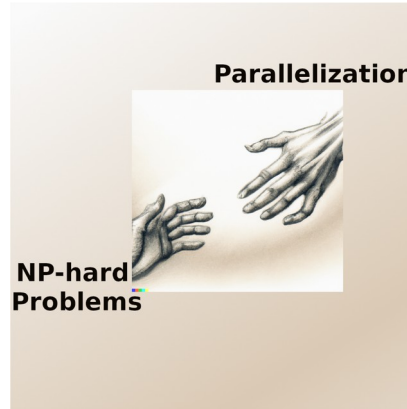


# NP-Hard Problems Meet Parallelization



23-Apr-2024



Rajesh Pandian M

**IIT MADRAS**

Indian Institute of Technology Madras

[www.cse.iitm.ac.in/~mrprajesh](http://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



... 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.

## 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

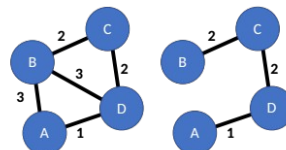
Jarim Seo  
Dept. of Computer Science  
Texas State University  
San Marcos, Texas, USA  
j\_s1195@txstate.edu

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.



SC'23

# 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

**Goal:** Solve largest benchmark instances from DIMACS/PACE Challenges

## 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

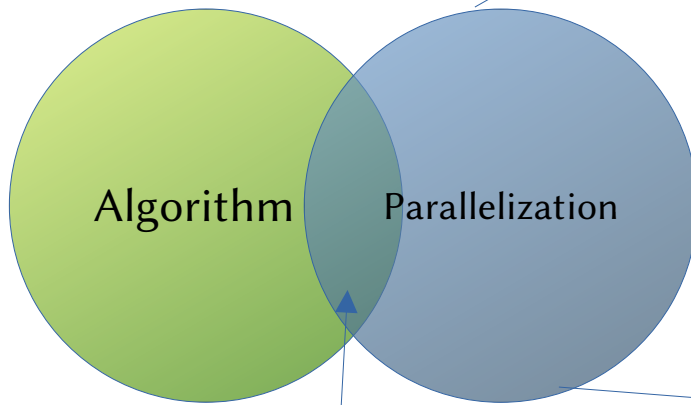
# Algorithms and Optimizations



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

Design parallelism-friendly algorithms

- Optimizations



Peak performance

## Data-driven versus Topology-driven Irregular Computations on GPUs

Rupesh Nare  
The University of Texas  
Austin, Texas, USA  
Email: nare@ices.utexas.edu

Martin Burtcher  
Texas State University  
San Marcos, Texas, USA  
Email: burtcher@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

**Abstract**—main data trees, etc. A its operator, iterated app nodes, in the usually time overlapping drives impl the operator no work to mentations t might be w algorithms i only support important. ( a topology counterhala In this p implementation goal is to um and how to generally fa a worklist, when certain implement that combin  
**Keywords**: i algorithmic

**Abstract**—The performance of graph programs depends highly on the algorithm, the size and structure of the input graphs, as well as the features of the underlying hardware. No single set of optimizations or one hardware platform works well across all settings. To achieve high performance, the programmer must carefully select which set of optimizations and hardware platforms to use. The GraphIt programming language makes it easy for the programmer to write the algorithm once and optimize it for different inputs using a scheduling language. However, GraphIt currently has no support for generating high-performance code for GPUs. Programmers must resort to re-implementing the entire algorithm from scratch in a low-level language with an entirely different set of abstractions and optimizations in order to achieve high performance on GPUs.

We propose G2, an extension to the GraphIt compiler framework, that achieves high performance on both CPUs and GPUs using the same algorithm specification. G2 significantly expands the optimization space of GPU graph processing frameworks with a novel GPU scheduling language and compiler that enables combining load balancing, edge traversal direction, active vertexset creation, active vertexset processing ordering, and kernel fusion optimizations. G2 also introduces two performance optimizations, Edge-based Thread Warps CTAs load balancing (ETWC) and EdgeBlocking, to expand the optimization space for GPUs. ETWC improves load balancing by dynamically partitioning the edges of each vertex into blocks that are assigned to threads, warps, and CTAs for execution. EdgeBlocking improves the locality of the program by reordering the edges and restricting random memory accesses to fit within the L2 cache. We evaluate G2 on 5 algorithms and 9 input graphs on both Pascal and Volta generation NVIDIA GPUs, and show that it achieves up to 5.11x speedup over state-of-the-art GPU graph processing frameworks, and is the fastest on 66 out of the 90 experiments.

**Index Terms**—Compiler Optimizations, Graph Processing, GPUs, Domain-Specific Languages

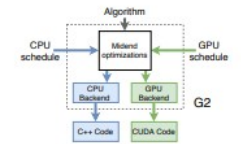


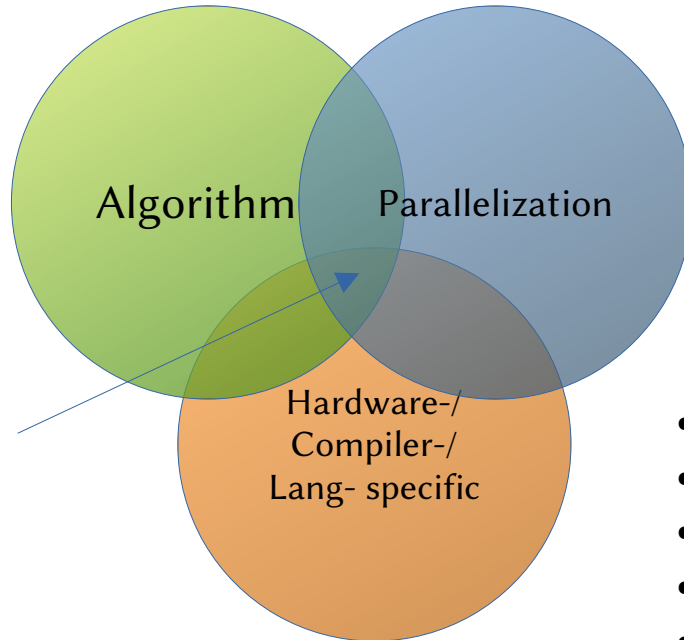
Fig. 1. G2 adds a new GPU backend to GraphIt that produces CUDA from the same GraphIt algorithm language and a scheduling language tail for GPU optimizations.

In prior work, we built the GraphIt DSL compiler [7] to generate high-performance CPU code from a high-level algorithm language. GraphIt achieves state-of-the-art performance on CPUs across different algorithms and graph inputs by introducing a scheduling language to tune optimizations. The algorithm language has primitives for topology-driven algorithms, data-driven algorithms, and priority-based algorithms. This algorithm and schedule representation makes it easy for the programmer to write an algorithm once and generate different highly-optimized implementations by simply changing the schedule. We will discuss in detail the Graph algorithm and scheduling languages in Section IV.

Apart from a large body of work for optimizing algorithms on different inputs on CPUs [6], [8], [9], [10], [11], [13], [14], [15], researchers have also used different hardware platforms for high-performance graph processing, including GPUs [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69], [70], [71], [72], [73], [74], [75], [76], [77], [78], [79], [80], [81], [82], [83], [84], [85], [86], [87], [88], [89], [90].

