Abstract
The Minimum Spanning Forest (MSF) problem finds usage in many different applications. While theoretical analysis shows that linear-time solutions exist, in practice, parallel MSF algorithms remain computationally demanding due to the continuously increasing size of data sets.

In this paper, we study the MSF algorithm from the perspective of graph structure and investigate the implications of the power-law degree distribution of real-world graphs on this algorithm.

We introduce the MASTIFF algorithm as a structure-aware MSF algorithm that optimizes work efficiency by (1) dynamically tracking the largest forest component of each graph component and exempting them from processing, and (2) by avoiding topology-related operations such as relabeling and merging neighbour lists.

The evaluations on 2 different processor architectures with up to 128 cores and on graphs of up to 124 billion edges, shows that Mastiff is 3.4–5.9× faster than previous works.

1 Introduction
Finding the Minimum Spanning Forest (MSF) is one of the basic graph algorithms with several usages in different fields of technology, science, and humanities [11, 22, 24, 41, 53, 58, 61]. Among different MSF algorithms, Borůvka’s algorithm [11] provides good opportunities for parallel execution. The algorithm is organized around a number of iterations. In each iteration, the lightest edge of each vertex is selected as an edge of MSF. Then, the graph is contracted over the selected edges, i.e., vertices in each component (formed by the selected edges) are merged and a new graph is created by relabeling the vertices and merging the neighbour lists. This new graph is used in the next iteration.

Contraction of the graph eliminates intra-component edges and makes the next iteration efficient. However, our evaluation shows that topology rewriting requires more than 50% of execution time (Section 3.1). The alternative is to not contract the graph. This approach has been shown to have the same asymptotic time complexity as rewriting the graph topology [22] as it is required to process all edges of the graph in each iteration. Thus, the practical choice is to spend time either contracting the graph, or to spend time traversing all edges.

On the other hand, the fast-growing size of real-world graphs and their special structure necessitate fast and efficient MSF algorithms. Many real-world graphs derived from social networks, the internet and the world-wide web, or from bio-informatics, show a skewed degree distribution, following a power-law distribution: a small fraction of vertices with very large degrees (known as hubs) are connected to a disproportionately large fraction of edges.

We start with studying the effects of power-law degree distribution of graphs on formation of components in consecutive iterations of Borůvka’s algorithm. Our study shows that as a result of the small-world property in power-law graphs, a great percentage of vertices tend to quickly connect to each other, resulting in a large and fast-growing component.
Moreover, in each iteration of Borůvka’s algorithm a component finds at most one edge to another component; therefore, it is more efficient to skip processing the fast-growing component (whose great number of edges requires a great amount of processing) and to allow other components (with much smaller number of edges to be processed) to attach themselves to this component.

Based on this, we introduce the MASTIFF algorithm that dynamically tracks the largest component in each graph component and exempts them from processing. Mastiff uses a disjoint-set mechanism [15, 51, 56] to efficiently manage the relationship between vertices without performing time-and memory-consuming topology operations. As a result, Mastiff reduces the additional memory space requirement from $O(|E|)$ to $O(|V|)$.

Experimental evaluation on 2 processor architectures with up to 128 cores and for graphs up to 124 billion edges shows that Mastiff is 3.4–5.9x faster than previous works.

This paper is structured as follows: Section 2 explains background material and Section 3 demonstrates the key observations that motivate the design of Mastiff. Section 4 introduces the Mastiff algorithm which is evaluated in Section 5. Section 6 discusses further related work and avenues for future work are presented in Section 7.

2 Background

2.1 Terminology

An undirected weighted graph $G = (V, E, W)$ has a set of vertices $V$, a set of edges $E$, and as set of weights $W$. Edge $(u, v, w)$ is an edge between vertices $u$ and $v$ and the weight of this edge is $w$. $G$ may have a number of connected components that are called graph components.

The Minimum Spanning Tree (MST) of a weighted, undirected, and connected graph is a tree over all vertices and with the minimum sum of the weights of the edges. If the graph has more than one connected component, the Minimum Spanning Forest is defined as the set of MSTs of all graph components.

In this paper, the term component is referred to a component of the MSF during its construction. A component contains a number of vertices that are connected by a subset of edges of the MSF.

