Efficient Parallel Shortest Path Algorithms

David R. Alves, Madan S. Krishnakumar, and Vijay K. Garg

The University of Texas at Austin,
Department of Electrical and Computer Engineering,
Austin, TX 78712, USA
{*dralves,madansk*}*@utexas.edu, garg@ece.utexas.edu*

***Abstract*—Finding the shortest path between nodes in a graph has wide applications in many important areas such as transportation and computer networks. However, the current reference algorithms for this task, Dijkstra's for single threaded environments and Δ-stepping for multi-threaded ones, leave performance and efficiency on the table by not taking advantage of additional information available about the graph. In this paper we present and experimentally evaluate novel algorithms SP_1, SP_2 and $ParSP_2$ that leverage these constraints to solve the problem faster and more efficiently in key metrics. In single threaded execution, we show how SP1 and SP2 out-perform Dijsktra's algorithm by up to 46%. In multi-threaded execution we show how our algorithms compare favorably to Δ-stepping algorithm in the ability to establish the shortest path between the source and the median node.**

***Index Terms*—Single Source Shortest Path Problem, Dijkstra's Algorithm**

I. Introduction

Graphs are increasingly prevalent: from virtual social networks, to physical road networks to everything in between, such as computer networks. When performing computations on graphs, a common problem is that of finding the shortest path (SP) between vertices of the graph. Many algorithms to solve this problem exist. Among the most well known are Disjktra's algorithm for single source shortest path [6] (SSSP), and Δ-stepping [16]. Most algorithms rely on simple edge relaxation and disregard additional information embedded in the structure of the graph, information that, as we show in this paper, can be leveraged to greatly increase the performance and efficiency of the SSSP algorithms.

Dijkstra's algorithm main loop consists in taking vertices off of a heap, marking their previously found distance as final, or *fixed*, and then going through all the outgoing edges to update the distances of the neighbors, adding/updating them in the heap if a lower distance was found. The key insight of algorithms SP_1 and SP_2 is that it is often possible to leverage implicit additional information about the structure of the graph to mark a vertex as *fixed*, i.e to mark its distance as final, without having to add it to the heap thus avoiding a $O(log\ n)$ operation (where n is the number vertices in the heap), using an $O(1)$ operation instead. $SP1$ leverages the notion that when all of the incoming edges to a vertex have been visited, then that vertex is *fixed*, possibly never placing that vertex in the heap. SP_2 additionally takes advantages of another constraint based on the minimum in-weight of a vertex to mark even more vertices as fixed. Algorithms SP_1 and SP_2, for general graphs, have a worst case asymptotic complexity that matches that of Dijkstra's algorithm for a sequential implementation; however, they always perform less heap operations than Dijkstra's algorithm. Additionally, they are more suitable for a parallel implementation because they allow multiple vertices to be explored in parallel unlike Dijkstra's algorithm which explores vertices in the order of their shortest cost. Algorithm SP_2 allows more parallelism than SP_1 at the expense of an additional $O(e)$ (pre-)processing. We present $ParSP_2$ that leverages this parallelism opportunity.

A preliminary version of SP_1 and SP_2 without implementation, optimizations or evaluation were reported in the informal publication [10]. The current work introduces the novel parallel algorithm $ParSP_2$ and the final, formal version of SP_1 and SP_2 with enhancements informed by real-world computing constraints, implementation and experimental evaluation.

In this paper we make the following contributions:

- We present a parallel version of SP_2, $ParSP_2$ and show how it compares favorably to Δ-stepping. Particularly in the ability to find final distances to the median node in the graph faster and behave better as the graph size increases.
- We formalize and conclude work on SP_1 and SP_2 and provide a working implementation which introduces several changes which greatly improve the algorithms' runtime behavior. Our implementation is freely available on Github[1].
- We experimentally evaluate our algorithms and compare them to several alternatives. Based on these results we provide insight as to which circumstances are more favorable for each of the benchmarked alternatives.

The rest of the paper is organized as follows: In Section II we formalize the problem, summarily present our approach and state our assumptions. In Section III we introduce prior work and how it relates to the work presented in this paper. In Sections IV to VII we introduce algorithms SP_1, SP_2 and $ParSP_2$.. In Section IX we introduce our implementation and the experimental design. In Section X we present and discuss our experimental findings and in section XI we summarize our findings and discuss future work.

This work was supported in parts by the National Science Foundation Grants CNS-1812349, CNS-1563544, and the Cullen Trust Endowed Professorship.

[1]https://github.com/dralves/sp1-sp2-galois

II. Preliminaries

Formally, the SSSP problem takes as input a weighted directed graph with n vertices and e edges. We are required to find $cost[x]$, the minimum cost of a path from the *source* vertex v_0 to all other vertices x where the cost of a path is defined as the sum of edge weights along that path.

Most SSSP algorithms are inspired by Dijkstra's algorithm [6] or Bellman-Ford [2], [8]. We present three algorithms in this paper in increasing order of work complexity. Algorithms SP_1 and SP_2 are most suitable for sequential implementations but are faster than Dijkstra's algorithm since they bypass the heap in many cases. Algorithm $parSP_2$ leverages new parallelism opportunities and is scalable with the number of threads, as we'll show in the experimental section. For general graphs, their worst case asymptotic complexity matches that of Dijkstra's algorithm for a sequential implementation; however, they always perform less heap operations than Dijkstra's algorithm.

