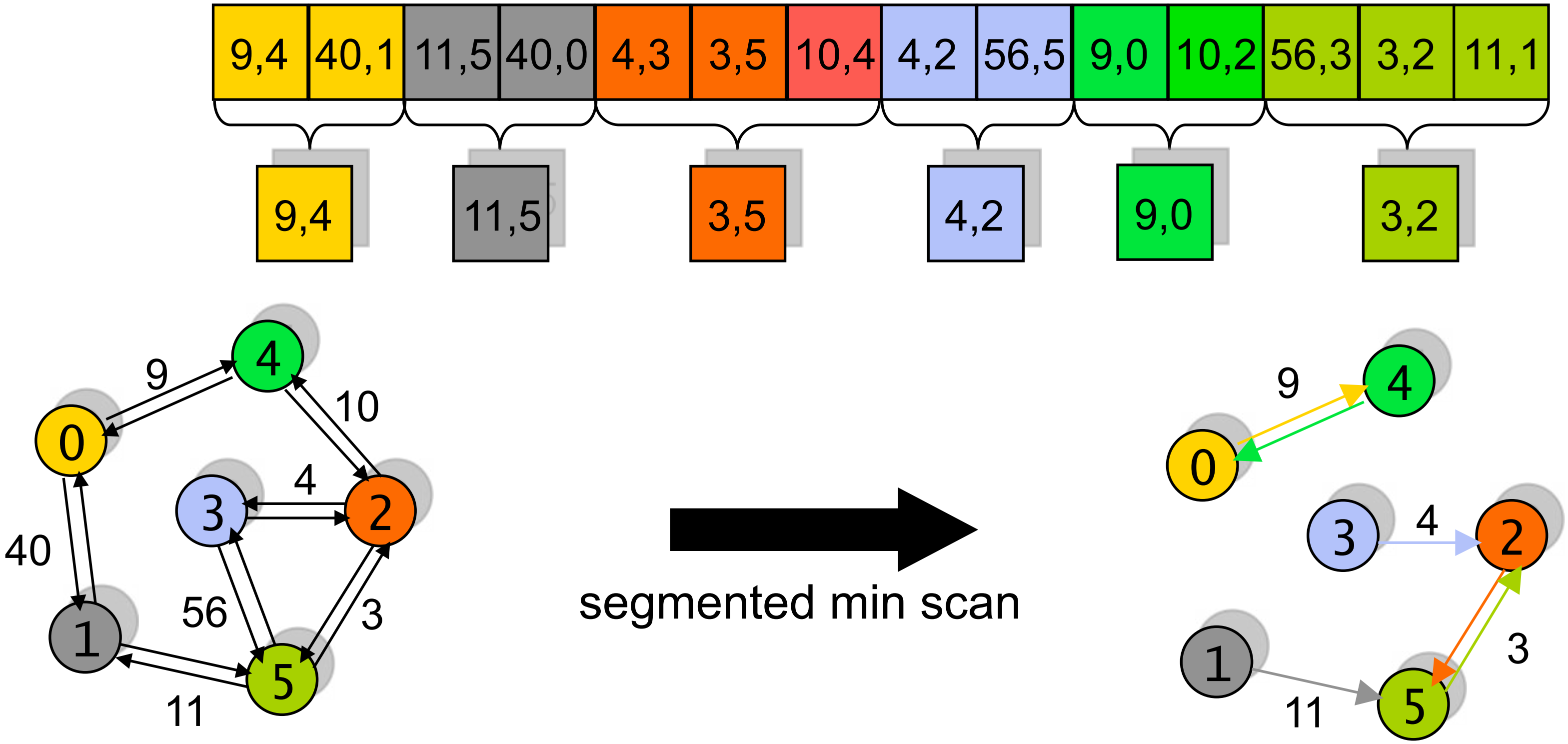
Compact edge list representation. Edges of vertex *i* following edges of vertex *i+1.* Each entry in Vertex array points to its starting of its adjacency list in the Edge list. Similar representation given in [Blelloch]

Primitive based MST - Algorithm

- **Find the minimum weighted edge to minimum outgoing vertex**
  - **Using *segmented min scan* on O(E) elements**
- Find and remove cycles by traversing successor or every vertex. Kernel of O(V)
- Select one vertex as representative for each disjoint component
- Mark the remaining edges in the output as part of MST
- Propagate representative vertex id. Using pointer doubling. Kernel of O(V)
- Merge vertices into supervertices. Using a *split* of O(V) with log V bit key size.
- Assign new ids to supervertices using a *scan* on O(V) elements
- Remove self edges per supervertex. Kernel of O(E)
- Remove duplicate edges from one supervertex to another. Split on supervertex ids along with edge weights. O(E) operation.
- Create a new vertex list from newly created edge-list. Scan of O(E)
- Recursively call again on newly created graph until one vertex remains

