Accelerated Steiner Tree Problem Solving on GPU with CUDA

Christian Mathieu[1] and Matthias Klusch[2]

[1] Saarland University, Computer Science Department, 66123 Saarbruecken, Germany
[2] German Research Center for Artificial Intelligence, 66123 Saarbruecken, Germany

Abstract. The Steiner Tree Problem in Graphs (STPG) is an important NP-hard combinatorial optimization problem arising naturally in many applications including network routing, social media analysis, power grid routing and in the context of the semantic web. We present the first parallel heuristics for the solution of the STPG for GPU with CUDA, and show that the achieved speedups for different kinds of graphs are significant compared to one of the fastest serial heuristics `STAR`.

Keywords: Graphs; Steiner Tree Problem; Parallel Algorithm; GPU; CUDA

1 Introduction

The Steiner Tree Problem in Graphs (STPG) is one of the most important combinatorial optimization problems in computer science and defined as follows:

Definition 1. *Given a weighed graph $G(V, E)$ with node set V, edge set $E \subseteq V \times V$ and a nonnegative weight function $w : E \to \mathbb{R}^+$, a solution tree $S(V', E')$ is a connected acyclic graph with node set $V' \subseteq V$ and edge set $E' \subseteq E \cap (V' \times V')$. We extend the weight function w to solution trees as $w(S) = \sum_{e \in E'} w(e)$. The Steiner tree problem in graphs (STPG) is as follows: For a given terminal set $V_t \subseteq V'$, find a* minimal-cost tree S in G that spans V_t. *Minimal cost* denotes that $w(S)$ is minimal among all possible solution trees in G spanning V_t. The vertices $v \in V_t$. are called *terminal nodes*, while the remaining vertices $v \in V' \backslash V_t$ are called *non-terminal nodes*

The STPG and close variants of it are used in a wide range of applications, including wire routing in very-large-scale integration circuits (VLSI) [11], network routing in wireless networks [20], multicast routing [12] and power distribution in electric grids [8]. Large graph databases, especially in the context of social networks or the semantic web, grow ever more important. When close relationships between multiple entities are of interest, the STPG often arises naturally. Many of these applications deal with very large problem sizes.
However, solving the STPG optimally grows prohibitively expensive with increasing problem size: Deciding the existence of a Steiner tree below a given cost is one of Karp's classical 21 NP-complete problems [14]. Thus, the optimization problem given in *Definition 1* is NP-hard. In fact, there is essentially

no known algorithm with worst-case time complexity better than exhaustive search, resulting in exponential runtime complexity. Due to the importance of the problem, there has been considerable effort into finding both serial [25, 17, 23, 24], distributed [20, 12] and parallel[22] heuristics approximating the optimal solution while offering far better runtime. On the other hand, there are also many implementations of other graph algorithms on GPU [10, 16, 26], including the Rectilinear Steiner Tree Problem [6]. However, to the best of our knowledge, there are no parallel heuristics for the STPG employing the GPU so far. To this end, we present the first parallel solution of the STPG for GPU with CUDA, called `STP_CUDA` [19], based on one of the fastest serial heuristics for the STPG, called `STAR` [15]. Our experimental evaluation shows that for different kinds of graphs, achieved speedup of `STP_CUDA` is significant compared to the reference implementation of `STAR` for CPU.
The remainder of this paper is structured as follows: NVIDIA's CUDA and Kasneci et al's `STAR` [15] are introduced in *Section 2*, while our parallel implementation is described in *Section 3*. Results of our practical evaluation are presented in *Section 4*, before we draw our final conclusions in *Section 5*.

2 Background

In this section, we briefly introduce CUDA and an approximated STP solver, called STAR [15], which serves as a basis of our parallel STP solving on GPU with CUDA, introduced in a subsequent section.

Approximated Solution with STAR. A very fast approximated STP solver called STAR [15] replaces the exhaustive search for a globally optimal solution with a subdivision approach that repeatedly recombines partial solutions using a shortest path heuristic. While unmodified `STAR`'s worst case runtime is exponential, the authors also introduced the *ϵ-improvement rule* which is an early abort criterion skipping recombinations resulting in minimal improvements, thus guaranteeing termination in a limited number of iterations.
Using this optimization, STAR has a time complexity of $\mathcal{O}(\frac{1}{\epsilon}\frac{w_{max}}{w_{min}}mk(S))$ for improvement threshold ϵ, query terminal count k, graph node count n, graph edge count m and largest and smallest edge weight w_{max} and w_{min}[15]. S denotes the time complexity of the chosen search algorithm, which is $nlogn + m$ for the reference implementation. It has been shown that STAR approximates the optimal solution to a factor of $(1 + \epsilon)(4\lceil logk \rceil + 4)$.
During its execution, STAR broadly operates in two phases. It first constructs an arbitrary tree connecting all terminals using edges of the encompassing graph. This tree can be thought of as an initial candidate solution. It is found by performing a breadth-first search from each terminal node until a single node is reached by all of the searches. The path to the corresponding starting terminal of each search is then backtracked to build a tree spanning all terminals. This initial tree is then passed to the second phase, which tries to iteratively improve it. The key idea is to visualize a tree as a collection of non-branching paths