There are two assumptions in our algorithms. First, we assume that all weights are strictly positive. This is a minor strengthening of the assumption in Dijkstra's algorithm where all weights are assumed to be non-negative. The second assumption is that we have access to incoming edges for any vertex discovered during the execution of the algorithm. Dijkstra's algorithm uses only an adjacency list of outgoing edges. This assumption is also minor in the context of static graphs. However, when the graph is used in a dynamic setting, it may be difficult to find the list of incoming edges. We assume in this paper that either the graph is static or that a vertex can be expanded in the backward direction in a dynamic graph.

Due to lack of space this paper doesn't include extensive proofs of correctness; these can be found in our informal preliminary work [10].

III. Related work

The single source shortest path problem has a rich history [6], [7]. One popular research direction is to improve the worst case complexity of Dijkstra's algorithm by using different data structures. For example, by using Fibonacci heaps for the min-priority queue, Fredman and Tarjan [9] gave an algorithm that takes $O(e+n\log n)$. There are many algorithms that run faster when weights are small integers bounded by some constant [1], [18], [19]. Our algorithms do not improve the worst case sequential complexity of the problem, but avoid many heap operations which produces a significant speedup as we'll show in later sections.

There are many related works for parallelizing Dijkstra's algorithm [4], [11]. The most closely related work is Crauser et al [4] which gives three methods to improve parallelism. These methods, in-version, out-version and in-out-version, allow multiple vertices to be marked as fixed instead of just the one with the minimum D value. The in-version marks as fixed any vertex x such that $D[x] \leq \min\{D[y] \mid \neg fixed(y)\} + \min\{w[v,x] \mid \neg fixed(x)\}$. This method is a special case of our algorithm SP_2. The implementation of in-version in [4] requires an additional priority queue and the total number of heap operations increases by a factor of 2 compared to Dijkstra's algorithm even though it allows greater parallelism. Our algorithm SP_2 uses fewer heap operations than Dijkstra's algorithm. The out-version in [4] works as follows. Let L be defined as $\min\{D[x] + w[x,y] \mid \neg fixed(x)\}$. Then, the out-version marks as fixed all vertices that have D value less than or equal to L. Our method is independent of this observation.

A popular practical parallel algorithm for SSSP is Δ-stepping algorithm due to Meyer and Sanders [16]. Meyer and Sanders also provide an excellent review of prior parallel algorithms in [16]. They classify SSSP algorithms as either *label-setting*, or *label-correcting*. Label-setting algorithms, such as Dijkstra's algorithm, relax edges only for fixed vertices. Label-correcting algorithms may relax edges even for non-fixed vertices. Our algorithms SP_1, SP_2 and $parSP_2$ are label-setting.

Δ-stepping algorithm is a label-correcting algorithm in which eligible non-fixed vertices are kept in an array of buckets such that each bucket represents a distance range of Δ. The parameter Δ provides a trade-off between the number of iterations and the work complexity. For example, when Δ is ∞, the algorithm reduces to Bellman-Ford algorithm where any vertex that has its D label changed is explored. When Δ equals 1 for integral weights, the algorithm is a variant of Dijkstra's algorithm. They show that by taking $\Delta = \Theta(1/d)$ where d is the maximum degree of a graph on n vertices, and random edge weights that are uniformly distributed in $[0,1]$, their algorithm takes $O(n+e+dM)$ where M is the maximum shortest path weight from the source vertex to any other vertex. There are many practical large-scale implementations of the Δ-stepping algorithm (for instance, by Madduri et al [15]) in which authors have shown the scalability of the algorithm. Chakravarthy et al [3] give another scalable implementation of an algorithm that is a hybrid of the Bellman-Ford algorithm and the Δ-stepping algorithm.

In summary, we present two single threaded algorithms for SSSP in this paper in order of increasing work complexity, SP_1 and SP_2 and present a parallel algorithm, $ParSP_2$ that uses the same techniques but in a multi-threaded setting. We only compute the cost of the shortest paths and not the actual paths because the standard method of keeping backward parent pointers is applicable to all of our algorithms. Algorithm SP_1 counts the number of incoming edges to a vertex that have been relaxed. When all incoming edges have been relaxed, we show that it is safe to mark this vertex as fixed. The algorithm SP_2 generalizes SP_1 to allow even those vertices to be marked as fixed which have incoming edges from non-fixed vertices under certain conditions. Both of these algorithms have fewer heap operations than Dijkstra's algorithm for the sequential case and allow more parallelism when multiple cores are used.

IV. Algorithms

In this section we introduce the algorithms which are the focus of this paper. We informally presented a preliminary version of SP_1 and SP_2 in [10] which includes proofs of correctness, so we will not restate the proofs here. For completeness we will describe the original algorithms, along

with the changes we've introduced in the context of this paper. We also present a parallel version of SP_2 ($ParSP_2$).

Dijkstra's algorithm (or one of its variants) is the most popular single source shortest path algorithm used in practice. For concreteness sake we use the version shown in Fig. 1 for comparison with our algorithm. The algorithm also helps in establishing the terminology and the notation used in our algorithm.

