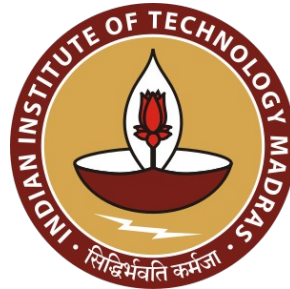


Accelerating Computation of Steiner Trees on GPUs

Rajesh Pandian M

CS16D003

www.cse.iitm.ac.in/~mrprajesh



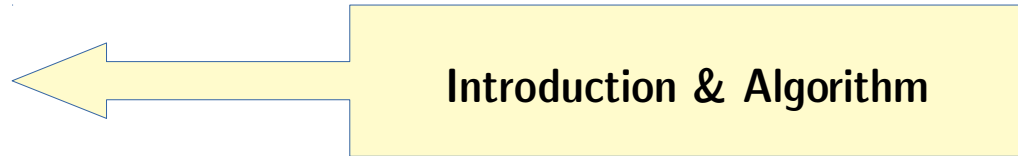
Acknowledgements

- Many thanks to my advisors N.S.Narayanaswamy & Rupesh Nasre.
- Thanks to P100 – GPU Server and TCS+PACE Lab members.
- This work evolved after the PACE Challenge 2018 [www.pacechallenge.org] on Steiner Tree
- This work is published in IJPP 2022.



Outline

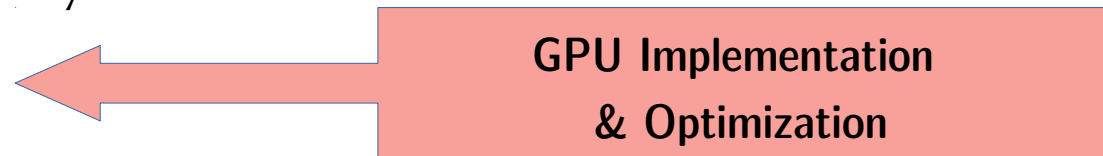
- Introduction - Steiner Tree Problem
- Definition & Example
- KMB algorithm



- Challenges in parallelizing KMB
- Design Choice of KMB
- CPU Optimization



- GPU Implementation
- SSSP Optimization – Sync, Compute, Memory
- Double Barrel and p-SSSP
- Experimental Results
- Summary



Steiner Tree Problem (STP)

Informally

Input : Undirected Graph $G(V, E, W, L)$ W is non-negative edge weights; $L \subseteq V$ terminals
Output : A tree with all terminals
Goal : Minimize the weight of the tree

- **Steiner Tree** - tree with all the terminals and zero or more non-terminals.
- **Terminals** or terminal vertices are special vertices which must be present in the tree
- **Non-terminals** or **Steiner vertices** are optional vertices – generally included in tree to minimize the overall weight of the resulting tree.
- Standard Graph-theoretic notation is used $n=|V|$, $m=|E|$ and additionally $k=|L|$
- Applications[Hwang et. al. 92]: VLSI design, network/vehicle routing, etc.



F.K. Hwang, D.S. Richards, P. Winter, *The Steiner Tree Problem*, Annals of Discrete Mathematics, Elsevier, 1992.

Steiner Tree Problem (STP) - Example

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

Output : Connected subgraph $T'(V' \ni L, E' \subseteq E)$ of G such that $\text{Min } W(E')$
// Minimum weighted tree with all terminals

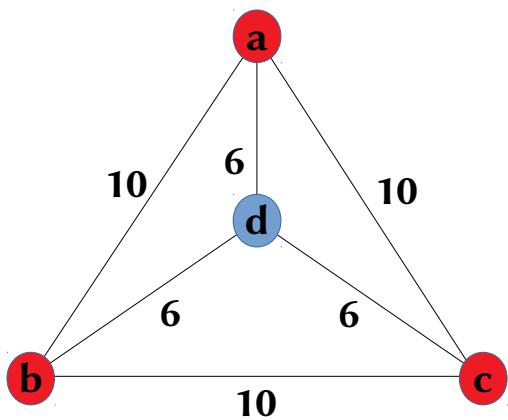


Fig 1 (a)

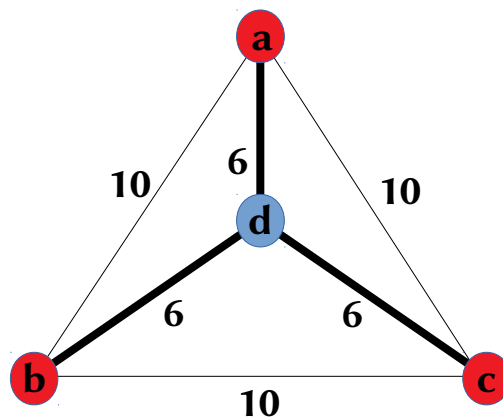


Fig 1 (b)



● Terminals
● Non-terminals



Steiner Tree Problem (STP) - Example

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

Output : Connected subgraph $T'(V' \supseteq L, E' \subseteq E)$ of G such that $\text{Min } W(E')$
// Minimum weighted tree with all terminals

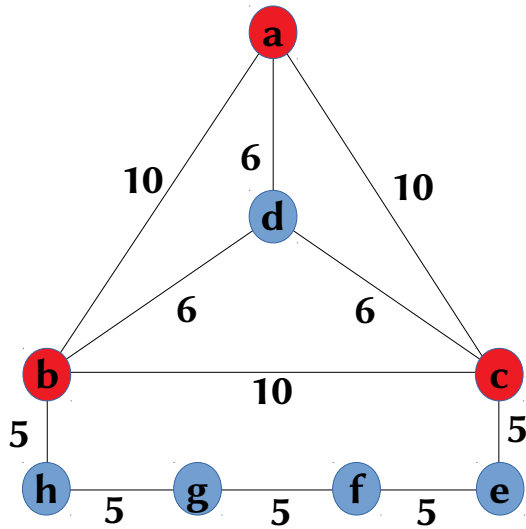


Fig 2 (a)

