A Generic and Highly Efficient Parallel Variant of *Borůvka*'s Algorithm

Cristiano da Silva Sousa
Department of Informatics
University of Minho
cristiano.sousa126@gmail.com

Artur Mariano
Institute for Scientific Computing
Technische Universität Darmstadt
artur.mariano@sc.tu-darmstadt.de

Alberto Proença
Department of Informatics
University of Minho
aproenca@di.uminho.pt

***Abstract*—This paper presents (i) a parallel, platform-independent variant of *Borůvka*'s algorithm, an efficient Minimum Spanning Tree (MST) solver, and (ii) a comprehensive comparison of MST-solver implementations, both on multi-core CPU-chips and GPUs. The core of our variant is an effective and explicit contraction of the graph. Our multi-core CPU implementation scales linearly up to 8 threads, whereas the GPU implementation performs considerably better than the optimal number of threads running on the CPU. We also show that our implementations outperform all other parallel MST-solver implementations in (ii), for a broad set of publicly available road-network graphs.**

I. Introduction

The Minimum Spanning Tree (MST) of a graph is the set of edges that connect every vertex contained in the original graph, such that the total weight of the edges in the tree is minimized. The problem crops up in several domains, although it plays a more relevant role in Very Large Scale Integration (VLSI) design and network routing [1]. The research on MSTs has been active for several decades, period within numerous MST-solvers and implementations have been proposed. In many application domains, such as ad-hoc networks, MST-solvers are often required, thereby demanding efficient implementations.

There are several MST-solvers, almost all of which are variants inspired by three seminal contributions: *Borůvka*'s algorithm, presented in 1926 [2], *Kruskal*'s algorithm, in 1956 [3] and *Prim*'s algorithm, a year later, in 1957 [4]. These algorithms have been implemented in several (parallel) computing devices (e.g. *Prim*'s on FPGAs [5], a marriage between *Prim*'s and *Borůvka*'s algorithms on multi-core CPU-chips [6], and *Borůvka*'s on GPUs [7]).

Sequential implementations of *Borůvka*'s, *Prim*'s and *Kruskal*'s algorithms are very competitive, and their performance varies with the input graph and used data structures [8]. Until the late 90s, the investigation around these algorithms revolved around implementation details to improve the performance of the algorithms, and some were shown to greatly influence the performance of the algorithms [1], [8]. From then on, parallelizing these algorithms has become a central point of research, as shown by the various efforts recorded in the literature, which we overview in Section III.

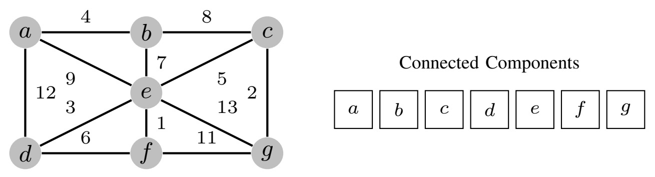
In this context, *Kruskal*'s algorithm is the least attractive candidate, due to its inherently sequential workflow. In contrast, *Prim*'s algorithm is more suited for parallelization but, it either breaks down to operations with reduced parallelization opportunities or ends up with overly complex parallel procedures, which require heavy use of fine-grained synchronization that substantially reduces the possible speedups [6], [9]. *Borůvka*'s algorithm, on the other hand, is naturally parallel, thereby becoming the strongest candidate for parallelization.

Graphic Processing Units (GPUs) have been gripping increasing attention due to their great potential for exploiting parallelism in regular, data-parallel algorithms. For irregular algorithms, such as those working on graphs, heavy hand-tuned code is necessary to attain good performance levels (e.g. [10]). Although GPUs are not tailored for irregular applications, they have been used, with satisfactory results, to implement graph algorithms where an operator is applied on every vertex, since the underlying execution model makes it intuitive to map a vertex per thread. This pattern is present in *Borůvka*'s algorithm, since an operator (searching for the lightest edge) is applied to all the vertices in every iteration.

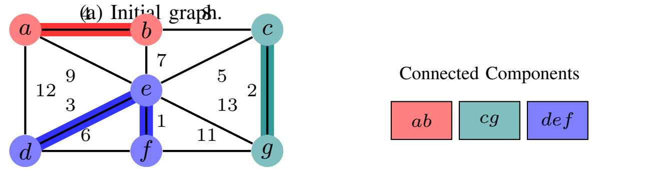
The contribution of this paper is two-fold. First, we present a parallel and platform independent variant of *Borůvka*'s algorithm that attains high performance and good scalability on multi-core CPU-chips and GPUs, in isolation. Second, we present a comprehensive comparison of the implementations of MST-solvers described in [11], [7], [12] and with the framework described in [13], wherein we include the implementations of our variant, which outperforms all the others.

Our proposal of an efficient parallel variant is based on specific design and implementation decisions, such as data representation (Compressed Sparse Row [CSR] format) and primitive selection that can be applied to enhance the performance of the algorithm, since data locality and data coalescing are improved on CPUs and GPUs, respectively. In particular, we introduce a new, very effective approach to perform a contraction of the graph (merging vertices into super-vertices). Our contraction process includes a very effective construction of the newly contracted graph, directly in the CSR format, since the elements of the new graph are known upfront. Moreover, our variant is platform-independent, i.e., it can be implemented on both CPU-chips and GPUs (and even on distributed systems, which we do not cover in this paper) without any modification. To this day, all implementations of *Borůvka*'s algorithm required specialized treatment for the underlying architecture.