We consider a directed weighted graph (V, E, w) where V is the set of vertices, E is the set of directed edges and w is a map from the set of edges to positive reals (see Fig. 2 for a running example). To avoid trivialities, we assume that the graph is loop-free and every vertex x, except the source vertex v_0, has at least one incoming edge.

```
var D: array[0 . . . n − 1] of integer
        initially ∀i : D[i] = ∞;
    fixed: array[0 . . . n − 1] of boolean
        initially ∀i : fixed[i] = false;
    H: binary heap of (j, d) initially empty;
D[0] := 0;
H.insert((0,D[0]));
while ¬H.empty() do
    (j, d) := H.removeMin();
    fixed[j] := true;
    forall k: ¬fixed(k) ∧ (j, k) ∈ E
        if (D[k] > D[j] + w[j, k]) then
            D[k] := D[j] + w[j, k];
            H.insertOrAdjust (k, D[k]);
endwhile;
```

Fig. 1: Dijkstra's algorithm to find the shortest paths to all nodes from v_0 .

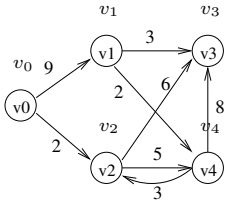

Fig. 2: A Weighted Directed Graph

Dijkstra's algorithm maintains $D[i]$, which is a tentative cost to reach v_i from v_0. Every vertex x in the graph has initially $D[x]$ equal to ∞. Whenever a vertex is discovered for the first time, its $D[x]$ becomes less than ∞. We use the predicate $discovered(x) \equiv D[x] < \infty$. The variable D decreases for a vertex whenever a shorter path is found due to edge relaxation.

In addition to the variable D, a boolean array *fixed* is maintained. Thus, every discovered vertex is either *fixed* or *non-fixed*. The invariant maintained by the algorithm is that if a vertex x is *fixed* then $D[x]$ gives the final shortest cost from vertex v_0 to x. If x is *non-fixed*, then $D[x]$ is the cost of the shortest path to x that goes only through fixed vertices.

A heap H keeps all vertices that have been discovered but are non-fixed along with their distance estimates D. We view the heap as consisting of tuples of the form $(j, D[j])$ where the heap property is with respect to D values. The algorithm has one main *while* loop that removes the vertex with the minimum distance from the heap with the method H.removeMin(), say v_j, and marks it as fixed. It then *explores* the vertex v_j by relaxing all its adjacent edges going to non-fixed vertices v_k. The value of $D[k]$ is updated to the minimum of $D[k]$ and $D[j] + w[j, k]$. If v_k is not in the heap, then it is inserted, else if $D[k]$ has decreased then the label associated with vertex k is adjusted in the heap. We abstract this step as the method H.insertOrAdjust$(k, D[k])$. The algorithm terminates when the heap is empty. At this point there are no discovered non-fixed vertices and D reflects the cost of the shortest path to all discovered vertices. If a vertex j is not discovered then $D[j]$ is infinity reflecting that v_j is unreachable from v_0.

V. Algorithm SP_1: Using predecessors

Dijkstra's algorithm finds the vertex with the minimum tentative distance and marks it as a fixed vertex. This is the only mechanism by which a vertex is marked as *fixed* in Dijkstra's algorithm. Finding the non-fixed vertex with the minimum D value takes $O(\log n)$ time when a heap or its variant is used. Our first observation is that if for any non-fixed vertex x, if all the incoming edges are from fixed vertices, then the current estimate $D[x]$ is the shortest cost. To exploit this observation, we maintain with each vertex i, a variable $pred[i]$ that keeps the number of incoming edges that have not been relaxed. The variable $pred[i]$ is decremented whenever an incoming edge to vertex i is relaxed. When $pred[i]$ becomes zero, vertex i becomes fixed. Determining a vertex to be fixed by this additional method increases the rate of marking vertices as fixed in any iteration of the while loop.

The second observation is that in Dijkstra's algorithm vertices are explored only in order of their cost. SP_1 explores vertices whenever it finds one that is fixed. Hence, in addition to the heap H, we maintain a set R of vertices which have been fixed but not explored, i.e., their adjacency lists have not been traversed. We also relax the invariant on the heap H. In Dijkstra's algorithm, the heap does not contain fixed vertices. In algorithm SP_1, the heap H may contain both fixed and non-fixed vertices. However, only those fixed vertices which have been *explored* may exist in the heap.

The algorithm SP_1 is shown in Fig. 3. The algorithm starts with the insertion of the source vertex with its D value as 0 in the heap. The algorithm consists of two *while* loops. The outer while loop removes one vertex from the heap. If this vertex is fixed, then it has already been explored and therefore it is skipped; otherwise, it is marked as fixed and inserted in R to start the inner while loop. The inner loop keeps processing the set R till it becomes empty.