# Optimizing for peak performance

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

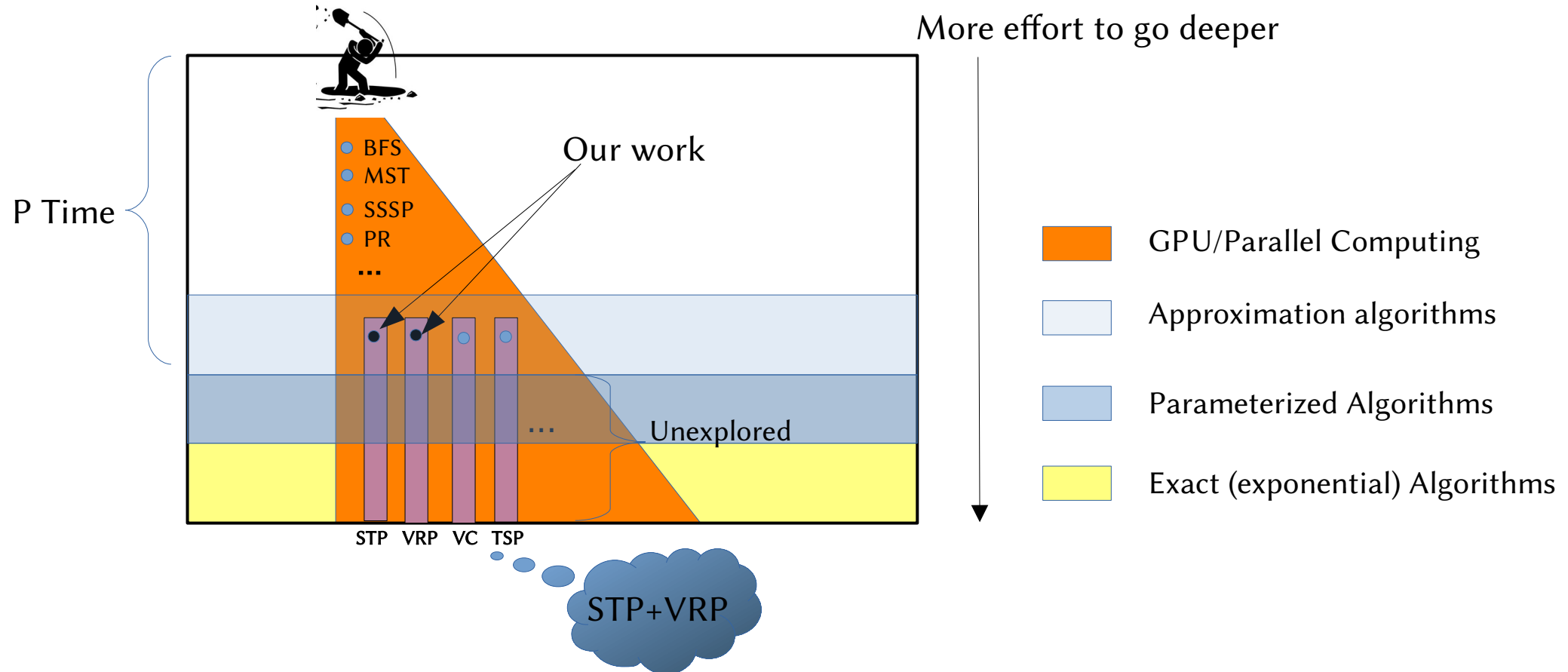


Peak performance

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

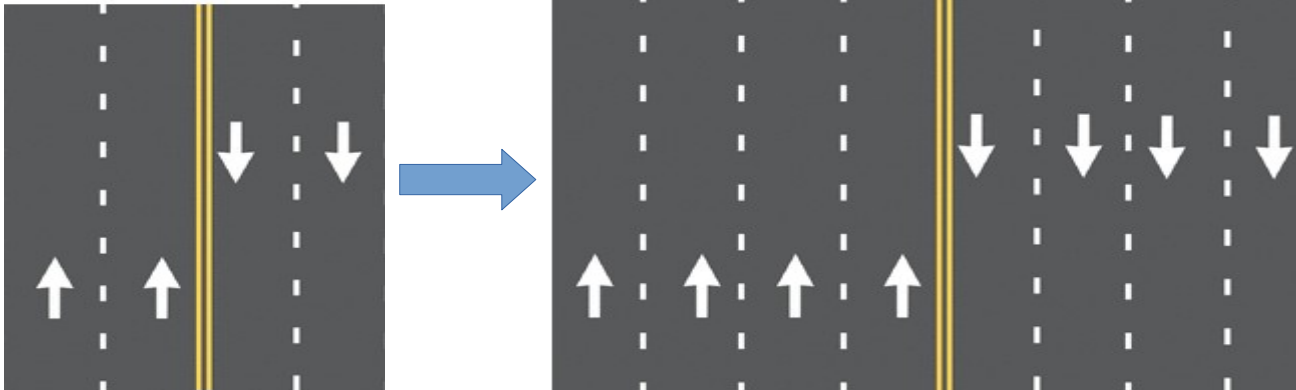
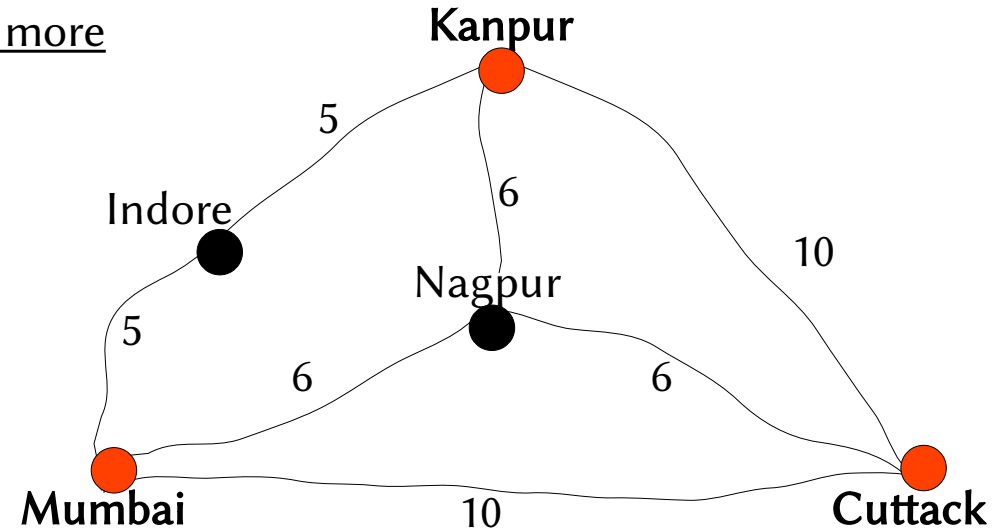
- Architecture-specific
- Shared memory
- Warp-Intrinsics
- Data accesses within reg/caches
- Vectorization loads/adds
- Language-specific 128b CAS CC9+
- Use profilers and <https://godbolt.org>

# 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  $\propto$  Distance





# Steiner Tree Problem (STP) – Example

**Input** : Graph  $G(V, E, W)$   $W:E \rightarrow \mathbb{Z}^+$  and  $L \subseteq V$  terminals.

**Output**: A tree  $T'(V' \supseteq L, E' \subseteq E)$  of  $G$  such that minimize  $W(E')$ .

// Minimum weighted tree with all terminals.

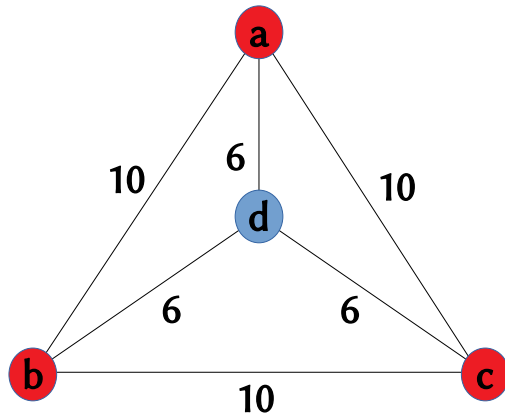


Fig 1 (a)

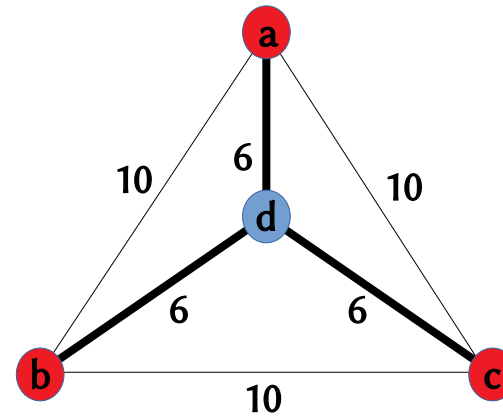


Fig 1 (b)



● Terminals  
● Non-terminals

# Steiner Tree Problem (STP) – Example

Input : Graph  $G(V, E, W)$   $W:E \rightarrow \mathbb{Z}^+$  and  $L \subseteq V$  terminals

Output : Minimum weighted tree with all terminals

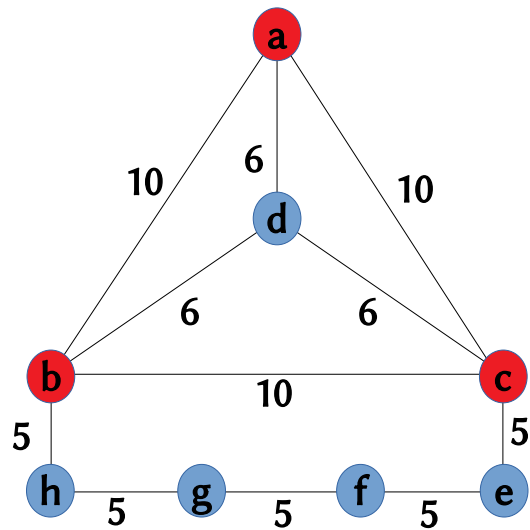


Fig 2 (a)

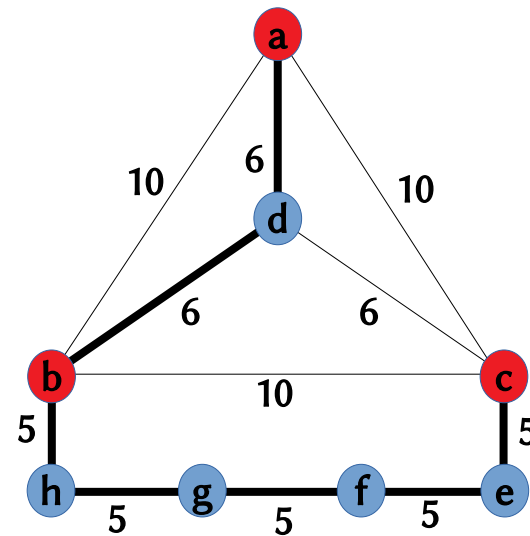


Fig 2 (b)

$|MST| = 37$   
 $|OPT| = 18$

● Terminals  
● Non-terminals

# Steiner Tree Problem (STP) - Hardness

Input : Graph  $G(V, E, W)$   $W:E \rightarrow \mathbb{Z}^+$  and  $L \subseteq V$  terminals

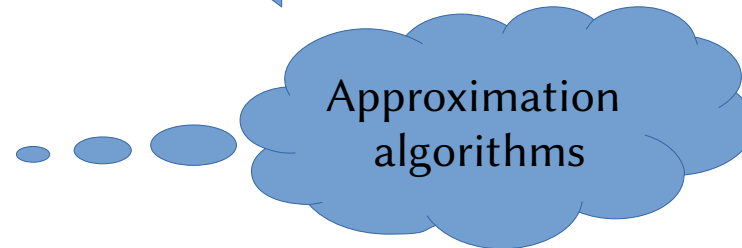
Output : Minimum weighted tree with all terminals

