Tree contraction, connected components, minimum spanning trees: a GPU path to vertex fitting

Raul H. C. Lopes
Ivan D. Reid
Peter R. Hobson

Particle Physics Group, School of Engineering and Design, Brunel University

September 9, 2014

Table of Contents

1. Motivation
2. Proposed solutions
3. Assumed
4. Minimal Spanning Trees
5. Algorithms
6. Parallel Kruskal
7. Well separated concerns (or ball decomposition)
8. Disclaimer

PBBS and Graph500

- PBBS: The Problem Base Benchmark Suite
  - Benchmark designed to compare
    - parallel algorithmic approaches;
    - parallel programming languages;
    - machine architectures;
  - Graph500
    - a benchmark to use graph problems to compare computer architectures and compiler design.
  - CMS (LHC experiment)
    - Minimum Spanning Tress computations are reported as part of *ZVMVST* in vertex finding.

Theoretical limit

- Connected components and Minimal spanning tree computations can be demanding.
  - Lower bound on work for computation of a Minimum Spanning Tree over an m edges graph is $O(m)$.
  - Lower bound on space also $O(m)$.
  - Euclidean Graph with 2^{21} vertices might demand $16TB$ of RAM just to represent the edges.

Theoretical limit

- Connected components and Minimal spanning tree computations can be demanding.
  - Lower bound on work for computation of a Minimum Spanning Tree over an m edges graph is $O(m)$.
  - Lower bound on space also $O(m)$.
  - Euclidean Graph with 2^{21} vertices might demand $16TB$ of RAM just to represent the edges.
- Challenging as a Computer Science problem.

Theoretical limit

- Connected components and Minimal spanning tree computations can be demanding.
  - Lower bound on work for computation of a Minimum Spanning Tree over an m edges graph is $O(m)$.
  - Lower bound on space also $O(m)$.
  - Euclidean Graph with 2^{21} vertices might demand $16TB$ of RAM just to represent the edges.
- Challenging as a Computer Science problem.
- Could it have applications in CMS or any other LHC project or even HEP in general? Do they work with large dense graphs?

Table of Contents

1. Motivation
2. Proposed solutions
3. Assumed
4. Minimal Spanning Trees
5. Algorithms
6. Parallel Kruskal
7. Well separated concerns (or ball decomposition)
8. Disclaimer

Kruskal plus lock-free transactions

- Parallel algorithm based on Kruskal MST algorithm with:
  - Lock-free transactions for edge selection based on database style distriburted reserve and commit of vertices.
  - Parallel tree contraction as form of disjoint find-union set implementation.

Parallel approximation algorithm

- Parallel approximation algorithm based on fair split that will lead to *Well Separated Pair Decomposition*.
    - Parallel tree implementation for vertices indexing that doesn't demand explicit edge representation.
    - Approximation algorithm that can lead to MST computation with $O(n \lg n)$ work, given n vertices.

Table of Contents

1. Motivation
2. Proposed solutions
3. Assumed
4. Minimal Spanning Trees
5. Algorithms
6. Parallel Kruskal
7. Well separated concerns (or ball decomposition)
8. Disclaimer

Graphs formally

- A graph G is a pair (V, E) where
  - V is a set of n vertices.
  - $E \subseteq \{\{v_i, v_j\} : v_i, v_j \in V\}$ is a set of m edges.
- a map w assigns weight to edges.

Metric for Edge weight

- Some sort of metric defines the weight of edges.
- Graph can possibly be Eucllidean.
    - w is the Euclidean distance between the respective vertices.
    - graph is possibly complete: an edge between each pair of vertices.

Table of Contents

1. Motivation
2. Proposed solutions
3. Assumed
4. Minimal Spanning Trees
5. Algorithms
6. Parallel Kruskal
7. Well separated concerns (or ball decomposition)
8. Disclaimer

Minimal spanning tree

- Spanning tree of a graph G

  a subgraph G that:
  - connects all vertices of G;
  - contains exactly one path connecting each pair of vertices.