of degree 2, called *loose paths*, which are connected to either nodes of degree ≥ 3 or terminal nodes. These crossroads where loose paths meet are called *fixed nodes*. This is illustrated in *Figure 1(a)*. Note that removing all *intermediate* (non-endpoint) nodes of a loose path from the tree decomposes the tree into two partitions, both of which are trees as well. *Figure 1(b)* shows the graph from *Figure 1(a)* after removing loose path 3. `STAR` now attempts to find a better (lower cost) tree by following the following scheme:

- All loose paths in the currently best solution candidate tree are found
- The loose paths are sorted by descending length
- `STAR` attempts to replace the longest loose path not yet attempted
- To replace the loose path, it is cut out of the tree, and the shortest path from any node of the first partition to any node in the second partition is found. This is a regular multi-source multi-target shortest path search.
- If the two partitions and the shortest path between them form a cheaper tree than the current best, it is the new current best and `STAR` repeats these steps on the newfound tree.
- If the newfound tree is not strictly better, it is discarded and the next loose path attempted.
- If all loose paths have been attempted but none resulted in an improvement, then the currently best solution is returned as `STAR`'s result. Note that this implies that the tree is a local optimum regarding `STAR`'s heuristic.

A schematic overview of `STAR`'s execution is shown in *Figure 2*. Loose paths, fixed nodes and their role during tree partitioning are depicted in *Figures 1(a)* and *1(b)*.

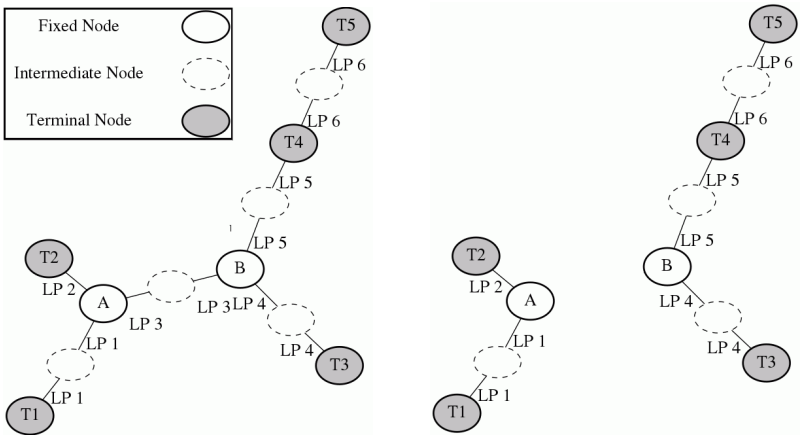
A pseudo-code listing of `STAR` is shown in *Algorithm 1*. *buildInitialTree()* denotes a first phase not depicted here, which connects all terminal nodes to a common tree. *partition()* returns the two partitions that result when removing all edges and intermediate nodes of *loosePath* from *tree*. By 'intermediate node' we mean all nodes of the path excluding the two end points (see *Figure 1*). *findShortestPath()* in Kasneci et al's reference implementation is a straightforward bidirectional shortest path search based on Disjkstra's Algorithm. We will later replace it with one of two parallel implementations. For further information on the reference implementation, refer to [15].

Algorithm 1 STAR(graph, terminals)

```
bestTree := buildInitialTree(graph, terminals)
newTree := improveTree(graph, terminals, bestTree)
while newTree.cost < bestTree.cost do
  bestTree := newTree
  newTree := improveTree(graph, terminals, bestTree)
end while
return bestTree
```

Algorithm 2 improveTree(graph, terminals, tree)

```
loosePaths := getLoosePaths(terminals, tree)
for loosePaths ordered by descending length do
  (partitionA, partitionB) := partition(tree, loosePath)
  path := findShortestPath(graph, partitionA, partitionB)
  newTree := merge(partitionA, path, partitionB)
  if newTree.cost < tree.cost then
    return newTree
  end if
end for
return tree
```