## 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 general      STP is NP-Hard



# 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]

$$|ALG| \leq 2 |OPT|$$



KMB

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  $G$

$\text{MST}(G')$

## Phase 3

// Construct  $G''$

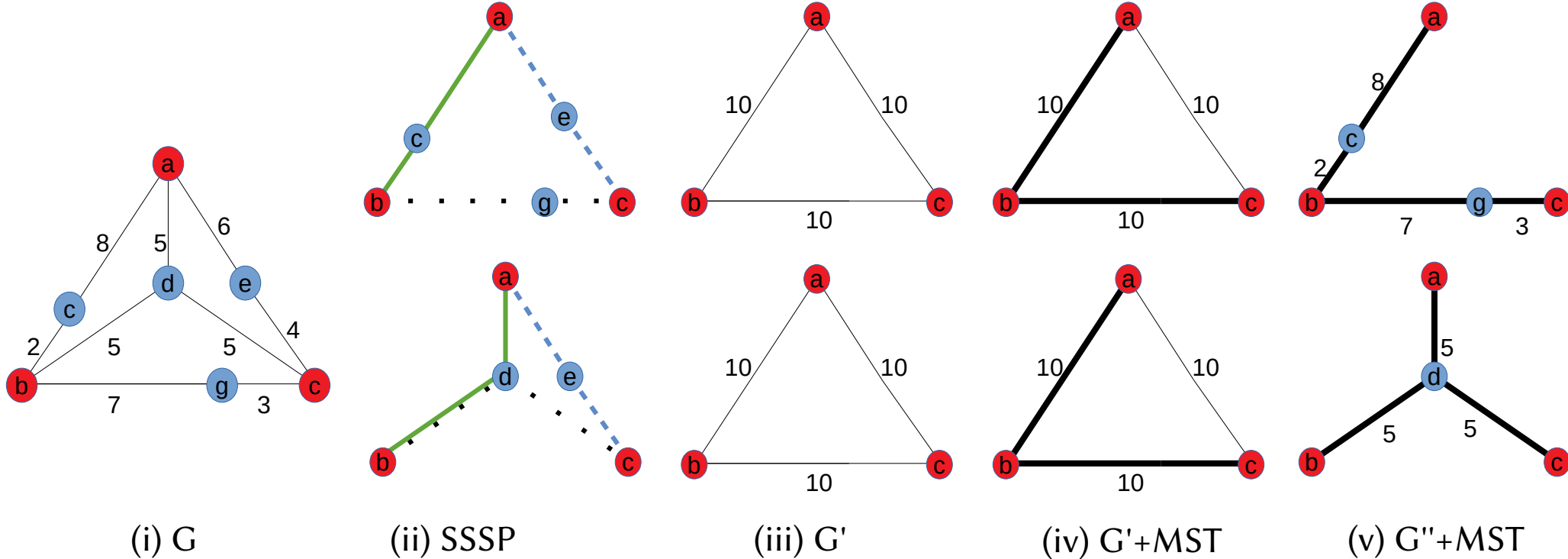
For every edge in  $\text{MST}(G')$  substitute the edges with the corresponding shortest path in  $G$

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

$\text{MST}(G'')$

Takeaway: One more invocation for SSSP/MST algorithm.  $G \rightarrow G' \rightarrow G''$

# KMB Algorithm – Running example



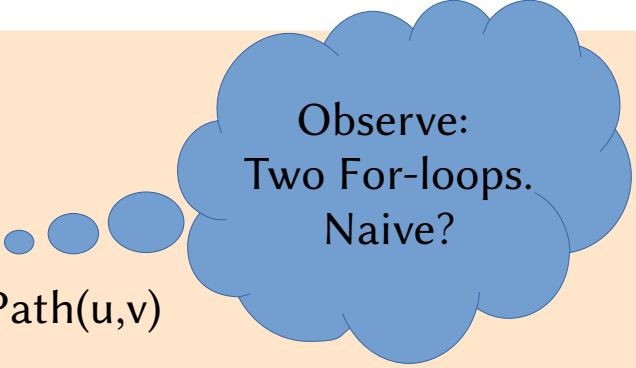
1) A parallel KMB may output a different answer. (2) Last MST may be avoided

**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 {  
     $P_{uv} = \text{ShortestPath}(u, v)$   
     $W'(u, v) = |P_{uv}|$   
  }  
}  
T' = \text{MST}(G', W')
```



Observe:  
Two For-loops.  
Naive?

## Phase 3

```
For (u,v) in edges of T' {  
   $G'' = G'' \cup P_{uv}$   
  //Add vertices & edges of  $P_{uv}$   
}  
  