Initially, each vertex of the graph is a component (with no edge between vertices) and then components of a graph component are gradually connected together by edges that shape the MST of that graph component.

As an example, Figure 1a shows a graph that has only one graph component. Figure 1b shows an intermediate level in constructing MST (which is also the MSF as the graph has only one component). Here, 3 components with names A, B, and C are seen. Each of these components include one or more vertices. Components A and C have two vertices, and component B has 6 vertices.

The cut property states that for each cut of the vertices, the lightest edge between two partitions is in the MSF. The cycle property states that the heaviest edge of each cycle of a graph is not in its MSF. For a graph with distinct weights of edges, only one MSF exists [22].

2.2 Borůvka’s Algorithm

Borůvka’s algorithm is performed in a number of iterations. Each iteration has 3 steps: (1) finding the lightest edge of each vertex (that specifies an edge in the MSF), (2) identifying the components connected by the lightest edges, and (3) merging vertices in the same component and making the new graph ready for the next iteration.

Figure 1 shows an example of the execution of Borůvka’s algorithm. In the first iteration, the edge between vertices 0 and 1 is selected as their lightest edge. Vertex 3 selects its edge with weight 4 to vertex 2 as the lightest edge. In the same way, other vertices select their lightest edges as edges of the MST.

Figure 1b shows how the 3 components are formed after the first iteration and each component contains a number of vertices of the main graph. The intra-component edges are removed and the new graph has 3 vertices (A, B, and C).

The edges of the new graph are the lightest inter-component edges. Vertices in component A have 3 edges to the vertices in component B with weights 2 and 3. The lightest edge is selected as the edge between components A and B. In the
same way, the edge with weight 3 is selected as the edge between components B and C.

The next iteration continues selecting the lightest edges of the new graph and merging more vertices until only one vertex remains for each component of the main graph (Figure 1c).

3 Motivation

3.1 Cost of Topology Rewriting

Figure 2 shows the practical cost of rewriting the graph topology to effect the contraction of connected vertices. The graph data sets are presented in Table 2, page 7. These measurements show the high cost of topology rewriting: between 56% and 81% of the execution time is spent in contracting the graph.

During each iteration of Borůvka’s algorithm, up to one edge is selected per component and the number of components reduces by up to half after each iteration; as a result, fewer edges are selected in the next iteration. On the other hand, if we avoid rewriting the graph topology, the required time for identifying the lightest edges remains constant; therefore, the time spent per lightest edge increases starkly as iterations progress.

This shows that a fundamental solution is necessary to make it beneficial to avoid rewriting the graph topology.

3.2 Formation of a Giant Component

Power-law graphs have a high degree of connectivity between the hubs. This strong interlinking of hubs aligns with the presence of a giant component in the graph. The hub vertices connect to a large portion of edges of the graph. Consequently, hubs share edges with most low-degree vertices. This pattern repeats when constructing the MSF. Hubs are likely to be incident upon the lowest-weight edges of many low-degree vertices, based on the statistical frequency of the number of their edges. Moreover, in some graphs the weights of edges that connect low-degree vertices to high-degree vertices are particularly smaller than the weights of edges between low-degree vertices. Consequently, the MSF grows very quickly around the hubs, resulting in the creation of a component containing a large percentage of the vertices.

Figure 3 confirms the creation of a giant component in the MSF. It shows the size of the largest component of each dataset, relative to the total graph size, after each iteration of Borůvka’s algorithm. It takes 3-6 iterations for the giant component to form.

Figure 4 shows the degree distribution of the graphs and their MSF. We see that MSF of the power-law graphs has a power-law degree distribution. The presence of hub vertices in the MSF shows that most of the hub edges of the main graph have been selected by the MSF. It explains that hubs (and components containing hubs) have more chance to be selected by other vertices as the destination of lightest edges as hubs have a large percentage of edges, i.e., a large percentage of the lightest edges.

4 MASTIFF

4.1 Mastiff Idea

Assume we have a graph component containing \( c \) components in an iteration of Borůvka’s algorithm. In this case, at most \( c - 1 \) lightest edges can be selected. Each of the components \([1, c - 1]\) can select an edge to a non-selected component. But the last component should select an edge...
to one of the previously selected components (as there is no other component). Moreover, since each two vertices can be connected by at most one edge, this edge is the lightest edge of both endpoints and the last component selects a repeated lightest edge.