(a) Loose Paths and Fixed Nodes

(b) Partitions after Removal of Loose Path 3

Fig. 1: Partitioning of Solution Candidate Tree

Parallel processing with CUDA. In a GPU using NVIDIAs Compute Unified Device Architecture (CUDA), multiple cores are clustered into coherent units called streaming multiprocessors (SM). CUDA segrates parallely executed code into enclosed units called *kernels*. A kernel can be distributed across hundres of cores, executing tens of thousands of lightweight threads concurrently. To synchronize this vast amount of threads, a simple barrier synchronization model is employed. Since groups of 32 threads (called a *warp*) are executed in true SIMD fashion on one SM, divergent control flow is to be avoided within a warp to keep performance high. To keep costly global memory accesses to a minimum, threads on one SM can use a local memory called *shared memory* to cooperate with other threads within the same core (or more precisely: within a *block*, a logical unit clustering some threads guaranteed to reside within the same SM). The device is further capable of joining multiple memory accesses within a contiguous region of memory, a so called *coalesced* memory access. These hardware details necessitate adapted program design avoiding inefficient memory layout and playing to the strengths of the architecture.

3 Parallel STP_CUDA Algorithm

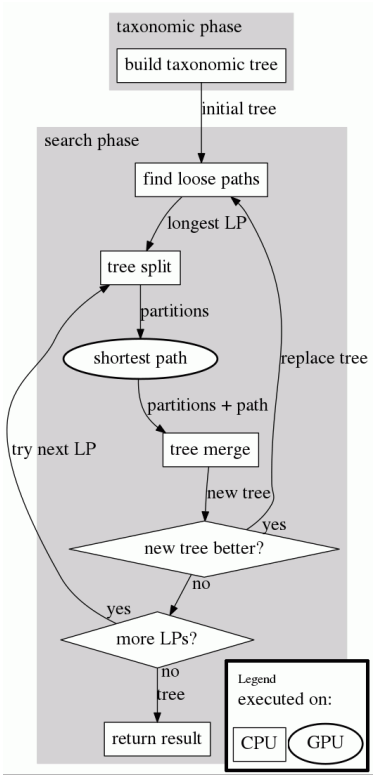


Fig. 2: STAR Overview

Since we found `STAR`'s runtime on large graphs to be mostly dominated by the shortest path heuristic (*Figure 6*), we replaced the serial implementation of Dijkstra's Algorithm [7] with two GPU-based parallel approaches. `CUDA_N`, the first of these, follows Martin et al.'s node-parallel approach closely [18], further employing an optimization found by Ortega-Arranz et al. [21].

The basic idea is to unroll DA's outer loop partially, processing multiple nodes from the queue at once. The second approach, `CUDA_E`, parallelizes Dijkstra's Algorithm further, traversing *edges* in parallel, not unlike Jia et al.'s algorithm for betweenness centrality [13]. This effectively unrolls DA's innermost loop as well, which traverses each processed nodes' outgoing edges. The general idea for parallelization is identical for both approaches:

Dijkstra's Algorithm keeps track of the length of the best known shortest path from any search source node to each reached node. This distance is called the *tentative distance*. During each iteration, the least tentative distance node (*min-node*) among unprocessed (not yet *settled*) nodes is found. Since Dijkstra's Algorithm forbids negative weight edges, no node can *relax* (i.e. update its tentative distance) a neighboring node to a tentative distance lower than its own. Since the min-node is by definition the least distance node not yet settled, it must have already converged to its true shortest path distance, since all other unsettled nodes have at least the same tentative distance and thus cannot reduce the min-node's tentative distance further. But the min-node is not necessarily unique. Depending on the edge weights in the graph, a potentially large number of nodes can be equally minimal at a given time. We call such nodes the F-set, since they are the *frontier* of nodes about to be settled (See *Figure 4*). The order in which nodes in the F-set are processed does not matter, since all nodes in the F-set have converged to their final state, i.e. there is no interdependence among these nodes. It is thus possible to expand them concurrently, as long as we take care to resolve conflicts while updating neighbors.

Ortega-Arranz et al. further showed that this definition of the F-set can be expanded to certain *almost-minimal* nodes which must have converged to their true shortest path distance as well [21]. A node n_i is almost minimal if $d_i \leq$

$d_{min} + min_j(w_{j,i})$, where d_i denotes the tentative distance of node i, $min_j(w_{j,i})$ denotes the least weight of any incident edge to i and d_{min} denotes the minimal tentative distance of any yet unsettled node. In other words, there is no node yet unsettled that could relax node i.

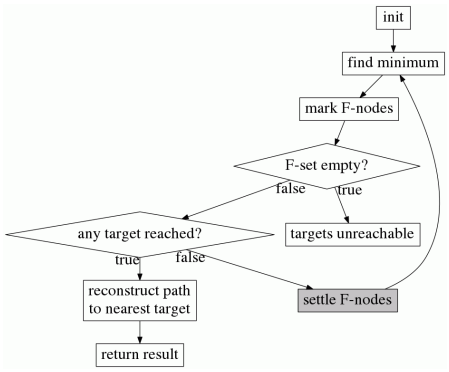


Fig. 3: Parallel Search Overview

both approaches are equivalent.