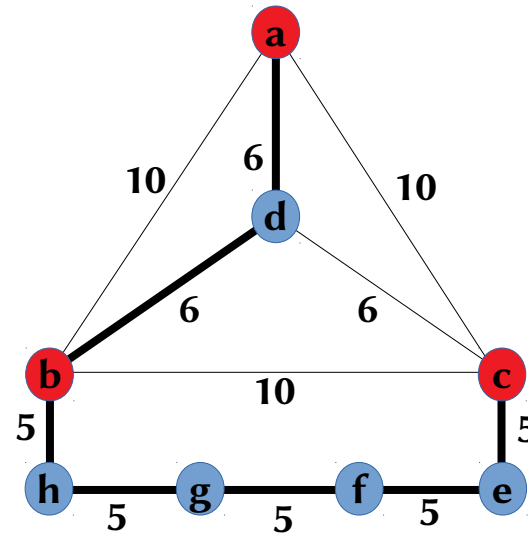
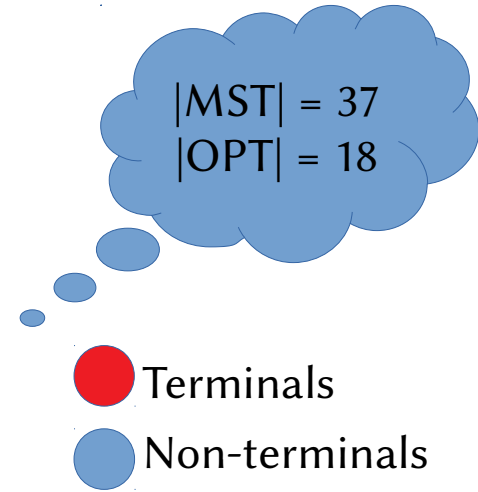


Fig 2 (b)



Steiner Tree Problem (STP) - Hardness

Input : Graph $G(V, E, W, L)$ $W:E \rightarrow Z^+$ and $L \subseteq V$ terminals
Output : Minimum weighted tree with all terminals

Take away

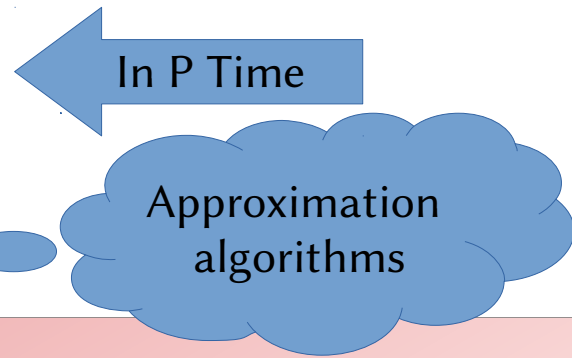
- MST solution is a valid feasible Steiner Tree solution
- However, solution can be arbitrarily bad w.r.t OPT.



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



Standard Graph-theoretic notation is used $n=|V|$, $m=|E|$ and additionally $k=|L|$



How to deal with NP-Hardness

- No Polynomial time algorithm can find optimal solution unless $P = NP$.
- 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]

$$|\text{ALG}| \leq 2 |\text{OPT}|$$



KMB



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

Comparison with related work

Solver	CPU	GPU	$k > 128$	Quality	Time taken
PACE2018 Winner [CIMAT Team]	✓		✓	☆☆	🕒🕒🕒🕒🕒
OGDF's KMB /JEA [BC19]	✓		✓	☆☆☆	🕒🕒🕒
CUDA STAR [MK15]		✓		-	-
Our KMBCPU [MNN22]	✓		✓	☆☆☆	🕒
Our KMBGPU-OPT [MNN22]		✓	✓	☆☆☆	🕒

average

Table 1 Characteristics comparison with related work and our work.



Challenges in parallelizing KMB

- Graph algorithms in general has an irregular access pattern.
 - Defies the scope of parallelizing
- Involvement of multiple primitive algorithms (such as SSSP and MST)
 - Dependence on an algorithm input from the output of previous algorithm
- Maintaining consistent parent information in SSSP along with distances.
 - Individual atomic instructions may not lead to atomic transactions.
- Parallel KMB may output different solutions during different invocations,
 - Makes it difficult to validate the solution,



Our Contributions

- Optimized CPU implementation for KMB algorithm
 - Novel SSSP-halt technique
 - Speed-up upto 15x (average 4x) improvement over JEA/OGDF's KMB[BC19]
- Optimized GPU implementation for KMB algorithm
 - Novel p-SSSP technique (multiple parallel-SSSP in parallel)
 - Speed-up upto 27x (average 4x) over sequential CPU [MNN22]
 - Speed-up upto 62x (average 20x) over sequential JEA/OGDF's KMB [BC19]



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

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

Phase 1

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''