Minimal spanning tree

- Spanning tree of a graph G

  a subgraph G that:
  - connects all vertices of G;
  - contains exactly one path connecting each pair of vertices.
- Minimum spanning tree of $G, mst(G)$:

  A spanning tree whose sum edge weights is minimal when compared to all spanning trees of G.

Minimal spanning tree

- Spanning tree of a graph G

  a subgraph G that:
  - connects all vertices of G;
  - contains exactly one path connecting each pair of vertices.
- Minimum spanning tree of $G, mst(G)$:

  A spanning tree whose sum edge weights is minimal when compared to all spanning trees of G.
- Minimum spanning forest of G : $msf(G)$

  A union of minimum spanning trees of each component of G.

Minimal spanning tree

- Spanning tree of a graph G

  a subgraph G that:
  - connects all vertices of G;
  - contains exactly one path connecting each pair of vertices.
- Minimum spanning tree of $G, mst(G)$:

  A spanning tree whose sum edge weights is minimal when compared to all spanning trees of G.
- Minimum spanning forest of G : $msf(G)$

  A union of minimum spanning trees of each component of G.

The challenge

- Benchmarks and all that
  - No parallel algorithm for **mst** computation in PBBS
  - 1 Parallel algorithm approximates **mst** through **msf**.
  - Efficiency of parallel **msf** algorithm is less than 30%.
  - No GPU algortihm for parallel **mst** or **msf** computation, even if a few algorithms for conncected components can be found.

The challenge

- Benchmarks and all that
  - No parallel algorithm for **mst** computation in PBBS
  - 1 Parallel algorithm approximates **mst** through **msf**.
  - Efficiency of parallel **msf** algorithm is less than 30%.
  - No GPU algortihm for parallel **mst** or **msf** computation, even if a few algorithms for conncected components can be found.
- Lower bounds do matter
  - $\omega(m)$: General **mst** in 3 or more dimensions cannot be computed in less than m comparisons, possibly close to n^2.
  - $\omega(m)$ again: 1 to $4TB$ for a dense graph with 1 million vertices.

The challenge

- Benchmarks and all that
  - No parallel algorithm for **mst** computation in PBBS
  - 1 Parallel algorithm approximates **mst** through **msf**.
  - Efficiency of parallel **msf** algorithm is less than 30%.
  - No GPU algortihm for parallel **mst** or **msf** computation, even if a few algorithms for conncected components can be found.
- Lower bounds do matter
  - $\omega(m)$: General **mst** in 3 or more dimensions cannot be computed in less than m comparisons, possibly close to n^2.
  - $\omega(m)$ again: 1 to $4TB$ for a dense graph with 1 million vertices.
- Why bother?
  - Kruskal algorithm can build an **mst** for a dense graph with $1K$ vertices in less than a second.
  - 2 days when number vertices gets to $1M$.

Table of Contents

1 Motivation

2 Proposed solutions

3 Assumed

4 Minimal Spanning Trees

5 Algorithms

6 Parallel Kruskal

7 Well separated concerns (or ball decomposition)

8 Disclaimer

A Lemma and 700 algorithms

- Lemma: Let X be any subset of vertices of G, and let e be lightest edge connecting X to $G - X$. Then e is part of the **mst** of G.

A Lemma and 700 algorithms

- Lemma: Let X be any subset of vertices of G, and let e be lightest edge connecting X to $G - X$. Then e is part of the **mst** of G.
- Kruskal: add lightest edge first.