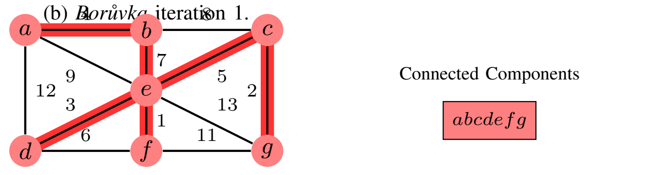
The rest of the paper is organized as follows. Section II introduces *Borůvka*'s algorithm. Section III compiles the related work that directly pertains to our variant. Section IV



(a) Initial graph.



(b) *Borůvka* iteration 1.



(c) *Borůvka* iteration 2.

Fig. 1: *Borůvka*'s algorithm.

presents our algorithmic variant. Section V shows how our implementations compare to their counterparts shown in Section III. Section VI discusses and generalizes those results. Section VII concludes the paper and presents future lines of research.

II. *Borůvka*'s Algorithm

In the 1920s, *Borůvka* was asked to find the most economic solution to construct an electrical power grid. The proposed algorithm, described in [2], first initializes each vertex as a connected component with a single element. A connected component is a subset of the graph, where any two vertices are connected to each other by a path, and no vertex is connected to a vertex of another component. Afterwards, the algorithm selects, for each component, the shortest edge that connects it to another component. The components that were connected by this selected edge are joined together, thus joining two components into a new one. This process is repeated until all the vertices are joined within the same, single component. The union of the edges selected at each iteration form the MST. A graphical description of the algorithm is shown in Figure 1.

Efficient implementations can be obtained using a disjoint-set structure. A disjoint-set allows to keep track of different elements (vertices) across non-overlapping subsets (connected components). Alternatively, the end-point vertices of each selected edge can be contracted into a single super-vertex, explicitly removing all the edges that connect vertices inside the same super-vertex. If multiple edges connect the same super-vertices, only the lightest is kept. With this strategy, edges that can never be part of the MST are quickly excluded. However, the average edge degree of each super-vertex can grow quickly if duplicated edges are not filtered out.

III. Related Work

The state of the art of both sequential and parallel implementations of MST-solvers is quite extensive and comprehensive literature compilations of the existent algorithms can be found in [1], [14]. In this section, we overview the literature that pertains directly to our work. Comparisons with distributed memory implementations (such as from the Parallel Boost Graph Library[1]) are out of the scope of this paper.

A. *SMP systems and multi-core CPU-chips*

The first parallel *Borůvka* implementation for shared memory Symmetric Multiprocessing (SMP) systems was presented in [6]. The authors implemented a variant of *Borůvka*'s algorithm that contracts the graph at each iteration. They also presented a new data structure, the flexible adjacency list, which, when compared to the adjacency list, is more suited for graph contraction on the CPU. Furthermore, a new parallel implementation is presented as a combination of *Prim*'s and *Borůvka*'s algorithms. This algorithm grows multiple instances of *Prim*'s algorithm from different starting vertices. When one *Prim* instance encounters another, it restarts from a different vertex. When all the vertices have been visited, the algorithm performs one iteration of *Borůvka*'s and restarts with multiple instances of *Prim*'s algorithm. A very conservative lock-free mechanism is employed to handle possible conflicts, thus incurring additional, excessive overhead. In contrast, our implementation is composed of kernels that are either embarrassingly parallel or implementable with minimal synchronization.

The same authors presented in [11] an algorithmic variant of *Borůvka*'s algorithm that uses colors to denote super-vertices, from here on referred to as *Cong2005*. There are two implementations of this variant, one with platform-specific assembly instructions, which we cannot use for comparisons purposes, and one with *pThread* mutexes, whose performance is shown in Section V. Our implementation both more generic than *Cong2005*, since no machine-specific assembly instructions are used, and more efficient for every tested case.

The *Galois* framework, presented in [13], is a system that automatically executes serial code on CPU-chips, in parallel. This framework includes a set of benchmarks, one of which being *Borůvka*'s algorithm. Executing any of the available benchmarks involves the use of the underlying framework, which is complex. As shown in Section V, our CPU implementation outperforms *Galois* both in sequential (with 1 thread) and in parallel (from 2 to 40 threads) executions.

B. *GPUs*

The first parallel implementation of an MST-solver on GPUs was described in [7], which we refer to as *Harish2009*. Using CUDA, the authors implemented a parallel variant of *Borůvka*'s algorithm. To distinguish super-vertices, colors are used and cycles are explicitly removed. Speedups were obtained in comparison to a CPU version presented in the paper. Our GPU implementation is significantly faster than *Harish2009*, achieving speedups between 2x and 26x, and our CPU implementation outperforms this implementation with 4 threads or more, as shown in Section V.

Also published in 2009, [10] describes a GPU implementation of *Borůvka* that resorts to explicit graph contraction, instead of colors, creating super-vertices at each iteration. The authors reported speedups in comparison to *Harish2009*,

[1]http://www.boost.org/

using parallel primitives from CUDA Data Parallel Primitives Library[2]. While this implementation outperforms *Harish2009*, it has some limitations: it packs vertex ids and weights into 32 bits, reserving 22 to 24 bits (configurable, at compile time) for vertex ids, and 8 to 10 for edge weights, which limits the number of vertices and edge weights of input graphs. As a result, the user has to change the weights of the edges on the graph, both if the graph is large or has high edge weights. Our implementation, on the other hand, does not depend on such parameters. We do not include this implementation in our comparative analysis, in Section V, as these restrictions limit the comparisons against all the other implementations.