We do not require that vertices in R be explored in the order of their cost. If R consists of multiple vertices then all of them can be explored in parallel. During this exploration other non-fixed vertices may become fixed. These are then added to R. The vertices $z \in R$ are explored as follows. We process all out-going adjacent edges (z, k) of the vertex z to non-fixed vertices k. This step is called *processEdgeSP1* in Fig. 3. First, we decrement the count $pred[k]$ to account

```
var D: array[0 . . . n − 1] of integer
        initially ∀i : D[i] = ∞;
    H: binary heap of (j, d) initially empty;
    fixed: array[0 . . . n − 1] of boolean
        initially ∀i : fixed[i] = false;
    Q, R: set of vertices initially empty;
    pred: array[0 . . . n − 1] of integer
        initially ∀i : pred[i] = | {x | (x, v_i) ∈ E} | ;

D[0] := 0;
H.insert((0, D[0]));
while ¬H.empty() do
    (j, d) := H.removeMin();
    if (¬fixed[j]) then
        R.insert(j);
        fixed[j] := true;
        while R ≠ {} do
            forall z ∈ R
                R.remove(z);
                forall k : ¬fixed(k) ∧ (z, k) ∈ E:
                    processEdgeSP1(z, k);
        endwhile;
        forall z ∈ Q:
            Q.remove(z);
            if ¬fixed[z] then
                H.insertOrAdjust (z, D[z]);
endwhile;

procedure processEdgeSP1(z, k);
 var changed: boolean initially false;
 pred[k] := pred[k] − 1;
 if (D[k] > D[z] + w[z, k]) then
     D[k] := D[z] + w[z, k];
     changed := true;
 if (pred[k] = 0) then
     fixed[k] := true;
     R.insert(k);
 else if (changed ∧ (k ∉ Q)) then
     Q.insert(k);
```

Fig. 3: Algorithm SP_1

for its predecessor z being fixed. Then, we do the standard edge-relaxation procedure by checking whether $D[k]$ can be decreased by taking this edge. If $pred[k]$ is zero, k is marked as fixed. Setting $fixed[k]$ to true also removes it effectively from the heap because whenever a fixed vertex is extracted in the outer while loop it is skipped.

Finally, if $D[k]$ has decreased and $pred[k]$ is greater than 0, we insert it in the heap with $H.insert$.

Consider the graph in Fig. 2. Initially $(0, D[0])$ is in the heap H. Since there is only one vertex in the heap H, it is also the minimum. This vertex is removed and inserted in R marking v_0 as fixed. Now, outgoing edges of v_0 are relaxed. Since $pred[1]$ becomes 0, v_1 is marked as fixed and added to R. The vertex v_2 has $pred$ as 1 and $D[2]$ as 2 after the relaxation of edge (v_0, v_2). The vertex v_2 is inserted in the Q for later insertion in the heap. Since R is not empty, outgoing edges of v_1 are relaxed. The vertex v_3 is inserted in Q and its D value is set to 12. The vertex v_4 is also inserted in Q and its D value is set to 11. At this point R is empty and we insert vertices in Q in H and get back to the outer while loop. Continuing in this manner, the algorithm terminates with D array as $[0, 9, 2, 8, 7]$.

The following lemma underlies SP_1:

Lemma 1. *Let v be any non-fixed vertex. Suppose all incoming edges of v have been relaxed, then $D[v]$ equals $cost[v]$.*

We next present the time complexity of the algorithm SP_1. Due to space constraints the full proof of the complexity study as well as of Lemma 1 is presented in [10], but the conclusion is captured in the following theorem:

Theorem 1. *SP_1 takes $O(e + n \log n)$ time with Fibonacci heaps for any directed graph and takes $O(e)$ time for directed acyclic graphs in which source node is the only one with with zero incoming edges.*

The worst case for SP_1 is when the vertex discovered last has outgoing edges to all other vertices. In such a worst-case scenario, SP_1 will not have any vertex that becomes fixed through processing of R and the algorithm will degenerate into Dijkstra's algorithm.

VI. Algorithm SP_2: Using weights on incoming edges and known minimums

We now strengthen our mechanism to mark vertices as fixed. SP_2 requires access to incoming edges for any vertex. If (v, k) is an edge, then we call v a *predecessor* of k. Note that predecessor is not an acyclic relation and k may also be a predecessor of v. Let a vertex k be discovered from a predecessor vertex z. Then, we compute $inWeight[k]$ as the minimum weight of incoming edges from all predecessors other than z. We exploit $inWeight$ as follows.

Lemma 2. *Let k be any non-fixed vertex discovered from the vertex z in any iteration of the outer while loop with d. If $(D[k] \leq d + inWeight[k])$ then $D[k]$ equals $cost[k]$.*

This mechanism comes at the space overhead of maintaining an additional array $inWeight[]$ indexed by vertices.

```
inWeight: array [0 . . . n − 1] of int
    initially ∀i : inWeight[i] = ∞;
procedure processEdgeSP2(z, k);
var changed: boolean initially false;
pred[k] := pred[k] − 1;

// Step 1: vertex k has been discovered.
// Compute inWeight
if (D[k] = ∞) ∧ (pred[k] > 0) then
    inWeight[k] := min{w[v, k] | (v, k) ∈ E, v ≠ z};

// Step 2: relax (z, k) edge
if (D[k] > D[z] + w[z, k]) then
    D[k] := D[z] + w[z, k];
    changed := true;

// Step 3: check if vertex k can be fixed.
if ((pred[k] = 0) ∨ (D[k] ≤ d + inWeight[k]) then
    fixed[k] := true;
    R := R.insert(k);
else if (changed ∧ (k ∉ Q)) then Q.insert(k);
```

Fig. 4: Algorithm SP_2: Algorithm SP_1 with *processEdgeSP2*

After incorporating Lemma 2, we get the algorithm SP_2 shown in Fig. 4. It is same as SP_1 except we use the procedure $processEdgeSP2$ instead of $processEdgeSP1$. In step 1, we compute $inWeight[k]$ when it is discovered for the first time, i.e., when $D[k]$ is ∞. If there are additional incoming edges, i.e., $(pred[k] > 0)$, we determine the minimum of all the

incoming weights except from the vertex z that discovered k. In step 2, we perform the standard edge-relaxation. In step 3, we check if the vertex k can be fixed either because it has no more predecessors, or for any non-fixed predecessor v, the relaxation of the edge (v, k) will not change $D[k]$. Observe that for sequential implementations, if R is maintained as a queue and all edge weights are uniform, then any vertex discovered for the first time will always be marked as fixed and will never be inserted in the heap. For such inputs, SP_2 will behave as a simple breadth-first-search.

