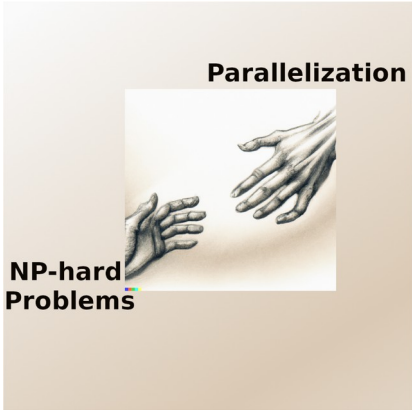
NP-Hard Problems Meet Parallelization

https://bit.ly/rajesh-talks



23-Apr-2024

Rajesh Pandian M
IIT MADRAS
Indian Institute of Technology Madras
www.cse.iitm.ac.in/~mrprajesh

Outline

- Motivation
  - Philosophy
  - Landscape
- Steiner Tree
  - Algorithm
  - Halt-Optimization
  - GPU-Optimization
  - Two-level parallelism
- Vehicle Routing
  - Local-search algorithm
- Summary
- Future Directions

Our Philosophy

... take a **fresh look** at some of the classic graph algorithms and devise **faster** and more parallel GPU and CPU implementations.

\+

NP-hard

\=

Our Philosophy

\- Fallin et al.

SC'23

A High-Performance MST Implementation for GPUs

Alex Fallin
Dept. of Computer Science
Texas State University
San Marcos, Texas, USA
waf13@txstate.edu

Andres Gonzalez
Dept. of Computer Science
Texas State University
San Marcos, Texas, USA
ag1548@txstate.edu

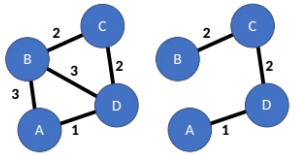
Jarim Seo
Dept. of Computer Science
Texas State University
San Marcos, Texas, USA
j_s1195@txstate.edu

Martin Burtscher
Dept. of Computer Science
Texas State University
San Marcos, Texas, USA
burtscher@txstate.edu

ABSTRACT

Finding a minimum spanning tree (MST) is a fundamental graph algorithm with applications in many fields. This paper presents ECL-MST, a fast MST implementation designed specifically for GPUs. ECL-MST is based on a parallelization approach that unifies Kruskal's and Borůvka's algorithm and incorporates new and existing optimizations from the literature, including implicit path compression and edge-centric operation. On two test systems, it outperforms leading GPU and CPU codes from the literature on all of our 17 input graphs from various domains. On a Titan V GPU,

lines. In this example, the cheapest distribution grid that allows everyone to deliver or receive electricity is the MST shown.



Current status

Irregular Mem. access

Poly-time Problems

- Parallelization is easier
- Algorithms are simpler
- Run few seconds on million/billion-scale
- Solution search space is small
- Exact solution

- **Examples**
  - Minimum Spanning Tree
  - Single Source Shortest Path

NP-Hard Problems

- Comparitively difficult
- Complicated algorithms
- Few hours for thousand-sized instances
- Solution search space is large
- Tradeoff: Solution vs Time

  - Steiner Tree Problem
  - Travelling Salesman Problem
  - Vehicle Routing Problem

- More practical applications

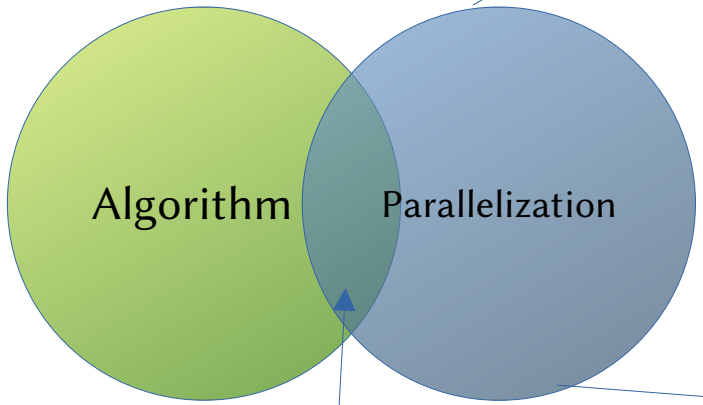
Goal: Solve largest benchmark instances from DIMACS/PACE Challenges

Algorithms and Optimizations

- Parallel Implementations chosen from
  - Efficient serial algorithm
  - Proven algorithm amenable for parallelization

Design parallelism-friendly algorithms

- Optimizations



Data-driven versus Topology-driven Irregular Computations on GPUs

Rupesh Nasre
The University of Texas
Austin, Texas, USA
Email: nasre@ices.utexas.edu

Martin Burtscher
Texas State University
San Marcos, Texas, USA
Email: burtscher@txstate.edu

Keshav Pingali
The University of Texas
Austin, Texas, USA
Email: pingali@cs.utexas.edu

Compiling Graph Applications for GPUs with GraphIt

Ajay Brahmakshatriya
MIT CSAIL
USA
ajaybr@mit.edu

Yunming Zhang
MIT CSAIL
USA
yunming@mit.edu

Changwan Hong
MIT CSAIL
USA
changwan@mit.edu

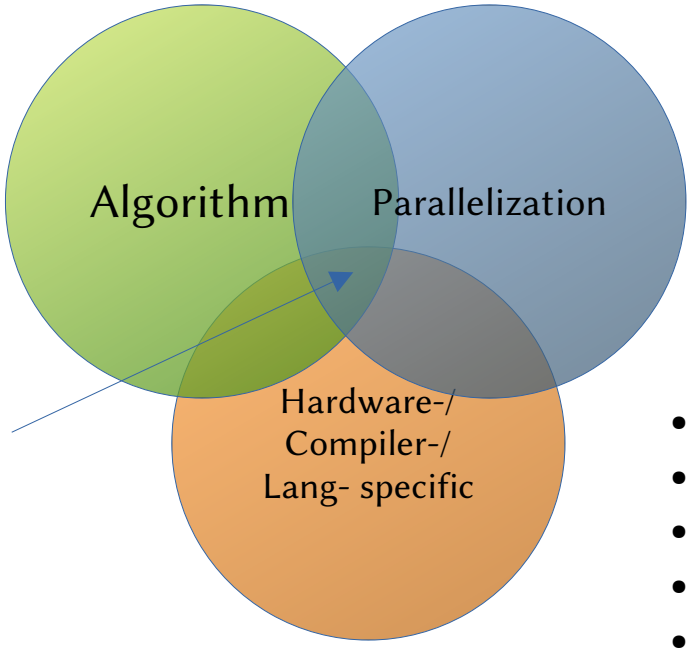
Shoaib Kamil
Adobe Research
USA
kamil@adobe.com

Julian Shun
MIT CSAIL
USA
jshun@mit.edu

Saman Amarasinghe
MIT CSAIL
USA
saman@csail.mit.edu

Optimizing for peak performance

- Input Charateristics
  - Diameter
  - Max degree
  - Road/social network
- Properties of substeps



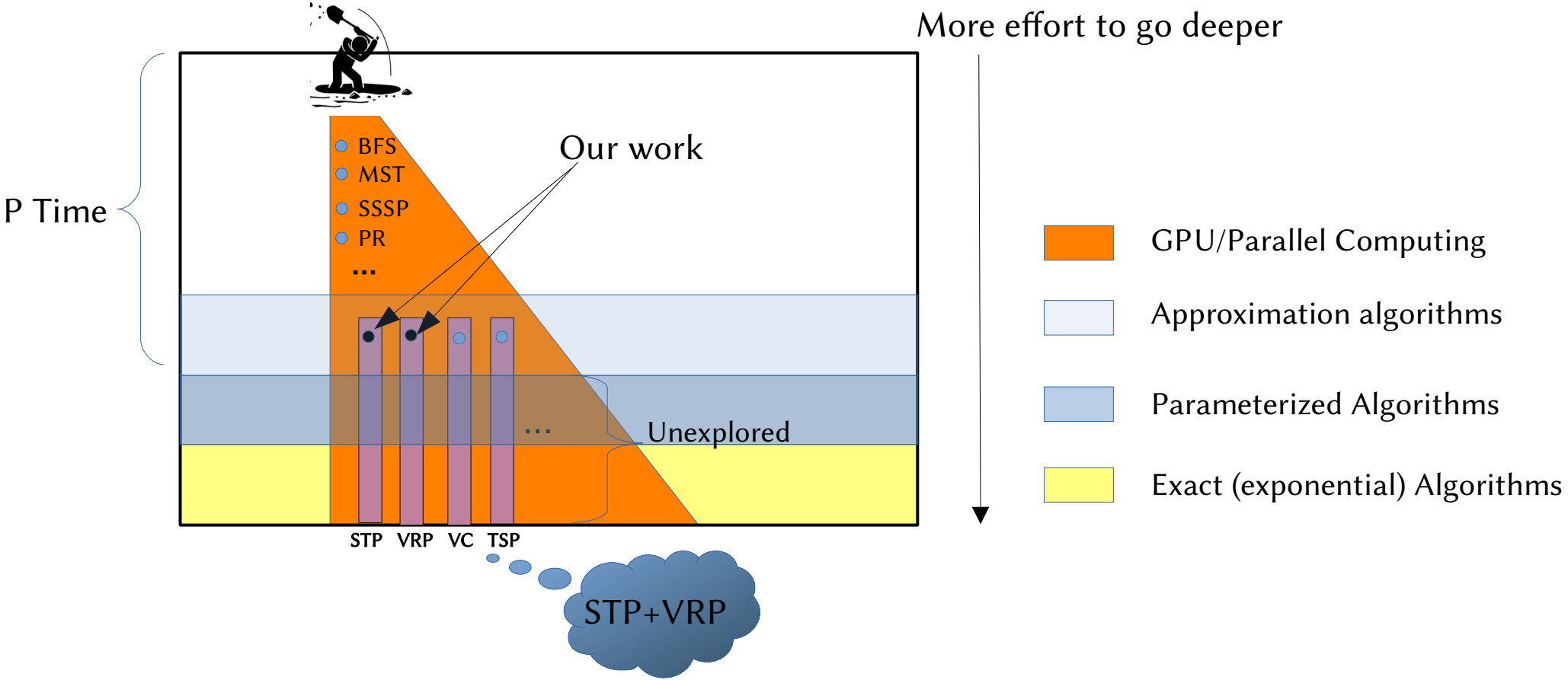
Peak performance

- Vertex-/Edge- centric
- Data-/Topology-driven
- Push-/Pull-based
- Load-balancing