In 2011, two other implementations were published [15], [9]. [15] focuses on the memory usage on the GPU, proposing an algorithmic variant of *Kruskal*'s that splits the edges by weight into partitions such that the maximum edge weight of a given partition is less than or equal to the minimum edge weight of any subsequent partition. The algorithm considers lighter edges before the heavier ones by processing one partition at a time, which results in a smaller memory footprint on the GPU. Unfortunately, we did not have access to this implementation. Moreover, their most efficient implementation also employs a bit-packing mechanism similar to [10], as such, it would not have been included in our comparison benchmarks, for the same reason we described previously.

Another GPU implementation of a parallel variant of *Prim*'s algorithm was presented in [9]. The two inner loops, i.e. finding the minimum edge and updating the candidate set, were parallelized with data-parallel primitives. The authors reported limited speedups with respect to a CPU implementation provided by the Boost Graph Library (BGL[1]). However, we were not able to obtain this implementation, and the most recent version of BGL (1.56.0) did not seem to deliver the correct results. The same algorithm was implemented on embedded systems and FPGAs in 2013 [5], but comparisons with these specialized devices are out of the scope of this paper.

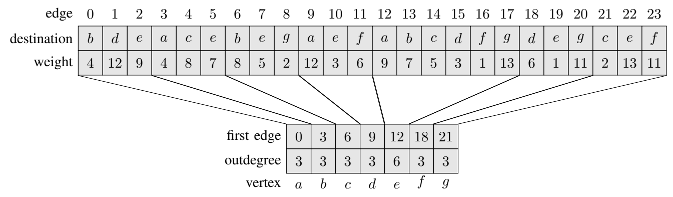
In 2013, a similar implementation to *Harish2009* was presented [12], from here on referred to as *Nasre2013*. The authors obtained no speedup in comparison with *Galois*.

C. Wrap up

While reviewing the literature, it stood out that there is a clear trade-off between the effort put in implementing the algorithms and the performance that is ultimately delivered. In order to boost performance, several implementations introduce parameters that somehow limit the usability of the application, making them tailored to specific graph types (e.g. reserved bits in [10], [15]). Up until the late 90s, the focus of optimization of these algorithms had been on graph representation and the usage of intermediate data structures such as heaps and disjoint-sets. Afterwards, the focus shifted to parallelization for SMP systems and, shortly after, to GPUs and multi-core CPU-chips.

In this paper, we present a parallel variant of *Borůvka*'s algorithm and efficient implementations of the proposed variant for multi-core CPU-chips and GPUs. We show that our implementations outperform the state of the art implementations

[2]http://cudpp.github.io/



edge	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
destination	b	d	e	a	c	e	b	e	g	a	e	f	a	b	c	d	f	g	d	e	g	c	e	f
weight	4	12	9	4	8	7	8	5	2	12	3	6	9	7	5	3	1	13	6	1	11	2	13	11

first edge	0	3	6	9	12	18	21
outdegree	3	3	3	3	6	3	3
vertex	a	b	c	d	e	f	g

Fig. 2: CSR representation of the example graph in Figure1.

described in *Cong2005*, *Harish2009*, *Galois* and *Nasre2013*. As a result, there is not, to the best of our knowledge, any disclosed implementation that outperforms ours.

IV. A PARALLEL VARIANT OF *Borůvka*'S ALGORITHM

In this section, we present our variant of *Borůvka*'s algorithm, addressing the key issues to a platform-independent variant: the graph representation and contraction in the CSR format.

A. Graph representation

The choice of the data structure for representing the graph is very relevant for the performance of the implementation. The most common representation in the literature is the adjacency list and the adjacency matrix. [6] introduces an extension to the adjacency list representation specifically for *Borůvka*'s graph contraction algorithm: each index can point to multiple lists of incident edges, making it much easier to merge vertice's edges. However, this representation is not suited for GPUs.

The CSR format is a compromise between adjacency list and adjacency matrix. It is often seen in the literature as the representation used in GPU implementations of graph algorithms. In this format, the graph is represented by four arrays:

- `destination` - an array of size $|E|$, which maps each edge to its destination;
- `weight` - an array of size $|E|$, which maps each edge to its weight;
- `first edge` - an array of size $|V|$, which maps each vertex to its first edge;
- `outdegree` - an array of size $|V|$, which maps each vertex to the number of outgoing edges it has.

To represent undirected graphs, all edges are duplicated to cover both directions. Figure 2 shows the CSR representation of the example graph shown in Figure 1.

When the graph structure might change, the use of the adjacency lists is more adequate, as CSR does not offer an easy way to alter the graph structure. This comes at a performance cost, since adjacency lists are usually implemented using linked-lists, while CSR is a more cache friendly approach. However, in Section IV-B, we show that our variant allows each contracted graph to be built, from the ground up, in the CSR format. This is possible because the numbers of vertices, edges and neighbors for each vertex of the contracted graph are known upfront. It is possible to derive `outdegree` from `first edge`. However, the `outdegree` array is required to build the graph in each contraction step.

