2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 978-1-6654-9747-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPSW55747.2022.00131

Parallel Minimum Spanning Tree Algorithms via Lattice Linear Predicate Detection

David R. Alves and Vijay K. Garg
The University of Texas at Austin
Department of Electrical and Computer Engineering
Austin, TX 78712, USA
Email: dralves@utexas.edu, garg@ece.utexas.edu

***Abstract*—We show that the problem of computing the minimum spanning tree can be formulated as special case of detecting Lattice Linear Predicate (LLP). In general, formulating problems as LLP presents two main advantages: 1) Different problems are formulated under a single, general framework, which defines the problem in terms of simple local predicates that must hold for the all the elements of a lattice, making the problem (and the solution) compact and easy to understand. 2) improvements on one set of problems can be transferable to other sets of problems; 3) since the problems are stated as a set of local predicates, which can be often tested with little or no synchronization it is often the case that new opportunities for parallelism present themselves.**

In this paper we introduce two parallel algorithms LLP-Prim and LLP-Boruvka that improve on the non-LLP counterparts in several ways. LLP-Prim reduces the number of heap operations required by Prim by allowing edges to be selected without entering the heap thus allowing for parallelism. LLP-Boruvka improves on Boruvka by reducing synchronization and thus once more improving parallelism opportunities.

Our experimental evaluation shows that LLP-Prim is faster than Prim's algorithm in both single threaded and multithreaded scenarios and that it provides a good tradeoff between parallelism and efficiency at low core counts. For higher core count scenarios we show how LLP-Boruvka improves on an efficient implementation of a parallel version of Boruvka.

***Index Terms*—Minimum Spanning Tree, Parallel Algorithms**

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 minimum spanning tree (MST) between vertices of the graph. In a fully connected graph, i.e. one where all the vertices are connected by at least one edge, the MST is a such such that it connects all the vertices in the graph in a way that minimizes the sum of the weights of the chosen edges. If the graph is not fully connected, then we have a minimum spanning forest (MSF), i.e. a set of minimum spanning trees each with the aforementioned property.

There are many well known algorithms that solve this problem. Among the most well known are Prim's [14], Boruvka's [11] and Kruskal's [7] algorithm for the MST. While these algorithms it's many derivations have been widely studied, we show how formulating Prim's and Boruvka's as a special case of detecting Lattice-Linear Predicates [1] opens the door for new efficiencies, and especially new opportunities for parallelization that result in a speedup experimentally. In this paper we introduce two new algorithms, *LLP-Prim* and *LLP-Boruvka*: First, we introduce the LLP formulation for these algorithms, and the increased parallelization that can be obtained from the formulation. Then we demonstrate, experimentally, how this parallelization results in speedups compared to efficient implementations of the non-LLP counterparts.

Suppose that we have an undirected weighted graph on n vertices with m edges. Our goal is to find the minimum spanning tree. It is known that if the graph is connected and all edge weights are distinct then there is a unique MST. If the graph is not connected, then there is a unique MSF. While *LLP-Boruvka* considers spanning forests, *LLP-Prim* considers a spanning tree, i.e. assumes the graph is fully connected. We briefly provide an intuition for *LLP-Prim*[1]: We model the search space of the problem in the form of a lattice. For *LLP-Prim* this lattice is the lattice of all possible choices of edges E, on for each of the $n-1$, vertices in the graph V other than the root (v_0). Each solution is represented by vector, G. $G[v]$ corresponds to the edge chosen for vertex v. An element of the lattice is a spanning tree if the set of edges spans all vertices and does not have any cycle. Our goal is to reach the element in the lattice with the minimum weight, which corresponds to the *Minimum Spanning Tree*. Formulating the MST problem as a Lattice-Linear predicate entails defining two functions: $forbidden$ and $advance$ that are applied *independently* to the elements of G. $forbidden(V, E, G, v)$ is a predicate that must be false for each element, for the solution to be valid, while $advance(V, E, G, v)$ is the function that allows to change $G[v]$ such that it will eventually not be $forbidden$. Once no element in G is $forbidden$ then we have our MST. Since each element of G can be tested for $forbidden$ *independently* this produces opportunities for parallelism.

In this paper we make the following contributions:

- We introduce an LLP formulation for Prim's algorithm and a derived serial and parallel implementation for the MST problem that includes novel optimizations.

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

[1]The formulation for *LLP-Boruvka* is different and will be introduced later in the paper

- We introduce an LLP formulation of Boruvka's algorithm and a derived implementation that includes novel optimizations that also applies to the minimum spanning forest problem.
- We experimentally evaluate the implementations of *LLP-Prim* and *LLP-Boruvka* comparing to well-known efficient implementations of Prim and Boruvka. Our implementation is freely available on Github[2].

The rest of the paper is sectioned as follows: In section II, we introduce the general form of Lattice-Linear Predicates. In section III we mention relevant related work. In section IV we give background on the classical Prim and Boruvka algorithms. In section V we introduce novel algorithms *LLP-Prim* and *LLP-Boruvka*. In section VII we experimentally evaluate these algorithms. Finally, in section VIII we make some final remarks and provide directions for future work.

II. Preliminaries: Lattice Linear Predicates

Let L be the lattice of all n-dimensional vectors of reals greater than or equal to zero vector and less than or equal to a given vector T where the order on the vectors is defined by the component-wise natural $\leq$. The minimum element of this lattice is the zero vector. The lattice is used to model the search space of the combinatorial optimization problem. The combinatorial optimization problem is modeled as finding the minimum element in L that satisfies a boolean *predicate* B, where B models *feasible* (or acceptable solutions). We are interested in parallel algorithms to solve the combinatorial optimization problem with n processes. We will assume that the systems maintains as its state the current candidate vector $G \in L$ in the search lattice, where $G[i]$ is maintained at process i. We call G, the global state, and $G[i]$, the state of process i.