In this way, at least two components between $c$ components select the same lightest edge; therefore, it is more efficient to process only $c - 1$ components.

As an example, in Figure 1b we have components A, B, and C that together will finally shape a MST. In this iteration, we can exempt each of the components from processing:

1. If we exempt component A, then component B selects the edge of weight 2 and component C selects the edge of weight 3.
2. If we exempt component B, then component A selects the edge of weight 2 and component C selects the edge of weight 3.
3. If we exempt component C, then both components A and B select their lightest edge which is the edge of weight 2 and the edge of weight 3 is selected in the next iteration.

As a very large component is formed in power-law graphs (Section 3.2), Mastiff selects the largest component as the best candidate to be exempted from processing.

As an example, in Figure 1b component B (that contains vertices with grey background) is the largest component and is exempted form processing in the second iteration. Components A and C find their lightest edge and the MST becomes completed. In the following lines, we show that the MSF result is not changed as a result of modifications applied by Mastiff.

**Theorem.** Exempting one component from a graph component does not change the selection of the lightest edges in MSF.

**Proof.** Assume edge $e = (u, v, w)$ is an edge between vertices $u$ and $v$ that finally appears in MST containing $u$ and $v$. In an iteration, the component containing $u$ is exempted from processing and $e$ is the lightest edge of this component, then two conditions can be supposed:

(i) $e$ is also the lightest edge of the component containing $v$, and therefore $e$ is selected in the current iteration, and

(ii) $e$ is not the lightest edge of the component containing $v$, and therefore $e$ is not selected until a future iteration where (a) $u$ is not exempted from processing and therefore $e$ is selected (as $e$ is the lightest edge of this component), or (b) the component containing $v$ does not have any lighter edges to components other than component containing $v$. In this case, no edges lighter than $e$ between components containing $u$ and $v$ can exist because such an edge results in the contradiction that $e$, as the lightest edge between $u$ and $v$, has been on the MST.

**Figure 5.** Vertex statuses and their transitions in Mastiff

### 4.2 High-Level Algorithm

In each iteration of the Mastiff algorithm, the largest component of each graph component is selected as EXEMPT and all other components select their lightest edges.

After selecting the lightest edges, it is required to create a new graph; however, as Mastiff does not process the largest components, we avoid time-consuming operations for creating the new graph (Section 3.1). In spite of that, we need to track (1) the status of each vertex, and (2) the relationship between vertices.

To track the condition of each vertex, Mastiff assigns Status to each vertex. Figure 5 shows different vertex statuses and their transitions:

1. Each vertex is initially in the ROOT status that specifies the vertex should be processed to identify its lightest edge.
2. When a vertex selects a lightest edge to another vertex, the edge is added to MSF and the status of the vertex is changed to MERGED.
3. If a ROOT vertex is identified by the Mastiff algorithm as an EXEMPT vertex, this vertex and all vertices that are MERGED into this vertex are exempted from processing.

As Mastiff dynamically selects the largest components as exempted, it is possible for the status of a vertex to return to ROOT from EXEMPT.

Mastiff uses a disjoint-set mechanism to track the relationship between vertices. A Parent array is used to specify the component of each vertex. Initially, each vertex is a root and upon selecting a lightest edge, the Parent of the vertex is set to the other endpoint of the selected edge.

For vertices with ROOT or EXEMPT statuses, Parent of the vertex is the same as the vertex. But, Parent of a vertex with MERGED status cannot be the same as the vertex.

After selecting the lightest edge in each iteration, it is enough to change the status of MERGED vertices and to update the Parent of vertices. Then, in selecting the lightest edge in the next iteration, the intra-component edges are filtered by using the Parent array.