Algorithm 1 Parallel *Borůvka* variant

```
Input: Undirected, connected and weighted graph $G(V, E)$
 1: while number of vertices $> 1$ do
 2:     Find minimum edge per vertex
 3:     Remove mirrored edges
 4:     Initialize colors
 5:     while not converged do
 6:         Propagate colors
 7:     Create new vertex ids
 8:     Count new edges
 9:     Assign edge segments to new vertices
10:     Insert new edges
```

vertex	a	b	c	d	e	f	g
vertex_minedge	ab	ba	cg	de	ef	fe	gc

(a) Find minimum edge per vertex.

vertex	a	b	c	d	e	f	g
vertex_minedge	$-$	ba	$-$	de	$-$	fe	gc

(b) Remove mirrored edges.

vertex	a	b	c	d	e	f	g
color	a	a	c	e	e	e	c

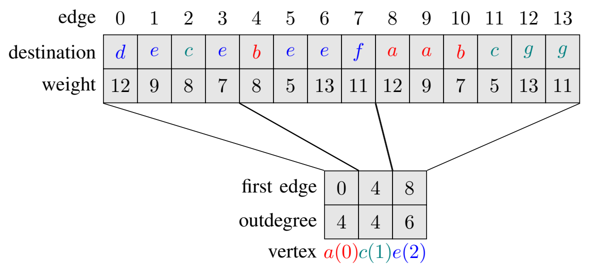
(c) Initialize and propagate colors.

vertex	a	b	c	d	e	f	g
flag	1	0	1	0	1	0	0
exclusive prefix sum	0	1	1	2	2	3	3

(d) Create new vertex ids.

vertex	$a(0)$	$c(1)$	$e(2)$
outdegree	4	4	6
first edge (after exclusive prefix sum)	0	4	8

(e) Count and assign edges.



edge	0	1	2	3	4	5	6	7	8	9	10	11	12	13
destination	d	e	c	e	b	e	e	f	a	a	b	c	g	g
weight	12	9	8	7	8	5	13	11	12	9	7	5	13	11

first edge	0	4	8
outdegree	4	4	6
vertex	$a(0)$	$c(1)$	$e(2)$

(f) Inserting edges and contracted graph.

Fig. 3: Progress of the algorithm after applying each kernel on the initial state of the example graph.

B. Algorithmic variant

Our algorithmic variant comprises a series of simple kernels, as shown in Algorithm 1[3]. All kernels, with exception of the two kernels that can be implemented with an exclusive prefix sum, are applied to each vertex as an operator, and each vertex can be processed independently of all others, only requiring a barrier synchronization between kernels. This sequence of kernels is repeated until one super-vertex remains:

[3]Assume w.l.o.g. that the graph is connected.

1) Find minimum edge per vertex: the algorithm starts off by selecting the minimum weight edge for each vertex. When the vertex has multiple edges with the same minimum weight, the edge with the smallest destination vertex id is selected. The selected edge id is stored in an array called `vertex_minedge`. Figure 3a shows the selected edges for each vertex of the example graph in the initial state (Figure 1a).

We do not resort to a segmented parallel reduction for this, as we would not be able to directly select the edge with the smallest destination vertex id, and additional computation would be required to remove cycles. Instead, only mirrored edges need to be removed.

2) Remove mirrored edges: mirrored edges are removed if the successor of a vertex successor is the vertex itself. When a mirrored edge is found, the edge is removed once from the `vertex_minedge` array, maintaining the edge by its endpoint with the largest vertex id. For instance, in Figure 3b, one of the mirrored edges is the pair eg and ge, since $g > e$, the edge eg, selected by vertex e, is removed while the edge ge, selected by vertex g, is maintained. The edges that remain in the `vertex_minedge` array are marked to be part of the MST.

3) Initialize and propagate colors: in order to contract the graph, connected components must be identified. Each connected component will be a super-vertex in the contracted graph. To this end, each vertex is initialized with the same color as their successor's id. If a vertex has no successor, because the edge has been removed in step 2), its successor is set to himself. The successors are then propagated by setting the successor of a vertex to its successor's successor. This process is repeated until it converges. Consider the newly created component by the vertices d, e and f, in Figure 3c: e sets its successor to himself since it has no selected edge, while d and f selected the edges de and fe, respectively, set their successor both to e. In this particular case, no propagation takes place as the successors converge immediately.

Steps 4) and 5) compose the core of our algorithmic variant, showing our approach to build the newly contracted graph in a very effective manner.

4) Create new vertex ids: after converging, any vertex successor that is the vertex itself will be the representative vertex for its component and is marked with `1` in a flag array. All other vertices are marked with `0`. An *exclusive prefix sum* is then computed on the flag array, assigning new vertex ids for the contracted graph. In Figure 3d, a, c and e are the representative vertices. After computing the prefix sum, these vertices are assigned the new vertex ids 0, 1 and 2, respectively.

By using a prefix sum we ensure that the new vertex ids are in order with respect to the old vertex ids, i.e., the smallest vertex id in the old graph will be part of the component whose representative is assigned the smallest new vertex id. Furthermore, this maintains any proximity between a vertex id and the id of its neighbors, all of which improve locality.

5) Count, assign, and insert new edges: to build edge arrays for the contracted graph, it is first necessary to identify how many edges each super-vertex will have, in order to assign new edge ids to the super-vertices. This is achieved using a simple kernel that counts the number of edges that cross the component for each vertex, and adds it to `outdegree` array

	CPU	GPU
#Devices	2	1
Manufacturer	Intel	NVIDIA
Model	E5-2670 v2	K20m
Launch date	Q3'13	Q1'13
μArch	Ivy Bridge	Kepler
#Cores	10	2496
Clock frequency	2500 MHz	706 MHz
L1 Cache	32 KB IC + 32 KB DC	16/32/48 KB/SM
L2 Cache	shared 256 KB/core	1.25 MB
L3 Cache	shared 25 MB/chip	n/a
Memory	64 GB	5 GB

TABLE I: System characteristics.