T'' = \text{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  $u \in L$  {

SSSP( $G, W, L, u$ ) with Halt

Compute  $W'$  incrementally

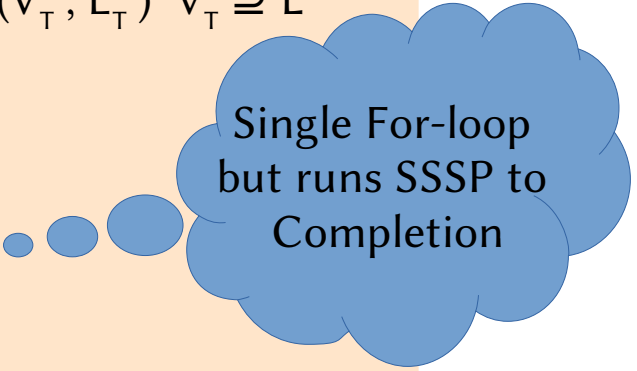
}

$T' = \text{MST}(G', W')$

Compute  $G''$  and its vertices, adjList using  $T'$

$T'' = \text{MST}(G'', W)$

return  $T''$



Single For-loop  
but runs SSSP to  
Completion



# CPU Implementation - Optimization

- SSSP-halt optimization

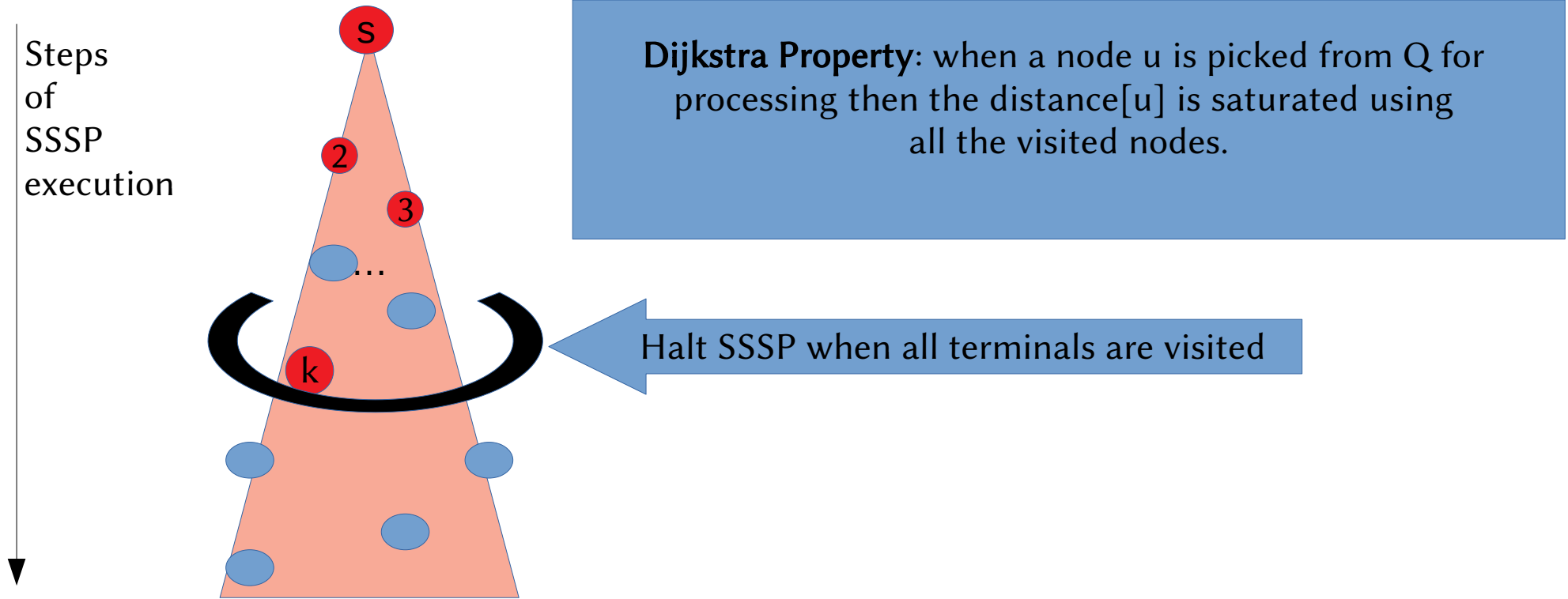


Fig. 4 SSSP-halt visualization

# 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  $u \in L$  {

**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

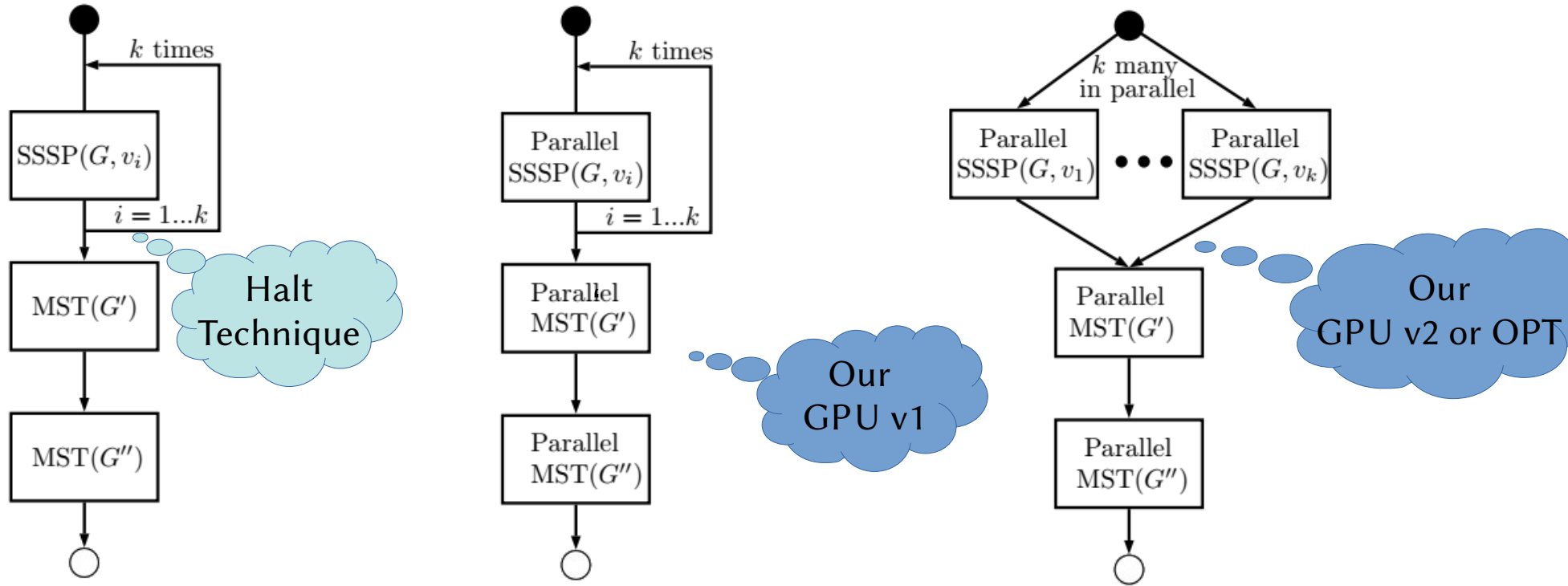
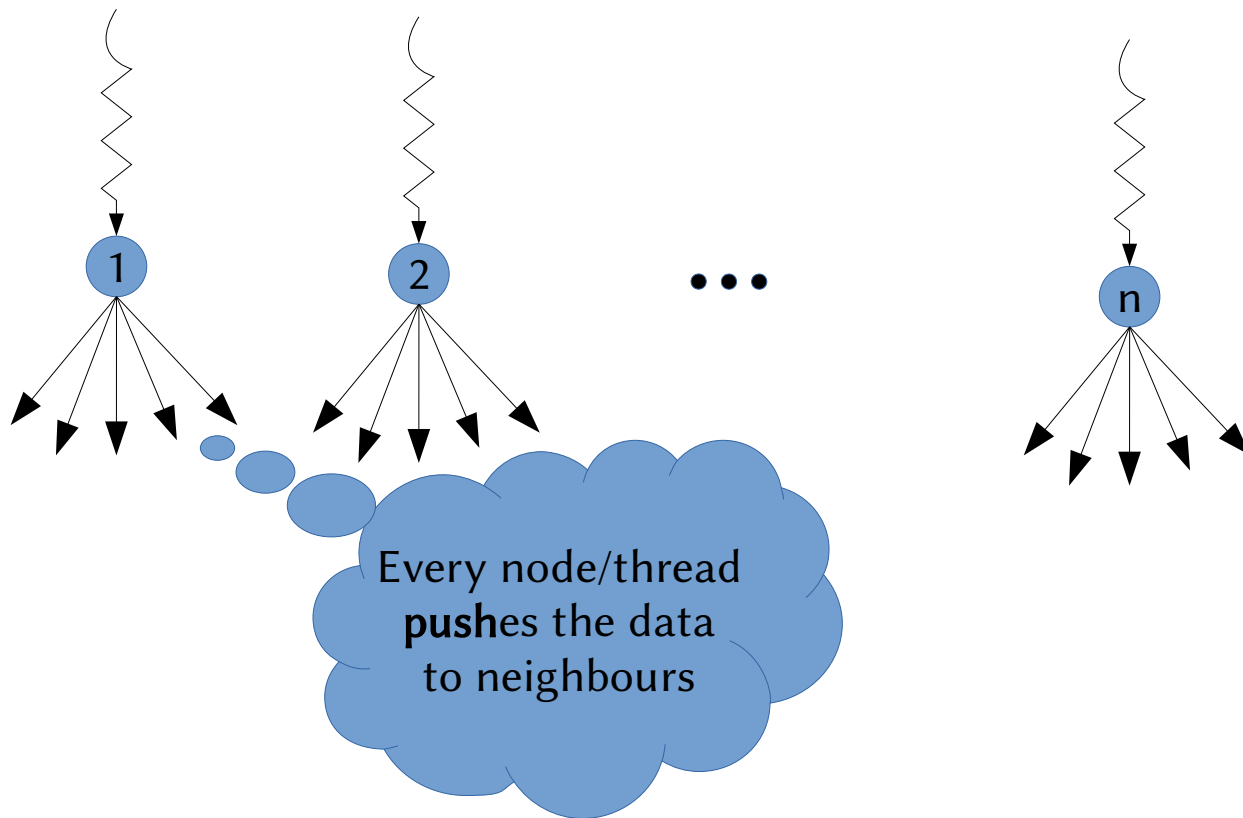


Fig. 5 Sequential vs GPU v1 vs GPU v2

KMBCPU  
KMBGPU

# GPU Implementation - SSSP



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

Fig. 6 push-SSSP

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

## MAIN

```
For s in L {  
  ThdsPerBlk = 512; // or 1024  
  Blks =  $\lceil n / \text{ThdsPerBlk} \rceil$ ;  
  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, ds, ps, changed, n) {
  u = tid // compute tid;
  If tid < n {
    For v ∈ adjacent[u] { // Using CSR arrays
      // Relax Operation (u, v, W(u,v))
      newCost = ds[u] + W(u, v);
      old = ds[v];
      If newCost < old
        Atomic-MIN(ds[v], newCost);
      // Updates Parent array
      If Atomic-MIN is success {
        ps[v] = u;
        changed = true;
      }
    }
  }
}

```

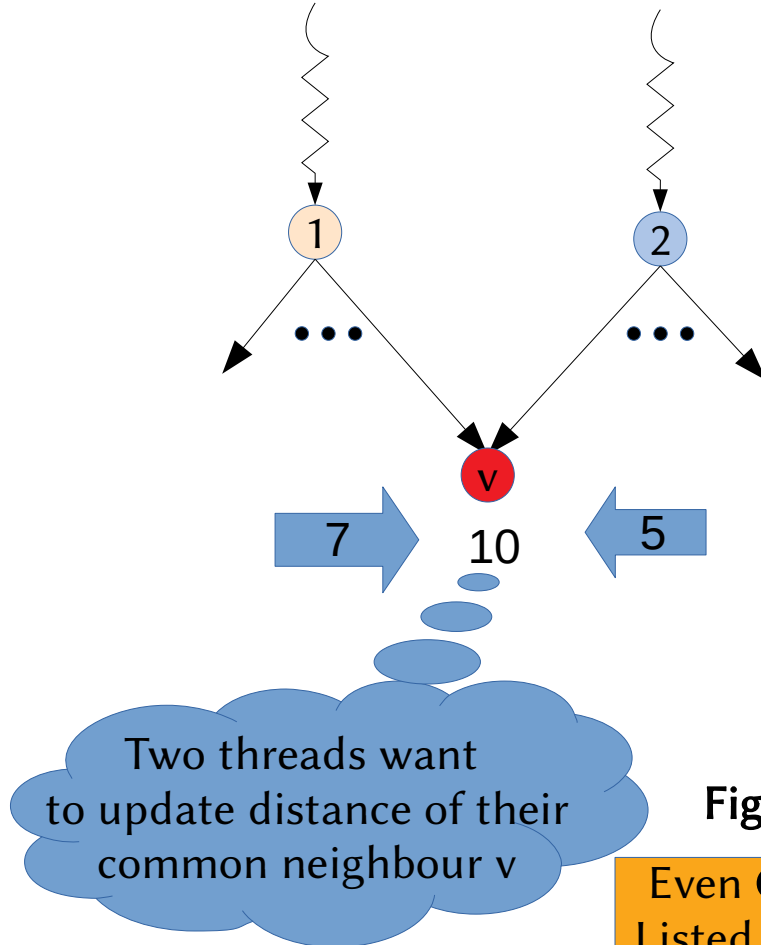
Note :

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



Is it enough?

# Parent update - Challenge



<snip>

..

$\text{newCost} = d_s[u] + W(u, v);$

$\text{old} = d_s[v];$

**If**  $\text{newCost} < \text{old}$

Atomic-MIN( $d_s[v]$ ,  $\text{newCost}$ );

// 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[]

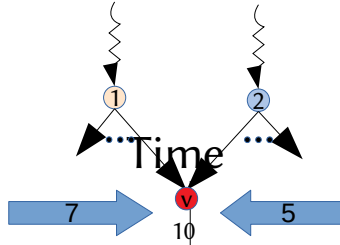
Thread 1

newCost=7  
old=10

d[v]=7 //oldA=10

p[v] = 1.

Wrong!