### 4.3 Mastiff Algorithm

Algorithm 1 shows the Mastiff algorithm. In addition to Status and Parent arrays explained in Section 4.2, Mastiff requires two other arrays (Lines 1-7):
Algorithm 1: Mastiff

Input: G(V, E)
Output: MSF

/* Declaring variables */
1 Vertex_Statuses = {ROOT, MERGED, EXEMPT};
2 Array MSF ; // Output
3 Array Status ; // Status of vertices
4 Array Parent ; // Parent of vertices
5 Array LE ; // Lightest Edges
6 Array CS ; // Component Size
7 RV_count ; // Root Vertices Count

/* Set max-degree vertex as graph component ID */
8 GC = connected_components(G) ; // Graph Components
9 Array MDV = {} ; // Max Degree Vertex
10 /* Finding the max-degree vertex of each graph component */
11 par_for v ∈ V
12    atomic_arg_max( MDV_{GC_v}, v, degree);
13    /* Updating graph component of vertices */
14    par_for v ∈ V
15        GC_v = MDV_{GC_v};

/* Initializing variables */
16 par_for v ∈ V
17    /* Each vertex is initially a component */
18        Parent_v = v;
19        CS_v = 1;
20    if GC_v == v then
21        /* v is the max-degree vertex */
22            Status_v = EXEMPT;
23    else
24        Status_v = ROOT
25 RV_count = |G.V| - |GC|;

/* Iterations */
26 while RV_count > 0 do
27       /* (Step-1) Find the lightest edges of components */
28         par_for {v ∈ V | StatusParent_v = ROOT}
29             le = ∅;
30             /* An intra-component edge */
31                 for e ∈ v.edges do
32                     if Parent_v == Parent_{e.dest} then continue;
33                         /* A heavier edge */
34                             if le.weight < e.weight then continue;
35                                 le = e;
36                                 atomic_min(LE_{Parent_v}, le);
37         /* (Step-2) Skip the repeated edges */
38         par_for {v ∈ V | Status_v = ROOT}
39             dest = Parent_{LE_v.dest} ; // destination component
40                 if v == LE_v.dest ∧ v > dest then
41                     /* (Step-3) Append vertices */
42                         LE_v = ∅;
43                 /* (Step-4) Update variables */
44                 par_for {v ∈ V}
45                     compress_path(Parent_v, v);
46                     if Parent_v ≠ v ∧ Status_v = ROOT then
47                         CS_{Parent_v} += CS_v ; // atomic
48                         Status_v = MERGED;
49                         RV_count == ∅ ; // race-free
50                 /* (Step-5) Set the greatest components as EXEMPT */
51                     Max_CS = ∅;
52                     par_for {v ∈ V | Status_v ≠ MERGED}
53                         Status_v = ROOT;
54                     atomic_arg_max( Max_CS_{GC_v}, v, CS);
55                     par_for {v ∈ V | is_set(Max_CS_v)}
56                         StatusMax_CS_v = EXEMPT;
57 return MSF;

- The LE array specifies the lightest edge of ROOT vertices.
- The CS array specifies the size of components, i.e., the number of vertices that have been merged into ROOT vertices. Initially, CS_v is set to 1 for each vertex v. At the end of each iteration the CS values of vertices are updated. We use CS to identify the largest component of each graph component.

As we explained in Section 4.1, Mastiff needs to identify the largest component of each graph component. So it starts with identifying the connected components of the main graph (Line 8). In the first iteration, Mastiff selects the vertices with maximum degree in their graph components as EXEMPT vertices.

To that end, it is required (1) to loop over all vertices in each component to identify the ID of the vertex with maximum degree (Lines 10-11), and then (2) to assign the ID of the vertex with maximum degree of each component as graph component ID of all vertices in that component (Lines 12-13).

atomic_arg_max(x, y, prop) (used in Lines 11 and 47) atomically compares the prop property of x and y and assigns y to x if y has a greater value of prop.

Then, the variables are initialized in Lines 14-21. The Parent of vertex v is set to v and CS of each vertex is set
to 1. The Status of all vertices is set to ROOT, except for the vertex of the maximum degree in each component whose status is set to EXEMPT. The RV\_count specifies the number of ROOT vertices (Line 21) and is used to identify the end of iterations (Line 22).

Each iteration of Mastiff (Lines 23–49) has 5 steps:

1. **Finding the lightest edge of ROOT vertices:** In this step (Lines 23–29), each ROOT vertex and vertices that are MERGED into ROOT vertices traverse their neighbour-lists and skip the intra-component edges (Line 26), and identify the lightest inter-component edges (Lines 27–28). This lightest edge is set as the lightest edge of the component (i.e., Parent of the vertex), if it is lighter than the current one (Line 29).