Finding Minimum outgoing edge

Append {w,v} for each edge per vertex and apply segmented min scan

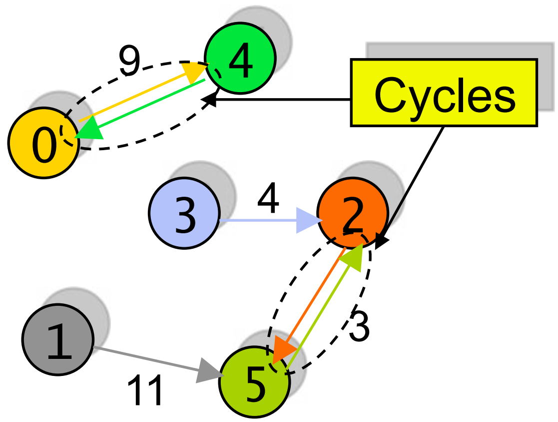


Append edge weight along with its outgoing vertex id per vertex. Apply a segmented min scan on this array to find the minimum outgoing edge to minimum outgoing vertex per vertex

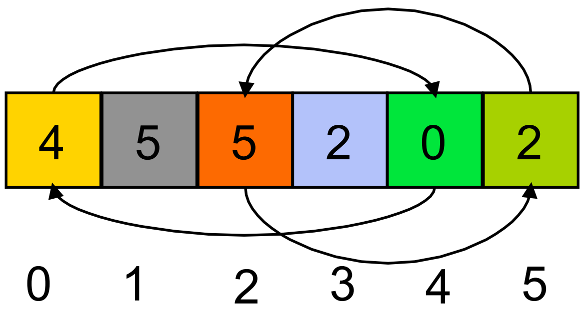
Primitive based MST - Algorithm

- Find the minimum weighted edge to minimum outgoing vertex. Using *segmented min scan* on O(E) elements
- **Find and remove cycles by traversing successor for every vertex**
  - **A Kernel of O(V)**
- **Select one vertex as representative for each disjoint component**
- **Mark the remaining edges in the output as part of MST**
- Propagate representative vertex id. Using pointer doubling. Kernel of O(V)
- Merge vertices into supervertices. Using a *split* of O(V) with log V bit key size.
- Assign new ids to supervertices using a *scan* on O(V) elements
- Remove self edges per supervertex. Kernel of O(E)
- Remove duplicate edges from one supervertex to another. Split on supervertex ids along with edge weights. O(E) operation.
- Create a new vertex list from newly created edge-list. Scan of O(E)
- Recursively call again on newly created graph until one vertex remains

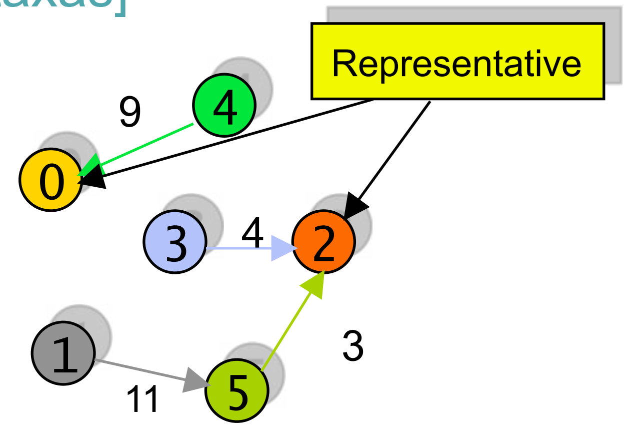
Finding and Removing Cycles



- For |V| vertices |V| edges are added, at least one cycle is expected to be formed
- It can be easily proved that cycles in an undirected case can only exist between two vertices and one per disjoint component [Johnson And Metaxas]



Remove edge of
Min(S(u),u) if
S(S(u))=u

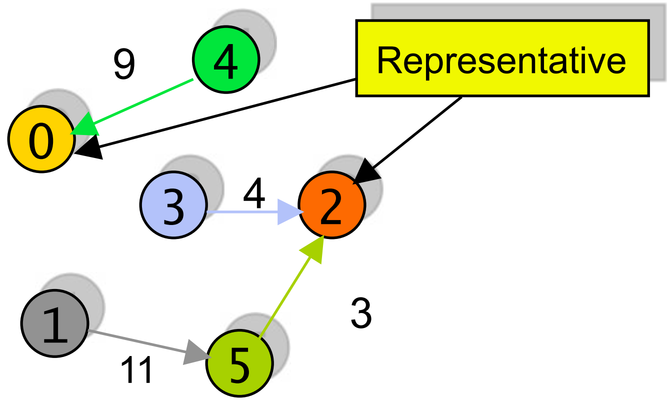


Create a successor array with each vertex's outgoing vertex id. Traverse this array if S(S(u))=u then u makes a cycle. Remove the smaller id, either u or S(u), edge from the current edge set. Mark remaining edges as part of output MST.