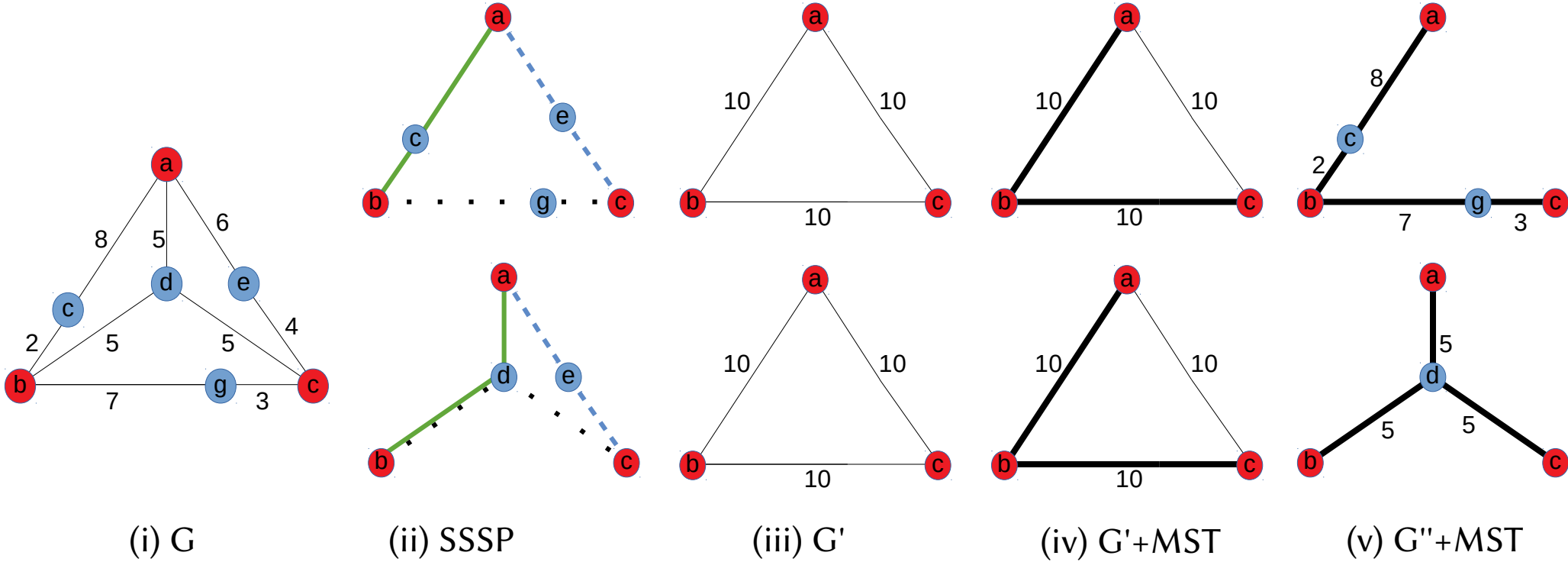
For every edge in MST(G') substitute the edges with the corresponding shortest path in G

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

MST(G'')



KMB Algorithm - Running example



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

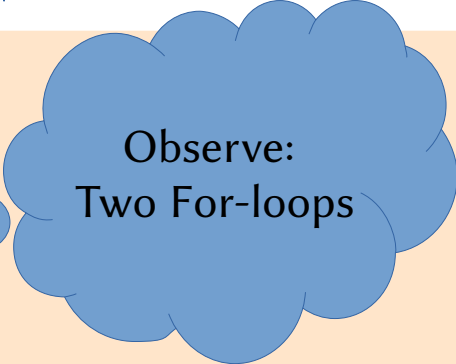
Fig. 3 Execution steps of KMB algorithm, where ● are terminals and ● are non-terminals.



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' = MST(G', W')
```



Observe:
Two For-loops

Phase 3

```
For (u,v) in edges of T' {  
   $G'' = G'' \cup P_{uv}$   
  //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 $T(V_T, E_T) \cdot V_T \supseteq L$

For $u \in L$ {

SSSP(G, W, L, u)

Compute W' incrementally

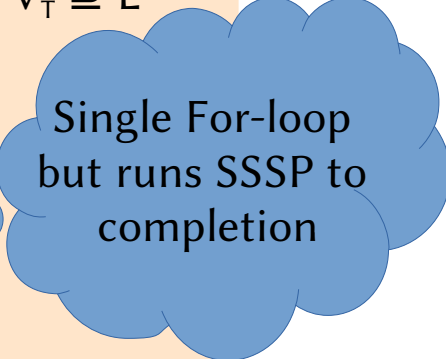
}

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

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

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

return T''



Single For-loop
but runs SSSP to
completion



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;

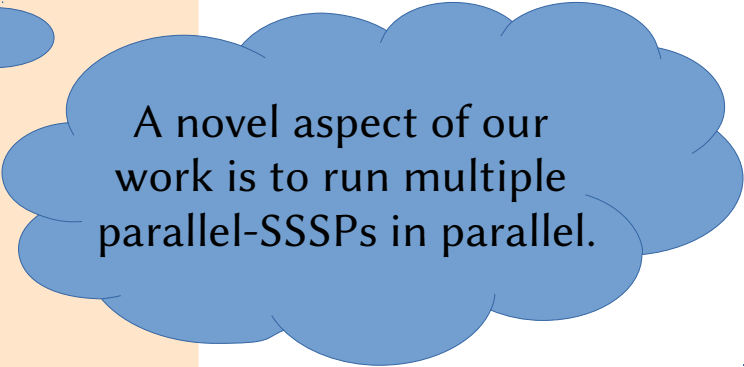
}

$T' =$ **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.



SSSP : Dijkstra vs BellmanFordMoore

- Runs in time $O((m+n) \log n)$
- Uses Fibonacci Min-Heap
- At each iteration,
 - Pick up node from Q
 - RELAXes all its neighbours
- Runs in time $O(nm)$
- No heap
- All edges are RELAXed at most $(n-1)$ times

For i from 1 to n-1:
For each edge (u, v) in E
RELAX(u,v, W(u,v))

In parallel setting it is difficult use Queue

RELAX all edges
Launched using n threads or m



Dijkstra and its RELAX operations

INPUT: $G(V,E,W)$, src

OUTPUT: $d[]$, $p[]$

INITIALIZE-SINGLE -SOURCE (G , src)

$Q = G.V$

while(! $Q.empty()$) {

$u = \text{ExtractMin}(Q)$;

 For v in $\text{Adj}[u]$

 RELAX(u,v, W)

}

```
RELAX( $u, v, W$ ) {  
    If  $u.d + W(u,v) < v.d$  {  
         $v.d = u.d + W(u,v)$   
         $v.p = u$   
    }  
}
```

INITIALIZE-SINGLE -SOURCE(G , src)