No.	Name	Description	#nodes	#edges
1	NY	New York City	264.346	733.846
2	BAY	San Francisco Bay Area	321.270	800.172
3	COL	Colorado	435.666	1.057.066
4	FLA	Florida	1.070.376	2.712.798
5	NW	Northwest USA	1.207.945	2.840.208
6	NE	Northeast USA	1.524.453	3.897.636
7	CAL	California and Nevada	1.890.815	4.657.742
8	LKS	Great Lakes	2.758.119	6.885.658
9	E	Eastern USA	3.598.623	8.778.114
10	W	Western USA	6.262.104	15.248.146
11	PT	Full Portugal	9.196.206	20.127.796
12	CTR	Central USA	14.081.816	34.292.496
13	USA	Full USA	23.947.347	58.333.344

TABLE II: Road-network graphs used in benchmarks.

of its corresponding super-vertex. Since multiple vertices may belong to the same super-vertex, an atomic function for the operations on the `outdegree` array has to be used.

We then compute an exclusive prefix sum on the `outdegree` array, assigning segments of edge ids to each super-vertex. The prefix sum ensures that the segments of the edge ids are assigned with accordance to the super-vertex id, i.e., the smallest edge ids are assigned to the smallest super-vertex id. This creates the new `first edge` array.

Once the edge ids are assigned to the super-vertices, all edges that cross components are added to the contracted graph. We first make a copy of the `first edge` array, which is going to be used to keep track of the current position to insert the new edge, since multiple vertices can belong to the same super-vertex. When a thread wants to add a new edge, it performs an atomic increment on this array, on the position of the super-vertex id. The old value, that is returned by the atomic function, is used as the id for the edge that is added. We discard intra-component edges by comparing the colors of the two end-points of each edge. However, we do not remove duplicate edges between pairs of super-vertices, as the benefit of doing this does not outweigh the incurred computational cost. Figure 3e shows the number of neighbors for each super-vertex. E.g. $a(0)$ has four: ad, ae, be and bc. Even though the first three connect the same super-vertices, they are still added.

Figure 3f shows the newly contracted graph, at the end of the iteration. The graph is built with low overhead, but with all the benefits of being able to use an array based data structure in the whole algorithm.

V. Results

All tests were carried out on a dual-socket NUMA system, specified in Table I. The CPU codes were compiled with `g++` 4.8.2, and GPU code with `nvcc` 5.5, both with `-O3` flag. We performed a series of empirical benchmarks of our implementations, against *Cong2005*, *Harish2009*, *Galois* and *Nasre2013*. The execution times reported for the GPU implementations include the time to transfer the input graph to device memory (the time to transfer the MST back to the host is not included, since it is negligible). To improve the accuracy of our measurements, we used the *k*-best measurement scheme with 5 measurements, *k* = 3 and a 5% tolerance, i.e., 5 tests are performed and the 3 best results, that are within the 5% tolerance of one another, are selected. The best of the 3 is then used in our results.

In our implementations, we follow a topological approach, having each thread operate on one vertex, for the GPU, and a set of vertices for each thread (one at a time), on the CPU. The GPU blocks are configured to use 1024 threads, organized in a single dimension, and enough blocks are configured to cover all the vertices of the graph. We resort to ModernGPU[4] 1.1 (MGPU) for the parallel primitives. Furthermore, we extended our implementation with the usage of texture memory for a small performance boost, wherein we store the four arrays that represent the graph at a given iteration. The new graph that is being contracted on each iteration remains stored in global memory. For our CPU implementation, we resorted to OpenMP, assigning chunks of vertices to each thread. For the use of parallel primitives, we resorted to Intel TBB 4.2, since OpenMP does not have have an exclusive prefix sum primitive.

As for test graphs, we used the USA road-network graphs from the 9th DIMACS challenge[5], and the OpenStreetMap's[6] Portuguese road-network, provided by Geofabrik[7], as described in Table II.

Figure 4 shows the execution time of all the selected GPU and CPU implementations (with 1 and 10 threads) for the set of input graphs. For *Cong2005*, we were only able to compute the graphs 1 to 6, since executions for the remaining graphs did not terminate in a timely manner, apparently due to a live-lock in the color propagation procedure. As shown in the figure, our CPU implementation outperforms both single-threaded *Cong2005* and *Galois* for all input graphs, running with a single thread. Both our CPU implementation, with 10 threads, and our GPU implementation, outperform all the other implementations. The CPU implementation attains speedups of between 1.35x and 12.71x, with respect to the fastest of the implementations under comparison, and the slowest, respectively, among all the used graphs. The GPU implementation attains speedups from 1.34x to 26.43x. Our results back up the superiority of our implementations, which is a direct consequence of the suitability of our algorithmic variant for parallel architectures.

Figure 5 shows the scalability of the CPU implementations for three particular representative input graphs (NE, PT and USA), and compares them with GPU implementations. Figure 5a shows the results for the NE graph, the largest graph