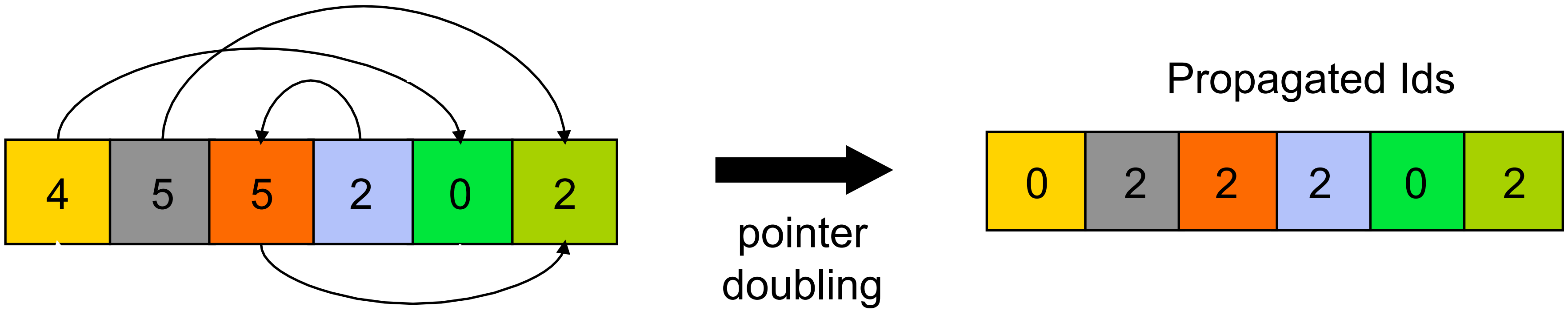
Primitive based MST - Algorithm

- Find the minimum weighted edge to minimum outgoing vertex. Using *segmented min scan* on O(E) elements
- Find and remove cycles by traversing successor or every vertex. A Kernel of O(V)
- Select one vertex as representative for each disjoint component
- Mark the remaining edges in the output as part of MST
- **Propagate representative vertex id.**
  - **Using pointer doubling. Kernel of O(V)**
- Merge vertices into supervertices. Using a *split* of O(V) with log V bit key size.
- Assign new ids to supervertices using a *scan* on O(V) elements
- Remove self edges per supervertex. Kernel of O(E)
- Remove duplicate edges from one supervertex to another. Split on supervertex ids along with edge weights. O(E) operation.
- Create a new vertex list from newly created edge-list. Scan of O(E)
- Recursively call again on newly created graph until one vertex remains

Propagating representative vertex id



The vertices whose edges are removed act as representative of each disjoint component called a supervertex

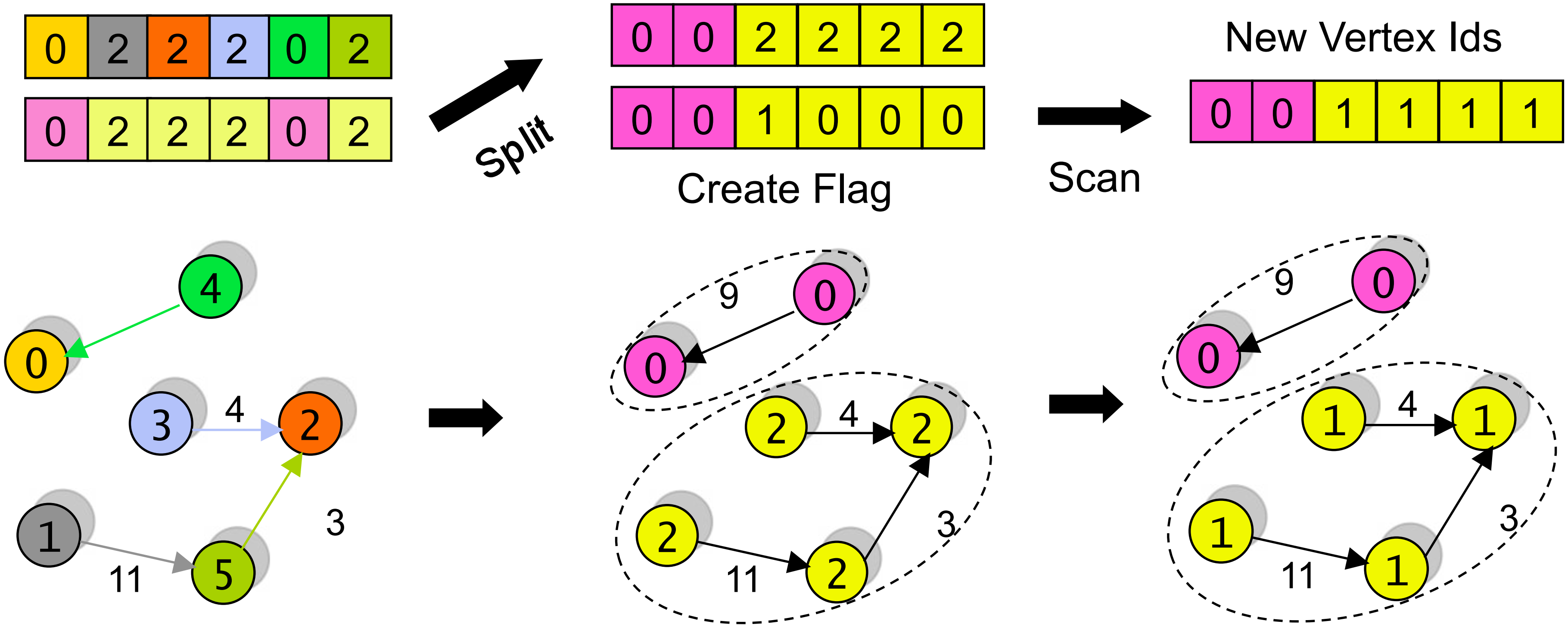


Employ pointer doubling to converge at the representative vertex in log of the longest distance from any vertex to its representative iterations.