For each v in $G.V$ {

$v.d = \infty$

$v.p = \text{NIL}$

}

$\text{src}.d = 0$

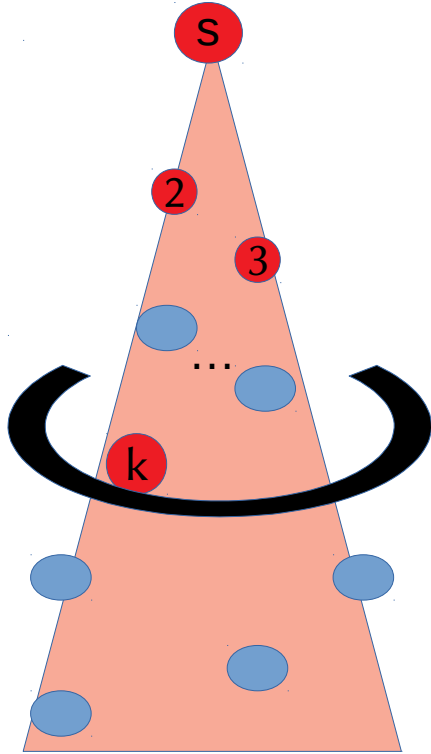


Source : CLRS book

CPU Implementation - Optimization

- SSSP-halt optimization

Steps
of
SSSP
execution



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

Halt SSSP when all terminals are visited

Fig. 4 SSSP-halt visualization



Design choice for parallelization

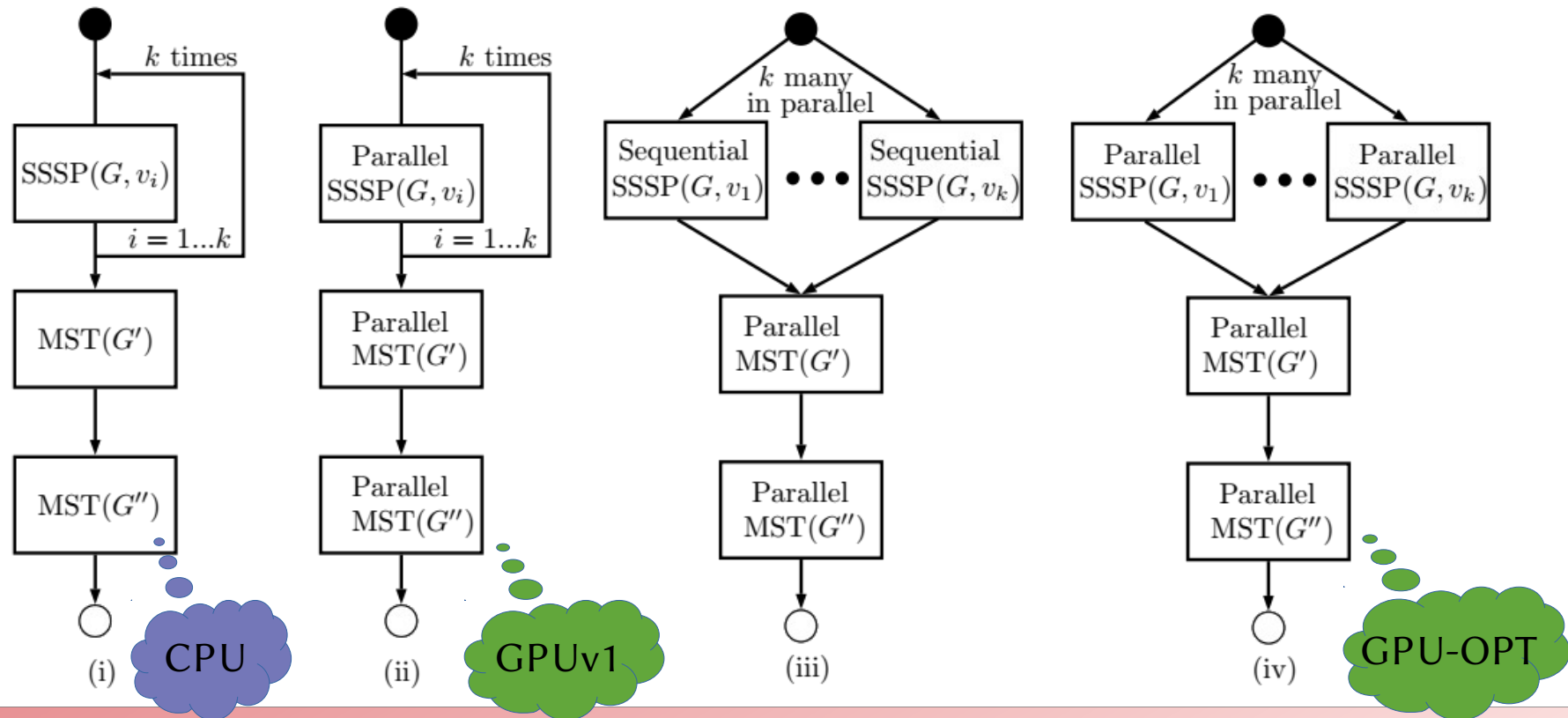
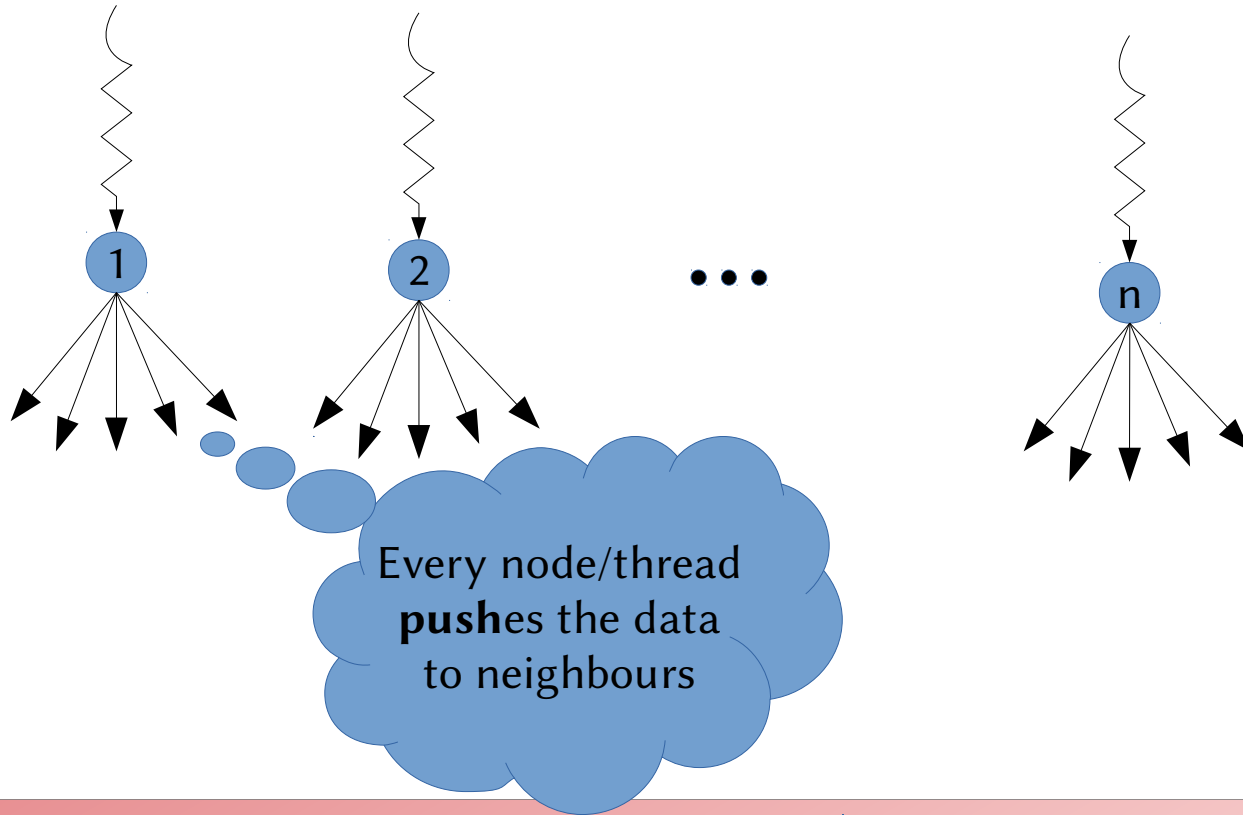