A Lemma and 700 algorithms

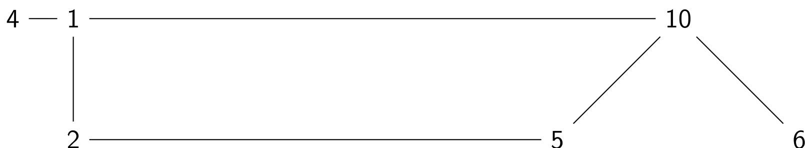
- Lemma: Let X be any subset of vertices of G, and let e be lightest edge connecting X to $G - X$. Then e is part of the **mst** of G.
- Kruskal: add lightest edge first.
- Prim-Dijkstra: start with a one vertex tree and add the lightest edge connecting the tree to a vertex not in it.

A Lemma and 700 algorithms

- Lemma: Let X be any subset of vertices of G, and let e be lightest edge connecting X to $G - X$. Then e is part of the **mst** of G.
- Kruskal: add lightest edge first.
- Prim-Dijkstra: start with a one vertex tree and add the lightest edge connecting the tree to a vertex not in it.
- Borůvka: start with n trees and join pairs with lightest edges. *(Extremely parallel, but...)*
- Separated blobs: subgraphs can be seen as blobs, worked on separately and in parallel and...

A graph

- A graph



- Edges in increasing order of weight:

  $(4, 1)$

  $(2, 1)$

  $(10, 5)$

  $(10, 6)$

  $(2, 5)$

  $(1, 10)$

Kruskal order

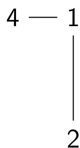
- Kruskal: loop invariant based on **mst** Lemma.
  start with empty t
  for each edge e in increasing order of weight
  if vertices of e are disconnected in t
  add e to t

Kruskal order

- Kruskal: loop invariant based on **mst** Lemma.
  start with empty t
  for each edge e in increasing order of weight
  if vertices of e are disconnected in t
  add e to t
- The tree
  4 — 1 10
  2 5 6

Kruskal order

- Kruskal: loop invariant based on **mst** Lemma.
  start with empty t
  for each edge e in increasing order of weight
  if vertices of e are disconnected in t
  add e to t
- The tree



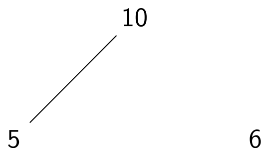
10

5

6

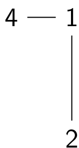
Kruskal order

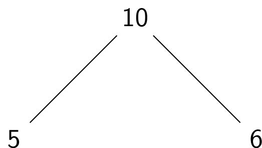
- Kruskal: loop invariant based on **mst** Lemma.
  start with empty t
  for each edge e in increasing order of weight
  if vertices of e are disconnected in t
  add e to t
- The tree



Kruskal order

- Kruskal: loop invariant based on **mst** Lemma.
  start with empty t
  for each edge e in increasing order of weight
  if vertices of e are disconnected in t
  add e to t
- The tree





Kruskal order

- Kruskal: loop invariant based on **mst** Lemma.
  start with empty t
  for each edge e in increasing order of weight
  if vertices of e are disconnected in t
  add e to t
- The tree

Table of Contents

1. Motivation
2. Proposed solutions
3. Assumed
4. Minimal Spanning Trees
5. Algorithms
6. Parallel Kruskal
7. Well separated concerns (or ball decomposition)
8. Disclaimer

Sequence of parallel bunch adds

- Process edges in order, but in parallel 2 steps a time if you have 2 processors.

  add $(4, 1)$ and $(2, 1)$ concurrently;

  $(10, 5)$ and $(10, 6)$ concurrently;

  $(2, 5)$ and $(1, 10)$ concurrently, but watch out!

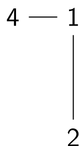
Sequence of parallel bunch adds

- Process edges in order, but in parallel 2 steps a time if you have 2 processors.

  add $(4, 1)$ and $(2, 1)$ concurrently;

  $(10, 5)$ and $(10, 6)$ concurrently;

  $(2, 5)$ and $(1, 10)$ concurrently, but watch out!
- The tree



10

5

6

- red edges created a cycle.