Primitive based MST - Algorithm

- Find the minimum weighted edge to minimum outgoing vertex. Using *segmented min scan* on O(E) elements
- Find and remove cycles by traversing successor or every vertex. A Kernel of O(V)
- Select one vertex as representative for each disjoint component
- Mark the remaining edges in the output as part of MST
- Propagate representative vertex id. Using pointer doubling. Kernel of O(V)
- **Merge vertices into supervertices.**
  - **Using a *split* of O(V) with log(V) bit key size.**
- **Assign new ids to supervertices.**
  - **Using a *scan* on O(V) elements**
- Remove self edges per supervertex. Kernel of O(E)
- Remove duplicate edges from one supervertex to another. Split on supervertex ids along with edge weights. Optional O(E) operation.
- Create a new vertex list from newly created edge-list. Scan of O(E)
- Recursively call again on newly created graph until one vertex remains

Bringing vertices together

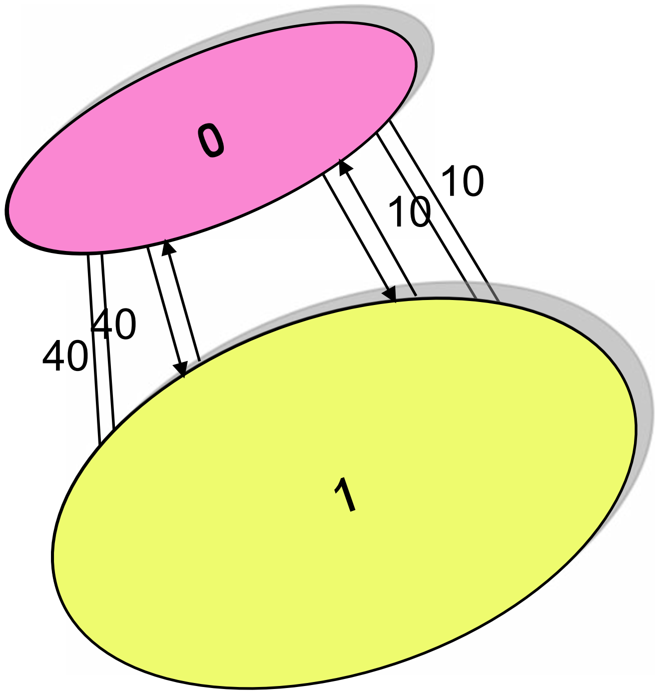


Split based on the supervertex id to bring together all vertices belonging to the same supervertex. Scan the flag to assign new ids.

Primitive based MST - Algorithm

- Find the minimum weighted edge to minimum outgoing vertex. Using *segmented min scan* on O(E) elements
- Find and remove cycles by traversing successor or every vertex. A Kernel of O(V)
- Select one vertex as representative for each disjoint component
- Mark the remaining edges in the output as part of MST
- Propagate representative vertex id. Using pointer doubling. Kernel of O(V)
- Merge vertices into supervertices. Using a *split* of O(V) with log V bit key size.
- Assign new ids to supervertices. Using a *scan* on O(V) elements
- **Remove self edges per supervertex.**
  - **Kernel of O(E)**
- **Remove duplicate edges from one supervertex to another.**
  - **Split edges on supervertex ids along with edge weights, Optional O(E) operation.**
- Create a new vertex list from newly created edge-list. Scan of O(E)
- Recursively call again on newly created graph until one vertex remains

Shortening The Edge list



Remove Edges with same vertex ids for both vertices

Append {u,v,w} for each edge

0,1,40	1,0,10	1,0,40	0,1,10

Split

0,1,10	0,1,40	1,0,10	1,0,40

Pick First Distinct {u,v} pair entry

0,1,10	1,0,10

Compact to create Edge-list and Weight-list

0,1,10	1,0,10

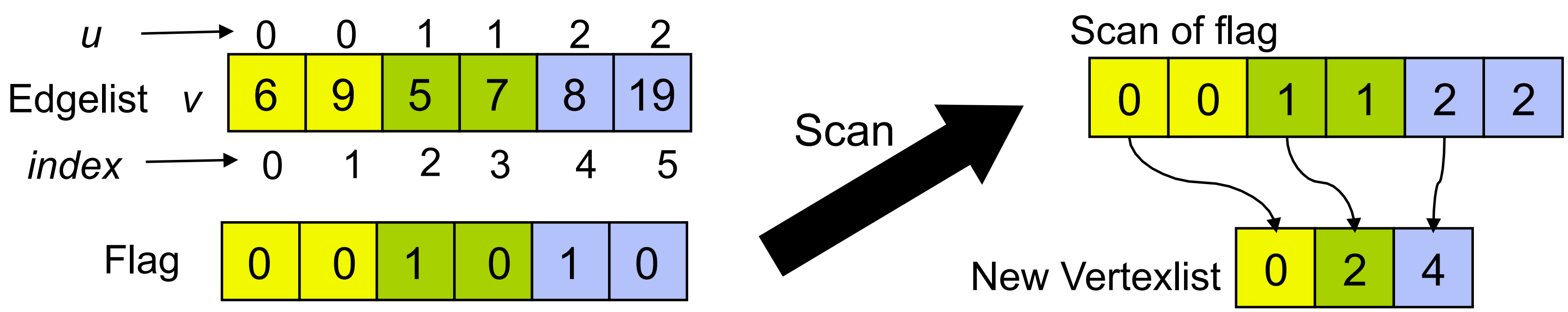
- Remove self-edges by looking at supervertex ids of both vertices
- Optionally remove duplicate edges using a 64-bit split on {u,v,w}. It is expensive O(E) operation and is done in initial iterations only.
- Pick first distinct {u,v} entry eliminating duplicated edges