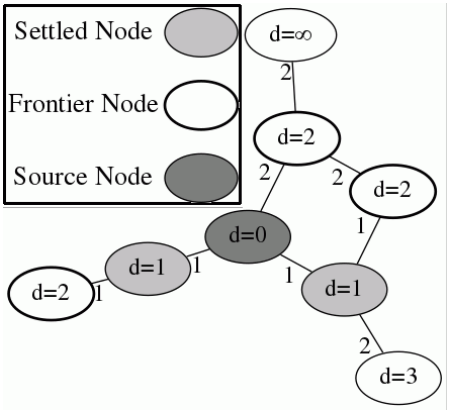


Fig. 4: Frontier Nodes of Search Fringe

most useful for static graphs.

Since this requires either processing all edge weights of each node in each iteration, or storing the minimal edge weight for each node, and since many of our graphs have uniform edge weights anyway, we opted to use Ortega-Arranz' *economical approach* instead. We only relax nodes in parallel when $d_i \leq d_{min} + w_{min}$ where w_{min} denotes the least weight of *any* edge in the graph. This is more conservative, but not dependent on the individual node's edge weights anymore. For uniform weight graphs,

To actually exploit this inherent parallelism, we proceed as follows: The graph is stored in GPU memory using a straightforward adjacency list representation. For each node, we store the offset of its first edge in the edge arrays. For each edge, we store source node index, target node index and weight. While storing the source node index might seem superfluous, it is very useful for our edge-parallel approach `CUDA_E` below. We further store the graphs minimal edge weight. The graph is initialized once and reused for subsequent searches and `STAR` invocations. Due to this, our algorithms are

During each search operation, tentative distances, F- and U-sets are initialised first and stored in form of a flag array. The F flag represents nodes that are about to be expanded (i.e. in the aforementioned *frontier*), whereas the U flag represents unsettled but visited nodes (i.e. potential candidates for the F-set during the next iteration). We then find the minimal tentative distance among U-nodes using a slightly modified parallel reduction [9]. We now mark all F-nodes, i.e. nodes n_i where $d_i \leq d_{min} + w_{min}$. If this results in no marked node, the target node(-s) must be unreachable and we abort (all possible paths are explored). If F-nodes were marked, we retrieve the least distance one. Note that,

since F-nodes are equivalent to nodes Dijkstra's Algorithm might have settled in this iteration, we are done if such a target exists. If no target was F-marked, we settle all F-nodes. To avoid data races, neighboring distance values are updated using the atomicMin operator. Parallel settling of nodes is where `CUDA_N` differs the most from `CUDA_E`. The general scheme of the search operation is depicted graphically in *Figure 3* and as pseudo-code in *Algorithm 3*.

Algorithm 3 findShortestPath(graph, partitionA, partitionB)

```
w_min := min{w(i,j) | edge (i,j) ∈ graph}                  # precalculated once
∀ nodes n ∈ graph : d(n) := ∞
∀ nodes s ∈ partitionA : d(s) := 0
d_min := ∞
U := {s | s ∈ partitionA}
F := {}
while true do
  d_min := min_{n∈U} d(n)                                  # parallel reduction[9]
  F := {n | (n ∈ U) ∧ (d(n) ≤ d_min + w_min)}              # parallel setting of flag array
  if |F| = 0 then
    return null                                            # no solution exists
  end if
  t_min := arg min_{t∈F∩partitionB} d(t)                   # any target about to be settled?
  if t_min exists then
    return constructPath(t_min)                            # backtrack from target node
  end if
  settle(U, F, d)                                          # This is where CUDA_N and CUDA_E differ!
end while
```

Node-parallel variant CUDA_N of STP_CUDA. Our node-parallel approach `CUDA_N` differs from `CUDA_E` in the strategy chosen when settling nodes in parallel. The idea behind `CUDA_N` is to distribute F-nodes over available cores, where they are settled in parallel. A single node is always processed by a single thread. To avoid the overhead of managing queue structures, U and F nodes are marked in a global flag array. We distribute *all* graph nodes among available cores, skipping nodes not in the F-set. Each thread then iterates over all neighbors of its currently allotted node in series. If successful relaxation seems possible (we don't know whether other threads are just relaxing the same node!), an atomicMin operation is executed to synchronize parallel write access. This is safe, since neither edge weights nor F-node tentative distances are going to change, thus no read after write conflicts occur. As mentioned above, the entire approach is mostly equivalent to [18].