[4]http://www.moderngpu.com

[5]http://www.dis.uniroma1.it/ challenge9/

[6]http://www.openstreetmap.org/

[7]http://download.geofabrik.de/europe/portugal.html

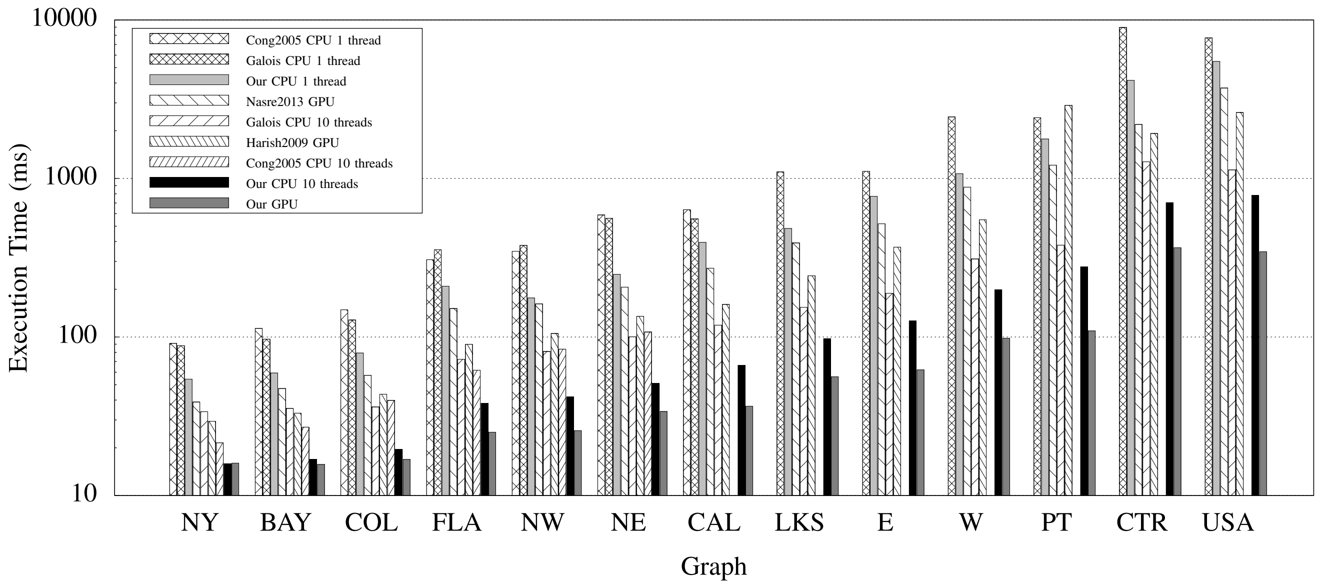
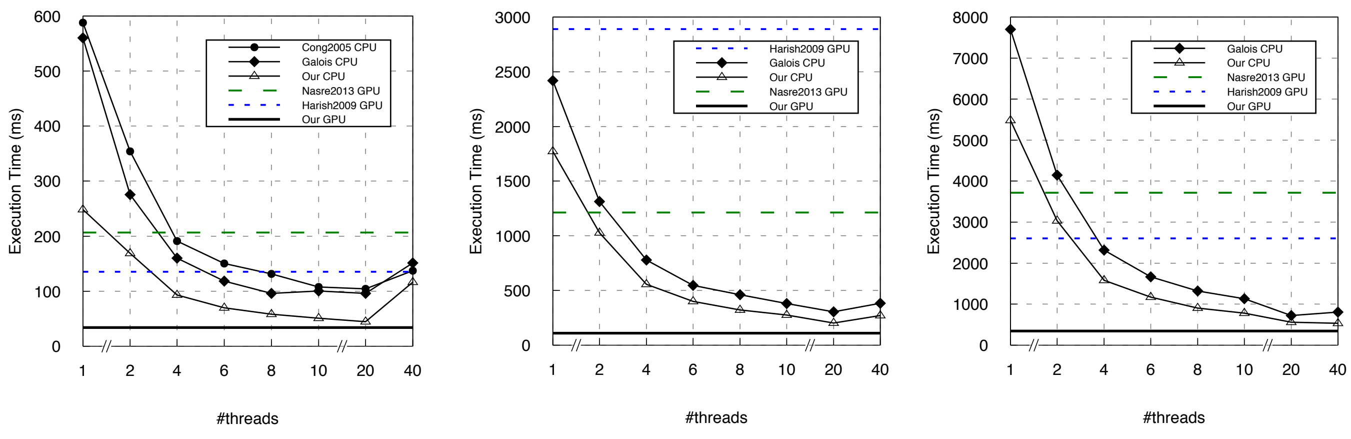
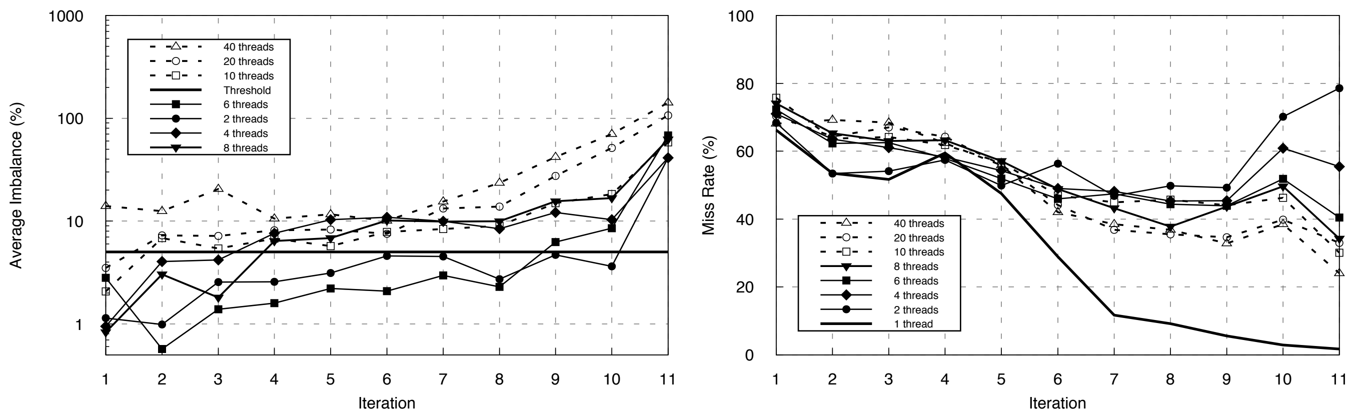


Fig. 4: Measured execution times for all road-network graphs



(a) Measurements for NE road-network graph. (b) Measurements for PT road-network graph. (c) Measurements for USA road-network graph.