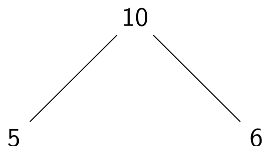
Sequence of parallel bunch adds

- Process edges in order, but in parallel 2 steps a time if you have 2 processors.

  add $(4, 1)$ and $(2, 1)$ concurrently;

  $(10, 5)$ and $(10, 6)$ concurrently;

  $(2, 5)$ and $(1, 10)$ concurrently, but watch out!
- The tree



- red edges created a cycle.

Sequence of parallel bunch adds

- Process edges in order, but in parallel 2 steps a time if you have 2 processors.

  add $(4,1)$ and $(2,1)$ concurrently;

  $(10,5)$ and $(10,6)$ concurrently;

  $(2,5)$ and $(1,10)$ concurrently, but watch out!

- The tree

- red edges created a cycle.

Sequence of parallel bunch adds

- Process edges in order, but in parallel 2 steps a time if you have 2 processors.

  add $(4, 1)$ and $(2, 1)$ concurrently;

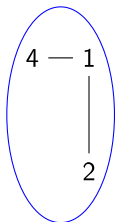
  $(10, 5)$ and $(10, 6)$ concurrently;

  $(2, 5)$ and $(1, 10)$ concurrently, but watch out!
- The tree

- red edges created a cycle.

Non interference demanded

- Blue lines delimit supervertices



10

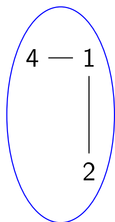
5

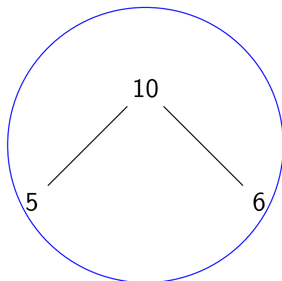
6

- Sequential execution helps the lazy programmer.
- Concurrent steps should not interfere with each other.

Non interference demanded

- Blue lines delimit supervertices

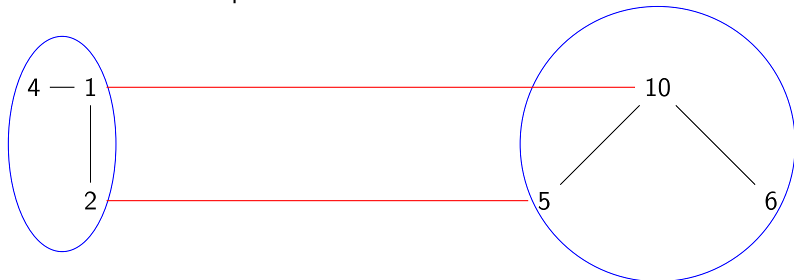




- Sequential execution helps the lazy programmer.
- Concurrent steps should not interfere with each other.

Non interference demanded

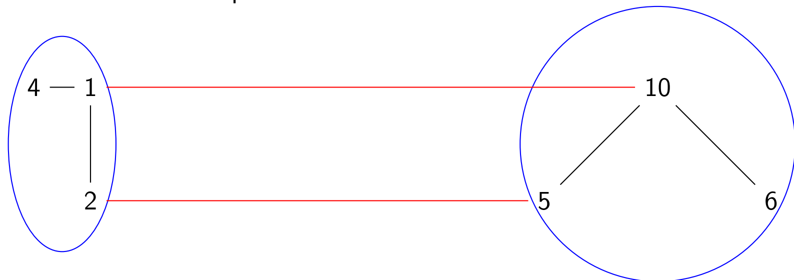
- Blue lines delimit supervertices



- Sequential execution helps the lazy programmer.
- Concurrent steps should not interfere with each other.

Non interference demanded

- Blue lines delimit supervertices



- Sequential execution helps the lazy programmer.
- Concurrent steps should not interfere with each other.

Linearization to avoid interference

- The two red edges interfere with each other.
- Their addition must be *linearized*.

Linearization to avoid interference

- The two red edges interfere with each other.
- Their addition must be *linearized*.
- Elipse on the left and circle on the right represent two different trees.
- Options:
  - Mutual exclusion through atomic execution.