Edge-parallel variant CUDA_E of STP_CUDA. The node-parallel approach employed by `CUDA_N` above potentially suffers from poor load balancing when processing real-world ontologies, which often have a power-law distribution

Algorithm 4 settle_N

```
for all n in parallel do
  if n ∈ F then
    for each neighbor m do                                  # this is serial
      if w(m) > w(n) + w(n,m) then
        w(m) := w(n) + w(n,m)                               # synchronized with atomicMin()
      end if
    end for
  end if
end for
```

of edge degree. When nodes differ greatly in edge degree, the workload differs between individual threads as well. Remember: `CUDA_N` has a single thread settle a single F-flagged node. This thread thus needs to iterate over all outgoing edges of the node. Differing edge degree results in unequal workload between threads. Our second approach thus distributes work not on a per-node but on a per-edge basis. When settling nodes in parallel, *all* graph edges (belonging to F-set nodes or not) are distributed among threads. For each edge, the F flag of its source node is checked. If the F-flag is not set, the edge is skipped, otherwise the target node's distance is updated. This effectively parallelises the settling of each single node. While this incurs significant overhead due to repeated access to source nodes as well as processing of unneeded edges, it offers much more fine grained load balancing, thus utilizing available processors more efficiently. As above, conflicts are resolved using the hardware-supported atomicMin operator. Thread safety follows from the same argumentation as with `CUDA_N` above.

Algorithm 5 settle_E

```
for all edges (n,m) in parallel do                          # rollout of inner loop of settle_N
  if n ∈ F then
    if w(m) > w(n) + w(n,m) then
      w(m) := w(n) + w(n,m)                                 # synchronized with atomicMin()
    end if
  end if
end for
```

4 Evaluation

Setting. All experiments below were executed using an Intel® Xeon® W5590 3.33GHz CPU with 32GB of DDR3-1333 main memory. The GPU-based components were executed using a Fermi-based Nvidia® GeForce® GTX590 card in standard memory configuration (i.e. 1536MB of GDDR5 memory per GPU).

While this card is a dual gpu card, only one GPU was employed during our tests. We compare the performance of three variants of STAR:

reference The purely CPU-based reference implementation, as defined by Kasneci et al.[15]. This implementation runs entirely host-sided and employs a straightforward serial implementation of Dijkstra's Algorithm[7] when searching for shortest paths.

cuda_n An implementation which performs the shortest path search *node*-parallel on GPU, i.e. distributing graph nodes among available threads. The implementation follows [21] closely.

cuda_e An implementation which performs the shortest path search *edge*-parallel on GPU, i.e. distributing graph edges among available threads, not unlike Jia et al.'s algorithm for betweenness centrality[13].

Both `cuda_n` and `cuda_e` perform the remaining components of `STAR` single-threaded on host side. To evaluate the performance of both parallel implementations as well as reference, we use object-relational queries in several large real-world ontologies. Test cases for each ontology were generated as follows:

- Each sample is a single object-relational query, defined by the set of terminal nodes and the ontology graph.
- The solution to a query is the resultant Steiner tree connecting the terminal nodes along edges of the ontology graph.
- Since query terminal count k affects runtime heavily, samples are generated for multiple values of k.
- k is set to increasing powers of two (starting with two), until main memory prohibits further increase (elaborated below).
- For each choice of k, 10 samples are generated.
- The query terminals of a given sample are chosen uniformly at random among all nodes of the ontology graph. If this results in an unsolvable instance (i.e. terminals are in disconnected components of the overall graph), the sample is discarded and a new sample generated until a valid sample is found.
- The evaluated algorithms all execute the same samples. The sample set is global, not per algorithm.
- The samples are executed by increasing k. Once at least one algorithm exceeds main memory during execution of a sample, all preceding samples of this k are discarded and no larger values of k are attempted. This is to ensure sample count is uniform among k, making runtime variance comparable.
- The protocolled performance metrics are wallclock runtime spent during execution and wallclock runtime spent during search. The former is defined as the time elapsed between issue of query until the resultant Steiner tree is returned. The latter is defined as the sum total of time spent inside shortest path searches during the former.
- We evaluate our parallel implementations regarding their *speedup* over reference as well as against each other. We define speedup of implementation i over implementation j as $\frac{t_j}{t_i}$, where t_i and t_j denote the measured runtime. If both implementations have equal runtime, this results in a speedup of 1.

– Time spent for creation of the ontology graph structure in memory does not affect runtime metrics. The graph is generated only once for each ontology and is considered part of the problem input. This includes both the graph representation in main memory as well as in GPU memory.