Finding an element in lattice that satisfies the given predicate B is called the *predicate detection* problem. Finding the *minimum* element that satisfies B (whenever it exists) is the combinatorial optimization problem. We now define *lattice-linearity* which enables efficient computation of this minimum element. A key concept in deriving an efficient predicate detection algorithm is that of a *forbidden* state. Given a predicate B, and a vector $G \in L$, a state $G[i]$ is *forbidden* (or equivalently, the index i is forbidden) if for any vector $H \in L$, where $G \leq H$, if $H[i]$ equals $G[i]$, then B is false for H. Formally:

Definition 1 (Forbidden State): Given any distributive lattice L of n-dimensional vectors of $\mathbf{R}_{\geq 0}$, and a predicate B, we define $\text{forbidden}(G, i, B) \equiv \forall H \in L : G \leq H : (G[i] = H[i]) \Rightarrow \neg B(H)$.

We define a predicate B to be *lattice-linear* with respect to a lattice L if for any global state G, B is false in G implies that G contains a *forbidden state*. Formally:

Definition 2 (lattice-linear Predicate): A boolean predicate B is *lattice-linear* with respect to a lattice L iff $\forall G \in L : \neg B(G) \Rightarrow (\exists i : \text{forbidden}(G, i, B))$.

[2]https://github.com/dralves/llp-mst

Once we determine j such that $forbidden(G, j, B)$, we also need to determine how to advance along index j. To that end, we extend the definition of forbidden as follows:

Definition 3 (advance): Let B be any boolean predicate on the lattice L of all assignment vectors. For any G, j and positive real $\alpha > G[j]$, we define $\text{forbidden}(G, j, B, \alpha)$ iff

$$\forall H \in L : H \geq G : (H[j] < \alpha) \Rightarrow \neg B(H).$$

Given any lattice-linear predicate B, suppose $\neg B(G)$. This means that G must be advanced on all indices j such that $\text{forbidden}(G, j, B)$. We use a function $advance(G, j, B)$ such that $\text{forbidden}(G, j, B, advance(G, j, B))$ holds whenever $\text{forbidden}(G, j, B)$ is true. With the notion of $advance(G, j, B)$, we have Algorithm 1. Algorithm 1 has two inputs — the predicate B and the top element of the lattice T. It returns the least vector G which is less than or equal to T and satisfies B (if it exists). Whenever B is not true in the current vector G, the algorithm advances on all forbidden indices j in parallel. This simple parallel algorithm can be used to solve a large variety of combinatorial optimization problems by instantiating different $\text{forbidden}(G, j, B)$ and $\alpha(G, j, B)$.

ALGORITHM 1: Algorithm LLP to find the minimum vector at most T that satisfies B

```
vector function getLeastFeasible($T$: vector, $B$: predicate)
var $G$: vector of reals initially $\forall i : G[i] = 0$;
while $\exists j : \text{forbidden}(G, j, B)$ do
    for all $j$ such that $\text{forbidden}(G, j, B)$ in parallel:
        if $advance(G, j, B) > T[j]$ then return null;
        else $G[j] := advance(G, j, B)$;
endwhile;
return $G$; // the optimal solution
```

III. Related Work

The LLP algorithm for combinatorial optimization is proposed in [15] where it is shown that variants of Gale-Shapley algorithm for stable marriage problem, Dijkstra's algorithm and Bellman-Ford algorithm for the shortest path problem, and Gale-Demange-Sotomayor algorithm for the market clearing prices can be derived from the LLP algorithm. In this paper, we add to this list a variant of Prim's algorithm called LLP-Prim algorithm and a variant of Boruvka's algorithm called LLP-Boruvka.

O. Boruvka [11] was one of the first to formulate the minimum spanning tree problem in the 1920's. Boruvka's algorithm starts by considering all the vertices of the graph a *component*. In each iteration the algorithm selects the minimum edge out of each component and then merges the components connected in this way. Another famous sequential algorithm for the minimum spanning tree problem is Prim's algorithm [14]. Prim's algorithm works by, starting from the initial vertex, always adding the minimum known edge to the tree where one vertex in that edge is in the tree and the other is not. The algorithm does this, one edge -*the minimum edge* - at a time, until the set of vertexes in the tree is the same as the set in the graph. By contrast our algorithm explores multiple edges at once, in parallel. Another popular deterministic minimum spanning tree algorithm is Kruskal's algorithm [7]. Kruskal's algorithm works by always selecting the minimum edge across the whole graph and connecting the vertexes on either end, as long as the edge does not form a cycle. Since this algorithm also relies on pulling the minimum element from a heap it is also hard to parallelize, in contrast to the LLP algorithm proposed here which explores multiple edges at the same time, in the R set. All of these three deterministic algorithms have a worst-case complexity of $O(m\ log\ n)$, our algorithms do not improve on this complexity.

A randomized linear time algorithm was proposed by Karger and [5] later demonstrated to run in linear time based on the work developed together with Klein, Tarjan [4] and by Pettie et al. [12]. This algorithm is based using on Boruvka's algorithm together with a linear time MST verifier. This principle was used by Cole, Klein, Tarjan to develop a linear time parallel algorithm [6]. We plan to compare directly with this approach when solving the MST problem with the generic LLP solver, but our approach thus far is different since our *LLP-Prim* algorithm is parallel and deterministic while the aforementioned parallel algorithm isn't and *LLP-Boruvka* only applies the LLP algorithm for each iteration and not globally.

The authors of the Galois framework [8] have studied the parallelization of both the classical Prim and Boruvka algorithms in [13]. In this study they reported the interesting finding that Boruvka's algorithm is highly parallelizable in the beginning, quickly tapering off after that. The authors of the Graph Based Benchmark Suite [2] also provide a highly parallel version of Boruvka's algorithm that they include with their benchmark suite. Out implementation of LLP-Boruvka shares some similarities with the work from Zhou, W. [16], but our formulation and the process through which we get the minimum edge within a component is different.