Primitive based MST - Algorithm

- Find the minimum weighted edge to minimum outgoing vertex. Using *segmented min scan* on O(E) elements
- Find and remove cycles by traversing successor or every vertex. A Kernel of O(V)
- Select one vertex as representative for each disjoint component
- Mark the remaining edges in the output as part of MST
- Propagate representative vertex id. Using pointer doubling. Kernel of O(V)
- Merge vertices into supervertices. Using a *split* of O(V) with log V bit key size.
- Assign new ids to supervertices. Using a *scan* on O(V) elements
- Remove self edges per supervertex. Kernel of O(E)
- Remove duplicate edges from one supervertex to another. Split edges on supervertex ids along with edge weights. Optional O(E) operation.
- **Create a new vertex list from newly created edge-list.**
  - **Scan of O(E)**
- Recursively call again on newly created graph until one vertex remains

Creating the Vertex list

- The Vertex list contains the starting index of each vertex in the edge list.
- In order to find the starting index we scan a flag based on distinct supervertex ids in the edge-list.
- This gives us the index where each vertex should write its starting value
- Compacting the entries gives us the desired vertex list



Primitive based MST - Algorithm

- Find the minimum weighted edge to minimum outgoing vertex. Using *segmented min scan* on O(E) elements
- Find and remove cycles by traversing successor or every vertex. A Kernel of O(V)
- Select one vertex as representative for each disjoint component
- Mark the remaining edges in the output as part of MST
- Propagate representative vertex id. Using pointer doubling. Kernel of O(V)
- Merge vertices into supervertices. Using a *split* of O(V) with log V bit key size.
- Assign new ids to supervertices. Using a *scan* on O(V) elements
- Remove self edges per supervertex. Kernel of O(E)
- Remove duplicate edges from one supervertex to another. Split edges on supervertex ids along with edge weights. Optional O(E) operation.
- Create a new vertex list from newly created edge-list. Scan of O(E)
- **Recursively call again on newly created graph until one vertex remains**

Recursive invocation

Iteration number	Number of Vertices	Number of Edges	
		After removing self edges only	After removing self and duplicate edges
0	1000000	9999930	-
1	233592	8467090	**646584**
2	38002	8075560	**79802**
3	2810	7991444	**22641**
4	77	2006114	**541**
5	1	0	**0**

Total Number of Iterations: $\sqrt{\log V}$ [Johnson And Metaxas]