Thread 2

newCost=5  
old=10

d[v]=5 //oldA=7

p[v]=2

<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

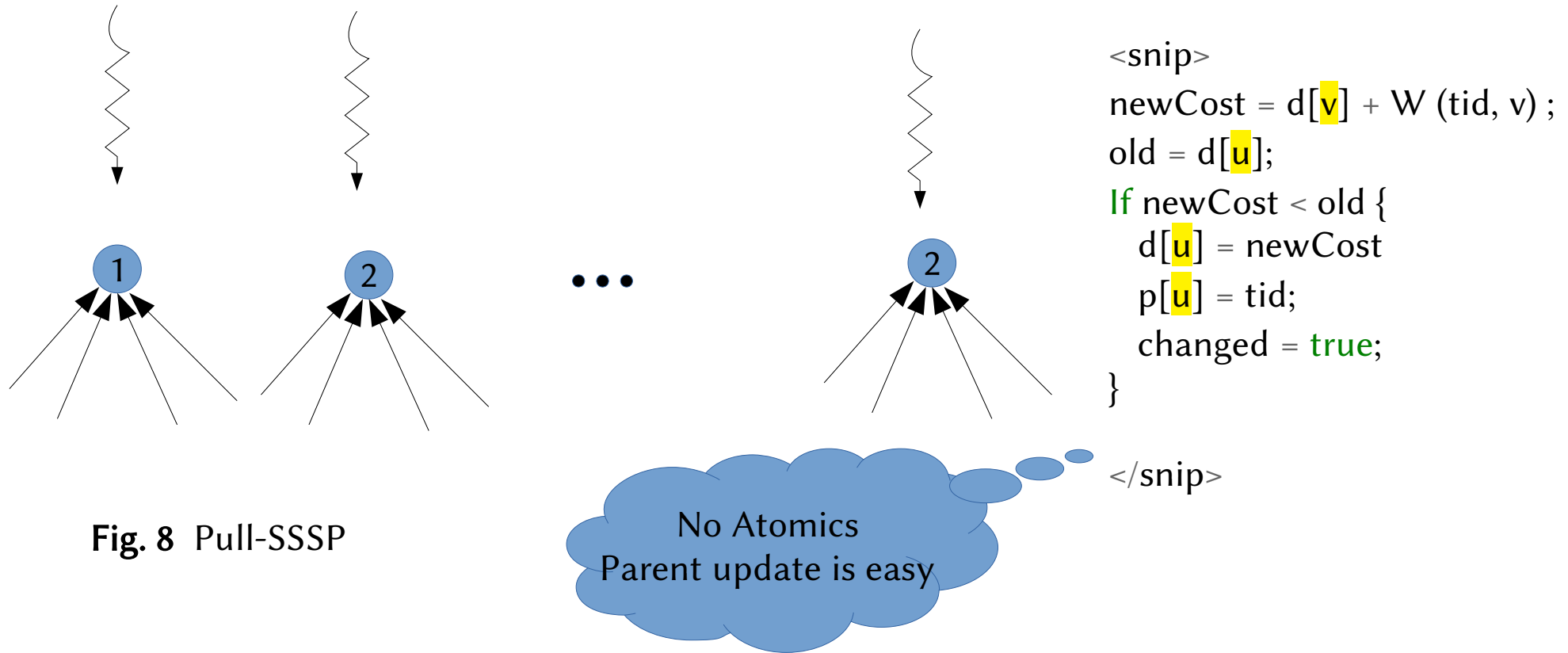


Fig. 8 Pull-SSSP

Because, one thread is writing to an index

# GPU Optimizations

---

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

$\Delta$  – max degree of the graph

# GPU Optimizations

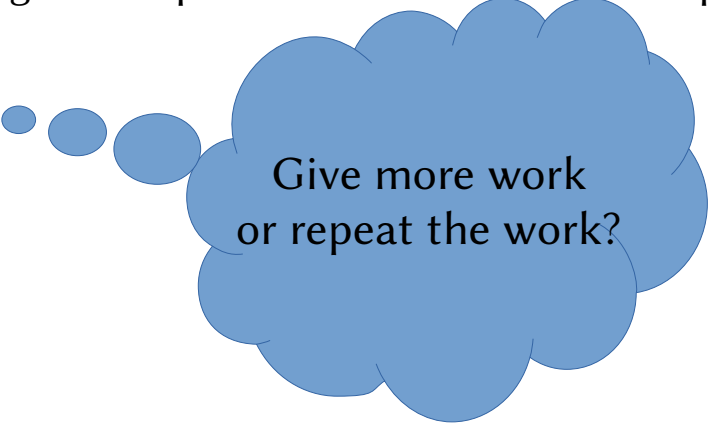
- Synchronization
  - Push
  - **Pull**
- Computation
  - Data-driven
  - Edge-based
  - **Controlled Computation unrolling**
    - $\Delta^2$
    - $2\Delta$
    - **$t\Delta$**
- Memory
  - **Shared memory**



$\Delta$  – max degree of the graph

# Compute optimization

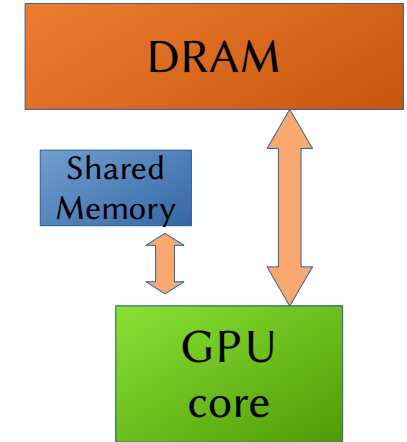
- Computation Unrolling
  - Instead of one thread doing  $\Delta$  work, perform more work per thread
  - Update also neighbours of neighbours ( $\Delta^2$ )
  - **Repeat the work**; Say 2 times or  $t$  times ( $2\Delta$  or  $t\Delta$ ); e.g. we do pull 3 times in the kernel – 3-pull
- Data-driven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



Give more work  
or repeat the work?

# 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  $\text{degree}(\text{node}) < 25$  we use shared, we move  $\text{CSR AdjList}[\text{node}]$  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( $d = \text{INTMAX}$ )

$d[\text{src}] = 0$

FixedPoint{

doRELAX( $G, d, \text{changed} \dots$ );

}



Result Array:  $d[2n]$

Initialize( $d = \text{INTMAX}$ )

$d[\text{src1}] = 0$ ;  $d[n + \text{src2}] = 0$

FixedPoint{

doRELAX( $G, \text{dist}, \text{changed}, \dots$ );

}

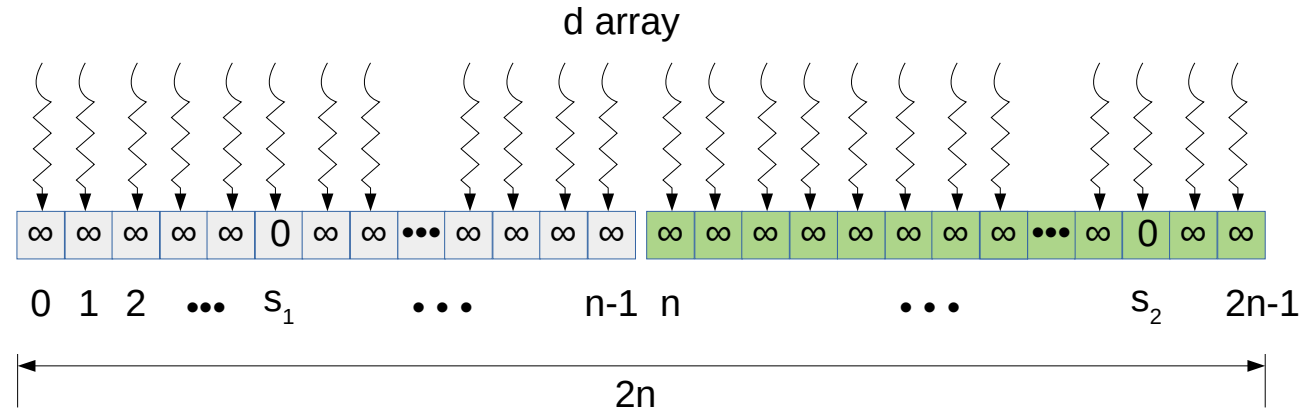


Fig. 9 Double-barrel approach.

Tunable  
arbitrary value

# 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

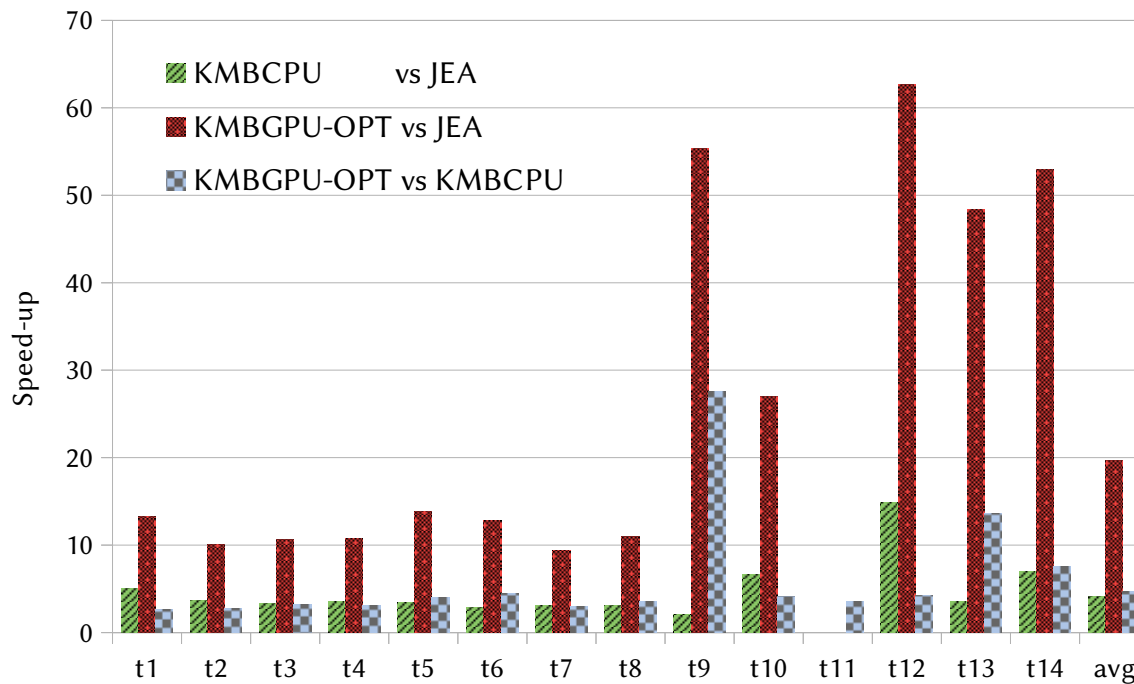
- Total 14 Graphs
  - 11 from PACE Challenge [PACE2018]
  - 2 from SteinLib
  - 1 from SNAP