    Concurrent access to one tree is disallowed by explicit use of atomic transactions. For example, through semaphores.

Linearization to avoid interference

- The two red edges interfere with each other.
- Their addition must be *linearized*.
- Elipse on the left and circle on the right represent two different trees.
- Options:
  - Mutual exclusion through atomic execution.

    Concurrent access to one tree is disallowed by explicit use of atomic transactions. For example, through semaphores.
  - Lineralization without atomic instructions

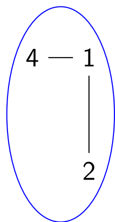
    Each concurrent step request the right to change a tree.
    Requests are prioritized.
    First request that does not create a circle is commited.
    Other requests are rejected and must be resubmitted.

Table of Contents

1. Motivation
2. Proposed solutions
3. Assumed
4. Minimal Spanning Trees
5. Algorithms
6. Parallel Kruskal
7. Well separated concerns (or ball decomposition)
8. Disclaimer

Limitations of parallel Kruskal

- Dense graphs always an obstacle.
  - All edges have to scanned: a dense graph with a million vertices might demand many terabytes of RAM memory.
  - Linearization of concurrent steps may lead to ineffcient sequential processing of too many edges.
- Mitigation: Prune 30% heaviest edges of dense graphs. However...



10

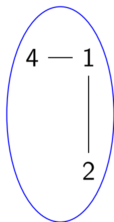
5

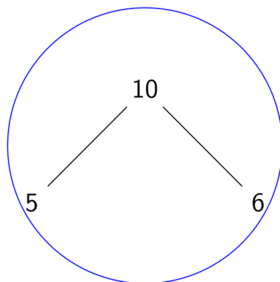
6

Red edges are in 30% heaviest.

Limitations of parallel Kruskal

- Dense graphs always an obstacle.
  - All edges have to scanned: a dense graph with a million vertices might demand many terabytes of RAM memory.
  - Linearization of concurrent steps may lead to ineffcient sequential processing of too many edges.
- Mitigation: Prune 30% heaviest edges of dense graphs.
  However...

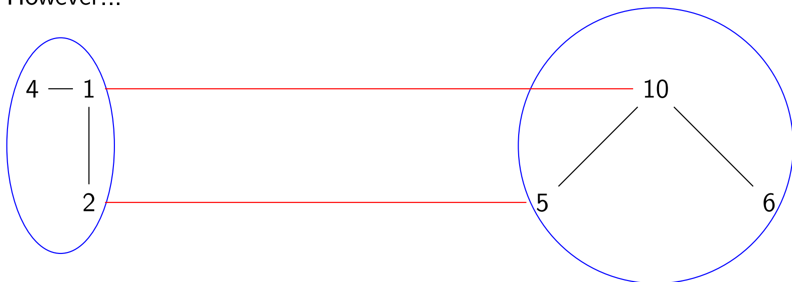




Red edges are in 30% heaviest.

Limitations of parallel Kruskal

- Dense graphs always an obstacle.
  - All edges have to scanned: a dense graph with a million vertices might demand many terabytes of RAM memory.
  - Linearization of concurrent steps may lead to ineffcient sequential processing of too many edges.
- Mitigation: Prune 30% heaviest edges of dense graphs. However...



Red edges are in 30% heaviest.

Vertex-centric processing

- Kruskal's limitations are edge scanning limitations
    - Scanning all edges in graph can be quadratic in space and work on the number of vertices.

Vertex-centric processing

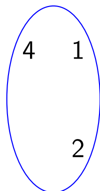
- Kruskal's limitations are edge scanning limitations
  - Scanning all edges in graph can be quadratic in space and work on the number of vertices.
  - Updating the **mst** in constructitions and the super-verticies will always lead to slow sequential execution of parallel algorithms due to linearization of concurrent executions.