  …

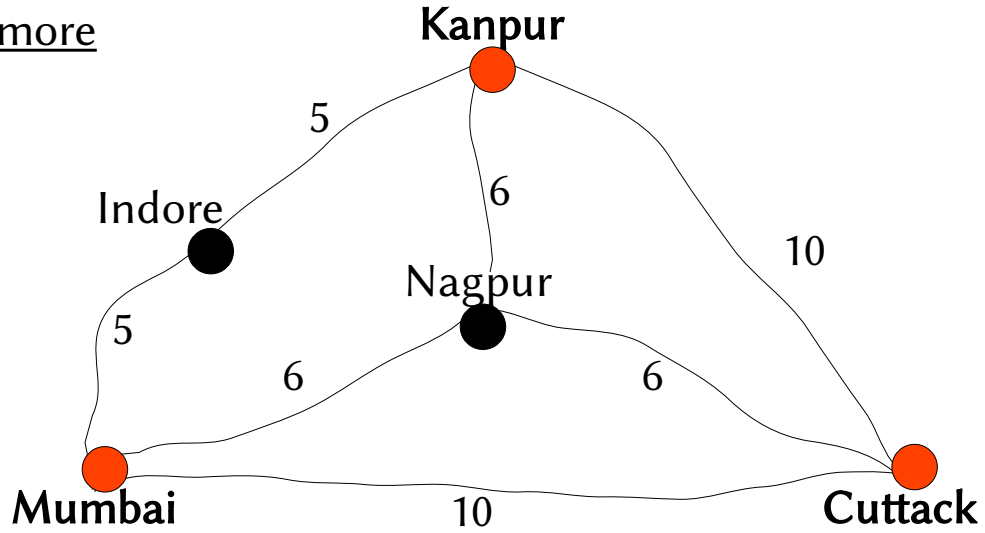
- Architecture-specific
- Shared memory
- Warp-Intrincics
- Data accesses within reg/caches
- Vectorization loads/adds
- Language-specific 128b CAS CC9+
- Use profilers and https://godbolt.org

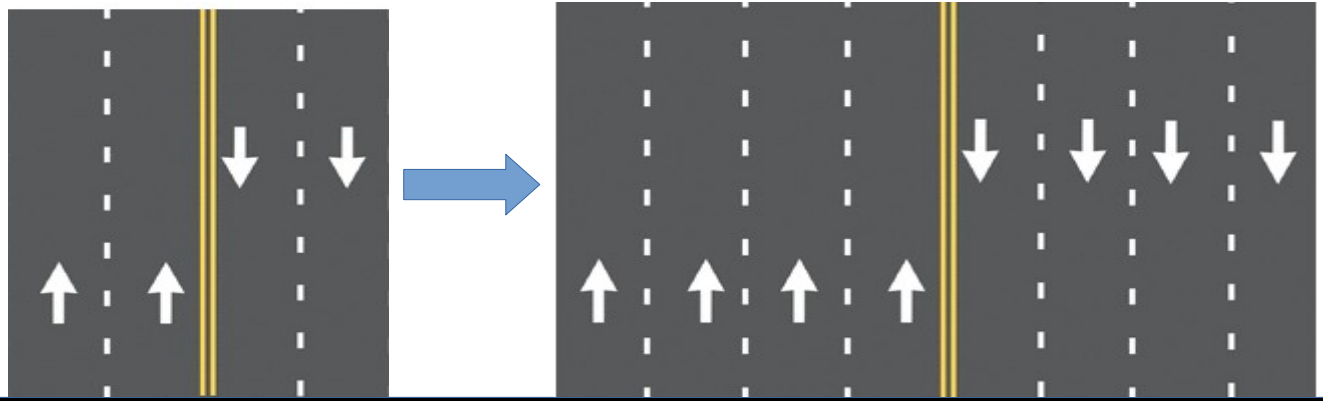
Landscape of Parallelization



Steiner Tree - Example

- Let's say Highway Department wants to add more lanes to connect K, M and C
- Say K, M & C are important.
- Overall budget should be minimum
- Budget ∝ Distance





Steiner Tree Problem (STP) - Example

Input : Graph G(V, E, W) W:E→ Z^+ and L⊆V terminals.
Output: A tree T'(V'⊇L, E'⊆E) of G such that minimize W(E').
// Minimum weighted tree with all terminals.

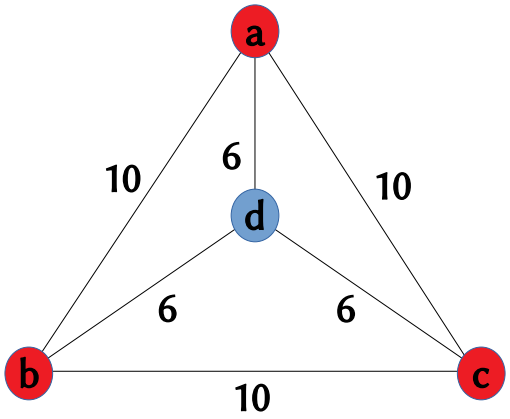


Fig 1 (a)

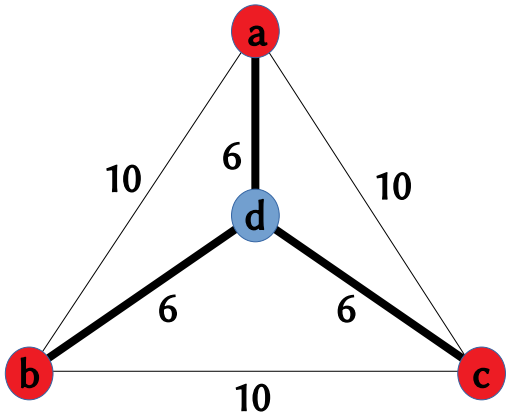
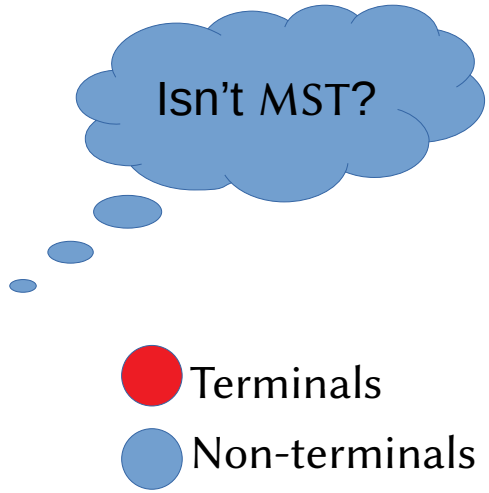


Fig 1 (b)



Steiner Tree Problem (STP) - Example

Input : Graph G(V, E, W) W:E→ Z^+ and L⊆V terminals

Output : Minimum weighted tree with all terminals

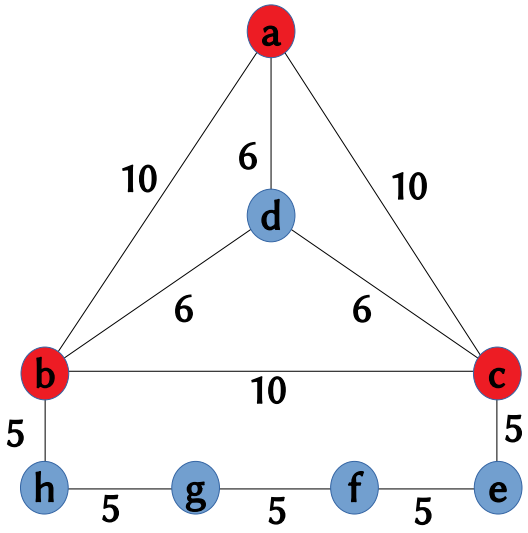


Fig 2 (a)

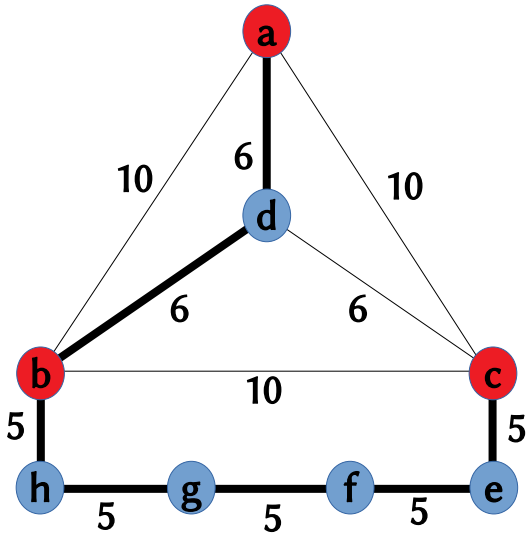


Fig 2 (b)

|MST| = 37
|OPT| = 18

Terminals

Non-terminals

Steiner Tree Problem (STP) - Hardness

Input : Graph G(V, E, W) W:E→ Z^+ and L⊆V terminals

Output : Minimum weighted tree with all terminals

Take away

- MST solution is a valid feasible Steiner Tree solution
- However, solution can be arbitrarily bad w.r.t OPT.

Special cases

- L = {u,v} or k = 2 STP=ShortestPath_In_G(u,v)
- L = V or k = n STP=MST(G)

In P Time

- In general STP is NP-Hard

Approximation algorithms

How to deal with NP-Hardness

- What could be naive solutions? Enumerate all Spanning trees.

Approximation algorithm

- Runs in Polynomial time.
- Outputs an approximate solution with some guarantee.
  - e.g 2 or some constant, log n, etc.

- There are several algorithms
  - Kou, Markowsky and Berman[KMB81]
  - Mehlhorn [M88]
  - Robins and Zelikovsky [RZ2000]

|ALG| <= 2 |OPT|

L. Kou, G. Markowsky, and L. Berman. A fast algorithm for Steiner trees. *Acta Informatica, 1981.*

KMB Algorithm G(V,E,W,L)

Phase 1

// Input G

Computes the shortest distance between every pair of terminals

Phase 2

// Construct G'= K_L

Build a graph G' over terminals, having edge-weights corresponding to the shortest distances computed in Phase 1

// Every edge in G' corresponds to a path in G

MST (G')

Phase 3

// Construct G''