Since any vertex is discovered at most once, computing $inWeight$ requires processing of all incoming edges of a vertex at most once. Hence, the cumulative time overhead is linear in the number of edges. If the graph is unweighted, then SP_2 is much faster than Dijkstra's algorithm when R is implemented as a queue.

We next present the time complexity of the algorithm SP_2. Due to space constraints the full proof of the complexity study as well as of Lemma 2 is presented in [10], but the conclusion is captured in the following theorem:

Theorem 2. *Suppose that R is implemented as a simple queue. SP_2 takes*

- $O(e+n\log n)$ *time with Fibonacci heaps for any directed graph,*
- $O(e)$ *time for directed acyclic graphs in which only the source node has zero incoming edges,*
- $O(e)$ *time for any unweighted directed graph.*

Hence, SP_2 unifies Dijkstra's algorithm with the topological sort for acyclic graphs as well as the breadth-first search for unweighted graphs. Consequently, it is faster than Dijkstra's algorithm when the input graph is close to an acyclic graph (i.e., has few cycles) or close to an unweighted graph (most weights are the same).

Lemma 2 captures a constraint similar to the one used in the in-version method of [4]. The in-version fixes any vertex k such that $D[k] \leq d + \min\{w[j,k] \mid \neg fixed(j), (j,k) \in E\}$. There are two differences. First, we do not include the weight of the edge that discovered k in our calculation of $inWeight$. Second, in [4] the implementation is based on maintaining an additional priority queue which adds the overhead of $O(e \log n)$ to the algorithm with ordinary heap implementation. SP_2 adds a cumulative overhead of $O(e)$. In sequential implementations, the in-version increases the number of heap operations, whereas SP_2 decreases this number.

VII. Algorithm $ParSP_2$: Leveraging parallelism opportunities

One of the key differences between our algorithms and Dijskstra's algorithm is that our algorithms have more opportunity for parallelism. In this section we present an algorithm that leverages this opportunity.

Figure 5 shows algorithm $ParSP_2$, which is a parallel version of SP_2 (commonalities with the previous algorithms removed for conciseness). This algorithms uses thread-safe data-structures for H and R. The algorithm executes as follows:

```
D[0] := 0;
H.insert((0, D[0]));
do in parallel
        if R ≠ {}:
            R.remove(z);
            forall k : ¬fixed(k) ∧ (z, k) ∈ E:
                processEdgeSP2(z, k, H);
            endfor;
            forall z ∈ Q:
                Q.remove(z);
                if ¬fixed[z] then
                    H.insertOrAdjust (z, D[z]);
        if R = {}: do single thread
            if H ≠ {}:
                (j, d) := H.removeMin();
                if (¬fixed[j]) then
                fixed[j] := true;
                R.insert(j);
while R ≠ {} ∨ H ≠ {}
```

Fig. 5: Algorithm $ParSP_2$

Multiple threads execute the main loop of the algorithm. While there are elements in R individual threads remove elements and proceed to execute $processEdgeSP2$ in parallel. If R is empty threads proceed to add remaining elements from Q to the heap and a single thread pops an element from the heap H and adds it to R. This continues until there are no elements in both R and H, in which case the algorithm is done.

One of the main reasons Dijkstra's algorithm is notoriously hard to parallelize is because its only underlying data-structure, the heap, is hard to parallelize itself. In fact implementing something analogous to the above for Dijkstra's algorithm actually worsens performance in both single and multi-threaded scenarios. This happens because, in the case of Dijkstra's algorithm, heap operations are the bulk of all operations and if these operations needs to be protected by a mutex, not only does this negate any advantage obtained from parallelization but also the additional context-switching penalty and synchronization penalty actually increase execution time. While $ParSP_2$ uses a trivially thread-safe heap H protected by mutex, R can be made much more efficient to use in a multi-thread scenario. Because R imposes no ordering constraints whatsoever it can be implemented using parallel efficient data-structures. In our implementation we leverage Galois's per-socket FIFO list for R [13]. This is a list that is partially thread-local, avoiding synchronization penalties when each thread can produce work for itself, only reverting using a mutex when this is no longer the case and work needs to be pulled from other threads. Similarly Q can be implemented as a thread-local set. There might be other parallel data-structures that yield even better results, such as compare-and-set based ones, but we leave exploring that aspect for future work.

VIII. Optimizations

As we set out to produce a high-performance implementation of SP_1 and SP_2, we noticed that some details hinder real-world performance, so we introduced a few empirically-driven changes, the more relevant of which we describe in this section.

A. Allow duplicates in the heap

High performance implementations of simple binary heaps are widely available, including in C++ the language we selected to implement the algorithms. The original implementation required an $insertOrAdjust$ method in the heap, to avoid adding a vertex to the heap more than once, thus minimizing its size. This method is not available in the commonly available array-based heaps. Adding this method required the heap implementation to grow in complexity to the point where this complexity would hurt the algorithm's runtime performance by almost one order of magnitude. We considered other types of heaps, such as Binomial Heaps and Fibonacci Heaps. In practice, these data structures hinder caching and force memory jumps which negate any theoretical advantage, at least in the case of the algorithms described in this paper.