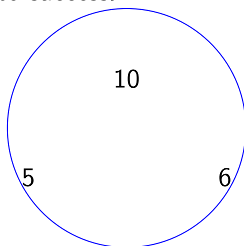
Vertex-centric processing

- Kruskal's limitations are edge scanning limitations
  - Scanning all edges in graph can be quadratic in space and work on the number of vertices.
  - Updating the **mst** in constructitions and the super-verticies will always lead to slow sequential execution of parallel algorithms due to linearization of concurrent executions.
  - Problem is intrinsic to Kruskal, Prim-Dijkstra and Borůvka algorithms.

Vertex-centric processing

- Kruskal's limitations are edge scanning limitations
  - Scanning all edges in graph can be quadratic in space and work on the number of vertices.
  - Updating the **mst** in constructitions and the super-verticies will always lead to slow sequential execution of parallel algorithms due to linearization of concurrent executions.
  - Problem is intrinsic to Kruskal, Prim-Dijkstra and Borůvka algorithms.
- The **mst** construction will be more efficient if vertex centered.
- Divide-and-conquer might be the path to success.





Pairs decomposition

A graph $G = (V, E)$ with n vertices and Euclidean distances is assumed.

- A k-d Tree or Ball Tree can be constructed with parallel work $O(n \lg n)$
- A well separated decompsition of vertices $W = (A_1, B_1), ..., (A_w, B_w)$ can be constructed in $O(n \lg n)$ work, see *CHEP 2013*.
- In $O(n \lg n)$ work a subset $E' \in E$ can be built where each $(a_i, b_i) \in E'$ is the Bichromatic Closest Pair of (A_i, B_i)

Origin of the decomposition

- The idea of decomposing the vertices in *balls* originates in a 1977 paper and theorem by Yao:
  - decompose given vertices to balls of up to 16 vertices, followed by computation **mst** of each ball, **mst** of super-vertices.
  - Time reduction obtained to $O((n \lg n)^{1.8}$ down from $O((n \lg n)^2)$
- Idea of decomposition to well separted balls originated in 1989 work by Vaidya, directed at sequential computations.
- $O(n \lg n)$ work bounds obtained with unform distributions.
- For non-uniform distributions good bounded approximations for spanning trees weight can be guaranteed.

Experimental comparisons

- Euclidean graphs with uniform distributions of 3-d. Time in seconds.
- GPU: Tesla 2070.
- CPU: Intel Xeon E5620, 2.40GHz.
- **spmst**: Separated pair decomposition **mst** on 1 thread.
- **4-spmst**: Separated pair decomposition **mst** on 4 thread.
- **8-spmst**: Separated pair decomposition **mst** on 8 thread.
- **gpu-spmst**: Separated pair decomposition **mst** on GPU.
- **Kruskal**: standard sequential Kruskal.

Number of points	2^{15}	2^{18}	2^{19}	2^{20}
Kruskal	19.09	-	-	-
sp**mst**	23.15	172.10	250.30	917.87
4-sp**mst**	9.67	78.21	119.19	447.71
8-sp**mst**	7.13	53.21	78.83	286.10
gpu-sp**mst**	1.01	7.61	11.51	40.96

Table of Contents

1. Motivation
2. Proposed solutions
3. Assumed
4. Minimal Spanning Trees
5. Algorithms
6. Parallel Kruskal
7. Well separated concerns (or ball decomposition)
8. Disclaimer

Much to do yet

- This VERY much work in development.
- Work needed in:
  - Testing of parallel Kruskal and the filtering policy based on J Bentley D Johnson's work.
  - Intensive test for both Kruskal and decomposition algorithms needed.
  - Transactional memory and delayed reservation should be tested with a parallel Borůvka algorithm.
  - No test has been performed so far with non-uniform distributions.
  - Intensive comparision of behaviours in presence of dense and sparse graphs has not been performed.
- This works is not part of the CMS development and we do not even know if CMS has a demand for the sort of algorithm here presented.