Ontology	$\|N\|$	$\|E\|$	mean edge degree	edge degree variance
OpenGalen	1.8M	8.7M	4.69	337k
IMDb	4.6M	29.8M	6.45	743
GeoSpecies	389k	4.1M	10.74	81k
Jamendo	484k	2M	4.32	48k

Table 1: Ontologies used for Runtime Testing

The ontologies chosen were (cf. Table 1):

OpenGalen An open source medical terminology, in development since 1990. The comparitively low edge count and high edge degree variance made it an interesting test case, since it introduced high branch divergence and poor load balancing for `cuda_n`. We used the OWL 2.0 based download of the entire GALEN ontology version 8, including Common Reference Model as well as all available extensions. [4]

IMDb The international movie database, a large database containing information about movies, television programs and video games. We built our test set using the plain text data files supplied by IMDb. We restricted the test set to the complete set of movies, actors, actresses, directors, producers and edges between those, as of 14th April 2014. The larger edge count and far lower edge degree variance provided an interesting contrast to OpenGalen above, since both favor the node parallel implementation `cuda_n` over the edge parallel `cuda_e`. [2]

GeoSpecies An ontology containing information about select plant and animal species, including their taxonomic relation and distribution area. Its lower edge and node count make it interesting to close the gap to the substantially smaller SteinLib test cases we will introduce below. [1]

Jamendo An ontology containing information about royalty free music tracks, records and the corresponding artists. It has about half the edge count of GeoSpecies and a comparable edge degree variance, further closing the gap to larger SteinLib test cases. [3]

For further information about the chosen ontologies, refer to *Table 1.* We further used SteinLib [5] test suites B, C, D and E to evaluate our implementation for high terminal count queries on small graphs. Since these queries are suboptimal for the parallelization approach chosen, they promise further insights into use cases where using our approach is *not* feasible. *Figure 5(a)* gives an overview how the SteinLib test cases and the ontology test cases compare in graph size and query terminal count. Note that both x and y-axis have been inverted to keep consistency with *Figure 5(b), 5(c) and 5(d)*, which had to be rotated to

avoid occlusion of high-k values.

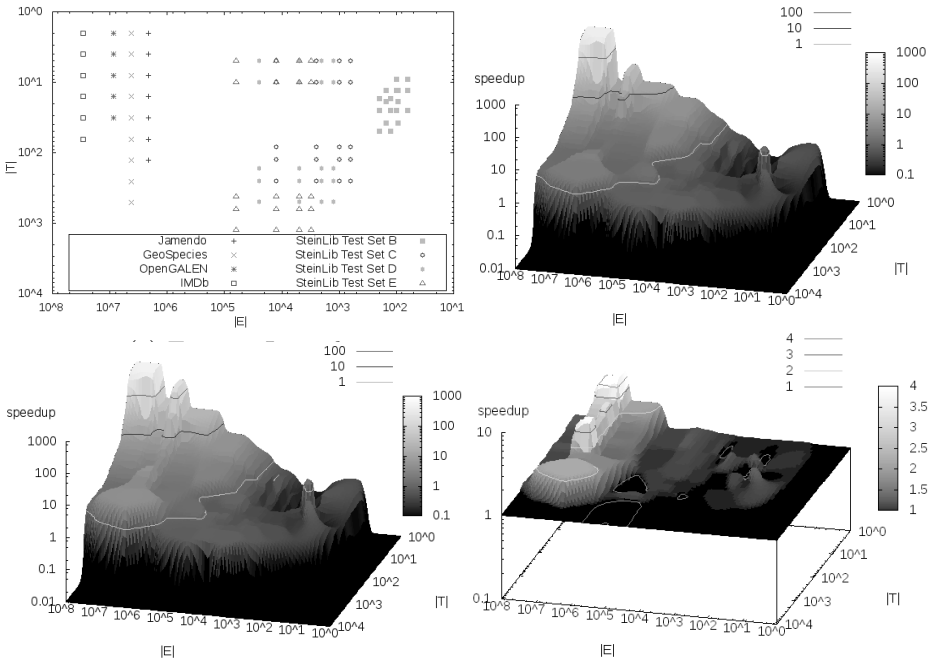
Results. Parallelizing the search operation of `STAR` promises substantial speedups when dealing with large graphs, especially at low query terminal counts. *Figure 5(b)* shows the speedup of `CUDA_N` over `REFERENCE`, while *Figure 5(c)* shows the same information for our edge-parallel approach `CUDA_E`. Both approaches achieved a total runtime speedup over reference of over two orders of magnitude for the larger ontology based test cases and smaller terminal counts.