Fig. 5 Design choices.



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);

SSSP-KERNEL<Blks,ThdsPerBlk>(., s, d_s , p_s , changed, n);

CopyTo(DArray, d_s); // From Device to Host.

CopyTo(PArray, p_s); // From Device to Host.

CopyTo(hChanged, changed); // From Device to Host.

}while (hChanged);

}

- Note we reuse $d[]$ $p[]$ across iterations
- We need the $p[]$ for knowing the intermediate vertices in the shortest path



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

```
SSSP-KERNEL(..,s, d_s , p_s , changed, n) {
```

```
u = tid // compute tid;
```

```
if tid < n {
```

```
  For v  $\in$  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 :

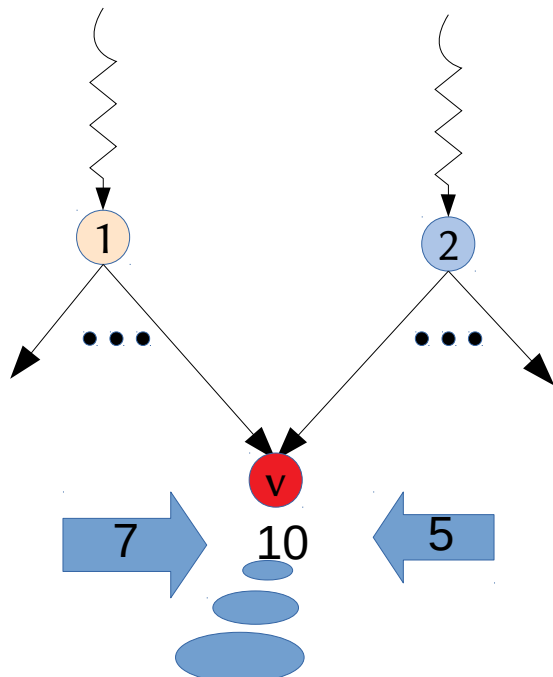
- Parent of v should be updated if the Atomic-MIN is success



Is it enough?



Parent update - Challenge



```
<snip>
```

```
..
```

```
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;
```

```
}
```

```
</snip>
```

Two threads want
to update distance of their
common neighbour v

Fig. 7 Challenges in parent update



Parent update - Challenge

Shared
d[], p[]

Thread 1

newCost=7
old=10

d[v]=7 //oldA=10

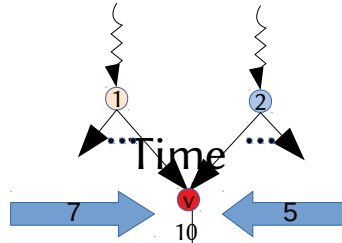
p[v] = 1.

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);
```

```
// Atomic-MIN is Success
```

```
If oldA != old {  
  // Update's Parent array  
  p[v] = u;  
  changed = true;  
}
```