IV. Background

In this section we cover Prim's and Boruvka's algorithm as to serve as a reference to the LLP versions introduced in section V.

The notion of a *fragment* is crucial in understanding all MST algorithms. A fragment is simply a subtree of the MST. Consider the graph in Fig. 1. The minimum spanning tree in this graph corresponds to the edges $\{2, 3, 4, 7\}$. The subtree formed by edges 3 and 4 is a fragment with three vertices $\{a, b, c\}$ and two edges $\{(a, c), (b, c)\}$.

A crucial property of MST is as follows:

Lemma 1: Let F be a fragment. Let e be the edge with minimum weight that is outgoing form F. Then, $F \cup \{e\}$ is also a fragment.

Proof: In the minimum spanning tree T, there must be at least one edge going out of the fragment F. Let that edge be f. If we add e to T and remove f, we get another tree T' with lower weight than T, a contradiction because we assumed that T is the minimum spanning tree. ■

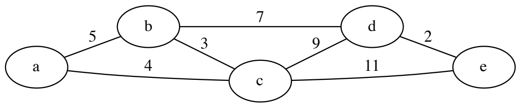

Fig. 1: An undirected weighted graph

A. Prim's Algorithm

Prim's algorithm is a greedy algorithm and is depicted in Algorithm 2. It builds the minimum spanning tree by increasing the size of a single *fragment* by adding the minimum weight outgoing edge of the fragment. It simply exploits Lemma 1 to increase the size of fragment until it becomes the MST. At any stage, Prim's algorithm has a fragment F. It finds the minimum outgoing edge from that fragment e. This edge can be viewed as the edge from the fragment to its nearest neighbor. Therefore, this algorithm is sometimes also known as the *nearest-neighbor-next* algorithm. To find the nearest neighbor, every vertex v maintains a label d which corresponds to the cost of adding v to the fragment. At every iteration, the algorithm chooses the vertex v with the minimum d value and adds it to the fragment. The array $fixed$ keeps track of the vertices in the fragment. Whenever, a new vertex v is *fixed* and added to the fragment, the d values for all adjacent vertex v' are updated as follows. We check whether the weight of the edge (v, v') is lower than the previous value of $d[v']$. If this is true, then $d[v']$ is updated to $w[v, v']$. We also use $parent$ pointer with each node which keeps track of the parent node v that is responsible for the d value of v'.

ALGORITHM 2: Prim: Finding the MST rooted at v_0 .

```
var d: array[0...n − 1] of integer initially $\forall i : d[i] = \infty$;
    $fixed$: array[0...n − 1] of boolean initially
  $\forall i : fixed[i] = false$;
    $H$: binary heap of $(j, key)$ initially empty;// $j$ is the vertex
  and $key$ is the tentative cost
    $parent$: array[0..n − 1] of int initially −1; // gives the
  parent structure of the minimum spanning tree
$d[0] := 0$;
$H$.insert((0,d[0]));
while $\neg H$.empty() do
    $(j, key) := H$.removeMin();
    $fixed[j] := true$;
    forall $k$: $\neg fixed(k) \wedge (j, k) \in E$
        if $(d[k] > w[j, k])$ then
            $d[k] := w[j, k]$;
            $parent[k] := j$;
            $H$.insertOrAdjust $(k, d[k])$;
endwhile;
```

Consider the graph shown in Fig. 1 again. For Prim's algorithm, we start from a fixed node. Suppose we start from the vertex a. Then, the nearest neighbor is c with the cost of 4. The next nearest neighbor to the fragment with vertices $\{a, c\}$ is the vertex b. The cost of adding b is 3. At this point, we have vertices $\{a, b, c\}$ in the fragment. The cost to add vertex d is 7 and to add the vertex e is 11. We add the vertex d to our fragment with the cost 7. Finally, e is added with the cost 2. Hence, the edges are added to the tree in the order $4, 3, 7, 2$.

B. Boruvka's Algorithm

In Prim's algorithm, we started with a trivial fragment including just the vertex v_0. We kept increasing the size of the fragment till it became a spanning tree. In Boruvka's algorithm depicted in Algorithm 3, we may have more than one fragment. We increase the size of all fragments by adding the minimum outgoing edge for each fragment.

ALGORITHM 3: Boruvka: Finding the MSF

```
Input: Undirected connected Weighted Graph: $(V, E, w)$.
Output: Minimum Weight Spanning Tree
var
    $T$: { set of edges } initially {};
    $cid$: array[1..$n$] of 0..$n$ initially all 0;
    $mwe$: array[1..$n$] of edge initially all $null$;
    $dist$: array[1..$n$] of 0..$n$ initially all $\infty$;
while $(|T| < n - 1)$ do
    $visited$: array[1..$n$] of boolean initially all $false$;
    for $i := 1$ to $n$ do
        if $(\neg visited[i])$
            // do a BFS in the graph $(V, T)$ from
            // vertex $i$ setting $cid$ of every visited vertex to $i$
            $BFS(i)$;
    for $(i, j) \in E$ such that $(cid[i] \neq cid[j])$ do
        if $w[i, j] < dist[cid[i]]$
            $dist[cid[i]] = w[i, j]$
            $mwe[cid[i]] = (i, j)$
        if $w[i, j] < dist[cid[j]]$
            $dist[cid[j]] = w[i, j]$
            $mwe[cid[j]] = (i, j)$
    forall $i$ do:
        $T := T \cup mwe[cid[i]]$;
endwhile
return $T$
```

We use T to denote the set of tree edges. Initially, T is empty. When we determine the components in (V, T) using BFS, we get that there are n components as each vertex is a component by itself when T is empty. The algorithm finds minimum weight outgoing edge for each component as follows. At any iteration, we use BFS to find the least numbered vertex that any vertex is connected to in the graph (V, T). This vertex serves as the identifier for the component of the node i, and we use the variable $cid[i]$ to store it. Once we have determined the component identity of all nodes, we move to the next step of determining the minimum weight outgoing edge for each component. We traverse all edges and for each edge that connects two different components we check whether it is cheaper than previously known outgoing edge for the component on either side. Once we have determined all minimum weight edges for every component, we add these to T and start the next iteration.