2. **Removing the symmetric edges:** In Lines 30–33, the lightest edges of the ROOT vertices are considered to check if an edge has been selected twice as the lightest edge between two components. If so, the lightest edge of one endpoint is discarded. This avoids adding one edge twice to the forest and also avoids making loops in the Parent array.

3. **Adding the selected edges to the forest:** In this step (Lines 34–37), the lightest edges of the ROOT vertices are added to the forest and the Parent of recently merged vertices is updated. As the Parent array is changed in this step, steps 2 and 3 cannot be fused.

4. **Updating the variables:** This step (Lines 38–43), starts with updating the Parent of each vertex.

Since a number of ROOT vertices may merge to each other in an iteration, it is necessary to identify a representative component as the Parent value for all of the MERGED vertices in a component to be able to filter intra-component edges in the first step of the next iteration. In Mastiff, the component of a vertex (i.e., the representative for all vertices in the same component) is its first non-MERGED Parent, that is identified by the compress_path() function in Lines 39. Then, this function performs full-path compression by updating the Parent of all intermediate MERGED vertices.

There are different ways to select the representative of a component in disjoint-set algorithms [1, 2, 5, 15, 51, 56]. Since the depth of trees is less than 20 (Section 5.6) and to avoid imposing overheads, we use the default vertex specified in the previous step (with Parent\_id = 0) as the representative.

After updating the Parent of the vertex (Line 39), if the vertex that has been merged into another vertex in the current iteration (Line 40): (i) the CS of the Parent vertex is updated in Line 41 (to be able to identify the largest components in the next step), (ii) the status of the vertex is changed to MERGED in Line 42, and (iii) the number of ROOT vertices (RV\_count) is reduced by one in Line 43.

5. **Updating the EXEMPT vertices:** This step is performed in Lines 44–49. First, a pass over all ROOT and EXEMPT vertices is made that identifies the ID of the vertex with maximum CS of each graph component (Lines 45–47). These vertices are representatives of the largest components of graph components (Section 4.1) and are marked as EXEMPT in the second pass in Lines 48–49.

During the execution of Mastiff, exactly one component of each graph component is EXEMPT; therefore, RV\_count is not changed in step 5.

### 4.4 Implementation

The atomic_arg_max() in Lines 11 and 47 is implemented as a loop of compare_and_swap(); however, as the maximum value is written, the majority of the accesses to this function are performed without atomic memory accesses.

In addition to the add operation in Line 41 which is performed atomically, there are two other cases that require protection from concurrent processing: (1) in Line 35 adding an edge to the MSF is protected by assigning a buffer for each thread to collect all its edges, and (2) in Line 43, RV\_count is protected by reduction. To that end, each thread has a private counter that is increased and then, the total sum of counters are reduced from RV\_count.

In step 2 of the while loop (Lines 30–33), the edges that are selected as the lightest edge of both endpoint components are identified and the selection of one endpoint is ignored. There is still another case that can result in a cycle when the graph does not have unique weights.

Assume that the graph has a cycle containing more than 2 vertices and each edge on this cycle has the same weight and, moreover, each of these edges of the cycle are the lightest edge of their endpoints. As we have more than 2 vertices in this cycle, random selection of the lightest edges of the vertices/components on this cycle results in adding a cycle to the MST.

To prevent formation of these same-weight lightest cycles, it is necessary to identify the lightest edge of each vertex with the minimum Parent ID. To that end, in Lines 27 and 29 we need to update the lightest edge if: (i) a new edge with lightest weight is found, OR if (ii) the new edge has the same weight but its Parent is smaller than the Parent of the current lightest edge.

### 5 Evaluation

#### 5.1 Experimental Setup

We present experiments on 2 machines with different processor architectures, listed in Table 1. The machines use CentOS 7.