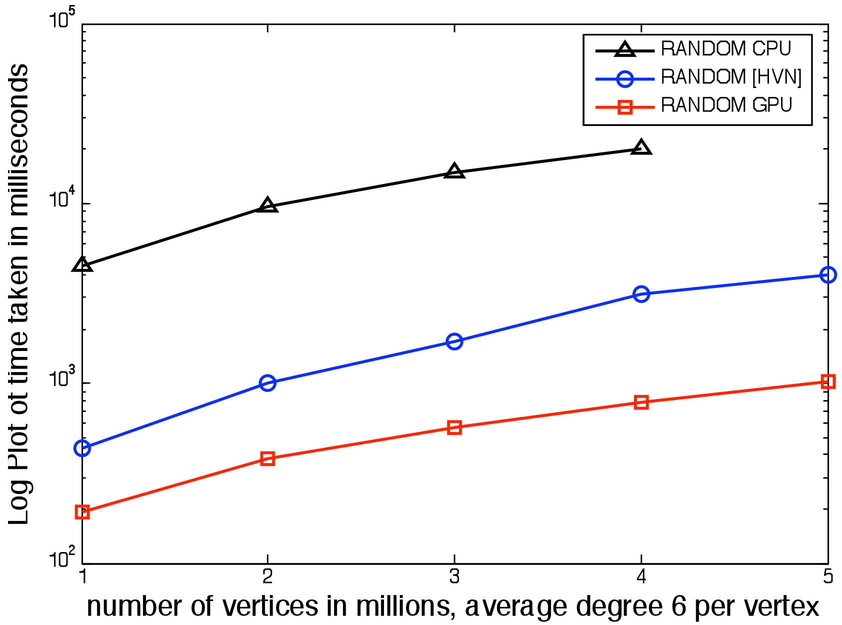
Duplicate Edge removal is optional

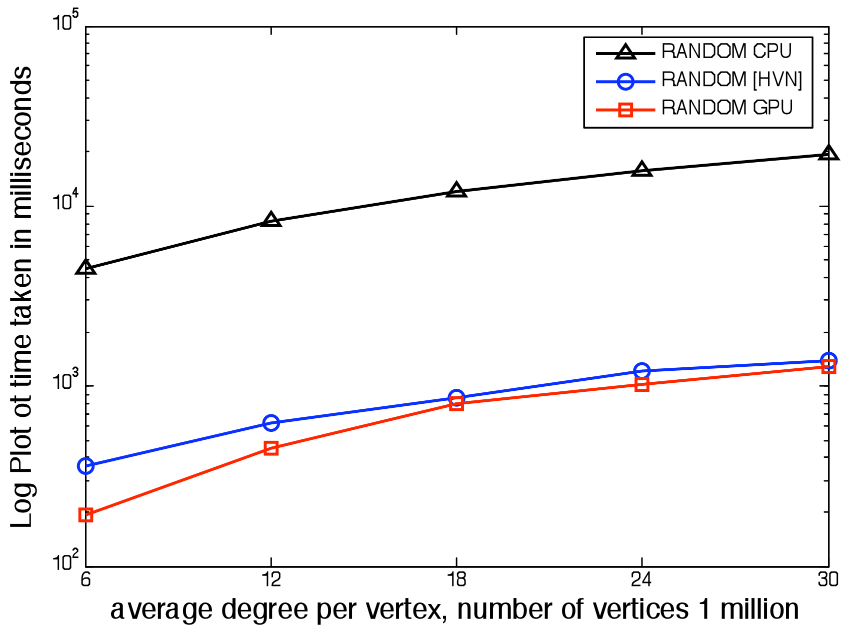
- A full 64-bit split {u,v,w} is an expensive operation
- Segmented scan compensates for this in later iterations

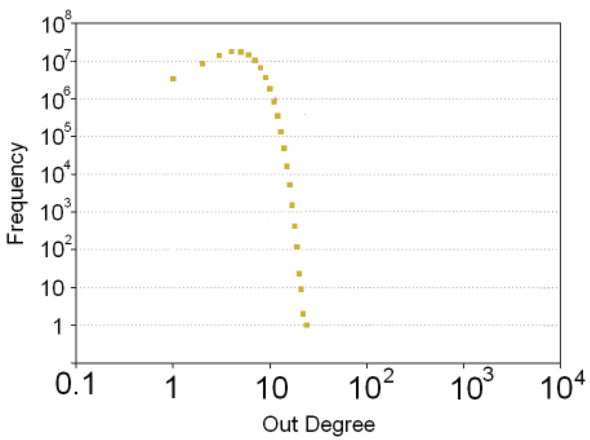
Experimental Setup

- Hardware Used:
  - Nvidia Tesla S1070: 240 stream processors with 4GB of device memory
- Comparison with
  - Boost C++ Graph Library on Intel Core 2 Quad, Q6600, 2.4GHz
  - Previous GPU implementation from our group on Tesla S1070 [HVN]
- Graphs used for experiments
  - GT Graph Generator [Bader and Madduri]
    - **Random:** These graphs have a short band of degree where all vertices lie, with a large number of vertices having similar degrees.
    - **RMAT:** Large number of vertices have small degree with a few vertices having large degree. This model is best suited to large represent real world graphs.
    - **SSCA#2:** These graphs are made of random sized cliques of vertices with a hierarchical distribution of edges between cliques based on a distance metric.
  - DIMACS ninth shortest path challenge