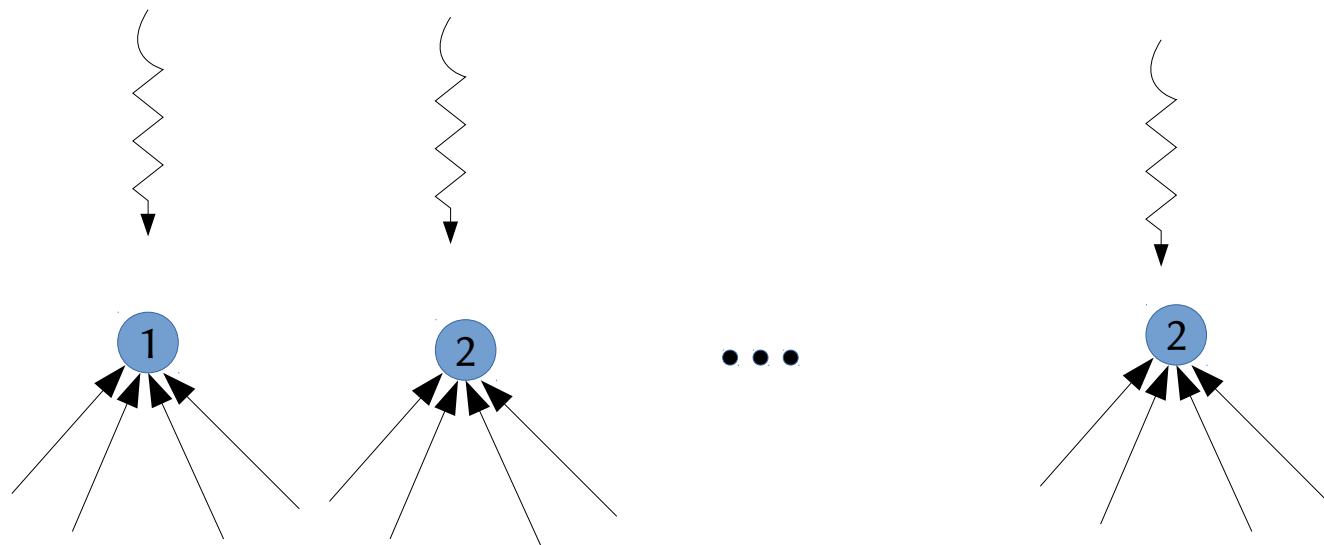
Wrong!

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? Locks?



Synchronization optimization • Pull



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

Fig. 8 Pull-SSSP

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

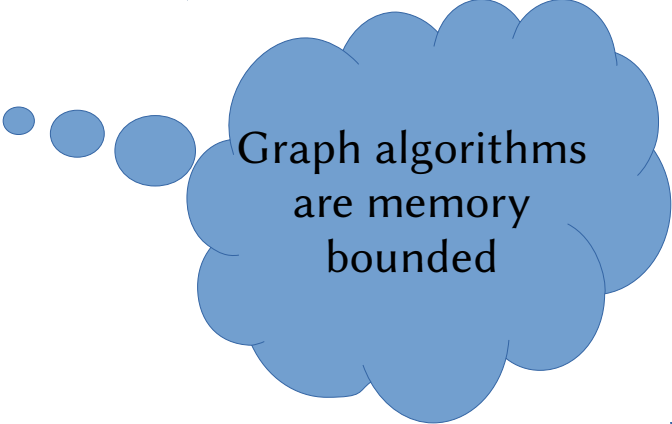
- Synchronization
 - Push
 - **Pull**
- Computation
 - Data-driven
 - Edge-based
 - **Controlled Computation unrolling**
 - Δ^2
 - 2Δ
 - **$t\Delta$**
- Memory
 - **Shared memory**



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 $t\Delta$); e.g. we do pull 3 times in the kernel – 3-pull
 - Empirically, we achieved best performance when $t=3$
- Data-driven
 - Needs Worklist (WL)
 - Active/Change nodes are inserted into WL
 - Only size of WL many threads launched
 - Need synchronization while inserting nodes in WL
- Edge based optimization
 - m -threads are launched
 - RELAXes one edge or a group of edges
 - Representation needs to be modified.

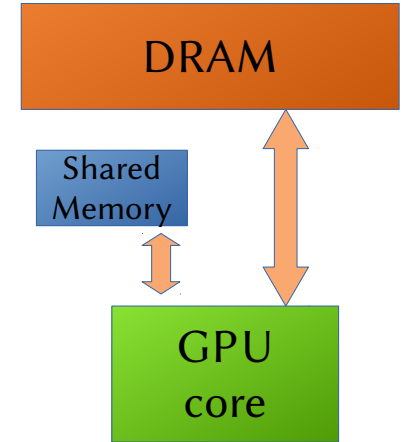


Graph algorithms
are memory
bounded



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 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, ...);
}
    