B. Heap compaction

The implementation of both our algorithms and Dijkstra's algorithm allows for duplicates in the heap, causing it to have extra "garbage" but improving its overall performance. However, both SP_1 and SP_2 additionally often mark nodes as *fixed* even if they are already in the heap. This means they will be thrown out on $H.pop$ and are just contributing to the height of the heap, thus worsening performance. To partially mitigate this problem we implemented a compaction mechanism that, on $insert$ and pop, before adding a new element goes through the end of the vector underlying the binary heap and discards vertices until it finds one node that is not *fixed*. This simple method helps mitigate long tail behavior we observed in our experiments: even though SP_1 and SP_2 would be considerably faster than Dijkstra's algorithm for the majority of the execution, they would tend to converge on the execution time of Dijskstra's algorithm near the end of the algorithm's run. This is because a lot of cycles were spent removing vertices from the heap even though they had already been *fixed*.

IX. Implementation and Experimental Design

In this section we present our implementation of the algorithms introduced in the previous section, as well as the experimental design underlying the results presented in the next section.

A. Code & Runtime framework

We chose the Galois Library [13] as our underlying runtime framework. Using Galois allows us to explore our own algorithms experimentally while relying on thoroughly tested parallel and benchmark constructs. Furthermore, Galois already provides high performance implementations of Dijskstra's algorithm and Δ-stepping, which provide an unbiased baseline for comparison with our own algorithms.

Our implementation of SP_1 SP_2 and $ParSP_2$, as well as the baseline algorithms for comparison Dijkstra's algorithm and Δ Stepping, is available as open source code on Github[2] under the same license as Galois itself (3-Clause BSD License). Nearly all of SSSP code relevant for this paper is contained in the *SSSP.cpp* file.

[2]https://github.com/dralves/sp1-sp2-galois

B. Measurements and metrics

In our experimental evaluation we focus on two core metrics, **Total Runtime** and **Median time to fixedness**, both in terms of their absolute values and in comparison to the baseline performance of Dijkstra's algorithm, referred to as **Speedup**. *Total runtime*, i.e. the time is takes to find the shortest path from previously chosen source vertex to all the other vertices, captures the algorithm overall performance. However, by itself, *Total runtime* is lacking in terms of capturing how fast individual nodes are marked as *fixed*. Capturing this last aspect is important because, in real world scenarios, finding the Shortest Path is seldom the goal of the computation. In fact, in most situations computing the shortest path is but a step in a larger computation and, as such, the sooner the follow on computation can start, the better. This advantage is captured in the *Median time to fixedness* metric, captures the median time that each algorithm takes to mark vertices as fixed (and thus allowing the follow up computation to start). We chose the median versus the average as it is statistically more robust to outliers and thus better captures real-world performance.

In the terms introduced in Section III, only *label-setting* algorithms (as is the case of Dijkstra's algorithm, SP_1, SP_2 and $ParSP_2$) are able to allow follow on computations to start early when the shortest path to a particular destination is found. *Label-correcting* algorithms, like Δ-stepping must run to completion which makes them very inneficient for this particular aspect. Thus, the *Median time to fixedness* in the case of Δ-stepping is the same as *Total runtime*, our experiments take this fact into account.

C. Graphs

The graphs considered for the experiments were either taken from graph challenge datasets or generated using synthetic graph generators. The Galois library provides a diverse set of graphs which includes random, scalefree, structured and road graphs [13]. Kronecker graphs used in the experiment were either generated using the SNAP kronecker graph generator [14] or obtained from graph challenge datasets like DIMACS [5], graph500 [17]. We additionally used graphs generated with ParMAT [12] and, in some cases, proceeded to eliminate cycles to generate directed acyclic graphs. Table I lists the graphs used in our experimental evaluation and their original names, to facilitate reproduceability.

D. Setup

The experimental system is a single SkyLake node of the Stampede2 supercomputer at the Texas Advanced Computing Center. Each SkyLake node is a 2-socket Intel(R) Xeon(R) CPU - Platinum 8160 @ 2.1GHz with 24 cores per socket and two threads per core and 192GB of RAM. Our experiments were limited to a maximum of 32 threads constrained to a

Dataset	Original Name	Name used	Type
Galois	USA Roads	USA Roads - 23M	road
Galois	r4-2e26	RMAT 2e26	random
DIMACS	coAuthorsDBLP	coAuthorsDBLP - 50k	citation
DIMACS	coPapersDBLP	coPaperDBLP - 400K	citation
DIMACS	belgium.osm.graph.bz2	Belgium Cities	road
Graph500	graph500-s21-ef16	Graph500 1.2M	scalefree
Graph500	graph500-s23-ef16	Graph500 4.5M	scalefree
SNAP	kron-2e23	Kronecker 2e24	scalefree
SNAP	kron-2e28	Kronecker 2e28	scalefree
SNAP	kron-2e30	Kronecker 2e30	scalefree
SNAP	kron-2e32	Kronecker 2e32	scalefree

TABLE I: Graphs used in experimental evaluation

single socket, both to exclude possible NUMA effects and to avoid saturating the CPU.