Results – Random Graphs

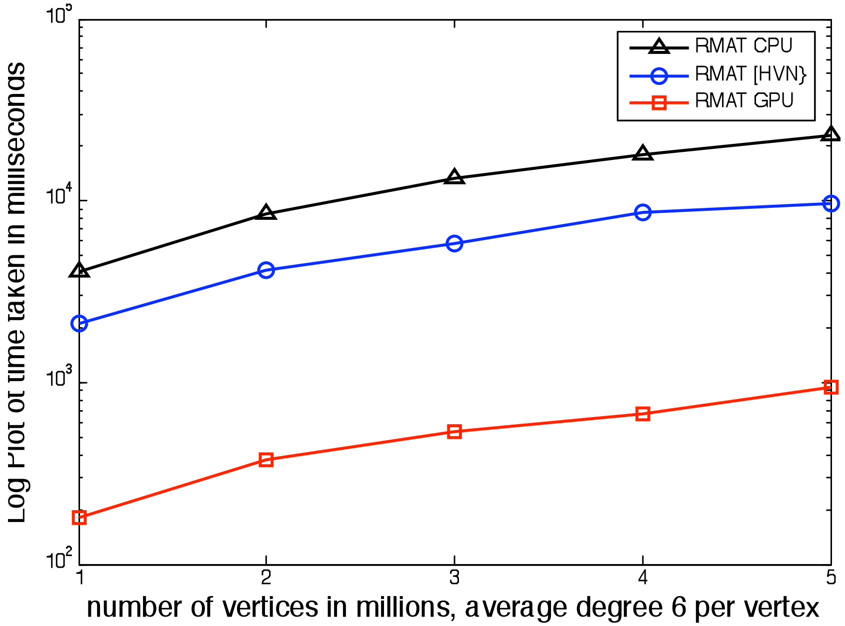


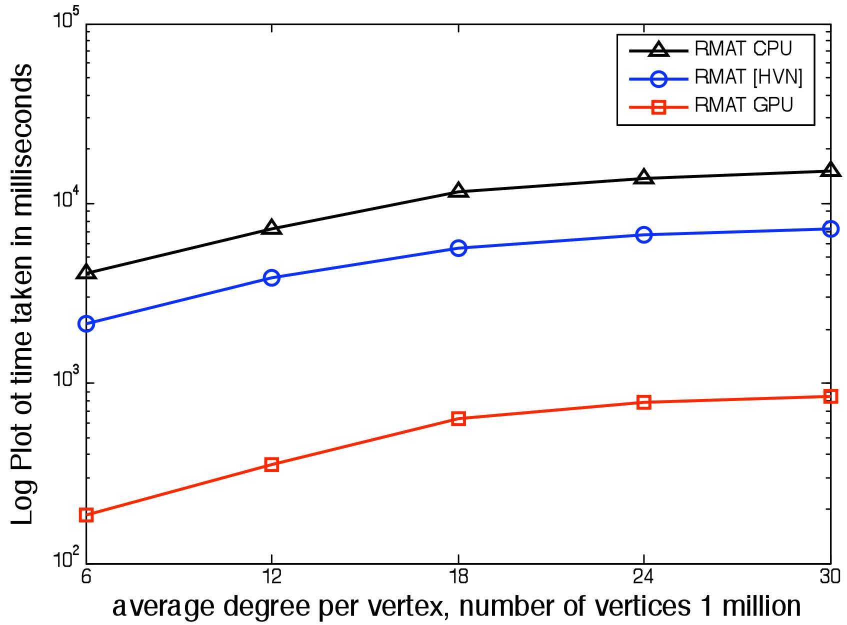


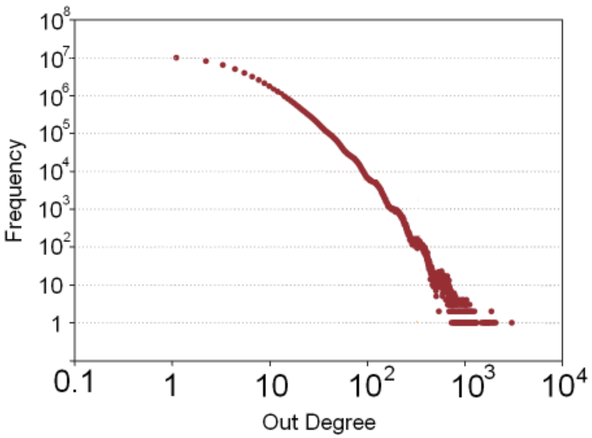


- A speed up of 20-30 over CPU and 3-4 over our previous GPU implementation.
- 5M vertices, 30M edges under 1 sec
- O(E) scans Vs O(V) threads writing atomically
- Actual number of atomic clashes are limited by an upper bound based on the warp size