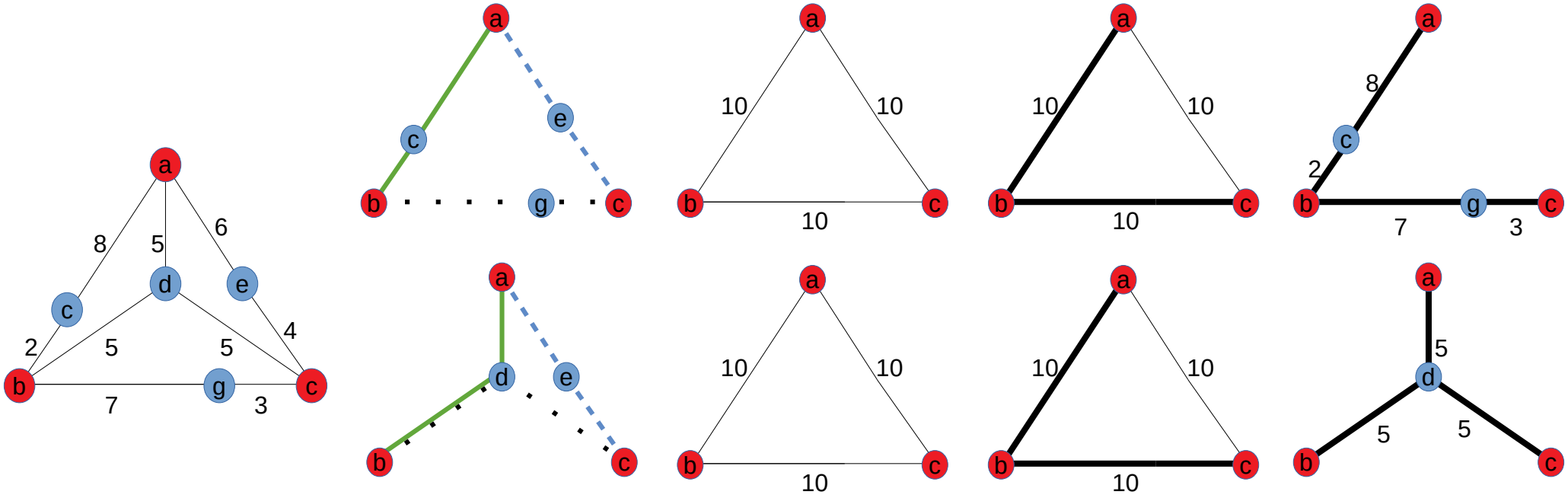
For every edge in MST(G') substitute the edges with the corresponding shortest path in G

// Collect all the edges & vertices of the corresponding path to construct G''

MST(G'')

Takeaway: One more invocation for SSSP/MST algorithm. G⟶G'⟶G''

KMB Algorithm – Running example

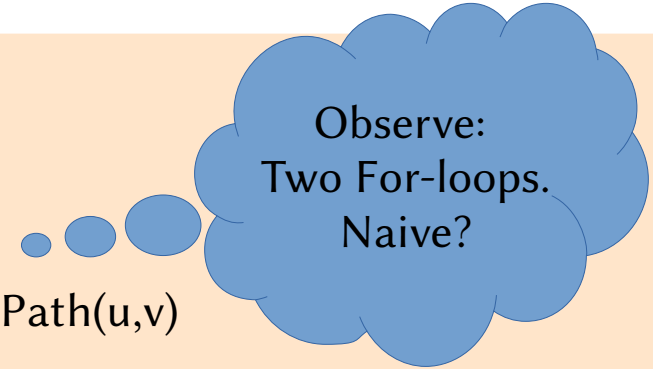


1) A parallel KMB may output a different answer. (2) Last MST may be avoided

Fig. 3 Execution steps of KMB algorithm, where ● are terminals and ● are non-terminals.

KMB Algorithm G (V,E,W,L)

Phases 1 & 2



For u in L {
  For v in L {
    P_{uv} = ShortestPath(u,v)
    W'(u,v) = $|P_{uv}|$
  }
}
T' = MST(G', W')

Phase 3

For (u,v) in edges of T' {
  G'' = G'' ∪ P_{uv}
  //Add vertices & edges of P_{uv}
}

T'' = MST (G'', W)

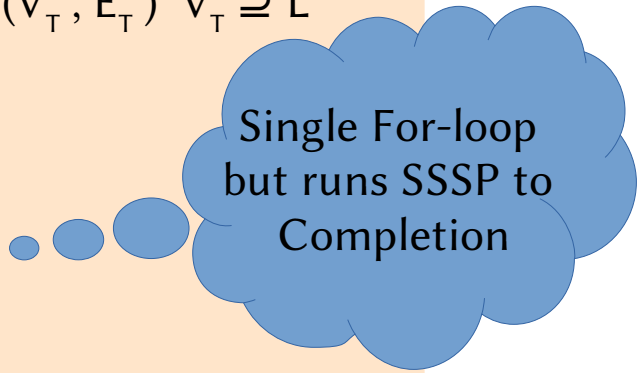
KMB Algorithm G (V,E,W,L)

Input: Graph G(V, E, W, L)
Output: 2-approx Steiner Tree T (V_T , E_T) $V_T \supseteq L$

For u ∈ L {
  SSSP (G, W , L, u) with Halt
  Compute W' incrementally
}
T' = MST(G' ,W')

Compute G'' and its vertices, adjList using T'
T'' = MST(G'' ,W)

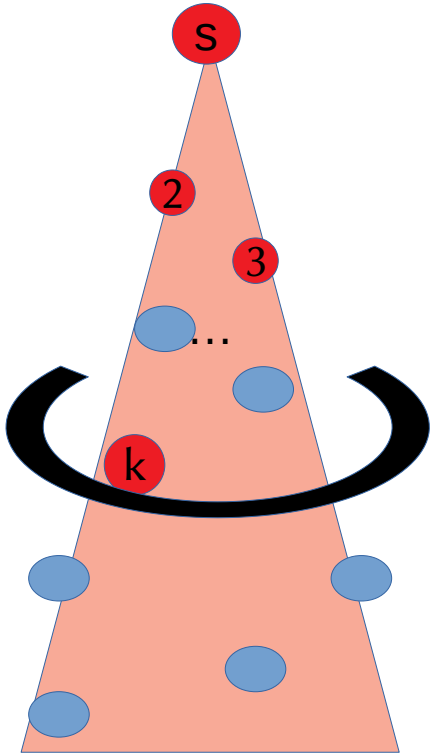
return T''



CPU Implementation - Optimization

- SSSP-halt optimization

Steps of SSSP execution



Dijkstra Property: when a node u is picked from Q for processing then the distance[u] is saturated using all the visited nodes.

Halt SSSP when all terminals are visited

Fig. 4 SSSP-halt visualization

KMB Algorithm G (V,E,W,L)

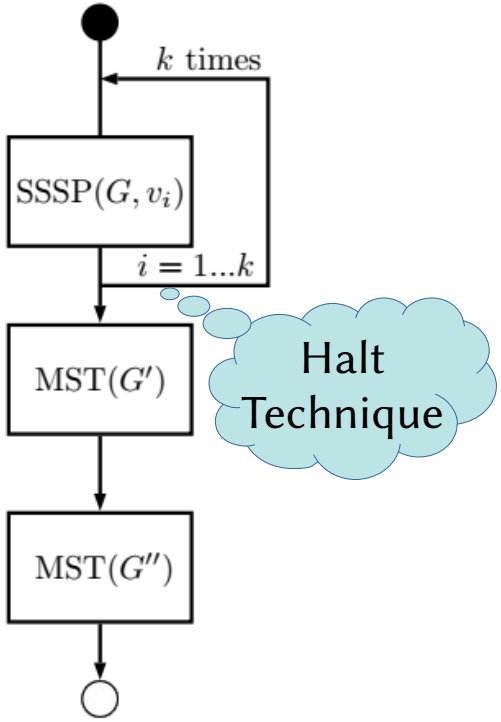
Input: Graph G(V, E, W, L)
Output: 2-approx Steiner Tree T (V_T , E_T) $V_T \supseteq L$

```
For u ∈ L {
    parallel SSSP(G, W , L, u);
    Compute W′ incrementally;
}
T' = parallel MST(G', W' );

Compute G'' and its vertices, adjList ;
T'' = parallel MST(G'', W );

return T''
```

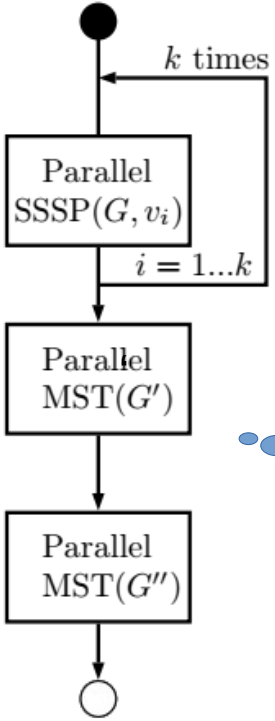
A novel aspect of our work is to run multiple parallel-SSSPs in parallel.