We use the Release build of our own fork of the Galois library for our experiment (available on github). The Galois library includes reference single source shortest path implementations of Dijkstra and Delta-stepping algorithms. The experiment uses Intel C++ Compiler version 18.0.2 and Boost C++ libraries.

X. Experimental Results

In this section we will present our experimental findings, derive insight and provide commentary on particularly interesting results.

A. Single Threaded Evaluation

In order to establish a baseline for the remaining experiments, the first experiment measures the speed up both for *Total Runtime* and *Median time to fixedness* of the non-parallel versions of SP_1 and SP_2 versus Dijkstra's algorithm.

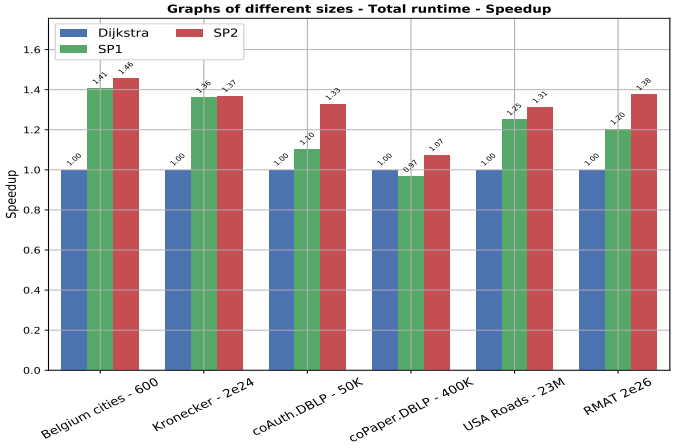

Fig. 6: Total runtime on graphs of various types/scales - Speedup vs Dijkstra's algorithm (higher is better))

As we can observe from Fig. 6 SP_2 always strictly outperforms Dijskstra's algorithm, presenting speedups from 1.07 up to 1.46, or between 7 and 46% faster. Perhaps more interesting is that, while SP_2 is consistently better than Dijkstra's algorithm, SP_1's performance varies quite a bit more, including actually being slower than Dijkstra's algorithm in for the coPapersDBLP graph. This implies that SP_1 single condition for fixedness is more sensitive to the graph morphology and that SP_2 is more robust to different types and scales of graphs, which makes sense since SP_2 also includes SP_1's condition.

B. Multi Threaded Evaluation

While SP_2 demonstrably works well on graphs of all scales, larger graphs can benefit from multi-threaded algorithms like $ParSP_2$. In the next experiment we evaluate how Dijkstra's algorithm, SP_1 and SP_2 compare with $ParSP_2$.

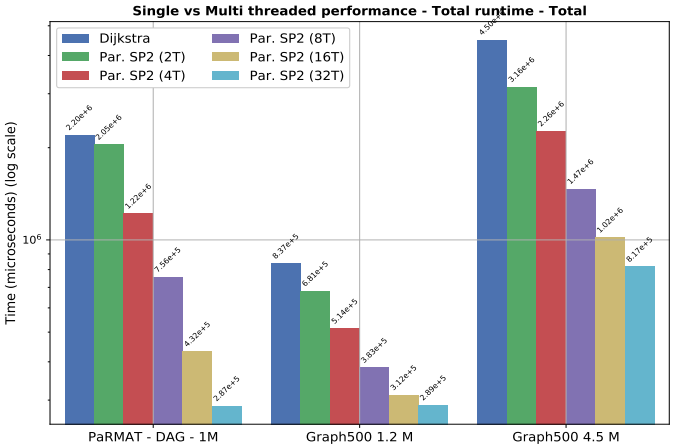

Fig. 7: Total runtime - ST vs MT (lower is better, log scale)

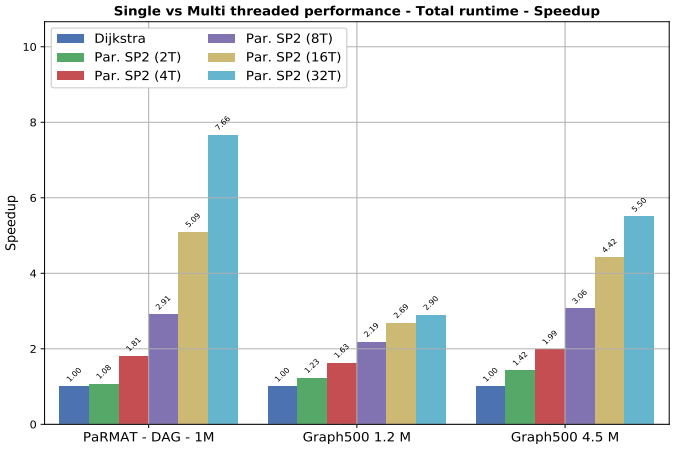

Fig. 8: Speedup: ST vs MT (higher is better)

Figures 7 and 8 shows the results of this experiment. Note that times are displayed in a logarithmic scale.

When executing with two threads, $ParSP_2$ is marginally faster than Dijkstra's algorithm in terms of total runtime, but it is able to mark the median vertex as fixed about 50% faster (not displayed). Looking the different types of graphs, we can see that $parSP_2$ scales well with the increase of number of threads, particularly in the case of the DAG graph. Note also that on graphs of the same morphology but different sizes, (Graph500 1.2M and 4.5M), the speedup of $ParSP_2$ increases from Graph500 1.2M to Graph500 4.5M for the same number

of threads. This happens because there are more opportunities for parallelism as the graph is bigger. We will explore this aspect more thoroughly in the next experiment.