Fig. 5: Scalability for 3 road-network graphs.



(a) Average load imbalance per iteration. (b) L3 cache miss rate per iteration.

Fig. 6: Metrics for CPU execution of the USA graph.

for which we were able to execute all implementations and combinations of threads. Figure 5b shows the results for the PT graph, wherein *Harish2009* performs particularly bad, and worse than CPU implementations running with a single thread. For this particular case, we profiled each kernel in *Harish2009* and concluded that the color propagation is inefficient on this particular graph. We believe that this is due to the structure of the PT graph, since the number of neighbors of each graph vertex varies considerably. Figure 5c shows the results for the largest graph in our data set. In all cases, the CPU versions outperform *Harish2009* and *Nasre2013* GPU implementations with relatively few threads. Another remark has to be made for *Cong2005*, whose execution time was very inconsistent. This is can be attributed to the non-determinism of the implementation and the use of mutex locks. Nevertheless, it showed to be competitive, when computing the MST of the NE graph.

Table III summarizes the speedup and efficiency attained by our CPU and GPU implementations, with respect to our CPU implementation running with a single thread for the same representative input graphs (NE, PT and USA). The CPU implementation does not scale for the NE graph, since the graph does not entail enough work for all the spawned threads. For the largest graph, USA, linear and almost linear speedups are achieved for up to 8 threads. There is neither benefit in using the second CPU-chip nor hyper-threading, which we attribute to load imbalance. The load imbalance in our application is originated by the scheduling at the vertex level, instead of scheduling at the edge level. This might lead to load imbalance since vertices with more edges take longer to be processed, even if the number of vertexes assigned to each thread is balanced. We used guided scheduling in our application, which only corrects this problem partially.

We took a deeper look at this problem. Figure 6a shows the average percentage of load imbalance, in terms of edges processed per thread, for each iteration of the kernel described in IV-B1, for different numbers of threads (2-40), when executing on the USA graph. Although we show all the 11 iterations of the algorithm, it should be noted that the first 7 are considerably more relevant than the others, since they represent a much larger chunk of the total execution time (>80%). We also plotted a threshold line at 5%, which we consider the threshold for significant impact from load imbalance on performance. As shown in the figure, the average imbalance increases with the number of threads, thereby hurting scalability. Although scheduling at the edge level would help to mitigate load imbalance, it would substantially increase the complexity of our algorithmic variant. In particular, we would need to resort to a large amount of synchronization and atomic operations, or a primitive for segmented reductions, whose possible implementations are very inefficient, together with a kernel to remove cycles.

We investigated further ways of improving the scalability of our CPU implementation. In particular, we experimented several combinations of thread affinity setups, even though none has shown to perform better than the others. In fact, we believe that there is no optimal thread affinity setup because the edges that are read by one of the threads, are never read by all the others.

	Input Graph					
	NE		PT		USA	
Threads	S	E	S	E	S	E
2	1.47x	74%	1.73x	86%	1.80x	90%
4	2.67x	67%	3.18x	79%	3.47x	87%
6	3.56x	59%	4.42x	74%	4.67x	78%
8	4.28x	54%	5.51x	69%	6.06x	76%
10	4.88x	49%	6.40x	64%	7.01x	70%
20	5.56x	28%	8.75x	44%	9.79x	49%
40	2.13x	5%	6.56x	16%	10.26x	26%
GPU	7.32x		16.21x		15.85x	

TABLE III: Speedup (S) and Efficiency (E) for 3 graphs with respect to our CPU implementation with a single thread.

Using PAPI [16], we measured the L3 cache miss rate[8] for each iteration of the main loop described in Algorithm 1, for different numbers of threads (1-40) on the USA graph. As shown in Figure 6b, for the single-threaded execution, the L3 cache miss rate starts to drop drastically at iteration number 4, and remains very low after iteration 7, where very few RAM accesses have to be made. The algorithm works on two different graphs (the current graph, and the new contracted graph that is built and used in the next iteration) at every iteration. This shows that one of these graphs fits in L3 cache at iteration number 6, and both graphs fit in L3 cache after iteration number 7. However, this behavior is not seen when running with multiple threads, which is an evidence of cache trashing, something that limits both the performance and the scalability of the application. We believe that this is something very difficult to avoid, since it has much more to do with the algorithm than the implementations. Surprisingly, it also happens that, in some cases, the miss rate is lower with larger number of threads (e.g. 2 threads vs 40 threads). This is connected to the load imbalance that originates during the execution of the application: cache contention is reduced, as many threads terminate before others.

VI. Discussion

Current implementations of MST-solvers have some drawbacks: they are too complex, address a limited set of input graphs (as seen in the early sections), or when generic, are considerably less efficient than the average efficient implementation (e.g. *Harish2009*, *Nasre2013*). In our comparisons, we also showed that GPU implementations of *Borůvka*'s algorithm are usually more efficient than CPU implementations, and that the implementations we propose outperform all other tested implementations, using the same algorithm for both multi-core CPU and CUDA implementations.