Subroutines? Gunrock

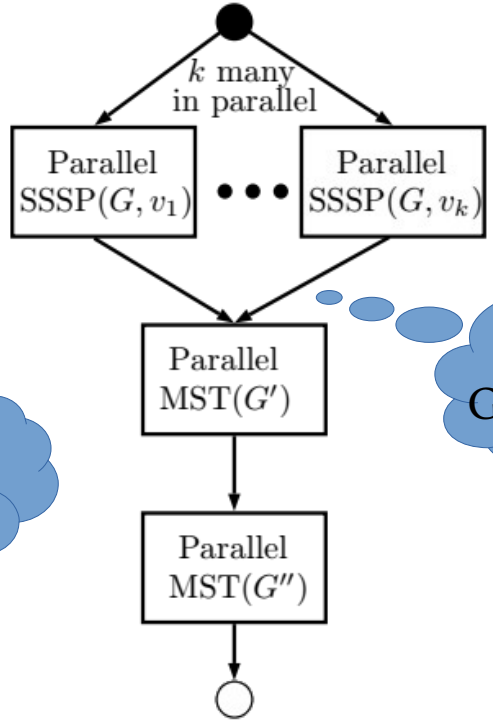
Design choice for parallelization



Halt Technique



Our GPU v1



Our GPU v2 or OPT

Fig. 5 Sequential vs GPU v1 vs GPU v2

KMBCPU
KMBGPU

GPU Implementation - SSSP

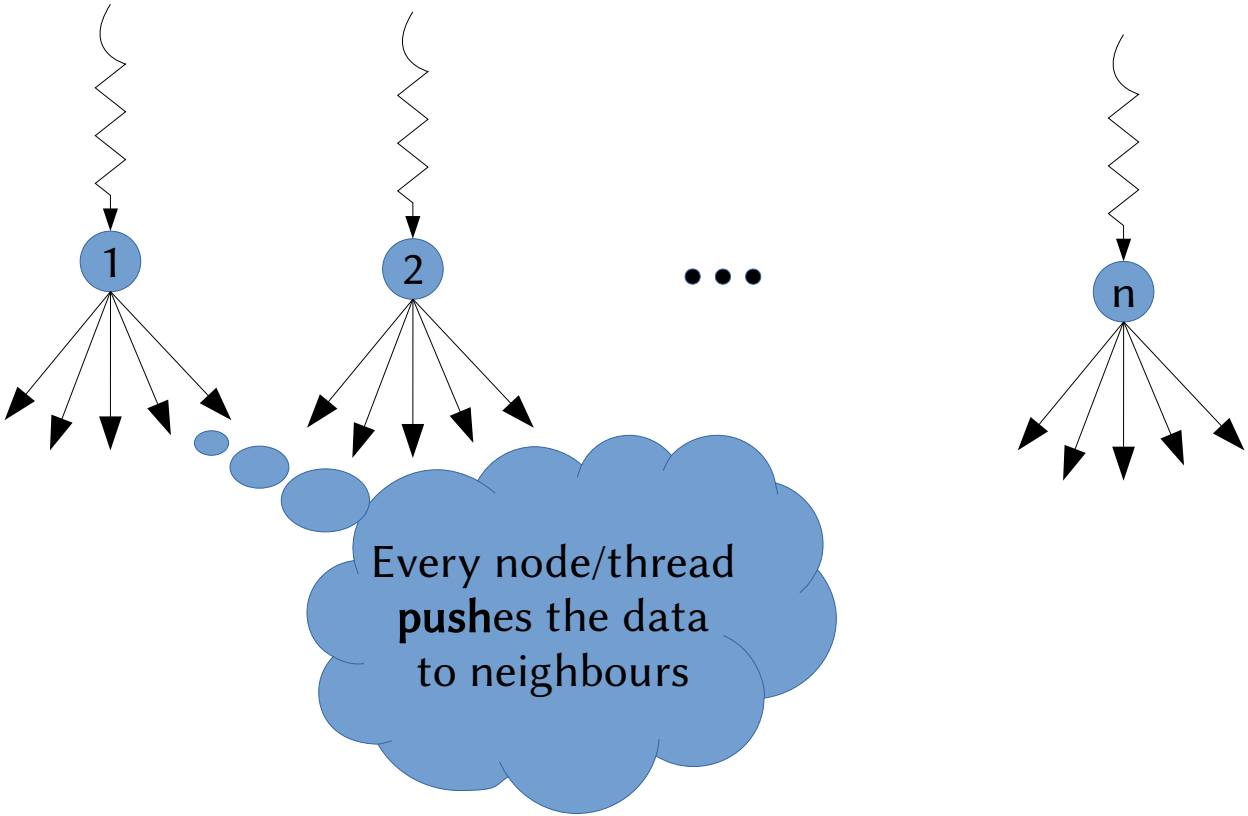


Fig. 6 push-SSSP

- n-threads
- One thread for each node
- Performs RELAX in parallel
- RELAXes its neighbours
- Till there is no change

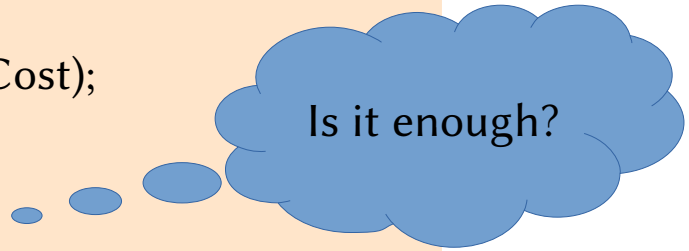
KMB Algorithm G(V,E,W,L)

```
MAIN
For s in L {
  ThdsPerBlk = 512; // or 1024
  Blks = ⌈n/ThdsPer Blk⌉;
  do {
     INIT-KERNEL<Blks,ThdsPerBlk>(s, d_s , p_s , n);
     RELAX-KERNEL<Blks,ThdsPerBlk>(.., s, d_s , p_s , changed, n);
     CopyTo(DArray, d_s );              //= = = = =
     CopyTo(PArray, p_s );              // From Device to Host
     CopyTo(hChanged, changed); // = = = = =
  } while (hChanged);
}
```

- We need the p[] for knowing the intermediate vertices in the shortest path

KMB Algorithm G(V,E,W,L)

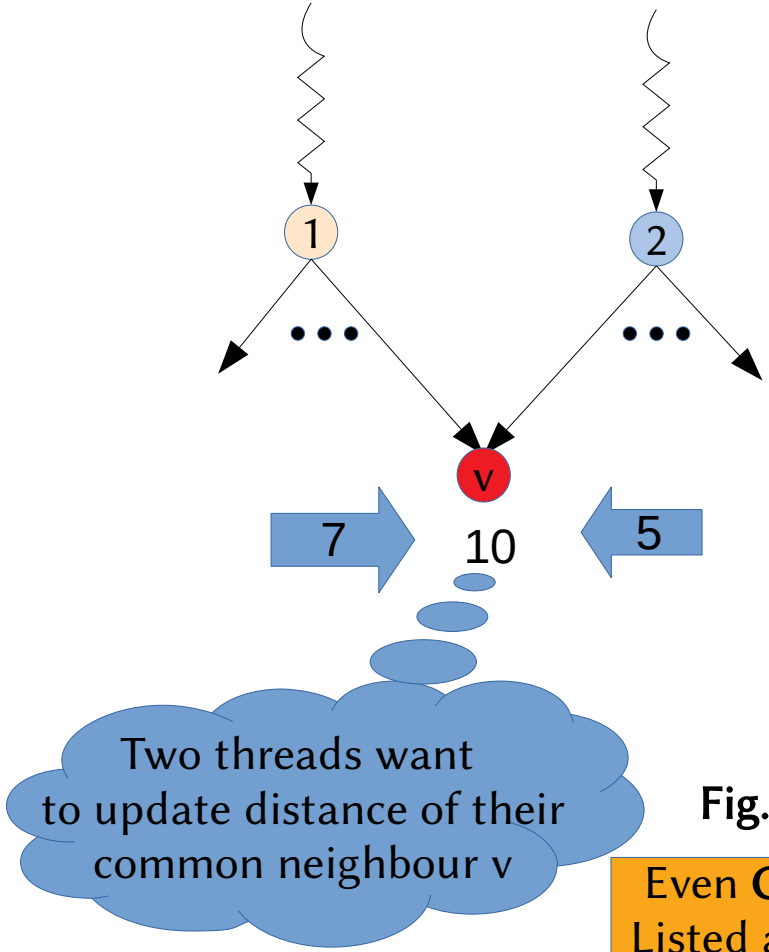
```
RELAX-KERNEL(..,s, d_s , p_s , changed, n) {
u = tid  // compute tid;
If tid < n  {
   For v ∈ adjacent[u] { // Using CSR arrays
       // Relax Operation (u, v, W(u,v))
       newCost = d_s[u] + W(u, v) ;
       old = d_s[v];
       If newCost < old
            Atomic-MIN(d_s[v], newCost);
       // Updates Parent array
      If Atomic-MIN is success {
           p_s[v] = u;
           changed = true;
       }
}}
```



Note :

- Parent of v should be updated if the Atomic-MIN is success

Parent update - Challenge




```
<snip>
..
newCost = d_s[u] + W(u, v) ;
old = d_s[v];
If newCost < old
        Atomic-MIN(d_s[v], newCost);
// Updates Parent array
If Atomic-MIN is success {
        p_s[v] = u;
        changed = true;
}
</snip>
```

Fig. 7 Challenges in parent update

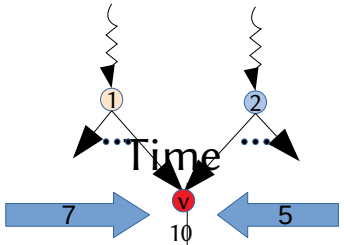
Even **Gunrock** has a challenges in updating p array consistently.
Listed as known issues https://github.com/gunrock/gunrock/releases/tag/v1.0

Parent update - Challenge

Shared d[], p[]

Thread 1

Thread 2



Thread 1	Thread 2
newCost=7 old=10	newCost=5 old=10
d[v]=7 //oldA=10	d[v]=5 //oldA=7
p[v] =1	p[v]=2

Wrong!

```
<snip>
newCost = d[u] + W (u, v) ;
old = d[v];
If newCost < old
   oldA=Atomic-MIN(d[v], newCost);

// Update's Parent array
If Atomic-MIN is success {

    p[v] = u;
    changed = true;
}
```

It is a challenge to find which "thread" updated d[v] to the minimum

How to update both distance and parent at the same time?

Synchronization optimization • Pull

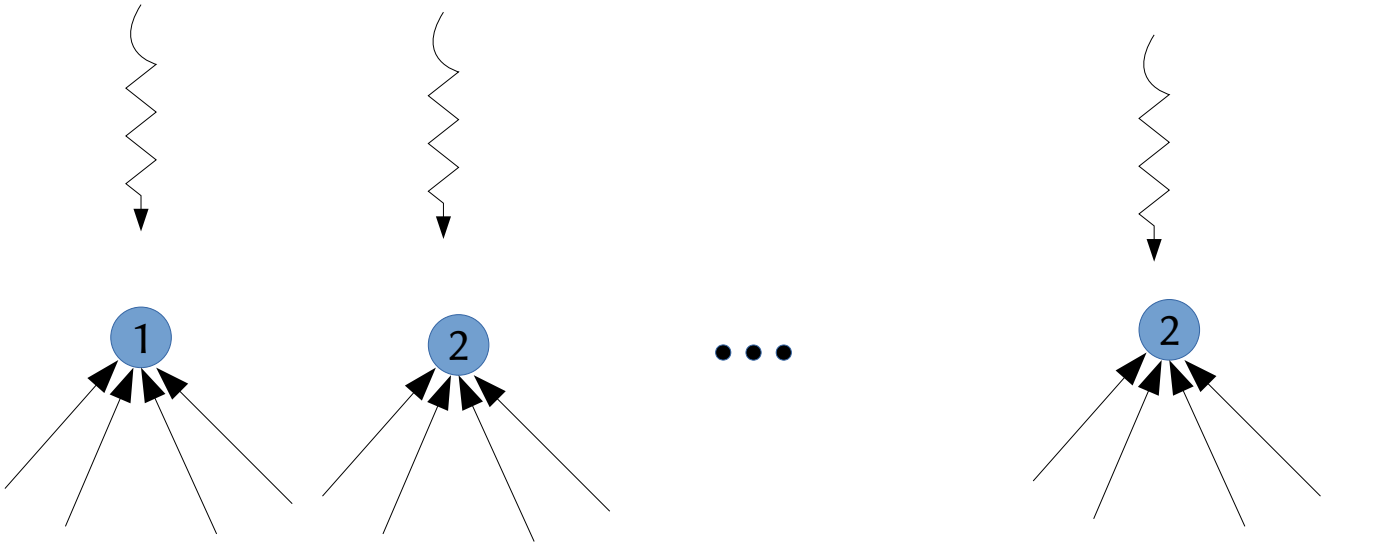


Fig. 8 Pull-SSSP

```
<snip>
newCost = d[v] + W (tid, v) ;
old = d[u];
If newCost < old {
   d[u] = newCost
   p[u] = tid;
   changed = true;
}

</snip>
```

No Atomics
Parent update is easy

Because, one thread is writing to an index

GPU Optimizations

- Synchronization
  - Push
  - Pull
- Computation
  - Data-driven
  - Edge-based
  - Controlled Computation unrolling
    - Δ^2
    - 2Δ
    - $t\Delta$
- Memory
  - Shared memory

Δ – max degree of the graph

GPU Optimizations

- Synchronization
  - Push
  - **Pull**
- Computation
  - Data-driven
  - Edge-based
  - **Controlled Computation unrolling**
    - Δ^2
    - 2Δ
    - **$t\Delta$**
- Memory
  - **Shared memory**