While the edge-parallel approach incurs overheads due to repeated calculations and additional edges it needs to iterate over, it still outperforms the node-parallel approach consistently, presumably due to better load balancing. This advantage is most pronounced for ontologies with high edge degree variance. *Figure 5(d)* shows the speedup of `CUDA_E` over `CUDA_N`. Note the prominent ridge for OpenGalen, an ontology with high edge degree variance, while speedup for the rather uniform IMDb is much more subtle.

For smaller graphs or queries with very high terminal count, the relative performance of both our parallel implementations suffers compared to the serial reference implementation. This has multiple reasons (cf. *Figure 2*). `STAR` needs to perform a tree split and merge each time a loose path is processed. For high terminal counts, a large number of shortest path searches needs to be conducted, while the effort for each individual search often even sinks. A higher terminal count often results in a larger, more branched solution tree, thus reducing the average distance between tree nodes, resulting in earlier search termination. A smaller graph size on the other hand reduces the overall amount of nodes the search needs to process, resulting in lower search overhead as well. In both cases, the benefit of parallelizing the search is reduced due to the lowered contribution of search to total runtime. For sufficiently short searches, the CPU even outperforms the GPU. We verified these assumptions with synthetic power-law graphs generated according to the Barabasi-Albert model (initial graph size 3, connection factor 2). *Figure 6* shows the time spent creating the initial tree, performing searches, and performing tree splits/merges for graphs of increasing node count.

Figure 6(a) shows results for a query terminal count of 4. The dominant factor contributing to runtime is clearly the time in shortest path searches, followed by the initial tree construction, which is a search as well. Effort for tree splits and merges is largely independent of graph size. This is unsurprising, since the size of the candidate solution tree is largely uneffected by graph size. In these graphs, the amount of nodes reachable along a path grows exponentially with path hop count (i.e. number of edges traversed). Contrast this with *Figure 6(b)*, which shows the results for queries with terminal count 128. The contribution of tree splits and merges increases significantly and now dominates time spent searching. This is not due to an increase in runtime per split/merge, but due to a higher amount of searches performed in total. The search effort for each individual search even decreases compared to the lower terminal count case, thus reducing opportunity for parallelism and runtime contribution of search itself. Since each kernel invocation on GPU adds a small constant runtime overhead

and since both GPU-based approaches iterate over all graph nodes (for `CUDA_N`) or edges (for `CUDA_E`) each search step, a sufficiently high terminal count results in poor performance compared to the serial reference implementation, especially when graph sizes are very small.
On our test platform, the break-even point where both parallel implementations consistently outperform the serial reference implementation is at about 10^4 graph edges, as long as query terminal counts don't exceed 10. For graphs with edge count higher than about 10^5, query terminal counts up to 10^3 still favored the parallel implementations. This of course is heavily dependent on implementation, input graphs and hardware, but the results still show that parallelizing `STAR`'s search heuristic promises substantial speedups for STPG instances in large graphs with moderate to small terminal counts.

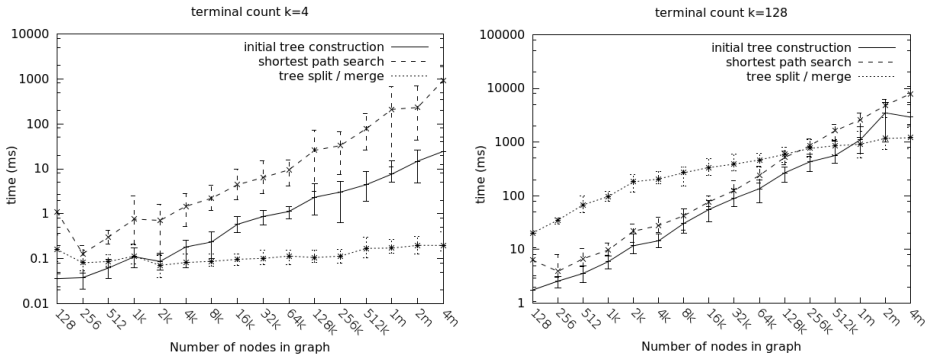


(c) Speedup `CUDA_E` over `REFERENCE` (d) Speedup `CUDA_E` over `CUDA_N`

Fig. 5: Speedups

5 Conclusions

While serial `STAR` itself is already a very fast heuristic for the Steiner tree problem in graphs, its runtime for queries with low terminal counts on large graphs is heavily dominated by its search heuristic. Parallelizing this search heuristic offers substantial speedups using affordable consumer graphics hardware. We evaluated



(a) $k = 4$ (b) $k = 128$

Fig. 6: STAR Component Scaling

both the viability of a standard node-parallel search approach and an edge-parallel improvement thereupon, and found both to be highly viable as long as graph sizes are large and query sizes are small enough. The edge-parallel approach promises much more robust performance when graphs with high edge degree variance are encountered. In our tests, both parallel implementations required graphs larger than about 10^4 edges to play to their strengths, limiting their feasibility on high-k problem instances in smaller graphs. Due to its more stable performance and consistently higher speedup, the edge parallel approach seems preferable to the node-parallel one.