For example, consider the graph in Fig. 1. Initially, T is empty and there are 5 components. We we compute mwe for each component, we get the edges $4, 3, 3, 2, 2$ as the minimum weight edges of a, b, c, d, e, respectively. Once, these edges are added we have two components: $\{a, b, c\}$ and $\{d, e\}$. We then find mwe of these two components as the edge 7. On adding this edge, we have chosen $(n - 1)$ edges and the algorithm terminates with the edges $\{2, 3, 4, 7\}$.

For complexity analysis, let n be the number of vertices and m be the number of edges. For simplicity, we analyze a version of Prim's algorithm in which instead of adjusting the key in the heap for a vertex, we simply insert the vertex in the heap. As a result the heap may have a vertex multiple times with different keys. When a vertex is removed, we check if it has already been fixed. If it is fixed, then we do not need to explore it and we can remove the next vertex in the heap. Since a vertex can be inserted in the heap only when an edge is explored, we get that there are at most m insertions from the heap. We can also conclude that there are at most m deletions from the heap. Since an insertion or a deletion from a heap takes $O(\log m) = O(\log n)$ time, we get the overall time complexity of $O(m \log n)$. *LLP-Prim*, introduced in the next section does not improve on this complexity.

V. LLP MST Algorithms

A. LLP-Prim

We now show an LLP based algorithm to find a minimum spanning tree rooted at a fixed vertex v_0. For simplicity, we assume that all edge weights are unique in the graph. If edge weights are not unique, then they can be made unique by incorporating identities of its endpoints.

When a minimum spanning tree is rooted at a fixed node, we can define an orientation of the edge directed towards the path to the root. We say that x is parent of y in the tree rooted at v_0, if (x, y) is an edge in the tree and x is closer than y to v_0. The problem of finding minimum spanning tree rooted at v_0 can be reformulated as finding the parent for every node other than v_0. We let these $n - 1$ nodes be responsible for choosing their parent edges. The variable S will be used as the *state* vector where $S[i]$ denotes the current choice of the edge for node i for $1 \leq i \leq n - 1$. Since all edge weights are unique, we simply use the weight of the edge as the identity of the edge. Initially, for any i, $S[i]$ is the least weight edge adjacent to node i. For example, all the choices for the graph in Fig. 1 are shown below. Since we will consider trees rooted at node a, we do not consider the node a.

```
b: 3 5 7
c: 3 4 9 11
d: 2 7 9
e: 2 11
```

Node b may choose as its parent edge, the edge 3, 5 or 7. Similarly, node e may choose 2 or 11. Initially, we have $G[b] = 3$, $G[c] = 3$, $G[d] = 2$ and $G[e] = 2$. In all there are $3 \times 4 \times 3 \times 2 = 72$ possible S vectors. These G vectors form a distributive lattice with $(3, 3, 2, 2)$ as the bottom vector and $(7, 11, 9, 11)$ as the top vector. If $G[j]$ equals the edge e, then we say that node j has proposed e as its parent edge. One of these vectors correspond to the minimum spanning tree and our goal is to find that vector efficiently.

The LLP algorithm for searching in this lattice is based on defining a predicate B such that the element in the lattice that satisfies the predicate is the desired element. The search for the element starts from the bottom element $(3, 3, 2, 2)$. If this element forms a spanning tree rooted at a, then we are done. Otherwise, we will find a node such that unless it advances to a heavier edge, the minimum spanning tree cannot be found. By repeating this procedure, we will reach the minimum spanning tree vector.

Given any G vector, we define a bipartition of the graph formed by directed edges in G as follows. There are $n-1$ edges in G and every node in $\{v_i \mid 1 \leq i \leq n-1\}$ has exactly one outgoing edge in G. We have that: *Any path starting from any node i and following the outgoing edge either ends at v_0 or ends in a cycle.*

We define a vertex to be *fixed* if by traversing the path starting from the edge proposed by that vertex leads to v_0. It is clear that the edge proposed by a non-fixed vertex can only lead to a non-fixed vertex and the edge proposed by a fixed vertex can only lead to a fixed vertex. Thus, any G partitions the set of vertices into *fixed* and *non-fixed* vertices. We define the set of cut edges between these two partitions as follows:

$$E'(G) := \{(i,k) \in E \mid fixed(i,G) \wedge \neg fixed(k,G)\}$$

Let (i, j) be the edge in E' with minimum $w[i, j]$ such that i is fixed and j is not fixed. We claim that process j must choose the edge (i, j) as its parent edge and all edges with lower weight than $w[i, j]$ that are adjacent to j cannot be parent edges of the minimum spanning tree.

Lemma 2: Consider any state vector S such that $w[i, j]$ is the minimum cut edge between $fixed(S)$ and $non-fixed(S)$, then the edge j to i is the parent edge in the minimum spanning tree.

Proof: Follows from the standard cut property of minimum spanning trees. ∎

Thus, any G such that $G[j] < w[i, j]$ cannot be part of the minimum spanning tree. When we advance $G[j]$ to that edge, v_j becomes fixed because by definition of E', the node i is fixed and now j's parent edge leads to a fixed node. Observe that additional nodes may become fixed if their proposed edge lead to v_j directly or indirectly.

We can then repeat this step until either all nodes become fixed and the algorithm can output G as the set of $n-1$ minimum spanning tree edges, or the edge set E' is empty. In the second case, the graph is not connected and there is no minimum spanning tree.