## 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.

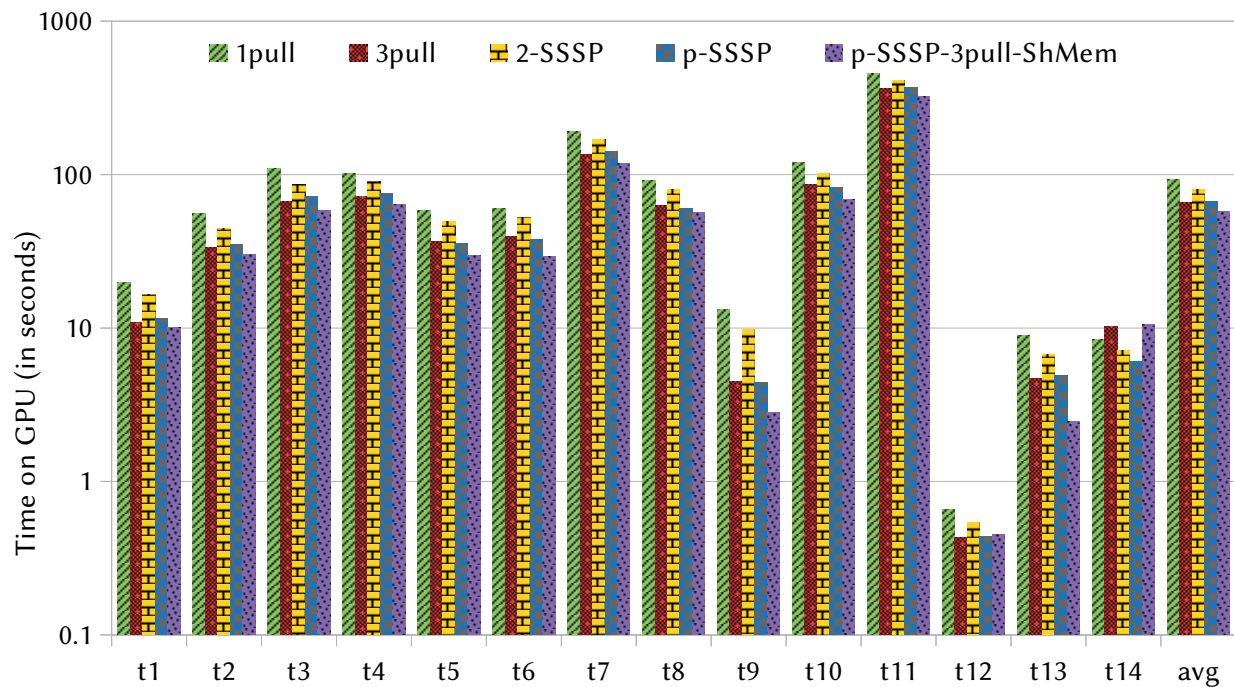
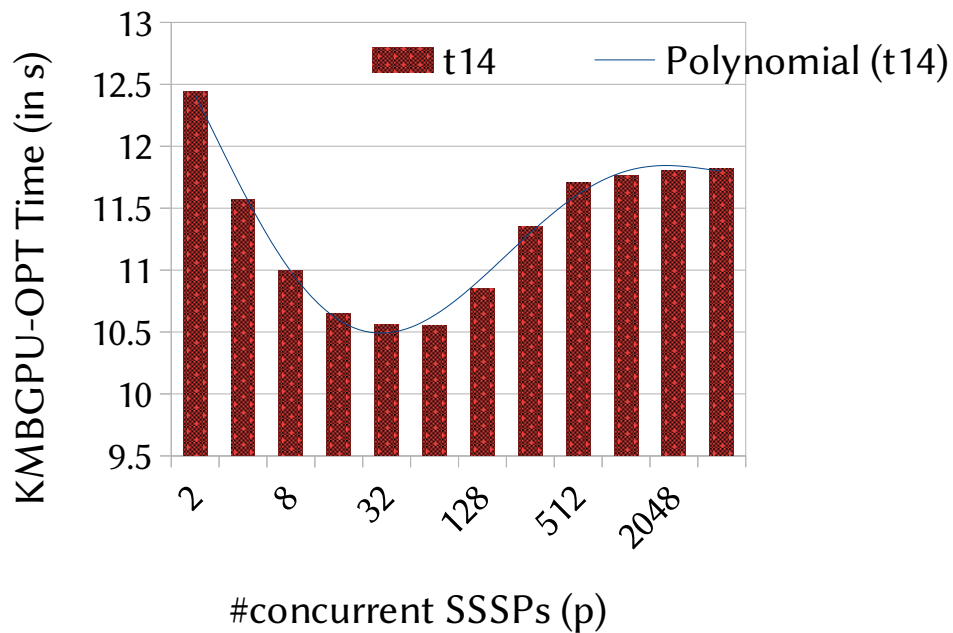


Fig. 11 Comparison of 1-Pull, 3-Pull, Double-barrel & KMBGPU-OPT

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

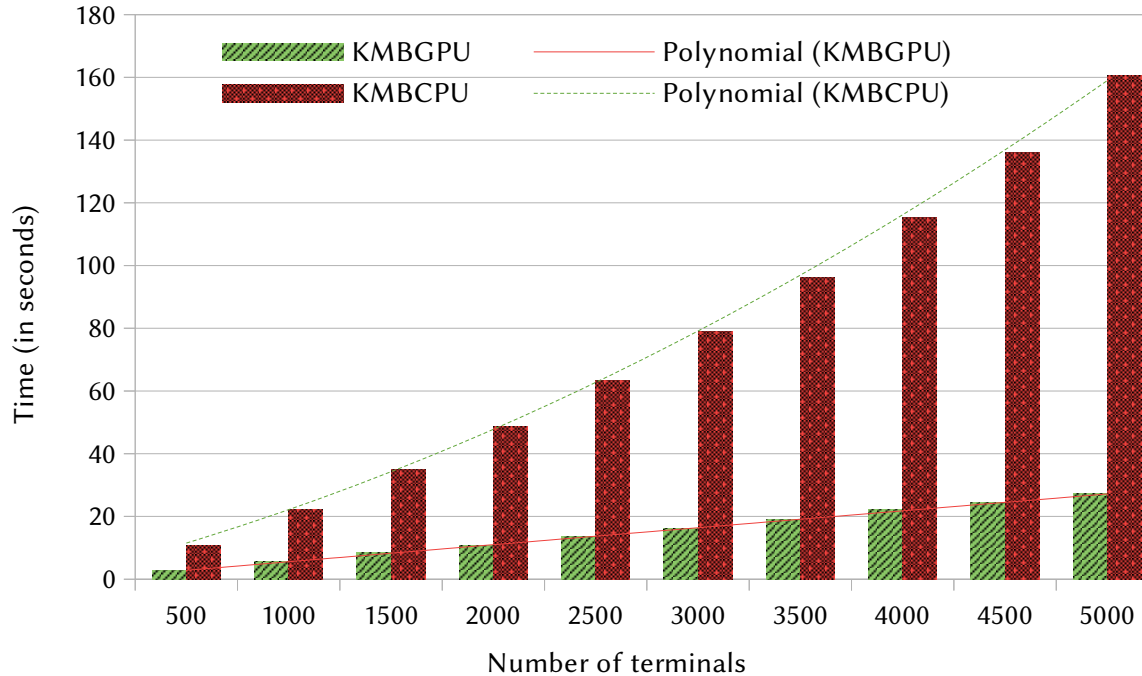
# Comparison of p-SSSP



**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.

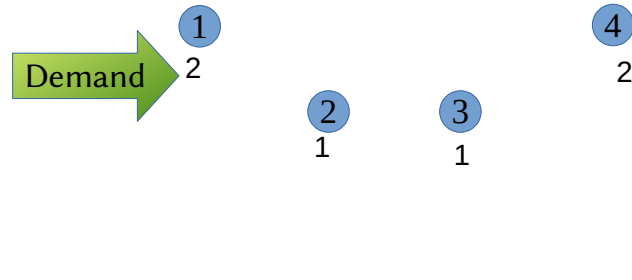
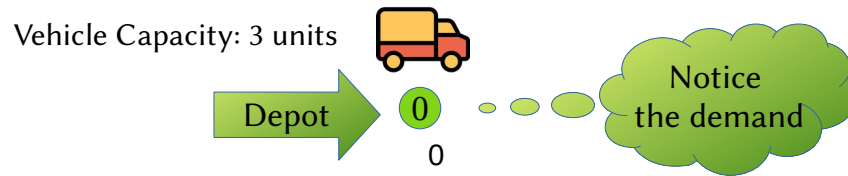
# Capacitated Vehicle Routing Problem (CVRP)



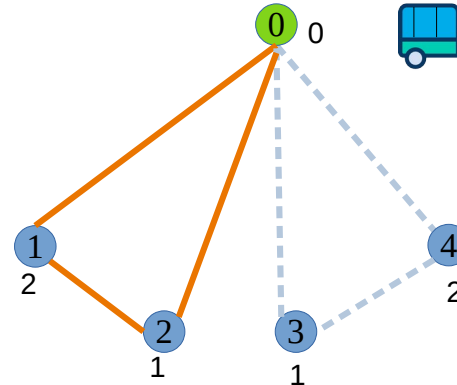
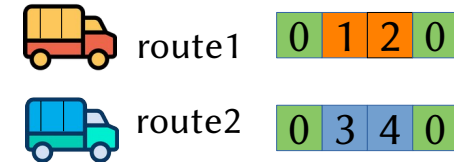
**Input** : Given  $n$  nodes (single Depot and customers) with their coordinates  $(x_i, y_i)$  and demands  $d_i > 0$  for  $i \in n$ , Vehicle capacity  $C$ . Node 0 is Depot and has zero demand.