Acknowledgement. This work was supported by the German ministry for education and research (BMBF) in the project INVERSIV under contract number 01IW14004.

References

1. `http://lod.geospecies.org/`
2. `http://www.imdb.com/`
3. `https://www.jamendo.com/`
4. `http://www.opengalen.org/`
5. `http://steinlib.zib.de`
6. Chow, W.K., Li, L., Young, E.F.Y., Sham, C.W.: Obstacle-avoiding rectilinear Steiner tree construction in sequential and parallel approach. Integr. VLSI 47(1), 105–114 (Jan 2014)
7. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
8. Duan, G., Yu, Y.: Power distribution system optimization by an algorithm for capacitated Steiner tree problems with complex-flows and arbitrary cost functions. Electrical Power & Energy Systems 25(7), pp. 515 – 523 (2003)
9. Harris, M.: Optimizing parallel reduction in CUDA. http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf (2007)

10. Hong, S., Kim, S.K., Oguntebi, T., Olukotun, K.: Accelerating CUDA graph algorithms at maximum warp. In: Proc. of 16th ACM Symposium on Principles and Practice of Parallel Programming, New York, NY, USA, 267–276, ACM (2011)
11. Ihler, E., Reich, G., Widmayer, P.: Class Steiner trees and VLSI-design. Discrete Applied Mathematics 90(1-3), 173 – 194 (1999)
12. Jia, X., Wang, L.: A group multicast routing algorithm by using multiple minimum Steiner trees. Computer Communications 20(9), 750 – 758 (1997)
13. Jia, Y., Lu, V., Hoberock, J., Garland, M., Hart, J.: Edge v. node parallelism for graph centrality metrics. In: GPU Computing Gems Jade Edition, 1st edition, chap. 2, 15–30, Morgan Kaufmann, USA (2011)
14. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations, 85–103 (1972)
15. Kasneci, G., Ramanath, M., Sozio, M., Suchanek, F.M., Weikum, G.: STAR: Steiner-tree approximation in relationship graphs. In: Proc. of IEEE International Conference on Data Engineering, Washington DC, USA, 868–879, IEEE (2009)
16. Katz, G.J., Kider, Jr, J.T.: All-pairs shortest-paths for large graphs on the GPU. In: Proc. of 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware. Eurographics Association, Switzerland, 47–55, ACM (2008)
17. Kou, L., Markowsky, G., Berman, L.: A fast algorithm for Steiner trees. Acta Informatica 15(2), pp. 141–145 (1981)
18. Martín, P.J., Torres, R., Gavilanes, A.: CUDA solutions for the SSSP problem. In: Proc. of 9th International Conference on Computational Science ICCS: Part I, 904–913, Springer, (2009)
19. Mathieu, C.: On approximated Steiner tree problem solving with CUDA. Bachelor thesis, Saarland University, Computer Science Department, Saarbruecken, Germany (2014)
20. Muhammad, R.: Distributed Steiner tree algorithm and its application in ad hoc wireless networks. In: Proc. of International Conference on Wireless Networks, 173–178 (2006)
21. Ortega-Arranz, H., Torres, Y., Llanos, D., Gonzalez-Escribano, A.: A new GPU-based approach to the shortest path problem. In: Proc. of International Conference on High Performance Computing and Simulation, 505–511 (2013)
22. Park, J.S., Ro, W., Lee, H., Park, N.: Parallel algorithms for Steiner tree problem. In: Proc. of 3rd International Conference on Convergence and Hybrid Information Technology (ICCIT), vol. 1, 453–455 (2008)
23. Rayward-Smith, V.J.: The computation of nearly minimal Steiner trees in graphs. Mathematical Education in Science and Technology 14(1), 15–23 (1983)
24. Robins, G., Zelikovsky, A.: Improved Steiner tree approximation in graphs. In: Proc. of 11th Annual ACM-SIAM Symposium on Discrete Algorithms SODA, 770–779, Society for Industrial and Applied Mathematics, USA (2000)
25. Takahashi, H., Matsuyama, A.: An approximate solution for the Steiner problem in graphs. Mathematica Japonica 24, 573–577 (1980)
26. Zhong, J., He, B.: Medusa: Simplified graph processing on GPUs. IEEE Trans. Parallel Distributed Systems 25(6), 1543–1552 (2014)