Algorithm 4 gives the boolean predicate required to search for the minimum spanning tree vector G. The algorithm has only one variable G, that denotes the element in the distributive lattice. We use the *always* section to define derived variables (or macros). A node is *fixed* if there is a path from that node to v_0. The edge set E' denotes all the edges that go from non-fixed vertices to fixed vertices. A node j is forbidden if it is the non-fixed vertex in the minimum weight edge in E'. The node j must advance to the minimum weight edge.

ALGORITHM 4: LLP Prim: Finding the MST .

```
var G: array[0..$n-1$] of real initially $\forall i : G[i]$ = minimum
edge adjacent to $i$;
always
    $fixed(j,G) \equiv$ there exists a directed path
                    from $j$ to 0 using edges in $G$
    $E'(G) := \{ (i,k) \in E \mid fixed(i,G) \wedge \neg fixed(k,G)\}$;
    forbidden(j)$\equiv \exists i : (i,j) \in E'$ such that it has min. weight
                                    $w[i,j]$ of all edges in $E'$
    advance(j) $G[j] := min\{w[i,j] \mid (i,j) \in E'\}$
```

We now give an efficient implementations of above LLP-Prim algorithm. Instead of recomputing $fixed$ after every iteration of LLP-Prim algorithm, we store whether a vertex has been $fixed$. Observe that once a vertex is fixed it stays fixed because its parent edge never changes. The set of cut edges in E' are maintained as a heap so that it is easy to determine the minimum of the set. The crucial difference from Prim's algorithm is in the way a non-fixed node is fixed. In Prim's algorithm, we only mark an element v as $fixed$ only when we remove it from the min-heap. In each iteration, we choose the vertex v with the least value of d from all non-fixed vertices. We then explore all the neighbors of v. The only difference is the way d is updated. Whenever a vertex is removed from the heap, all its outgoing edges can be explored in parallel. Prim's algorithm suffers from the sequential bottleneck (similar to Dijkstra's algorithm for the shortest path): in every iteration of the *while* loop, exactly one vertex becomes fixed. In LLP-Prim algorithm not only that vertex v is fixed but also all the vertices that can reach v directly or indirectly using their current proposed edges are also fixed.

Note that Prim's Algorithm grows the fragment of MST one vertex at a time. We call all the vertices that are part of the fragment as *fixed*. All of these vertices know their final parent in the MST rooted at v_0. Prim's algorithm determines the next vertex to be fixed as the vertex that is nearest to fixed vertices. We now claim that any non-fixed vertex such that it is connected to one of the fixed vertices with a minimum weight edge (MWE) can also be fixed. Observe that initially all nodes such that their minimum weight edge points to v_0 are fixed. In every iteration, a node k can become fixed in either of the two ways

1) It is the the nearest neighbor to the fragment. This is the usual way in Prim's algorithm.
2) It is connected to a fixed node z via a minimum weight edge. This edge could be the minimum weight edge for z or for k.

Note that when a node becomes fixed, it may result in additional nodes becoming fixed due to the second way of becoming fixed. We continue to add nodes to the fixed set until we are not able to add any more nodes by this method.

ALGORITHM 5: LLP-Prim: Early Fixing Algorithm $MST1$

```
Input: Undirected Weighted Graph: $(V, E, w)$.
Output: Minimum Weight Spanning Tree
struct Node: {
min_edge: e $\in$ E, set beforehand to the minimum Edge
  connected to Node
fixed: boolean,
parent: v $\in$ V, initially not set
dist: integer, initially $\infty$}
var $H$: binary heap of $Node$ initially empty;// Node.dist is the
tentative cost
$R$: bag of $Node$ initially empty

$source.dist := 0$;
$R.insert(source)$;
while !$R.empty()$ in parallel do
    $j = R.pop()$
    $j.fixed = true$
    forall $k$: $\neg k.fixed \wedge (j,k) \in E$
        if ($k.min\_edge = w[j,k]$) then:
            $k.dist := w[j,j]$
            $k.fixed := true$
            $k.parent := j$
            $R.push(k)$
        else if $k.dist > w[j,k]$ then:
            $k.dist := w[j,k]$
            $k.parent := j$
            $H.push(k)$
    if $R.empty()$ && !$H.empty()$ then:
        $R.push(H.pop())$
endwhile;
```

The algorithm starts with the insertion of the source vertex with its d value as 0 in the R set. Instead of removing the minimum vertex from the heap in each iteration and then exploring it, the algorithm always extracts vertexes from the R set first, and only when it's empty does it go to the heap to pop the minimum element. If the value popped from R is then marked as fixed and 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. Some vertices may initially be non-fixed but R may become fixed later when R is processed. To avoid the expense of inserting these vertices in the heap, we collect all such vertices which may need to be inserted or adjusted in the heap in a separate set called Q. Only, when we are done processing R, we call $H.insertOrAdjust$ on vertices in Q.

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 *processEdge1* in Algorithm 5. First, we check if this edge is in the set MWE. If this is the case, then we know that this edge can be added to the tree and the node k can be marked as fixed. Setting $fixed[k]$ to true removes it effectively from the heap because whenever a fixed vertex is extracted in the outer while loop it is skipped. The vertex k can be added to R for future processing. If (z, k) is not in MWE, then we check if the existing distance $d[k]$ is bigger than the weight of the edge (z, k). If this is the case, we update $d[k]$ and make z as the parent of k. Finally, if $d[k]$ has decreased, we insert it in Q so that once R becomes empty we can call H.insertOrAdjust() method on vertices in Q. This algorithm can be terminated as soon as $n-1$ edges have been chosen.