Results – RMAT graphs

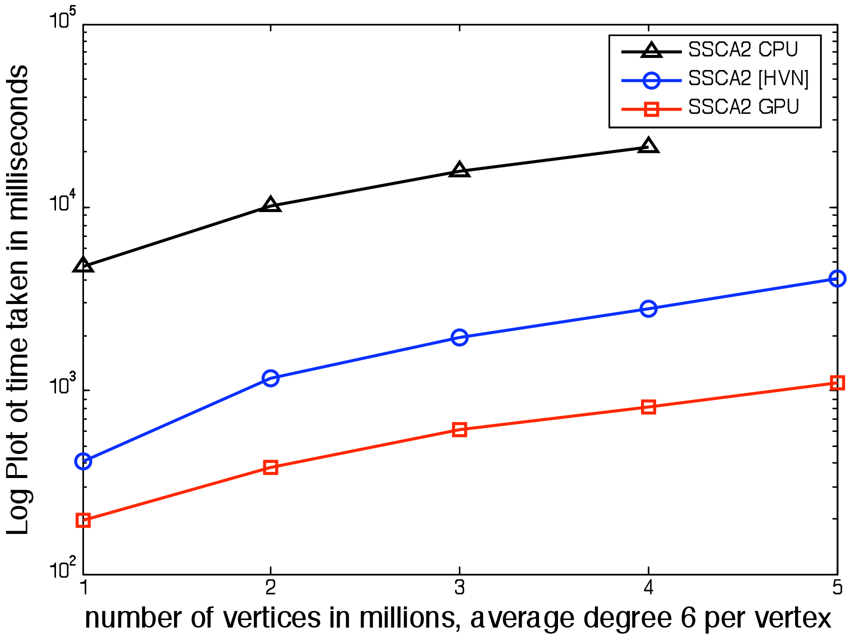


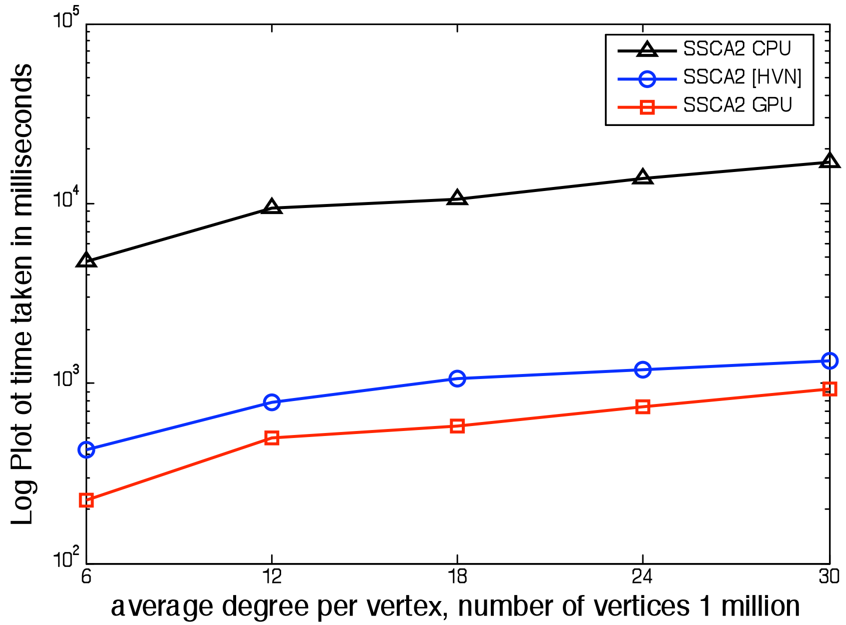


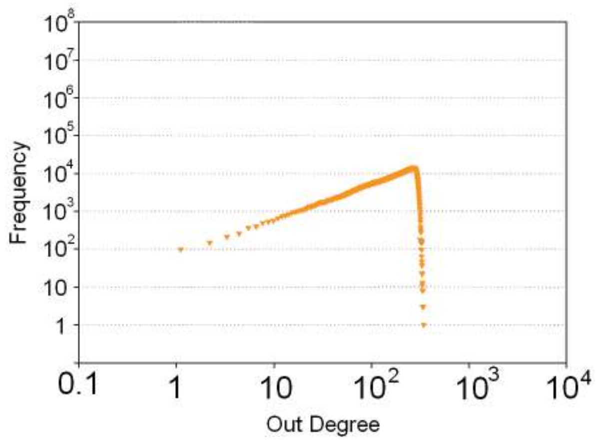


- A speed up of 40-50 over CPU and 8-10 over our previous GPU implementation.
- 5M vertices, 30M edges under 1 sec
- High load imbalance due to large variation in degrees for loop based approach.
- Primitive based approach performs better

Results – SSCA2 graphs



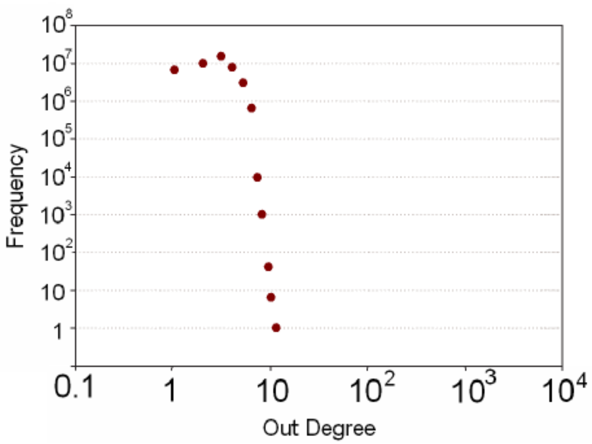




- A speed up of 20-30 over CPU and 3-4 over our previous GPU implementation.
- 5M vertices, 30M edges under 1 sec

Results – DIMACS Challenge

- A speed up of 20 over CPU and 2-3 over our previous GPU implementation.
- Linear nature (maximum frequency for degrees 2-3) of these graphs lead to smaller loops in our previous approach [HVN]



Name	Vertices	Edges	Time in Milliseconds		
			CPU	[HVN]	Ours
NY	264K	733K	780	76	**39**
San F	321K	800K	870	85	**57**
Colorado	436K	1M	1280	116	**62**
Florida	1.07M	2.7M	3840	261	**101**
NW-USA	1.2M	2.8M	4290	299	**124**
NE-USA	1.5M	3.9M	6050	383	**126**
California	1.9M	4.6M	7750	435	**148**
Great L	2.7M	6.9M	12300	671	**204**
USA-E	3.5M	8.8M	16280	1222	**253**
USA-W	6.2M	15.2M	32050	1178	**412**

Conclusion and Future Work

- Irregular steps can be mapped to data-parallel primitives efficiently of generic irregular algorithms
- Recursion works well as controlled via CPU
- We are likely to see many graph algorithms being ported to GPUs using such primitives as it has the potential to regularize irregular problems as common to graph theory

Thank you!

Questions?