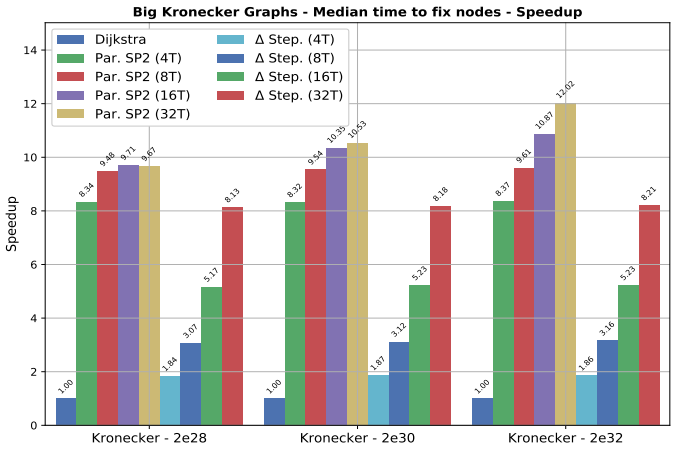

Fig. 9: Median time to 'fix' vertices - Speedup vs Δ-stepping (higher is better)

The final experiment that we performed was to evaluate the scalability of $ParSP_2$ as we increase the number of threads and the size of the graphs. We chose graphs of identical morphologies but different scales and compared all graphs with Δ-stepping. Δ-stepping is well known for displaying a good ability to scale to multiple threads as can be seen in Figure 9. However this ability doesn't translate well when the size of the graph increases. As we can see the speedup relative to Dijkstra's algorithm *Mean time to fixedness* remains constant as the graph size increases. On the other hand $ParSP_2$'s speedup increases with the size of the graph, for the same number of threads. This makes sense since as the graph gets bigger there are more opportunities to keep nodes in the R list and thus mark nodes as *fixed* while bypassing the heap. We can thus conclude that, while Δ-stepping scales better with the number of threads for the same size graph, our algorithm scales better with the size of the graph, for the same number of threads.

XI. Conclusions and Future work

In this paper we presented and experimentally evaluated three novel *label-setting* algorithms for the Shortest Path problem: SP_1 and SP_2 are single-threaded algorithms that take advantage of available information about the graph to mark nodes as *fixed*, avoiding heap insertions and removals; $ParSP_2$ is a multi-threaded algorithm that leverages the parallelism opportunities created by SP_1 and SP_2. We present an implementation of our algorithms and an evaluation that shows how they demonstrate significant speedups in several types of graphs, not only in terms of *total runtime*, but also in the median time that it takes to calculate the shortest path from the source vertex to any given node.

In future work, we will explore possible improvements to *label-correcting* algorithms like Δ-stepping, using similar techniques, namely the possibility of using *fixedness* as a prioritization metric that drives what is processed in each round, together with the currently used distance.

References

[1] Ravindra K Ahuja, Kurt Mehlhorn, James Orlin, and Robert E Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM (JACM)*, 37(2):213–223, 1990.

[2] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[3] Venkatesan T Chakaravarthy, Fabio Checconi, Prakash Murali, Fabrizio Petrini, and Yogish Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):2031–2045, 2017.

[4] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A parallelization of dijkstra's shortest path algorithm. In *International Symposium on Mathematical Foundations of Computer Science*, pages 722–731. Springer, 1998.

[5] Camil Demetrescu, Andrew V Goldberg, and David S Johnson. The Shortest Path Problem, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, November 13-14, 2006. *The Shortest Path Problem*, 2009.

[6] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959. URL: https://doi.org/10.1007/BF01386390.

[7] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, April 1972. URL: https://doi.org/10.1145/321694.321699.

[8] L. A. Ford. Network flow theory. Technical report, 1956.

[9] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987. URL: http://doi.acm.org/10.1145/28869.28874.

[10] Vijay K Garg. Removing Sequential Bottleneck of Dijkstra's Algorithm for the Shortest Path Problem. *arxiv.org*, December 2018. URL: http://arxiv.org/abs/1812.10499.

[11] Michael Kainer and Jesper Larsson Träff. More parallelism in dijkstra's single-source shortest path algorithm, 2019. URL: http://arxiv.org/abs/1903.12085.

[12] Farzad Khorasani, Rajiv Gupta 0001, and Laxmi N Bhuyan. Scalable SIMD-Efficient Graph Processing on GPUs. *PACT*, 2015.

[13] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009. URL: http://iss.ices.utexas.edu/Publications/Papers/ispass2009.pdf.

[14] Jure Leskovec and Rok Sosič. SNAP: A General Purpose Network Analysis and Graph Mining Library. *ACM transactions on intelligent systems and technology*, 8(1):1–20, October 2016.

[15] Kamesh Madduri, David A Bader, Jonathan W Berry, and Joseph R Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 23–35. SIAM, 2007.

[16] Ulrich Meyer and Peter Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[17] R C Murphy, K B Wheeler, BW Barrett Cray Users Group, and 2010. Introducing the graph 500. *richardmurphy.net*.

[18] Rajeev Raman. Recent results on the single-source shortest paths problem. *SIGACT News*, 28(2):81–87, June 1997. URL: http://doi.acm.org/10.1145/261342.261352.

[19] Mikkel Thorup. On ram priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000.