Let us run this algorithm on the graph in Fig. 1. We first insert the vertex a in the heap with distance 0. We remove a from the heap, fix it and insert it into R. We then process all edges of vertices in R. When we process the edge (a, c), we find that it is the mwe for a. Hence, c is fixed and added to R. We continue processing edges of a. The next vertex discovered is b. It is added to Q. We then continue processing R. The next vertex is c. When we process the edge (c, b), we find that it is mwe for b (and c). Hence, b is now added to R. We now remove the vertex b from R and process its edges. Since vertices a and c are already fixed, we discover d and insert it in Q. Now, R is empty and we start inserting vertices from Q into the heap. Since c is already fixed, we insert only the vertex d in the heap. We are now ready to remove the minimum from the heap. The vertex d is removed and fixed. When we explore the edges of d, we find that the edge (d, c) is mwe and the vertex c is also fixed. At this point all vertices have been fixed and we get the edges $\{4, 3, 7, 2\}$ as the MST.

Note that this algorithm requires every vertex to know its minimum weight edge. If this information is not available, then every node can determine this information in parallel by processing all its adjacent edges. In a sequential algorithm, the set MWE can be computed when the graph is input.

B. LLP-Boruvka

VI. LLP-Boruvka Algorithm

In this section, we continue with the recursive version of the algorithm, but implement each recursive instance of Boruvka's algorithm with the LLP algorithm. As before, we assume that input to our algorithm is a connected graph without any self-loops. The LLP algorithm uses the single variable G. For each instance, we let the minimum weight edge (mwe) adjacent to each vertex v be denoted by $mwe[v]$. We denote this directed graph by H: the set of vertices is V and the set of edges is $\{(v, w) | (v, w) = mwe[v]\}$. We initialize $G[v]$ with the node w when $mwe[v] = (v, w)$ except when $mwe[w] = (w, v)$ and $v < w$. For the latter case, we make $G[v]$ equal to v. As a result of this initialization, the structure $G[v]$ forms a rooted tree. This initialization is simply the steps of getting the rooted tree in Algorithm 6. We say that the edge (v, w) is *incident* to v when $G[v]$ equals w for w different from v.

Once we have rooted trees for the graph, we convert the rooted trees to rooted stars using pointer-jumping. A node j is considered forbidden if $G[j] \neq G[G[j]]$. It is advanced by setting $G[j]$ to $G[G[j]]$. The algorithm stops this iteration when no node is forbidden. We show that each instance is indeed an LLP algorithm.

ALGORITHM 6: LLP-Boruvka: Finding MSF

```
Input: Undirected Weighted Graph: $(V, E, w)$.
Output: Minimum Spanning Forest
function $Boruvka(V, E)$
    var $G$: array[1..$|V|$], $T = \{\}$
    forall $v$ in $V$ in parallel do:
        // Choose the minimum weight edge from $v$
        $(v, w) = mwe(v)$

        // Choose $v$'s parent, break symmetry with $w$
        $G[v]$ := if $mwe(w) \neq (v, w) \Rightarrow w$
            else if $mwe(v) = mwe(w)$ and $v < w \Rightarrow v$
            else $\Rightarrow w$

        // Add to the MSF
        $T := T \cup (v, w)$

    // Evaluate in parallel
    forbidden(j)$\equiv G[j] \neq G[[G[j]]$
    advance(j) $\equiv G[j] := G[G[j]]$

    if $|E| = 0$ return $\{\}$
    else
        $E' := \{(G[v], G[w]) | ((v, w) \in E) \wedge (G[v] \neq G[w])\}$
        $V' := \{v \in V \mid G[v] = v\}$
        return $T \cup Boruvka(V', E')$
```

We first show a basic lemma:

Lemma 3: The following is an invariant of the program: $G[v]$ is reachable from v in the directed graph H.

Proof: The invariant is true initially because $G[v]$ is set to w where (v, w) is an edge in H. The only statement executed in the program is $G[v] := G[G[v]]$ which preserves the invariant because $G[G[v]]$ is reachable from $G[v]$. ■

We also observe that H is a collection of rooted trees and if w is reachable from v, then there is a unique path from v to w. Furthermore, the weight of edges along any path is strictly decreasing. This is because if v points to w and w points to u then the weight of the edge (w, u) must be strictly smaller than the the weight of the edge (v, w) by the property that each vertex points to the minimum weight edge incident to it. We are now ready to show the following lemma:

Lemma 4: Each instance of finding connected components is an LLP algorithm that terminates.

Proof: We consider the lattice of vectors. For component j, we keep the weight of the minimum weight edge in the path from j to $G[j]$. We define the predicate B as

$$B \equiv \forall j : G[j] = G[G[j]]$$

We show that predicate is lattice-linear. Suppose the predicate B is not true. This implies that there exists j such that $G[j]$ is different from $G[G[j]]$. We show that the component j is forbidden. $G[j]$ cannot be the root of a tree in H because the root points to itself (and therefore $G[j]$ equals $G[G[j]]$. Let k be equal to $G[j]$. We just showed that k is not equal to the root. Therefore, $G[k] \neq k$ even if $G[k]$ changes. Hence, $G[j] \neq G[G[j]]$ continues to hold. The component j in G vector is advanced by setting $G[j]$ to be $G[G[j]]$.