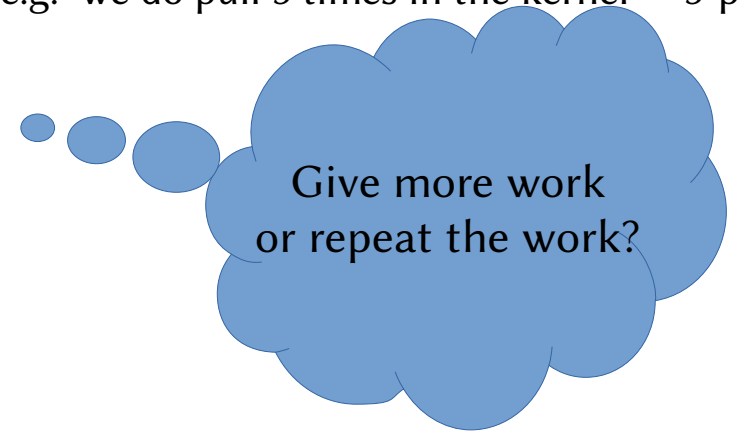
Δ – max degree of the graph

Compute optimization

- Computation Unrolling
  - Instead of one thread doing Δ work, perform more work per thread
  - Update also neighbours of neighbours (Δ^2)
  - **Repeat the work**; Say 2 times or t times (2Δ or tΔ); e.g. we do pull 3 times in the kernel – 3-pull

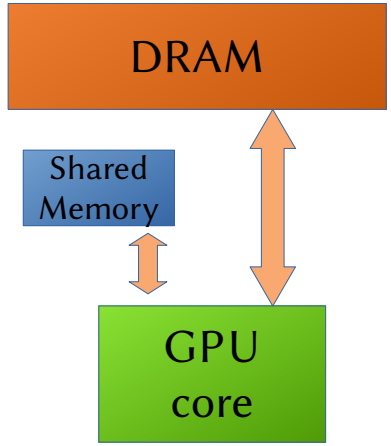
- Data-driven
  - Needs Worklist (WL)
  - Active/Change nodes are inserted into WL

- Edge-based optimization
  - m-threads are launched
  - RELAXes one edge or a group of edges



Memory optimization

- Programmable shared memory can be useful
- When there are multiple reads to DRAM
- We can move data to shared memory



- For e.g. In 3-pull, we moved CSR AdjList to shared
- As the neighbours AdjList is accessed 3 times
- Of the total 48K per block
- when using 512 threadPerBlock we have 24 words to store per thread

- Hence, if degree(node) < 25 we use shared, we move CSR AdjList[node] to Shared
- With shared memory we achieve 25% of improvement in 3-pull

Double-barrel approach

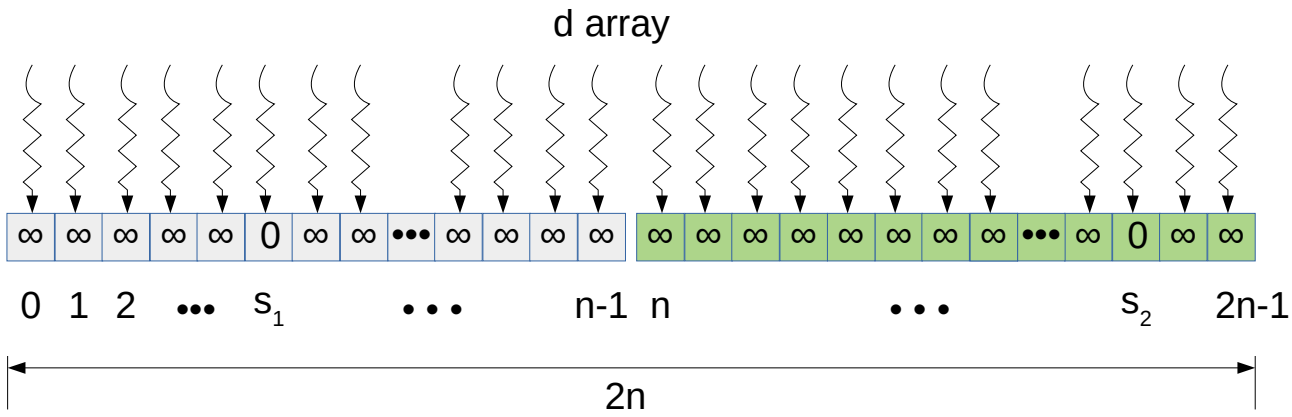
- SSSP happens in parallel
- To run two SSSP, we have to run one after the other
- Instead we use Double-barrel approach
- This can be generalized (p-SSSP)

In our Double-barrel approach, we run two individually parallel SSSPs also in parallel.

Image source: https://stock.adobe.com/

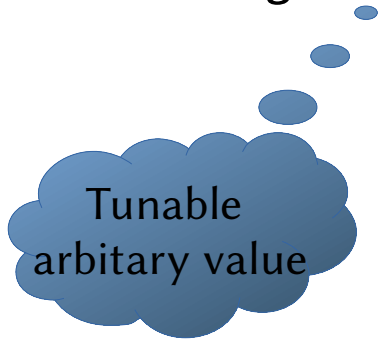
Double-barrel approach

```
Result Array: d[n]
Initialize(d=INTMAX )
d[src] = 0
FixedPoint{
      doRELAX(G, d, changed ...);
}
```




```
Result Array: d[2n]
Initialize(d=INTMAX)
d[src1] = 0; d[n+src2] = 0
FixedPoint{
      doRELAX(G, dist, changed, ...);
}
```

Fig. 9 Double-barrel approach.



Key takeaways so far

- Solving Steiner Tree Problem is NP-hard
- KMB Algorithm, a 2-approximation algorithm
- CPU implementation has SSSP-halt optimization
- SSSP with parent array update was challenging
- Pull-based SSSP is great for KMBGPU even without SSSP-halt
- Parallel-SSSPs in parallel (p-SSSP)

Experimental setup & Graphsuite

CPU

- Intel(R) Xeon(R) E5-2640 v4 @ 2.40GHz
- 64GB RAM

GPU

- Tesla P100 @ 1.33 GHz
- 12GB global memory

-
- GCC 7.3.1 with O3
- CUDA 10.2

Graphsuite

- Total 14 Graphs
  - 11 from PACE Challenge [PACE2018]
  - 2 from SteinLib
  - 1 from SNAP

Baselines

- PACE'18 Winner – CIMAT [PACE2018]
- ODGF's KMB/JEA [BC19]

- CIMAT Team - https://github.com/HeathcliffAC/SteinerTreeProblem
- S. Beyer and M. Chimani, Strong Steiner Tree Approximations in Practice, JEA 2019.

Experiments: Speed-up

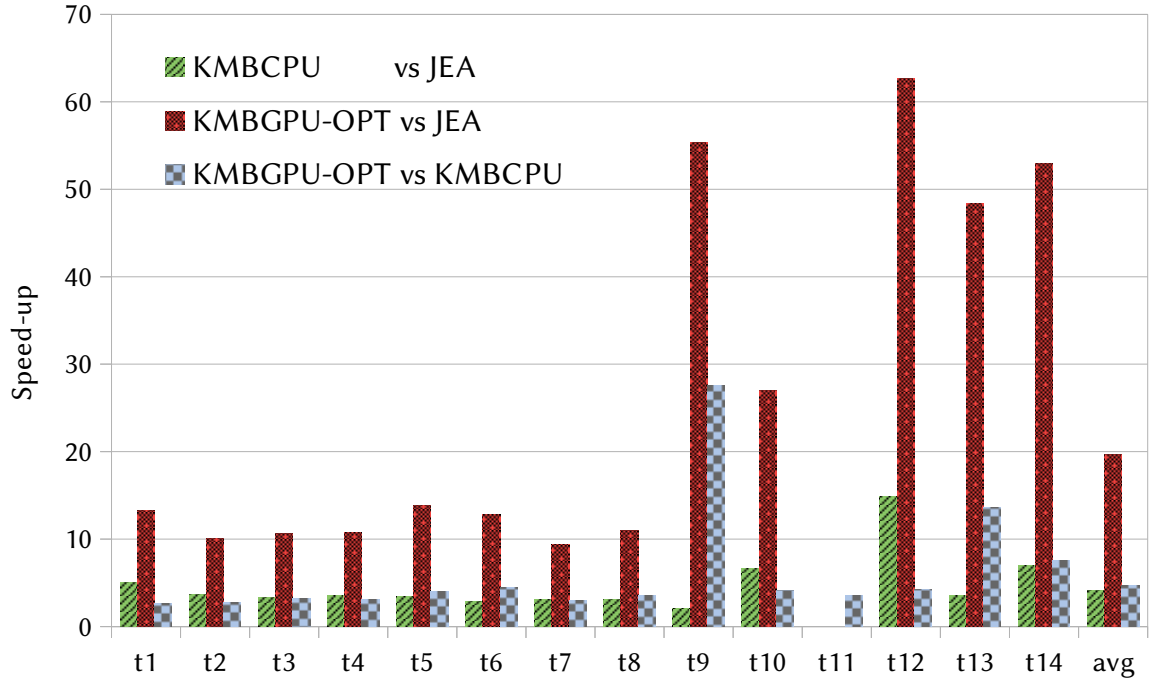


Fig. 10 Speed-up comparisons of the implementations (higher is better). JEA timed-out on t11

Takeaway: KMBCPU and KMBGPUOPT is better than JEA

Comparison of GPU time with Shared mem.

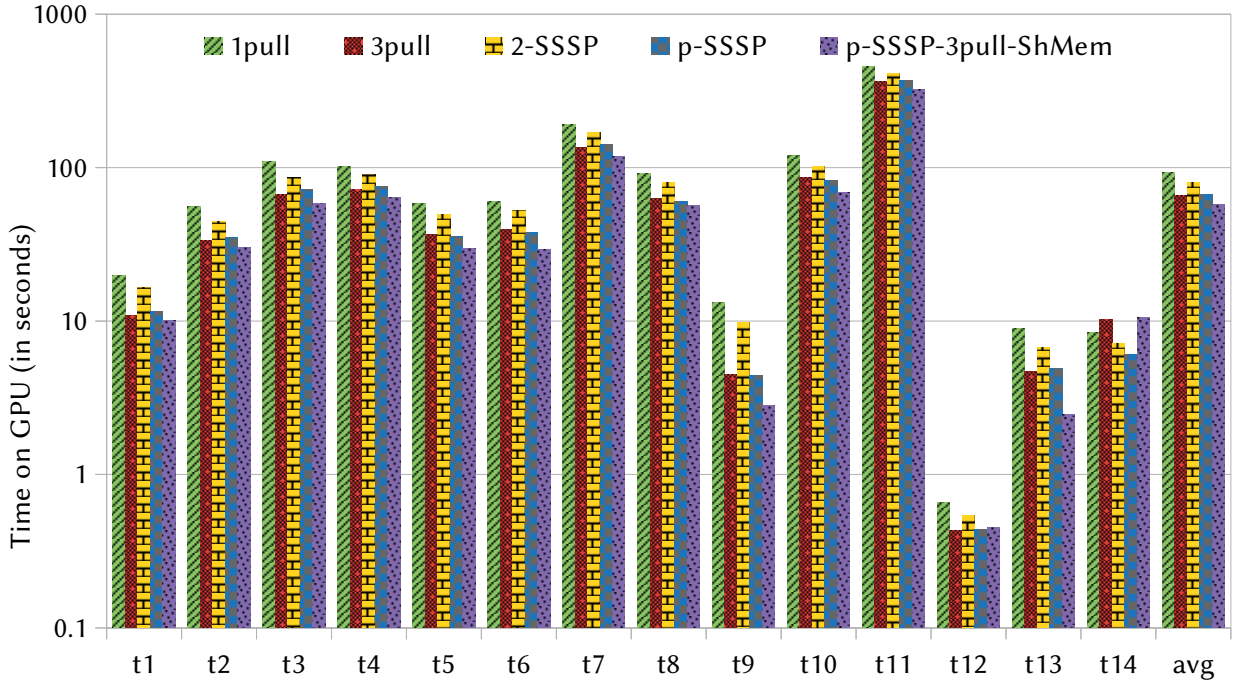


Fig. 11 Comparison of 1-Pull, 3-Pull, Double-barrel & KMBGPU-OPT

Takeaway: Combining GPU optimizations p-SSSP, 3-Pull & Shared memory performs best.