Table 2 shows the datasets and their sources: “Konec” (KN) [7, 34, 42], “NetworkRepository” (NR) [10, 13, 16, 47, 52], “Laboratory for Web Algorithmics” (LWA) [7–10, 35], and “Web Data Commons” (WDC) [36, 38, 39]. Datasets types are Road Networks (RN), Knowledge Graph (KG), Social Network (SN), and Web Graph (WG). Numbers of symmetric edges are shown in billions and numbers of vertices are in millions, counted after removing zero degree vertices.
<table>
<thead>
<tr>
<th>Dataset</th>
<th>SkyLakeX</th>
<th>Epyc</th>
</tr>
</thead>
<tbody>
<tr>
<td>GB Rd</td>
<td>0.13</td>
<td>0.11</td>
</tr>
<tr>
<td>US Rd</td>
<td>0.39</td>
<td>0.22</td>
</tr>
<tr>
<td>WWiki</td>
<td>0.23</td>
<td>0.15</td>
</tr>
<tr>
<td>LJ</td>
<td>0.43</td>
<td>0.23</td>
</tr>
<tr>
<td>LJ Grp</td>
<td>2.01</td>
<td>1.49</td>
</tr>
<tr>
<td>Twtr10</td>
<td>4.97</td>
<td>3.65</td>
</tr>
<tr>
<td>Twtr</td>
<td>3.4</td>
<td>3.5</td>
</tr>
<tr>
<td>WebB</td>
<td>3.2</td>
<td>2.8</td>
</tr>
<tr>
<td>Twtr Mpi</td>
<td>8.9</td>
<td>10.1</td>
</tr>
<tr>
<td>Frndstr</td>
<td>11.4</td>
<td>8.1</td>
</tr>
<tr>
<td>SK</td>
<td>9.2</td>
<td>3.2</td>
</tr>
<tr>
<td>WbCc</td>
<td>26.6</td>
<td>16.5</td>
</tr>
<tr>
<td>UKDls</td>
<td>13.3</td>
<td>10.7</td>
</tr>
<tr>
<td>UU</td>
<td>12.8</td>
<td>11.4</td>
</tr>
<tr>
<td>UK Dmn</td>
<td>13.4</td>
<td>10.5</td>
</tr>
<tr>
<td>CluWEB12</td>
<td>118.7</td>
<td>22.2</td>
</tr>
<tr>
<td>UK 14</td>
<td>97.0</td>
<td>27.0</td>
</tr>
<tr>
<td>WDC 14</td>
<td>-</td>
<td>45.7</td>
</tr>
</tbody>
</table>

### Table 2. Datasets

Graphs are represented in CSX (Compressed Sparse Row / Column) [49] format with $|V| + 1$ index values of 8 bytes per index value and $|E|$ elements for edges. Each edge contains 4 bytes as neighbour ID and 4 bytes as weight. Weights are assigned randomly.

We implemented Mastiff in the C language using the OpenMP API [19], Libnuma, and papi [57] libraries. We use the interleaved NUMA memory policy and to have a better load balance[50] in processing edges, we use edge-balanced partitions [54]. Other loops over vertices are performed using vertex-balanced partitions. The gcc -O3 flag.

We evaluate Mastiff in comparison to implementations of Borůvka’s algorithm in (1) GBBS [20] (commit 38964eb, OpenMP) and in (2) Galois [43] (release 6).

#### 5.2 Comparison to Previous Works

Table 3 compares the execution time of Mastiff to Borůvka’s implementations in GBBS and Galois. GBBS uses edge bucketing [18, 48, 63] that runs Borůvka’s algorithm multiple times and each time for a bucket of edges (separated based on weight of edges). Galois uses a disjoint set structure to avoid rewriting neighbour lists after selecting the lightest edges. Galois performs a preprocessing step to sort edges of vertices that have not been included in the execution time.

The SkyLakeX machine has been used for graphs smaller than ClueWeb12 and Table 3 shows that Mastiff is 5.9 times faster than Galois and 3.2 times faster than GBBS. The Epyc machine has been used for all datasets and Table 3 shows that Mastiff is 3.5 times faster than GBBS on this machine.

#### 5.3 Analysis of Evolution of Components and Vertex Status

Figure 6 shows the distribution of vertices in different iterations. Note that the number of iterations may slightly differ as the results are collected from both machines. To separate the vertices merged to an EXEMPT vertex from those merged into a ROOT vertex, the status of the Parent has been shown for the MERGED vertices. The number of “Merged (Exempt)” vertices in this figure shows the size of the giant components.