Dataset	Original Name	Name used	Type
Galois	USA-road-d.USA	USA Roads - 23M	road
Graph500	graph500-s25-ef16	Graph500 18M	scalefree

TABLE I: Graphs used in experimental evaluation

We now show that the LLP algorithm terminates. When $G[j]$ is advanced, the weight of the last edge from j to $G[j]$ can only strictly decrease. Since there are at most $n-1$ edges in H, the algorithm must eventually terminate with B as true. ■

Let us run this algorithm through the graph Fig. 1. We compare vertices lexicographically and we use $(vertex, parent, chosen\_edge)$ here instead of G for simplicity of exposition. In the first iteration, we start by choosing, in parallel, the minimum weight edge and the parent for each of the vertices. We obtain the following set $\{(a, c, 4), (b, b, 3), (c, b, 3), (d, d, 2), (e, d, 2)\}$, $T = 4, 3, 2$. We then proceed to evaluate $forbidden$ and $advance$ in parallel and without synchronization, which in this very simple case just results in changing the parent of a to b, resulting in $\{(a, b, 4), (b, b, 3), (c, b, 3), (d, d, 2), (e, d, 2)\}$. In preparation for the second iteration we choose vertices such that $vertex = parent$, so vertices $\{b, d\}$ and edges such that the parents of either end differ, so in this case edges $\{7, 9, 11\}$. In the second iteration, again starting with choosing the minimum weight edges and the parent, resulting in $\{(b, b, 7), (d, b, 7)\}$), $T = 7$. The next phase has nothing to do in this second iteration, since $G[j] = G[G[j]]$ for all vertices, so the algorithm returns the union of all T's, the minimum spanning tree $T = \{4, 3, 2, 7\}$. Note that, within a round, Algorithm 6 requires little to no synchronization between vertices of the MSF.

VII. Experimental Evaluation

We use the Galois Library [8] as our underlying runtime framework for *Prim* and *LLP-Prim* and the Graph Based Benchmark Suite [2] (GBBS) for *Boruvka* and *LLP-Boruvka*. Using existing frameworks allows us to explore our own algorithms experimentally while relying on thoroughly tested parallel and benchmark constructs.

A. Experimental Setup

The experimental system is a single custom configured virtual machine of type C2 allocated through Google Compute Engine [3]. The virtual machine boosts 48 vCPUs and 96GB's of memory. Our experiments were limited to a maximum of 32 threads constrained to a single socket, to exclude possible NUMA effects. We use the Release build of our own fork of the Galois library for our experiment (available on github) as well our fork of the Graph Based Benchmark Suite [2], which contains competitive implementations of Prim and a fast parallel implementation of Boruvka.

We use two types of graphs for our benchmarks: graphs from graph500 [10] benchmarks are synthetic Kronecker [9] graphs that are generated to be used in the benchmarks and are thus a good reference to run our algorithms against. To make sure that our results also apply to real world graphs, we also test the algorithms in the USA road network, which is 24M node graph with real weights. Table I summarizes the graphs use in the experiments, including their vertex counts and types.

B. All Single vs Multi Threaded

In this section we take a look at the single-threaded versus the multi-threaded performance and give some intuition about the behavior observed.

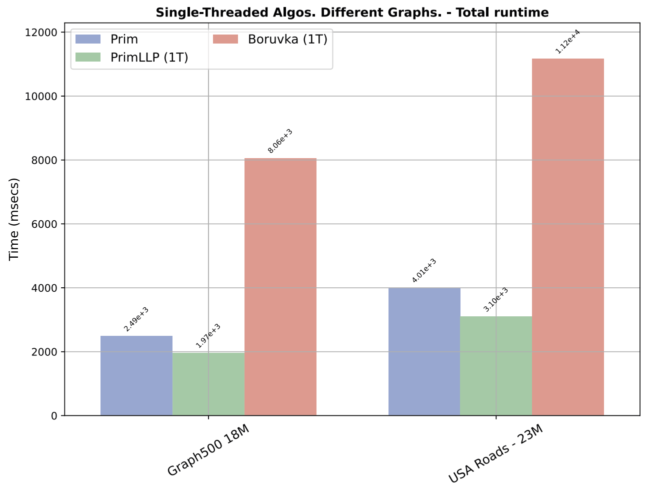

Fig. 2: Prim, LLP-Prim, Boruvka, single-threaded, Road network graph and Graph500 18M

Fig. 2 compares *Prim*, *LLP-Prim (1 Thread)*, and *Boruvka (1 Thread)* in the *USA Roads* graph and in the *Graph500 18M* graphs. These graphs were chosen to illustrate different morphologies, but similar sizes. As we can see, both *Prim* and *LLP-Prim (1T)* are much faster that the the Boruvka version, in a single thread scenario, by a factor of about 3x, with *LLP-Prim (1T)* showing an advantage of about 21% and 27% vs *Prim* in in the graph 500 and road graphs, respectively. This is expected, since Prim is a greedy algorithm, it fares better in a single threaded scenario and since *LLP-Prim (1T)* does less work than *Prim*, it's faster in absolute terms. Note that we didn't include *LLP-Boruvka (1 Thread)*, in this comparison as we'll look at how it compares vs *Boruvka* in the next section.

Fig. 3 on the other hand shows quite a different scenario. When increasing the number of threads, *Boruvka* based algorithms start dominating the benchmark. In this case around 8 threads both *Boruvka* and *LLP-Boruvka* become faster and, with almost linear speedup. While *LLP-Prim* does show some speedup it is not linear and even starts to regress at about 8 threads. The reason behind this is that the algorithm has exhausted the parallelism opportunities (i.e. processing vertices in parallel), and thus adding more threads brings diminishing returns.

Comparing *LLP-Boruvka* and *Boruvka* in Fig 3 we can observe that *LLP-Boruvka* is faster than parallel *Boruvka* up to