Comparison of p-SSSP

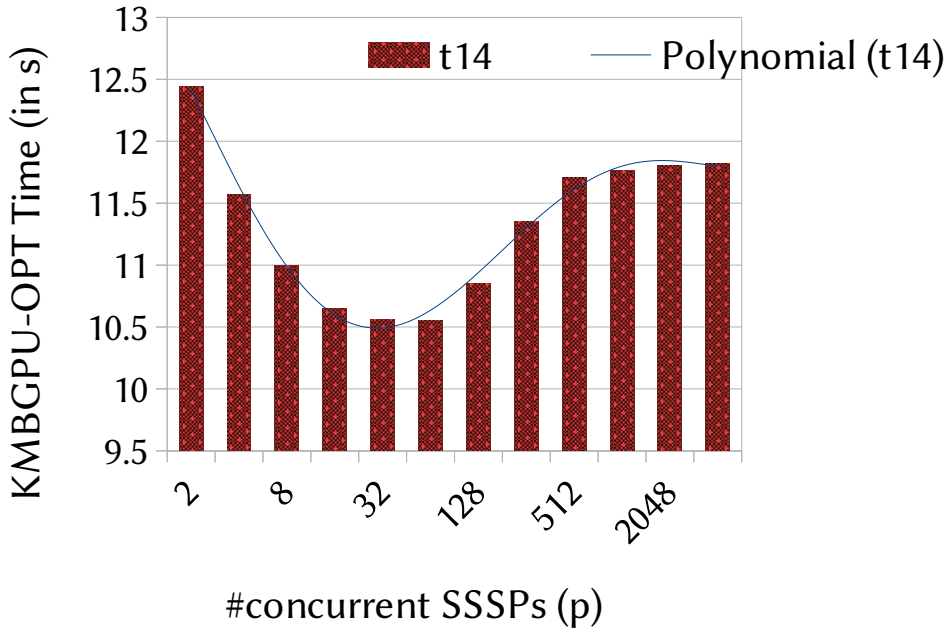


Fig. 12 KMBGPU with varying p-SSSP for the same graphs t14 (Smaller is better).

Takeaway: As we increase the #parallel SSSPs it reaches a point and then increases.

Experiments – Scalability of GPU and CPU

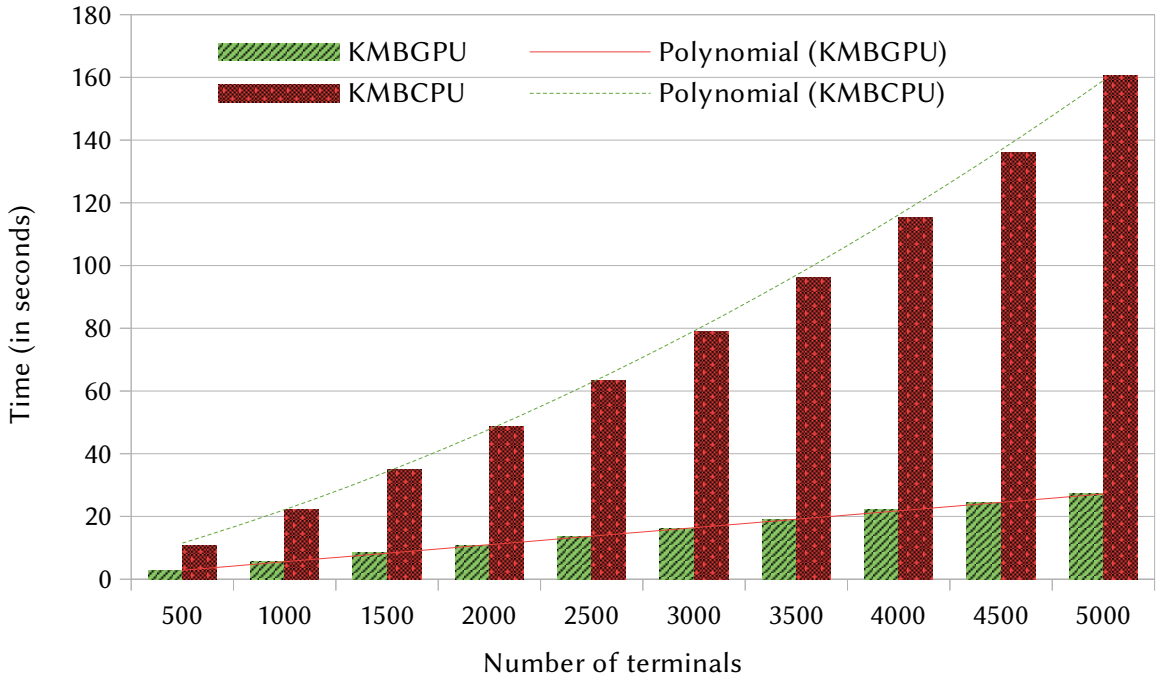


Fig. 13 Scalability plot on **t14** with increasing terminal size (lower is better)

Takeaway: KMBGPU-OPT scales better than KMBCPU

Summary - STP

- Optimized CPU implementation for KMB algorithm
  - Novel SSSP-halt technique
  - Speed-up of 4x (average) over JEA/OGDF's KMB[BC19]

- Optimized GPU implementation for KMB algorithm
  - Novel p-SSSP technique (multiple parallel-SSSP in parallel)
  - Speed-up of 20x (average) over sequential JEA[BC19]

</work>

S. Beyer and M. Chimani, Strong Steiner Tree Approximations in Practice, JEA 2019.

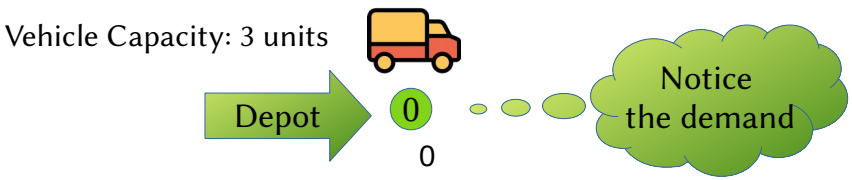
Capacitated Vehicle Routing Problem (CVRP)

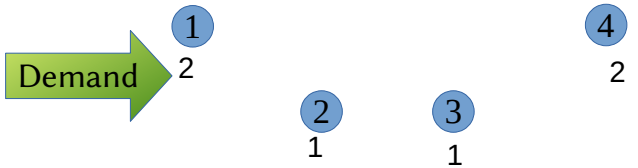
Input : Given *n* nodes (single Depot and customers) with their coordinates (x_i, y_i) and demands $d_i > 0$ for $i \in n$, Vehicle capacity *C*. Node 0 is Depot and has zero demand.

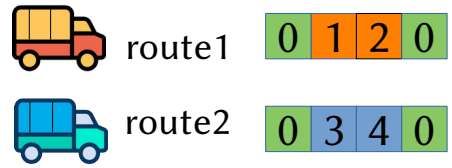
Output: Set of routes serving all the customers respecting the vehicle capacity from/to Depot.

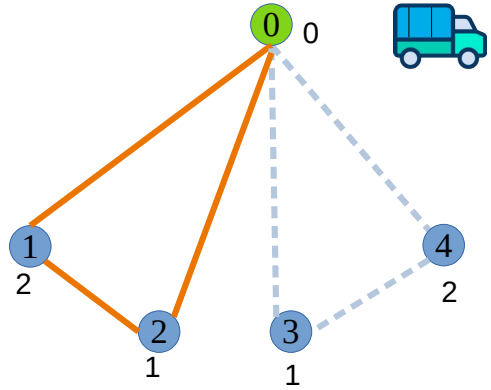
Goal : Minimize total distance travelled.

If Capacity = sum of demands d_i ,
CVRP ➜ Travelling Salesman Problem









NP-Hard

CVRP Limitations

Current state-of-the-art

- work only on smaller instances
- has a large solution Gap
- takes a lot of time

Instance	Number of customers	Time (s)	
		Base2	Base1
Flanders2	30,000	8,355	2,534
Flanders1	20,000	7,768	2,031
Brussels1	15,000	7,164	871

Table 4: State-of-the-art GPU methods are time-consuming.

RQ1. Can we invent a simpler algorithm?

RQ2. Can we reduce Gap on large instances?

RQ3. Design Parallelization friendly algorithms?