Figure 6 shows that after the first iterations, most of MERGED vertices have an EXEMPT Parent and are skipped by Mastiff. It also shows that the fraction of vertices in ROOT and “Merged (Root)” status drops very quickly. As a result, the percentage of vertices that are processed by Mastiff shrink dramatically over iterations and as Figure 7 demonstrates, the execution times of iterations are reduced.
Figure 6. Status distribution of vertices after Mastiff iterations - For the MERGED vertices, the parent status has been shown in parenthesis.

This confirms that Mastiff has been successful in achieving its design goal to reduce the work performed in iterations while avoiding rewriting the graph topology (Section 3.1).

5.4 Execution Breakdown

Figure 6 shows that as the number of iterations are increased, the EXEMPT component includes more vertices. Figure 7 shows the execution breakdown of Mastiff on the Epyc machine. It depicts the percentage of time passed in the initialization step (Lines 1-22 of Algorithm 1), followed by the percentage of time passed in different iterations. Figure 7 shows that after the first iterations, the execution times of iterations are reduced. This is the result of growth of the EXEMPT component that reduces the number of vertices that are processed for selecting the lightest edge of each ROOT component.

5.5 Hardware Events

Figure 8a compares the last level cache misses in the execution of Mastiff in comparison to Borůvka on the SkyLakeX machine. It shows that Mastiff reduces cache misses by 2.2 times, on average. Figure 8b compares the memory accesses (load and store instructions) and shows that mastiff reduces memory accesses by 1.6 times, on average. The comparison of hardware instructions in Figure 8c shows that Mastiff reduces hardware instructions by 1.4 times, on average.

For graphs with a high number of vertices (such as Web-Base or graphs larger than UK-Delis), the numbers of memory accesses and hardware instructions are increased by Mastiff. That is the result of 6 par_for loops in each iteration of Mastiff that are performed for all vertices. However, the actual data required for processing the loop bodies is fetched only for vertices with specific status; moreover, the fraction of the vertices with a relevant status decreases over time. Therefore, the Status array is accessed always but the actual data is rarely required. In this way, memory operations are mostly read accesses to the Status array that is prefetched and also kept in cache. As a result, the total cycles are reduced by 3.3 times on the SkyLakeX machine.

On the other hand, steps 2, 3, and 5 in Algorithm 1 are performed on the components (as ROOT vertices) and the number of memory accesses and hardware instructions in these steps can be significantly reduced by using a sparse frontier (worklist) for components. Figure 9 shows that after 2 iterations less than 5% vertices have ROOT and EXEMPT statuses.

5.6 Depth of Components’ Trees

In step 4 of Algorithm 1, function compress_path() traverses all MERGED parents of a vertex until finding the root of the tree (which is a vertex with ROOT or EXEMPT status) and then updates the Parent of all intermediate vertices.

Figure 10 illustrates the maximum depth of components in each iteration of the Mastiff algorithm. It shows that the maximum depth of the trees is less than 20.
6 Related Work

6.1 MSF

MSF algorithms are categorized into mainly three types:

1. Borůvka’s algorithm [11], that is explained in Section 2.
2. Jarnik’s algorithm [25] (also known as Prim’s algorithm [46]) starts from a vertex and iteratively grows the tree by selecting the lowest-weight edge between the vertices of the tree (i.e., the previously selected vertices) and a non-selected vertex.
3. Kruskal’s algorithm [33] iteratively selects the lowest-weight edge that connects an endpoint of the previously selected edges to a not selected vertex. This continues until all vertices are marked as selected.

A parallel and distributed algorithm for Borůvka’s algorithm is presented in [15] that in each iteration merges vertices that are in the same component and removes self-edges of the component. This paper introduces the supervertex algorithm as a new algorithm to accelerate pointer jumping. Locality of Borůvka’s algorithm has been explored in [17] and a GPU implementation of Borůvka’s algorithm is presented in [60] that packs weights of edges and destinations in the same array. Edge bucketing has been proposed in [18, 63] to accelerate searching for the lightest edges in Borůvka’s algorithm.

Edge bucketing has similarities to $\Delta$-stepping [40] in Single-Source Shortest Paths that processes (relaxes) edges in different steps and only after ensuring the shortest distances in the previous step have been settled.

Parallel implementation of Prim’s algorithm introduced in [4] that selects a number of start points and simultaneously grows distinct trees. Upon identifying a neighbour in another tree, the tree stops growing and vertices on the same tree are merged to a new vertex. This process is continued until no edges remain. A similar technique has been used in [44] that merges two components upon finding a conflict.
Parallelization of the searching for the lightest edge of Prim’s algorithm has been introduced in [37].