**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  $\rightarrow$  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 **Par**allel implementation
- Combining **M**ST and **D**FS
- Uses Local-search approach
- Uses Randomization approach

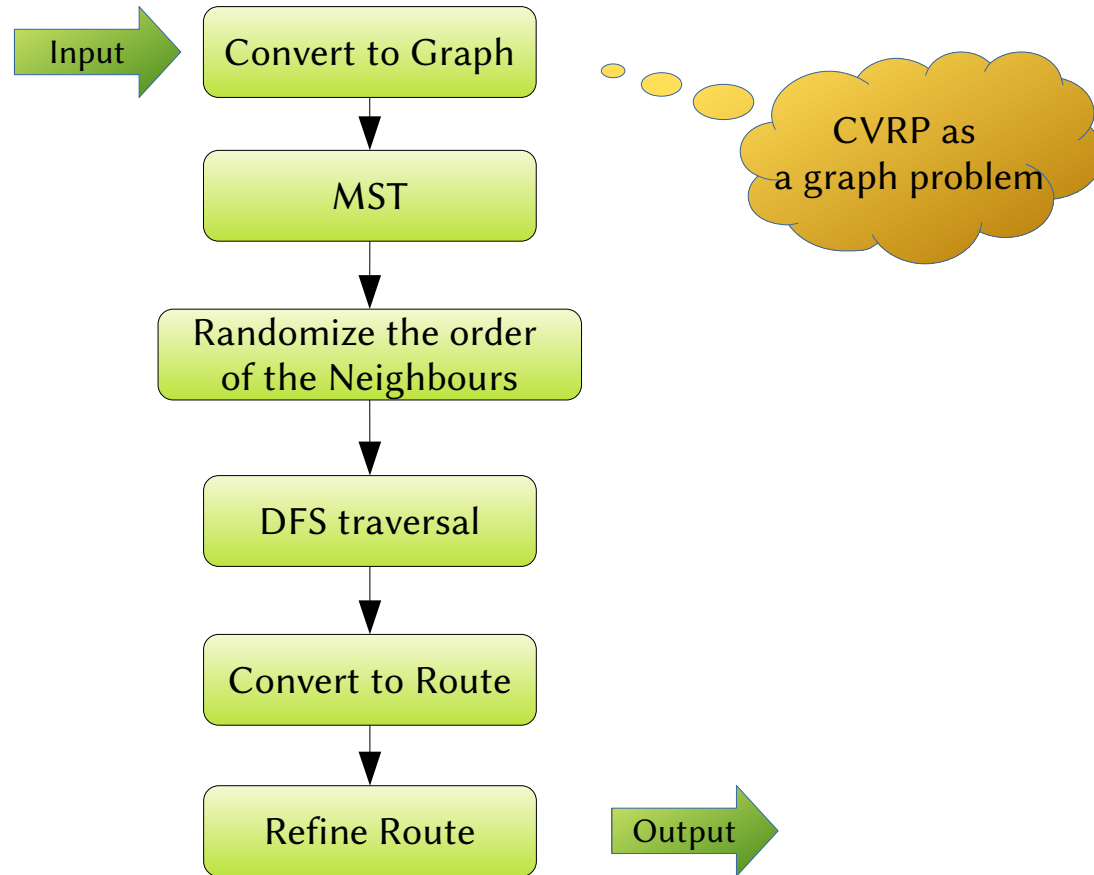
$$Gap = \frac{Z_S - Z_{BKS}}{Z_{BKS}} \times 100$$

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

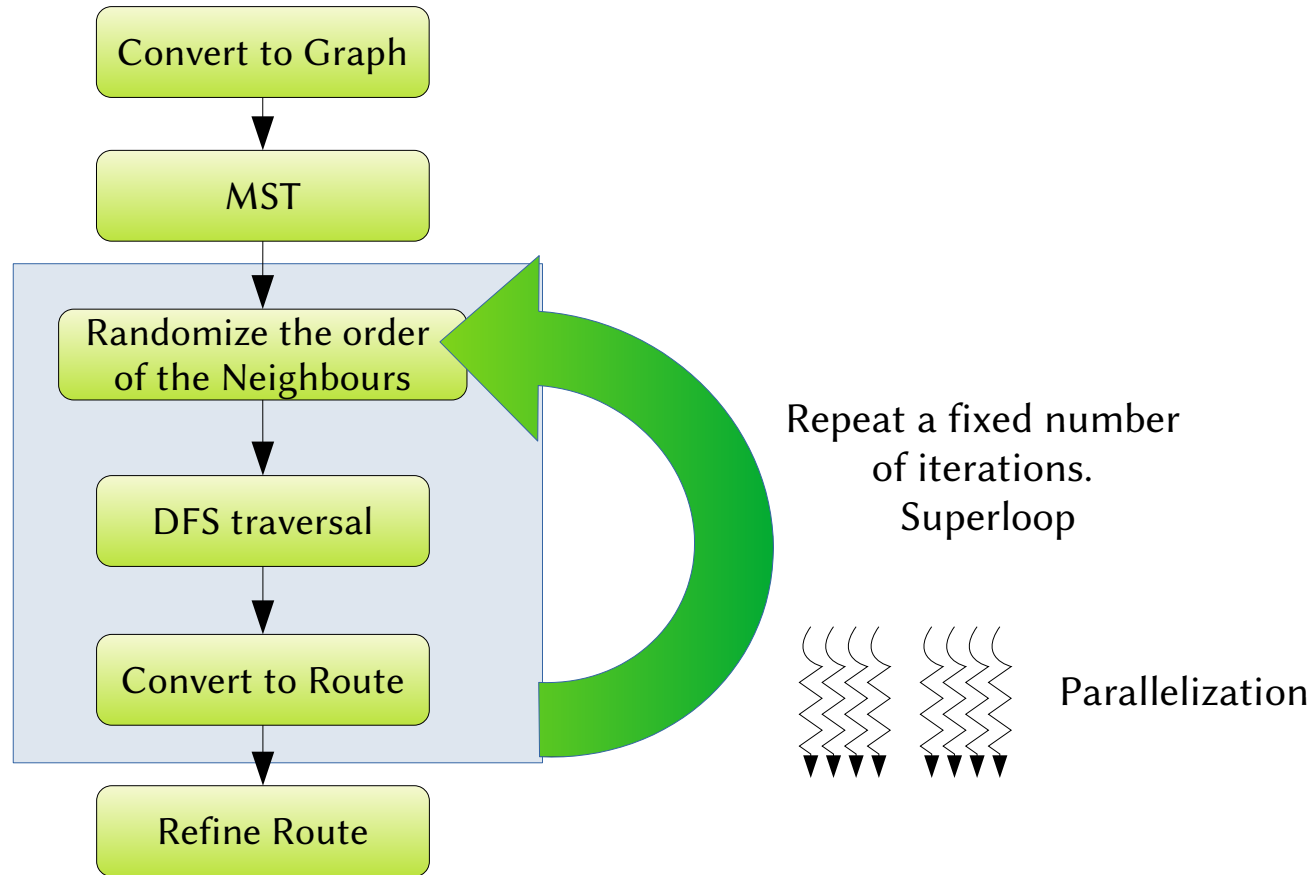
**Baseline2:** M. Abdelatti and M. Sodhi. An improved GPU-accelerated heuristic technique applied to the Capacitated Vehicle Routing Problem, GECCO, 2020.