Our **ParMDS**

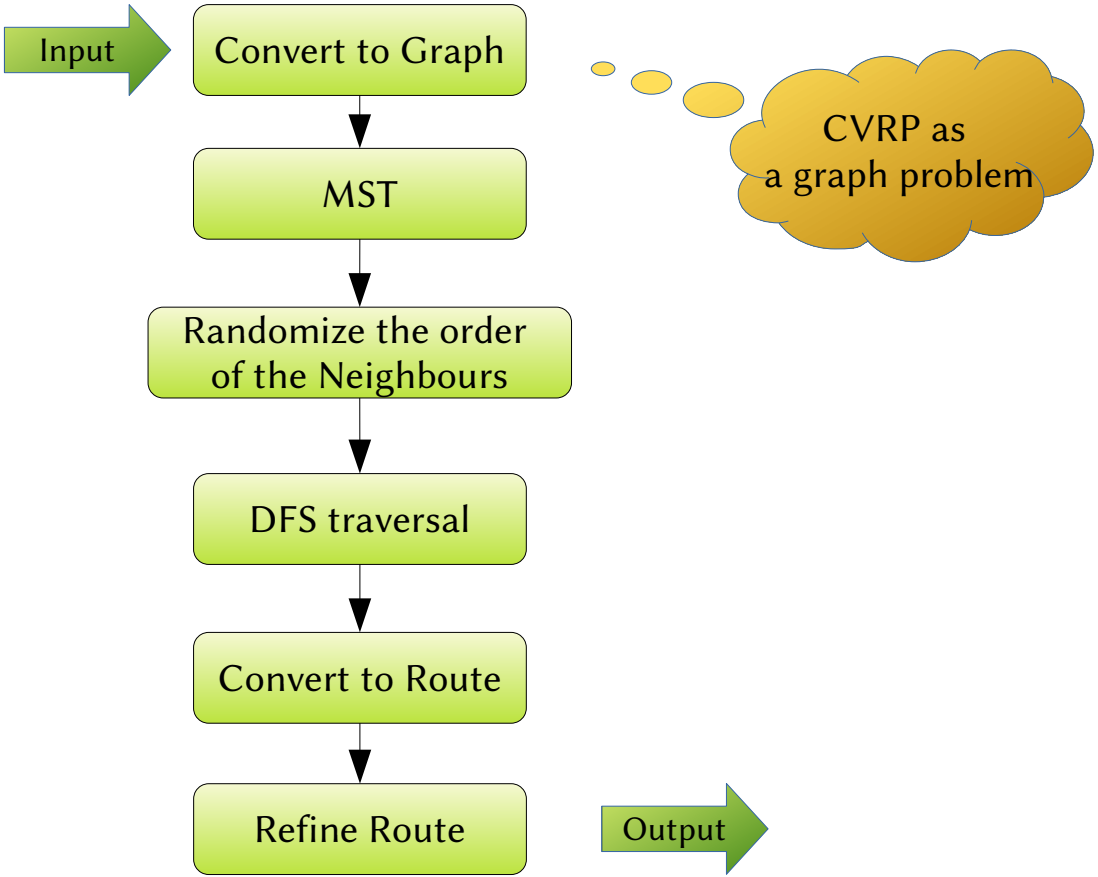
- Serial and Parallel implementation
- Combining MST and DFS
- Uses Local-search approach
- Uses Randomization approach

$$Gap = \frac{Z_S - Z_{BKS}}{Z_{BKS}} \times 100$$

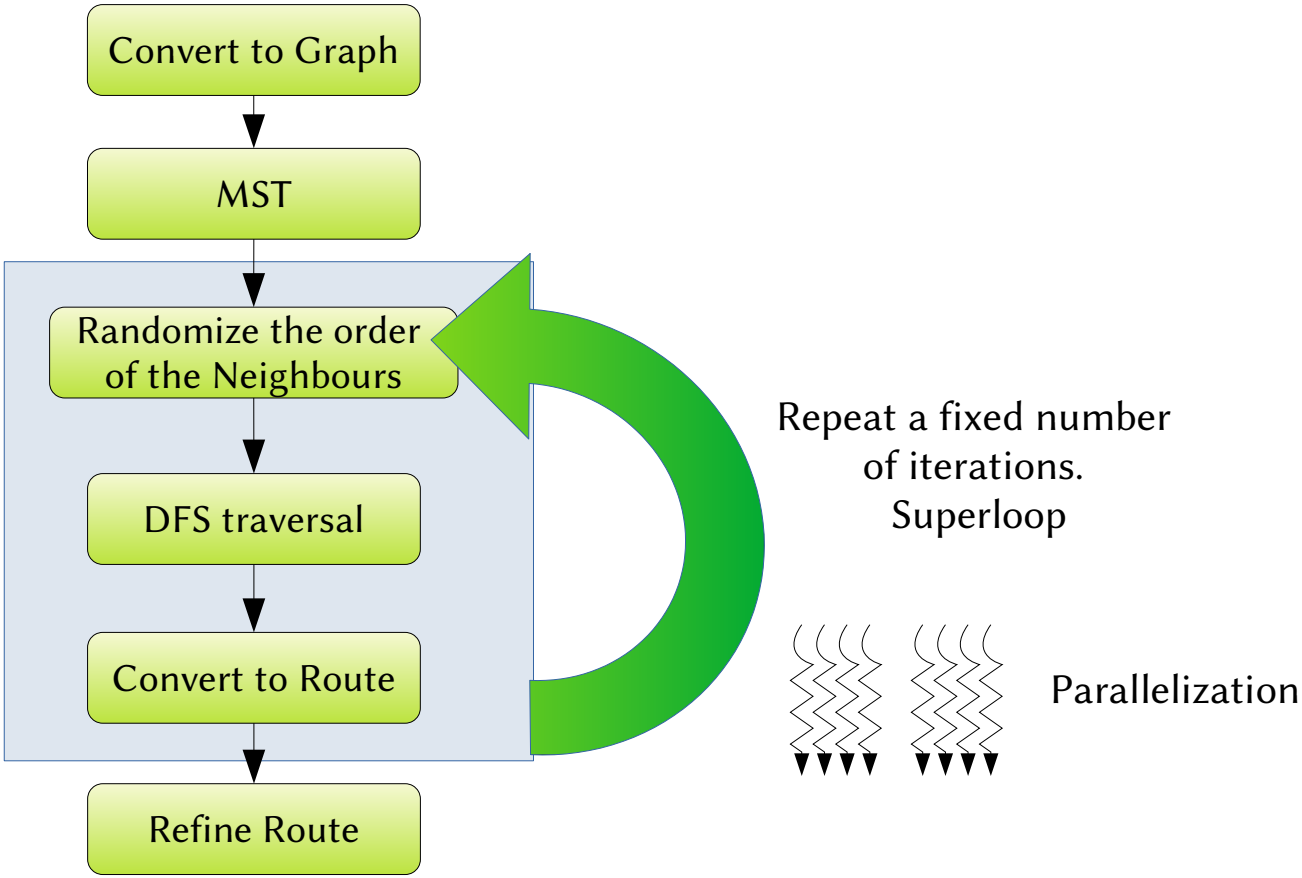
Baseline1: P. Yelmewad and B. Talawar. Parallel Version of Local Search Heuristic Algorithm to Solve Capacitated Vehicle Routing Problem, Cluster Computing, 2021.

Baseline2: M. Abdelatti and M. Sodhi. An improved GPU-accelerated heuristic technique applied to the Capacitated Vehicle Routing Problem, GECCO, 2020.