```

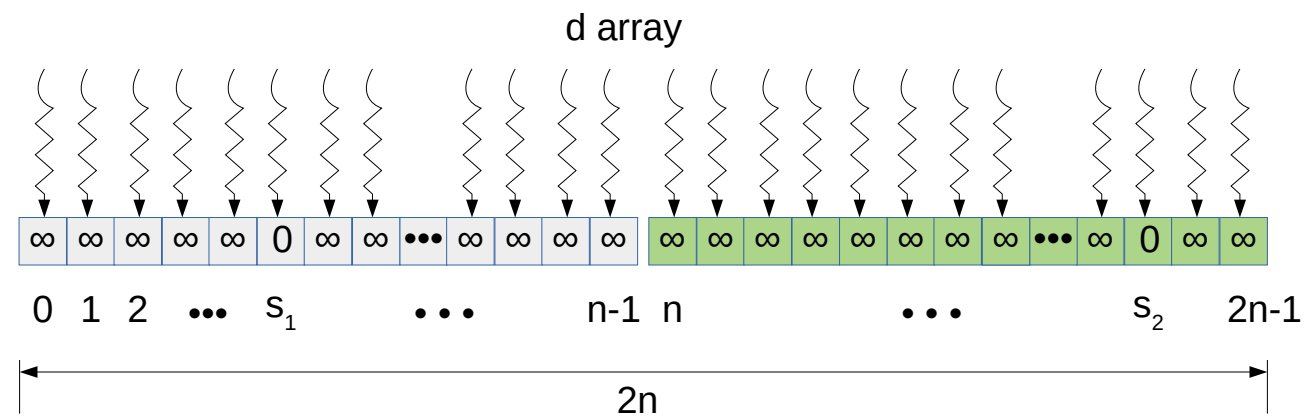


Fig. 9 Double-barrel approach.



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

- CentOS Linux release 7.5
- GCC 7.3.1 with O3
- CUDA 10.2

Graphsuite

- Total 14 Graphs
 - 11 from PACE Challenge [PACE2018]
 - 2 from SteinLib
 - 1 from SNAP
- n : 17K – 235K
- m : 27K – 498K
- k : 0.1K – 6K

Baselines

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

- PACE 2018 - <https://pacechallenge.org/2018/steiner-tree/>
- CIMAT Team - <https://github.com/HeathcliffAC/SteinerTreeProblem>
- S. Beyer and M. Chimani, Strong Steiner Tree Approximations in Practice, JEA 2019.



Experiments - Comparison of solution quality

Why?

STP is an NP-hard,
our algorithm finds a solution
closer to optimum.

Solution Quality

- KMBGPU-OPT, KMBCPU and JEA are similar vs OPT
- KMBGPU-OPT and KMBCPU are better than PACE on all instances



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

Experiments - Speed-up

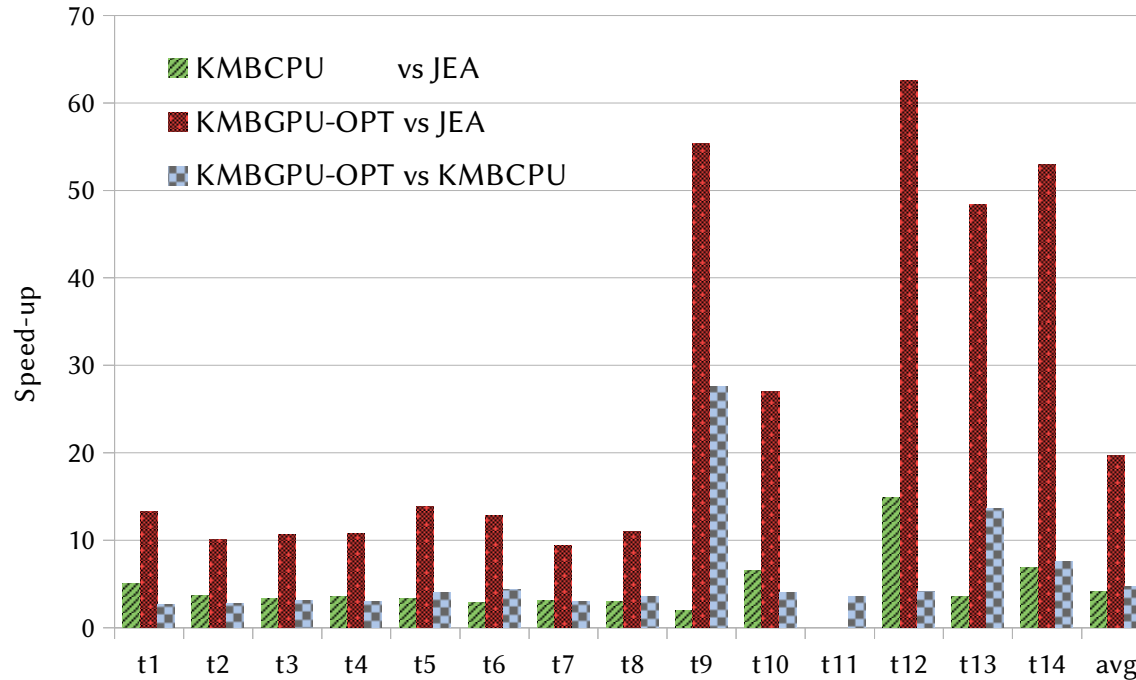


Fig. 10 Speed-up comparisons of the implementations (higher is better). JEA timed-out on t11

Takeaway: KMBCPU and KMBGPUOPT is better than JEA



Comparison of GPU time with Shared memory