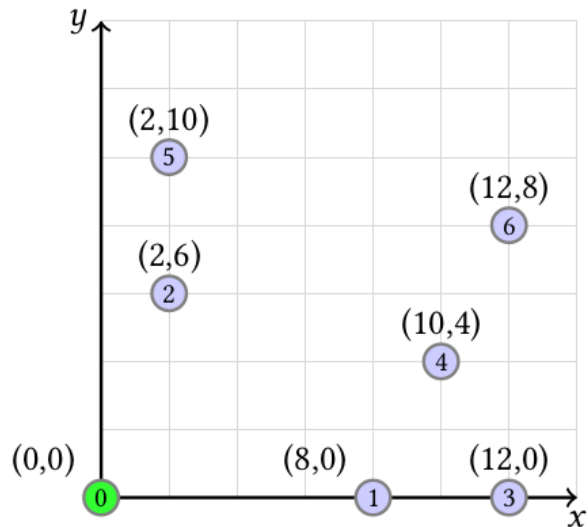
# Overview - ParMDS



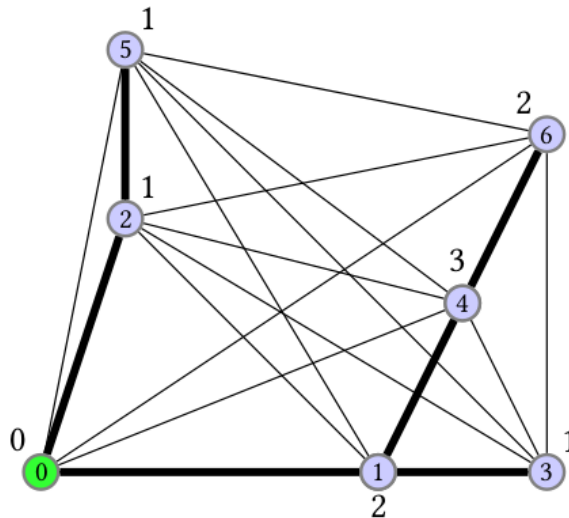
# Overview - ParMDS



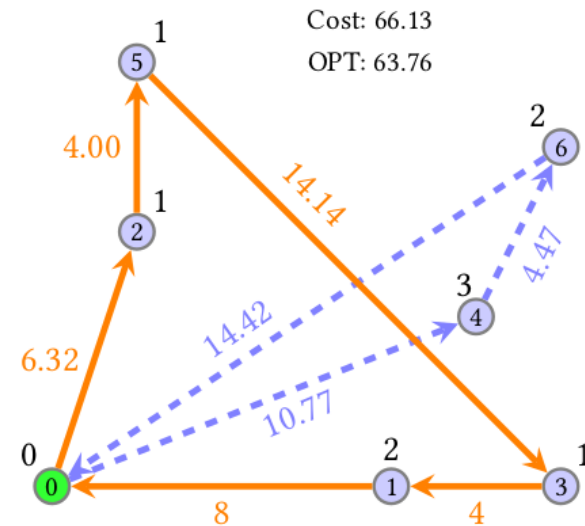
# Example - Overview



(a) Input instance  $I$



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

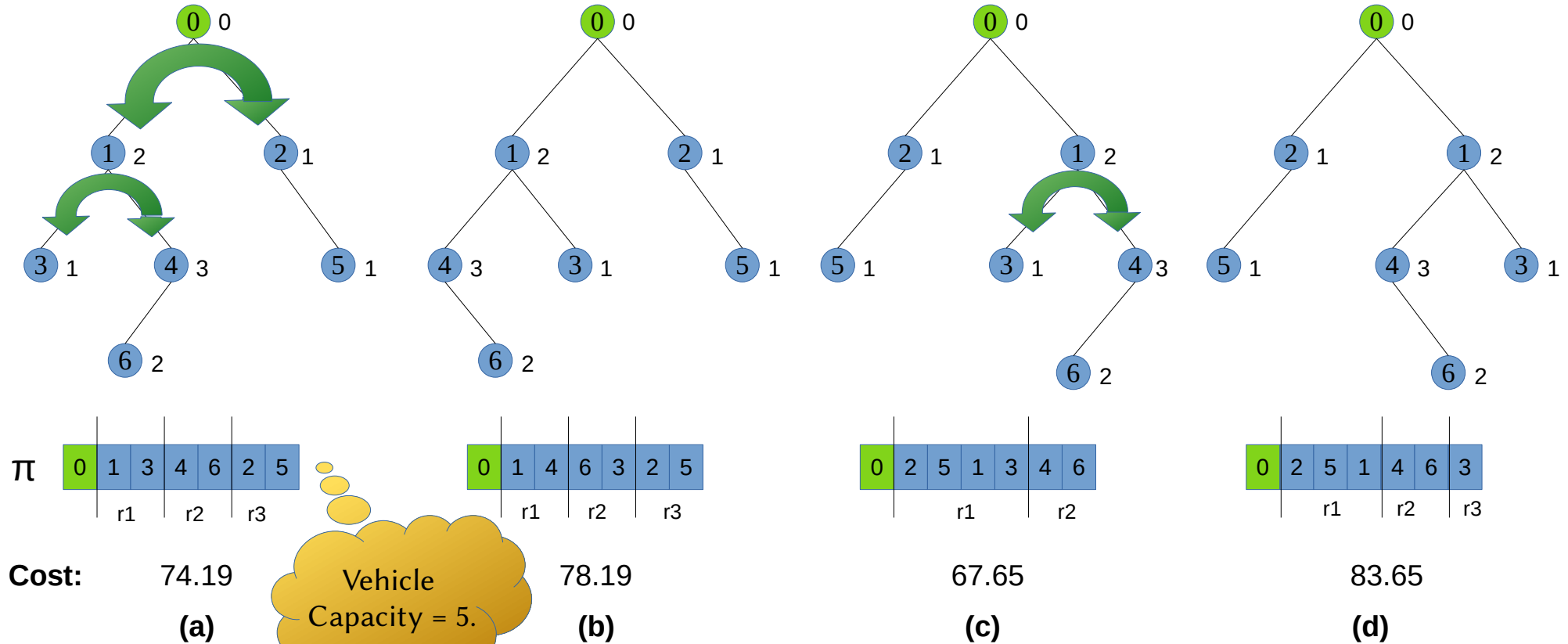


(c) Final routes generated by ParMDS

Cost: 66.13  
OPT: 63.76

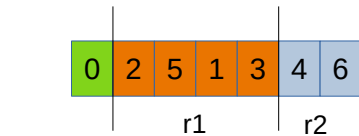
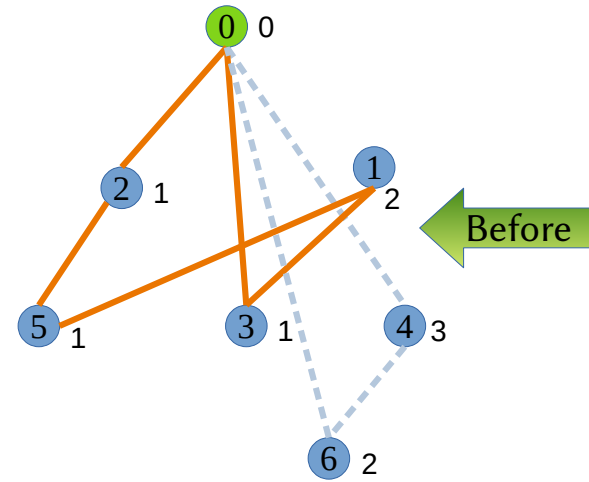
ParMDS on an example input instance with  $n = 7$  and Vehicle Capacity = 5.

# Example – DFS and Randomization



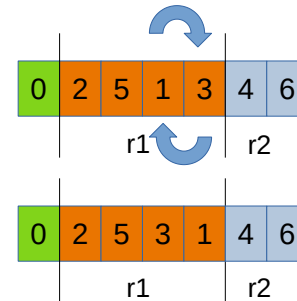
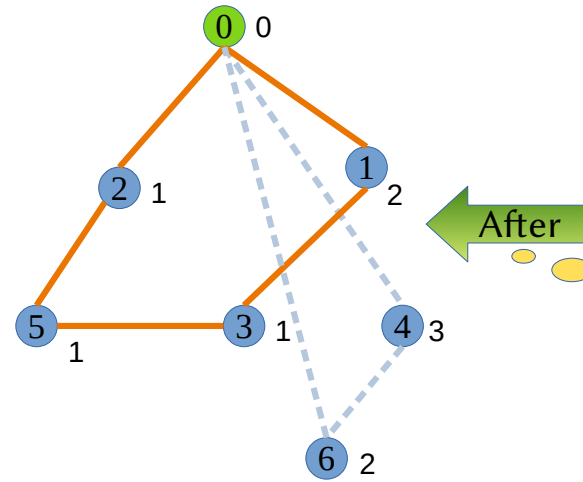
**Takeaway:** Randomizing neighbours of MST may yield a different DFS ordering. Hence, a different route!

# Intra-route optimization - 2Opt



Cost: 67.65

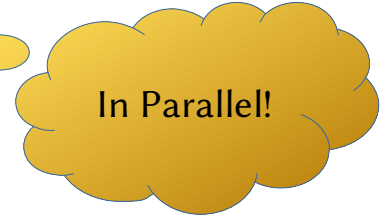
(a)



67.65

66.13

(b)



# 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   $T \leftarrow \text{PRIMS\_MST}(G)$                                 /* Step 1 */
2   $C_R \leftarrow \infty$ 
3  for  $i \leftarrow 1$  to  $\rho$  do    /* Superloop */ /* Parallel */
4       $T_i \leftarrow \text{RANDOMIZE}(T)$  /* Shuffle Adjacency List */
5       $\pi_i \leftarrow \text{DFS\_VISIT}(T_i, \text{Depot})$            /* Step 2 */
6       $R_i \leftarrow \text{CONVERT\_TO\_ROUTES}(\pi_i, Q, D)$  /* Step 3 */
7       $C_{R_i} \leftarrow \text{CALCULATE\_COST}(R_i)$           /* Parallel */
8      if  $C_{R_i} < C_R$  then
9           $C_R \leftarrow C_{R_i}$                 /* Current Min Cost */
10          $R' \leftarrow R_i$                 /* Current Min Cost Route */
11     end
12 end
13  $R \leftarrow \text{REFINE\_ROUTES}(R')$                 /* Step 4 */
14 return  $R, C_R$ 

```

Zoom-in

```

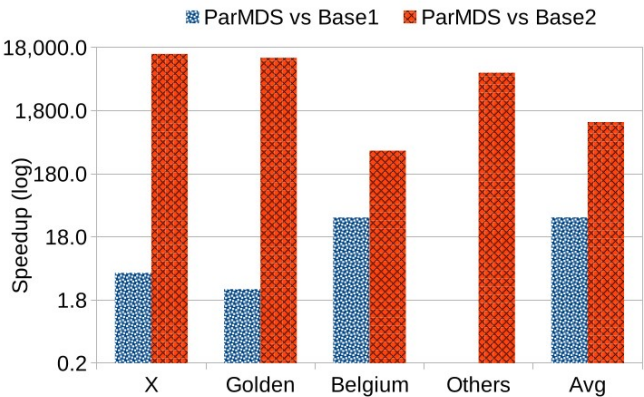
/* Standard: stride = 1; */
/* Strided : stride = #CPU cores */
/* Parallel for loop: Standard/Strided */
1 for  $i \leftarrow 1; i \leq \rho; i = i + \text{stride}$  do
2     for  $v \in V$  do
3         /* seed  $\leftarrow$  constant or  $i$  or  $\text{rand}()$  */
4          $\text{SHUFFLE\_NEIGHBORS}(\text{AdjList}(v), \text{seed});$ 
5     end
6 end
7 ...

```

# Experiments



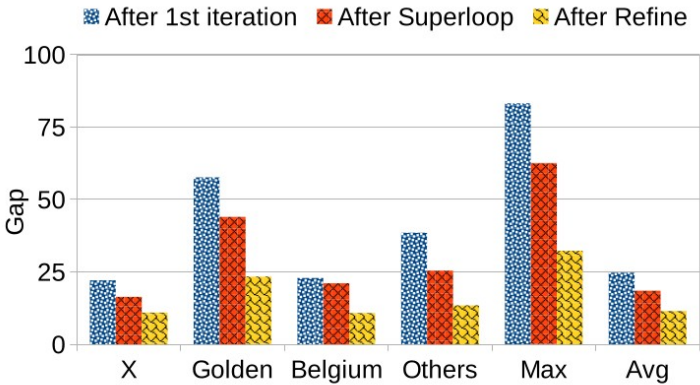
- 130 Instances of CVRPLIB
- Intel Xeon CPU E5-2640 v4
- Baselines on GPU
  - NVIDIA's Tesla P100
  - CUDA 11.5
- Our Code uses
  - **SeqMDS**: GCC 9.3.1
  - **ParMDS**: nvc++ compiler NVIDIA's HPC SDK 22.11



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

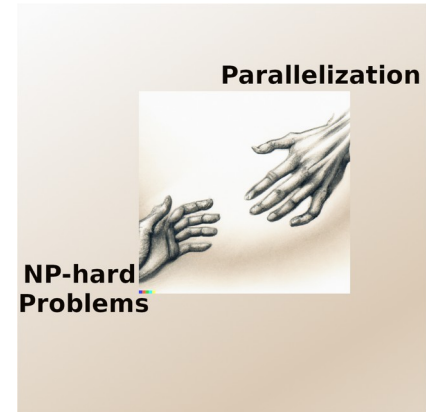


Gap at the end of each step

# Summary



- Fresh perspective to design parallelism-friendly algorithms
- Performance: Algorithmic-, Parallelism- & Platform- specific Optimizations
- Our techniques are applicable
  - 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 multi-core GPU and GPU



Thank you!

Questions?