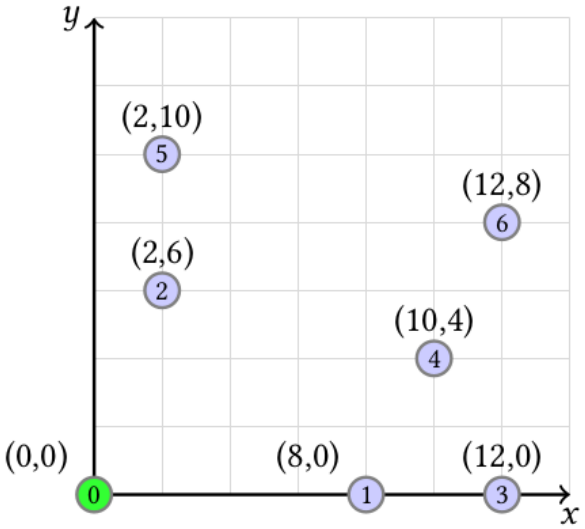
Overview - ParMDS



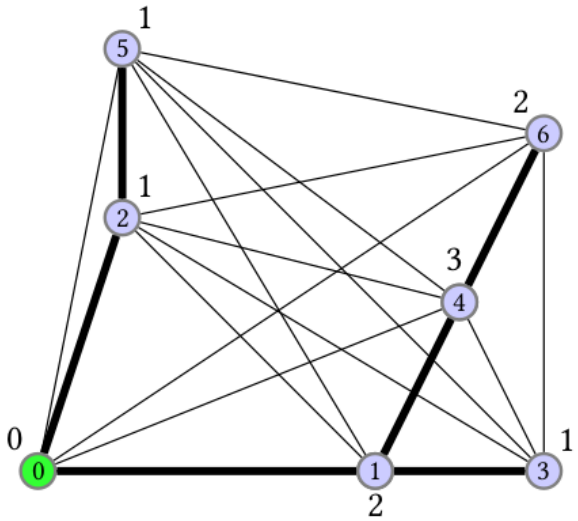
Overview - ParMDS



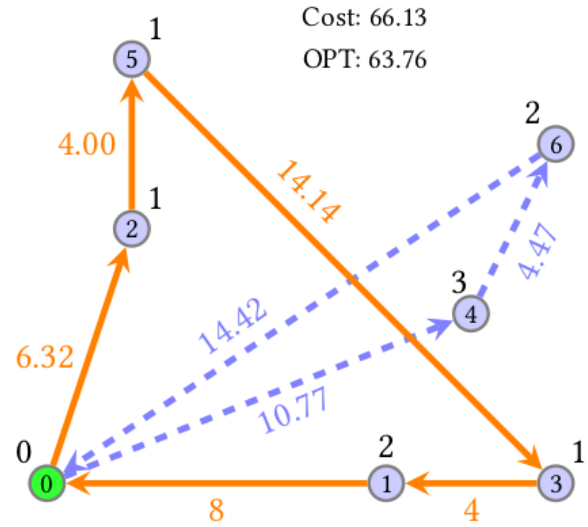
Example - Overview



(a) Input instance I



(b) Graph for I, along with node-demands

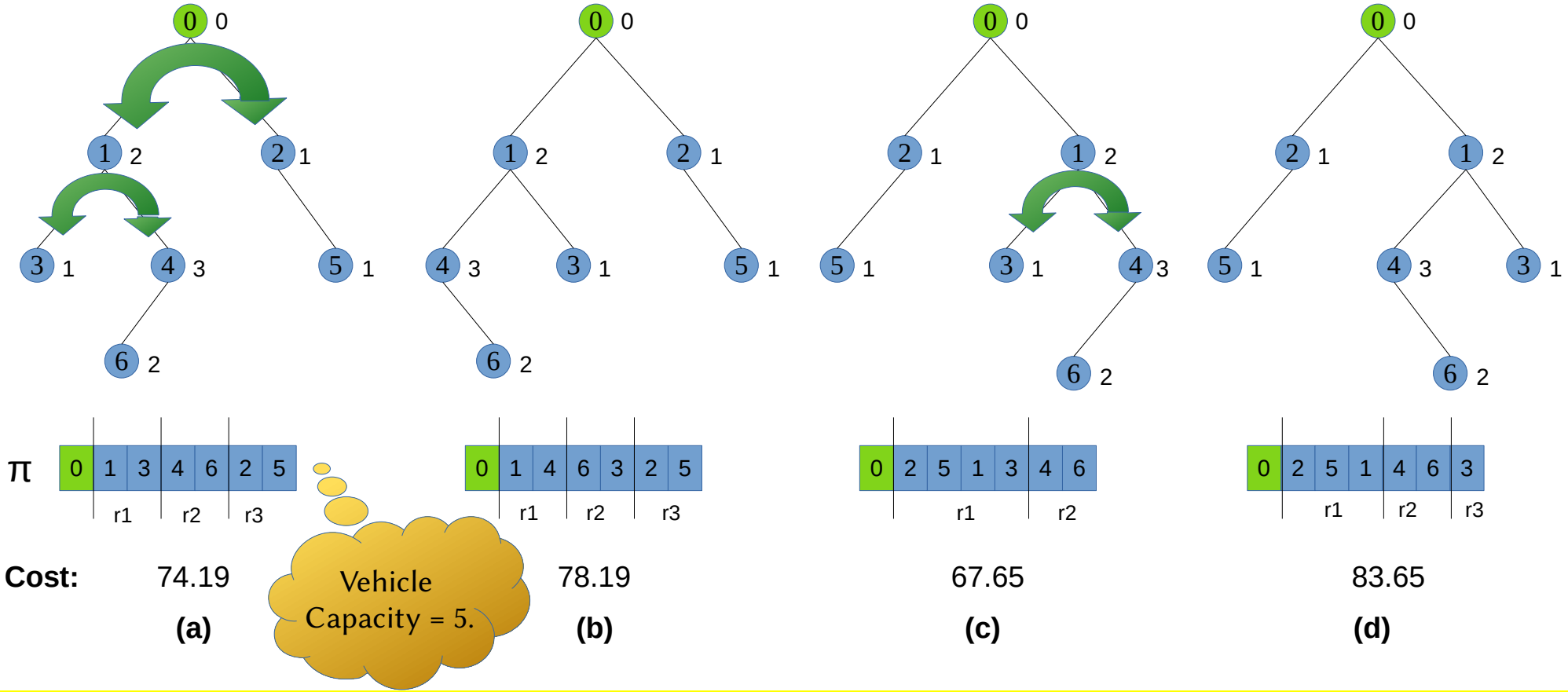


(c) Final routes generated by ParMDS

Closer to Optimal Cost

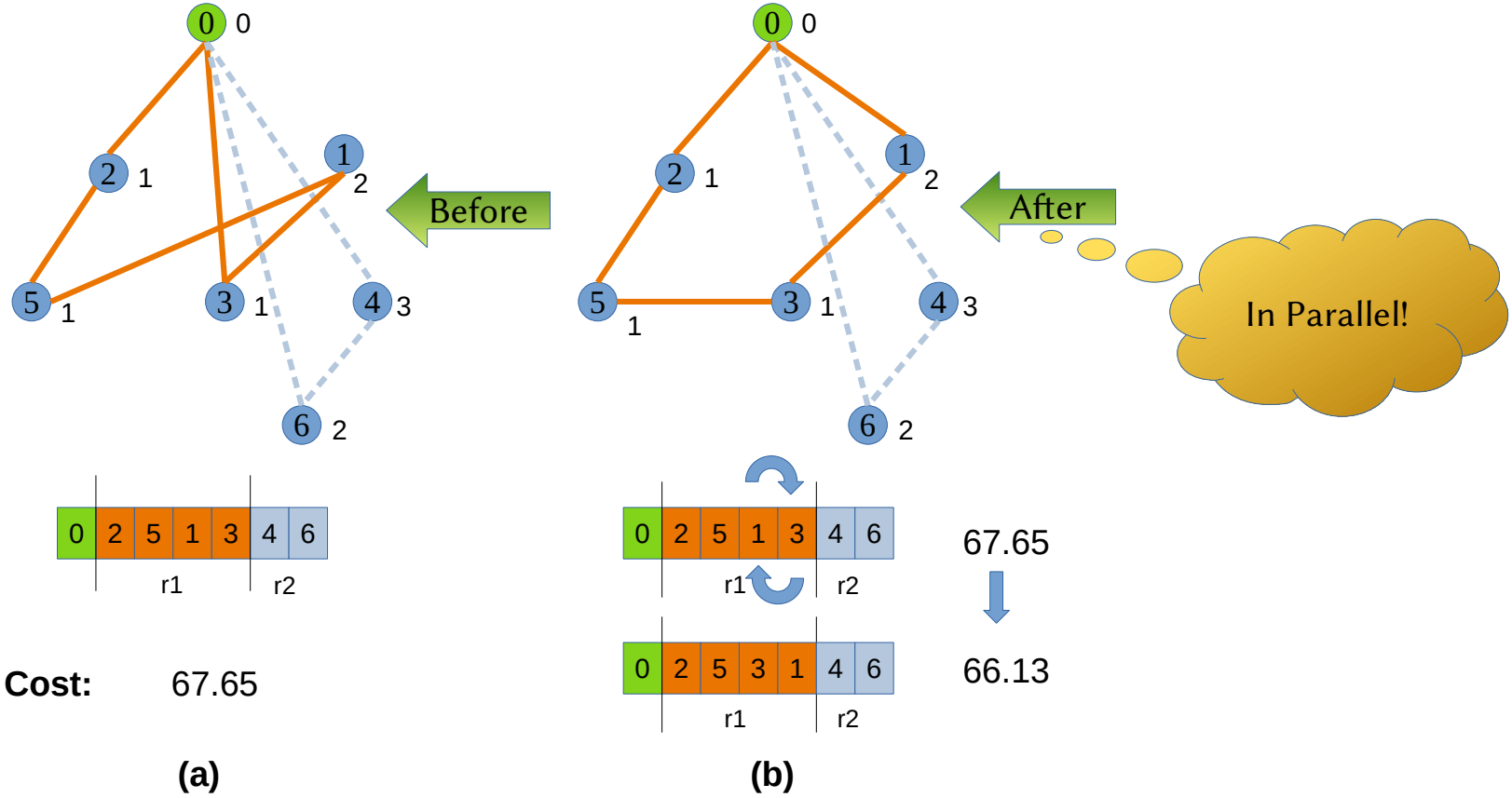
ParMDS on an example input instance with n = 7 and Vehicle Capacity = 5.

Example – DFS and Randomization



Takeaway: Randomizing neighbours of MST may yield a different DFS ordering. Hence, a different route!

Intra-route optimization - 2Opt



ParMDS Algorithm

Input: $G = (V, E)$, Demands $D := \bigcup_{i=1}^{n} d_i$, Capacity Q
Output: R, a collection of routes as a valid CVRP solution
C_R, the cost of R

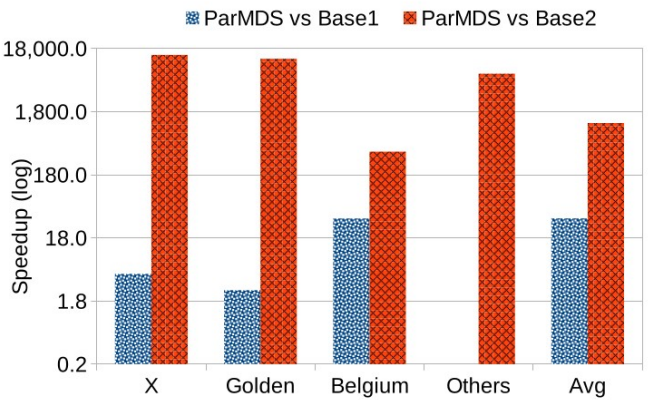
```
1  T ← PRIMS_MST (G)                                   /* Step 1 */
2  C_R ← ∞
3  for i ← 1 to ρ do        /* Superloop */ /* Parallel */
4  |   T_i ← RANDOMIZE (T) /* Shuffle Adjacency List */
5  |   π_i ← DFS_VISIT (T_i, Depot)                    /* Step 2 */
6  |   R_i ← CONVERT_TO_ROUTES (π_i, Q, D)             /* Step 3 */
7  |   C_Ri ← CALCULATE_COST (R_i)                     /* Parallel */
8  |   if C_Ri < C_R then
9  |   |   C_R ← C_Ri                                 /* Current Min Cost */
10 |   |   R' ← R_i                                   /* Current Min Cost Route */
11 |   end
12 end
13 R ← REFINE_ROUTES (R')                              /* Step 4 */
14 return R, C_R
```

Zoom-in

```
/* Standard: stride = 1;                               */
/* Strided : stride = #CPU cores                       */
/* Parallel for loop: Standard/Strided                 */
1 for i ← 1; i ≤ ρ; i = i + stride do
2 |   for v ∈ V do
  |   |   /* seed ← constant or i or rand()            */
3 |   |   SHUFFLE-NEIGHBORS(AdjList(v), seed);
4 |   end
5 |   ...
6 end
7 ...
```

Experiments

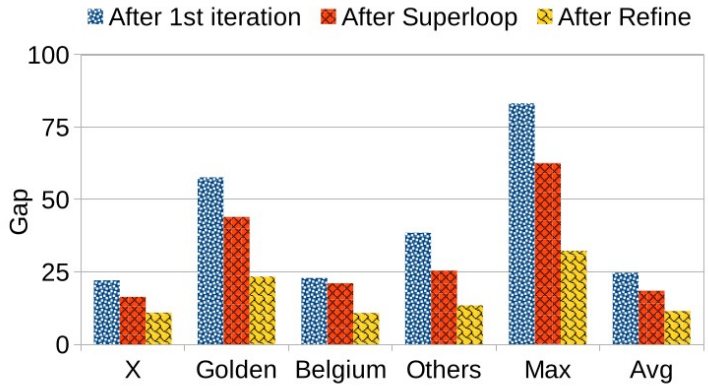
- 130 Instances of CVRPLIB
- Intel Xeon CPU E5-2640 v4
- Baselines on GPU
  - NVIDIA's Tesla P100
  - CUDA 11.5
- Our Code uses
  - **SeqMDS:** GCC 9.3.1
  - **ParMDS:** nvc++ compiler NVIDIA's HPC SDK 22.11



Speedup of ParMDS vs. baselines

Method	Execution Time (s) using Random
SeqMDS	1,722.44
ParMDS-Standard	1,522.26
ParMDS-Strided	186.50

More detailed analysis in our paper



Gap at the end of each step

Summary

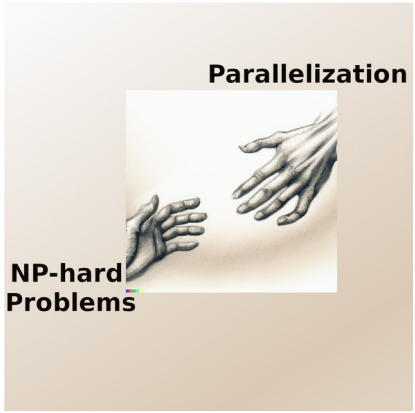
- Fresh perspective to design parallelism-friendly algorithms
- Performance: Algorithmic-, Parallelism- &Platform- specific Optimizations

- Our techniques are applicable in general
  - Two-level parallelism (p-SSSP) technique
  - Strided parallel Local-search

- Promising future directions in our area (next 3-10 years)
  - Paradigm specific parallelization: Greedy, Dynamic Programming
  - Most STL algorithms would run parallel // My Prediction
  - Single source code for muti-core and GPU

Thank you!

Questions?



https://bit.ly/rajesh-talks