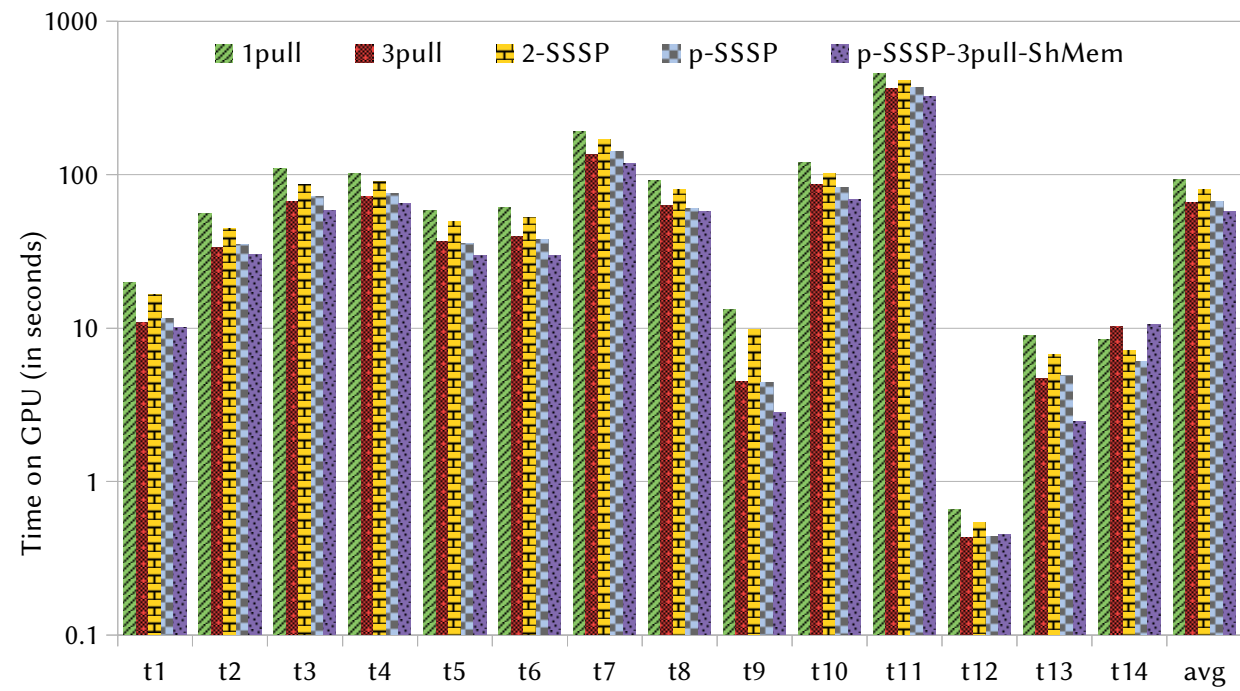


Fig. 11 Comparison of 1-Pull, 3-Pull, Double-barrel & p-SSSP+3-Pull+shared memory (smaller is better). Note: 1-Pull is KMBGPU whereas p-SSSP-3pull-ShMem is KMBGPU-OPT

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



Comparison of p-SSSP

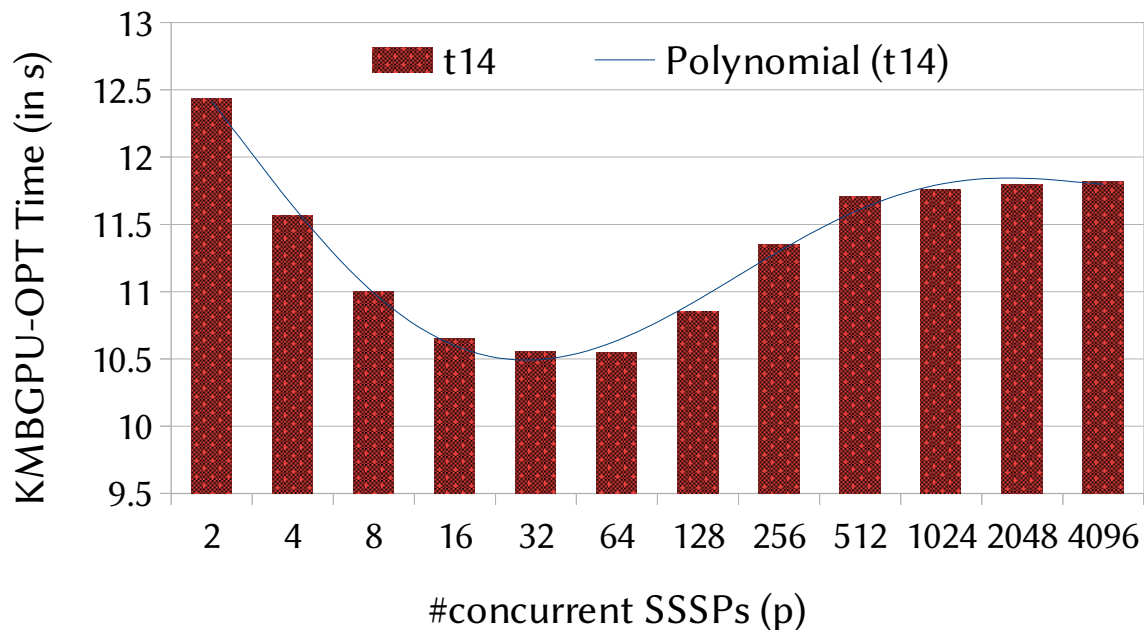


Fig. 12 KMBGPU with varying p-SSSP for the same graphs t14 (Smaller is better).

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



Experiments - Scalability of GPU and CPU

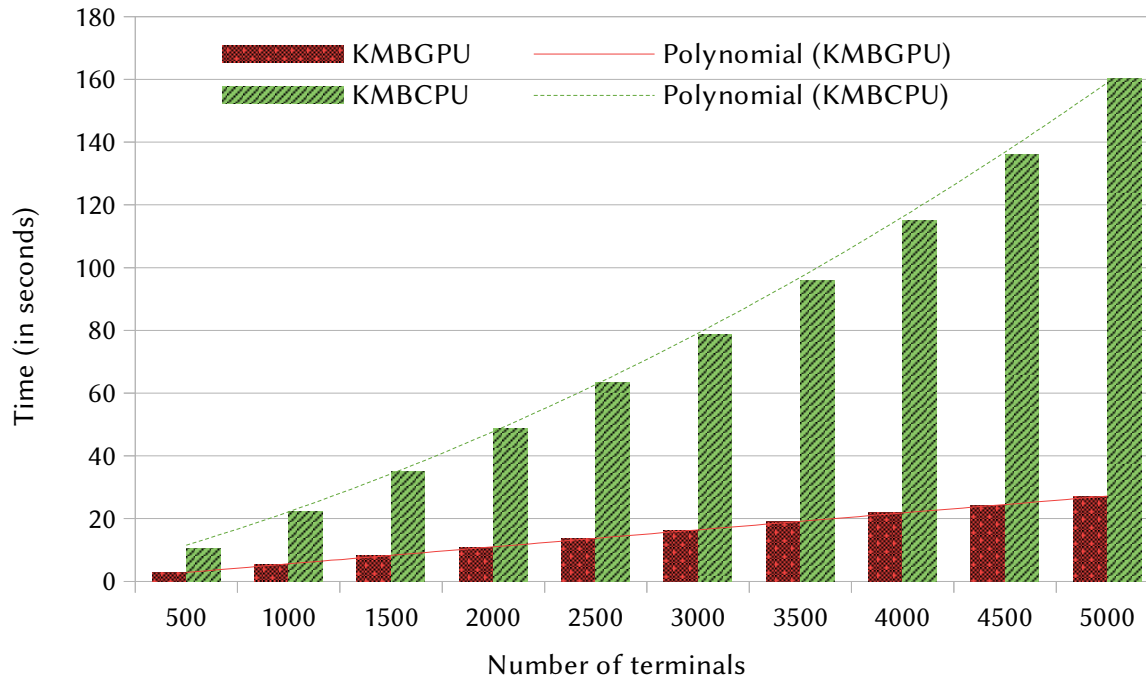


Fig. 13 Scalability plot on **t14** with increasing terminal size (lower is better)

Takeaway: KMBGPU-OPT scales better than KMBCPU



Summary

- SSSP halt-optimization benefits CPU.
- Pull and p-SSSP optimization benefits GPU.
- Our output Steiner tree can be used as initial tree for other local search algorithms.
- Our technique is applicable when multiple parallel instances of an operator are used.

Future work

- KMBCPU can be extended to multicore-CPU.
- KMBGPU-OPT can be extended to multi-GPU.
- Capacitated Vehicle Routing Problem
- Build a GPU graph library for aiding NP-Hard problems.



Thank you.