Edge bucketing is used in [48] to reduce the overhead of Kruskal’s algorithm and to avoid accessing all edges in each iteration. The opportunity to parallelize searching for the lightest edge and also merging has been studied in [37]. Helper threads are used in [26] to identify cycles in Kruskal’s algorithm and to remove the heaviest edges of the cycles.

Distributed memory implementations of MSF have been studied in [15, 37, 45] and GPU-based ones in [44, 48, 60].

A comprehensive study and analysis of MSF algorithms, their complexities, and parallelization opportunities has been presented in [22]. Finding replacements in MSF has been studied in [3].

Rewriting the neighbour list of vertices has been explored in some studies [22, 23]. While it is not efficient for Mastiff to rewrite neighbour lists of all vertices in ROOT components, further investigation is required to identify if it is useful to rewrite the neighbour list of high-degree vertices in some iterations.

6.2 Structure-Aware Graph Algorithms


PowerLyra [14] reduces the communication cost by using vertex-cut partitioning for low-degree vertices and edge-cut for high-degree vertices. In this way, PowerLyra ensures that replicas of low-degree vertices are not increased and processing high-degree vertices experience better load balance.

To provide better load balance in using CPU and GPU integrated devices, FinePar [62] assigns high-degree vertices to CPU while processing low-degree vertices by GPU.

VEBO [55] introduces a partitioning algorithm that distributes high-degree vertices on different partitions, while trying to assign equal number of edges to partitions.

The implications of real-world graphs on SpMV-based graph processing is studied in [28, 29] by investigating the connection between different vertex classes of the graphs. It is also explained how the structure of a power-law graph provides better Push Locality (in traversing a graph in the push direction), or Pull Locality (for traversing a graph in the pull direction).

iHTL [27] is a structure-aware SpMV with optimized locality in processing power-law graphs. iHTL extracts dense sub-graphs containing incoming edges to in-hubs and processes them in the push direction; while processing other edges in the pull direction.

Thrifty [30] is a structure-aware label propagation Connected Components algorithm that optimizes work efficiency by introducing Zero Planting and Zero Convergence techniques to accelerate label propagation and to prevent processing all edges of the graph in pull iterations. In this way, Thrifty processes only a small percentage of edges.

Lotus [31] optimizes locality in Triangle Counting (TC) by separating hub edges from non-hub edges and dividing TC into 3 steps. In this way, Lotus (1) provides a compact presentation for hub edges and optimizes the cache capacity usage, (2) concentrates random memory accesses in each step to a small data structure that is easier to be maintained in cache, and (3) prunes unnecessary searches.

7 Conclusion and Future Work

This paper investigates the formation of components in Borůvka’s algorithm in processing power-law graphs. Based on the novel observations, we introduce the MASTIFF algorithm that accelerates MSF by avoiding processing the largest component in each graph component, and by avoiding topology operations such as merging neighbour lists and relabeling vertices and edges. The evaluation shows that Mastiff is 3.4–5.9 times faster than previous works.

The following cases are our suggestions for future work:

- In addition to MSF, writing graph topology is a time- and memory-consuming step in several graph algorithms like Louvain [6], maximum weighted clique [12], and graph coloring [59]. It is an open question how to exploit the structure of graphs for these algorithms to avoid topology rewriting.
- As explained in Section 5.5, there is an opportunity to reduce memory accesses by using a sparse data structure for tracking ROOT and EXEMPT components.

Source Code Availability

Source code repository and further discussions are available online in https://blogs.qub.ac.uk/GraphProcessing/MASTIFF-Structure-Aware-Minimum-Spanning-Tree-Forest/.

Acknowledgments

This work is partially supported by the High Performance Computing center of Queen’s University Belfast and the Kelvin-2 supercomputer (EPSRC grant EP/T022175/1) and by DiPET (CHIST-ERA project CHIST-ERA-18-SDCDN-002, EPSRC grant EP/T022345/1).

First author is supported by a scholarship of the Queen’s University Belfast and the Department for the Economy, Northern Ireland.

We are grateful for the constructive feedback and suggestions of our ICS’22 reviewers.

References