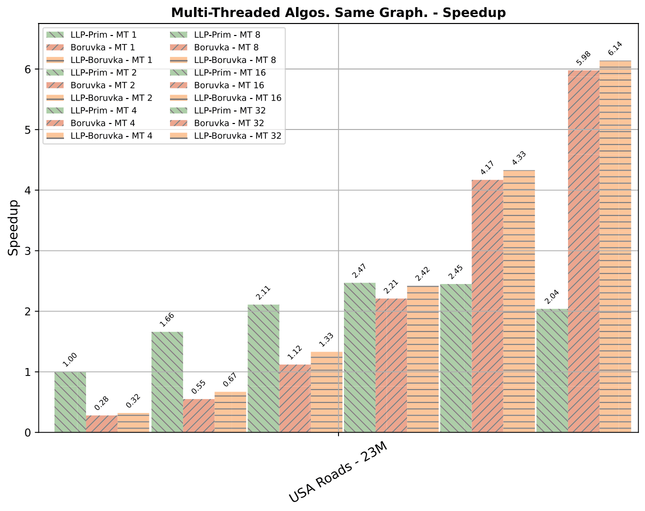

Fig. 3: LLP-Prim, Boruvka, LLP-Boruvka multi-threaded, USA Road network graph

32 threads, though the difference starts to taper off at higher core counts. When the number of threads available is small (or even one) *LLP-Prim (1T)* is the fastest. This seems to indicate that in scenarios of low thread count where only one or few threads are available, *LLP-Prim (1T)* is a better choice that both *Boruvka* based algorithms and *Prim* as it allows for some speedup it there are a few threads available while still performing very well in single thread scenarios.

C. Parallel Algorithms on Different Graphs

In this section we take a closer look at the parallel algorithms and their behavior on different graphs.

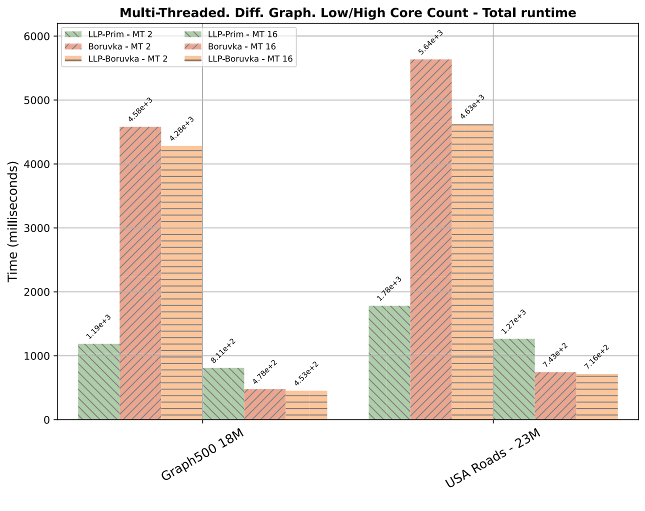

Fig. 4: LLP-Prim, Boruvka, LLP-Boruvka. Low/High core count. Different graphs

Fig. 4 compares all the parallel algorithms *Boruvka*, *LLP-Boruvka* and *LLP-Prim* at low and high core counts for different types of graphs. Once more we can see that *LLP-Prim* is faster at low core counts, whereas the Boruvka based algorithms fare better at higher core counts. More importantly, we can observe that that *LLP-Prim* performs best in graphs with more edges. The USA Road Network graph has, on average, a lower number of edges per vertex than the graph 500 benchmark graphs. This means that the graph 500 graphs present more opportunities for parallelism for *LLP-Prim*. On other other hand, at higher core counts Boruvka based algorithms have the most advantage with *LLP-Boruvka* being faster than *Boruvka*, though not by much.

We also tested the algorithms in graphs of different sizes and the same morphology. But we only had immediate access smaller graphs than the ones used in the experiments above the results were analogous and didn't show any additional insight.

VIII. Conclusions and Future Work

In this paper we demonstrated how the minimum spanning tree problem can be modeled in terms of Lattice-Linear Predicates and we used this theoretical framework to extract more parallelism from Prim's and Boruvka's classical algorithm for the minimum spanning tree.We introduce two novel parallel algorithms *LLP-Prim* and *LLP-Boruvka*. *LLP-Prim* is an improvement over Prim's algorithm and is suitable for low core count scenarios, whereas *LLP-Boruvka* is an improvement over the version of Boruvka's algorithm and is more suited for high core count scenarios. In future work we will explore and evaluate LLP-based formulations for other combinatorial optimization problems.

References

[1] C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[2] L. Dhulipala, J. Shi, T. Tseng, G. E. Blelloch, and J. Shun. The Graph Based Benchmark Suite (GBBS). *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 1–8, 2020.

[3] Google. About machine families — google compute engine documentation. https://cloud.google.com/compute/docs/machine-types.

[4] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM (JACM)*, 42(2):321–328, 1995.

[5] D. Karker. Random sampling in matroids, with applications to graph connectivity and minimum spanning trees. *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, pages 84–93, 1993.

[6] P. Klein, R. Cole, and R. Tarjan. A Linear-Work Parallel Algorithm for Finding Minimum Spanning Trees. *Proceedings of the 6th Symposium on Parallel Algorithms and Architectures*, pages 11–15, 1994.

[7] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[8] M. Kulkarni and M. Burtscher. Lonestar: A Suite of Parallel Irregular Programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[9] J. Leskovec and C. Faloutsos. Scalable modeling of real graphs using Kronecker multiplication. *Proceedings of the 24th international conference on Machine learning - ICML '07*, pages 497–504, 2007.

[10] R. C. Murphy, K. B. Wheeler, and B. B. C. U. Group. Introducing the graph 500. *richardmurphy.net*, 2010.

[11] J. Nešetřil, E. Milková, and H. Nešetřilová. Otakar Borůvka on minimum spanning tree problem Translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1-3):3–36, 2001.

[12] S. Pettie and V. Ramachandran. A Randomized Time-Work Optimal Parallel Algorithm for Finding a Minimum Spanning Forest. *SIAM Journal on Computing*, 31(6):1879–1895, 2002.

[13] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. *ACM SIGPLAN Notices*, 46(6):12–25, 2011.

[14] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[15] C. Scheideler, M. Spear, and V. K. Garg. Predicate Detection to Solve Combinatorial Optimization Problems. *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–245, 2020.

[16] W. Zhou. *A Practical Scalable Shared-Memory Parallel Algorithm for Computing Minimum Spanning Trees*. PhD thesis, Carnegie Mellon University, 2017.