A. Use of primitives

Another important angle of discussion, regarding implementations of MST-solvers, is that some are described as a stack of parallel primitives (e.g. [10]). However, the use of many parallel primitives is a problem, due to two different aspects. First, they add complexity to the code, also because layout conversion procedures are necessary to pile primitives up, in a single workflow. Second, they limit portability, since many GPU parallel primitives are either unavailable or inefficient on other platforms. The solution for this is to present an

[8]We used the `PAPI_L2_TCM` and `PAPI_L3_TCM` counters.

abstract algorithmic variant that can be efficiently implemented on different platforms, possibly using primitives but without the need to resort to them, as we do in this paper.

In our variant, the two kernels that resort to a parallel prefix sum (create new vertex ids and assign edge segments to new vertices), were previously implemented as a simple kernel using atomics. The relative elapsed time of these kernels was negligible but we noticed that our implementations of these kernels were impairing data locality and causing uncoalesced global memory accesses on the CPU and GPU, respectively. When replaced with parallel primitives, the speedup of these kernels was negligible, but major speedups were attained for all other kernels.

B. Topology- vs data-driven

Two different approaches are normally followed to implement operator based algorithms (such as *Borůvka*'s): a topology-driven approach (all nodes are active) or a data-driven approach (some nodes are active, kept in a work-list) [17]. With *Borůvka*'s coloring approach, vertices that do not have edges crossing super-vertices will be inactive in a data-driven approach, a problem that increases with the density of the graph. A topology-driven approach on multi-core CPU-chips may lead to load imbalance for sparse graphs, while on GPUs may result in inefficient use of SMs. Although this approach configures the GPU kernel with enough blocks to cover all graph vertices, as the algorithm progresses each block will have less work. On the other hand, a topology-driven approach, with *Borůvka*'s graph contraction, leads to a more efficient implementation, since graph contraction keeps all nodes active at each iteration, and explicitly reduces the graph size, thereby reducing the memory footprint and improving spatial locality.

C. Analysis of conducted benchmarks

Our benchmarks also enable us to draw a number of conclusions. For starters, some implementations behaved differently from expected in a couple, particular cases. For instance, *Harish2009* performed worse on a graph with a broad spectrum of vertex degrees. Also, the performance of *Cong2005* was very irregular, even when considering the same input graph computed multiple times.

Regarding the scalability of CPU implementations of *Borůvka*'s algorithms, our implementation, *Cong2005* and *Galois* scaled poorly after 8 or 10 threads on our benchmarking machine, or with small graphs (under 2 million vertices). As we showed in Section V, scalability is hurt by load imbalance, a problem that is not easy to fix, and cache misses, that are due to the irregular memory access pattern of the algorithm, as it is usually the case in graph algorithms.

VII. Conclusion and Outlook

This paper presents (i) a parallel algorithmic variant of *Borůvka*'s algorithm and its assessment on multi-core CPU-chips and GPUs (implementations are publicly available [9]) and (ii) a first-hand comprehensive empirical comparison of several disclosed state of the art CPU and GPU implementations of MST-solvers. The benchmarks that we carried out, with public domain graphs, showed that our CPU implementation outperformed all disclosed parallel CPU implementations, and our GPU implementation outperformed all disclosed MST-solver implementations.

The literature review showed that implementations of MST-solvers are limited in the number and type of graphs that they can work on, and generic implementations are usually inefficient. We fill this gap by presenting implementations that are not only very efficient, as they can also solve every type of graph without the need to adjust parameters.

In the future, we will merge our applications into an heterogeneous CPU+GPU implementation. We also plan to extend our approach to dense graphs, and modify our implementation, if needed, in order to maintain high performance levels.

Acknowledgments

We thank G. Cong for providing the implementation shown in [11], and R. Nasre and F. Correia for insightful discussions.

References

[1] R. L. Graham *et al.*, “On the history of the minimum spanning tree problem,” *Annals of the History of Computing*, vol. 7, no. 1, 1985.

[2] O. Borůvka, “O jistém problému minimálním (about a certain minimal problem),” *Práce Mor. Pírodoved. Spol. v Brne III*, vol. 3, 1926.

[3] J. B. Kruskal, “On the shortest spanning subtree of a graph and the traveling salesman problem,” *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[4] R. C. Prim, “Shortest connection networks and some generalizations,” *Bell system technical journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[5] A. Mariano *et al.*, “Hardware and software implementations of Prim's algorithm for efficient minimum spanning tree computation,” in *Embedded Systems: Design, Analysis and Verification*. Springer, 2013.

[6] D. A. Bader *et al.*, “Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs,” in *Parallel and Distributed Processing Symposium*. IEEE, 2004, p. 39.

[7] P. Harish *et al.*, “Large graph algorithms for massively multithreaded architectures,” *Centre for Visual Information Technology, IIIT, Hyderabad, India, Tech. Rep. IIIT/TR/2009/74*, 2009.

[8] B. M. Moret *et al.*, “An empirical assessment of algorithms for constructing a minimum spanning tree,” *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science*, vol. 15, 1994.

[9] W. Wang *et al.*, “Design and implementation of GPU-based prim's algorithm,” *IJMECS*, vol. 3, no. 4, p. 55, 2011.

[10] V. Vineet *et al.*, “Fast minimum spanning tree for large graphs on the GPU,” in *Conference on High Performance Graphics*. ACM, 2009.

[11] G. Cong *et al.*, “Lock-free parallel algorithms: An experimental study,” in *HiPC 2004*. Springer, 2005, pp. 516–527.

[12] R. Nasre *et al.*, “Morph algorithms on GPUs,” in *Proceedings of the 18th ACM SIGPLAN symposium on PPoPP*. ACM, 2013, pp. 147–156.

[13] K. Pingali *et al.*, “The tao of parallelism in algorithms,” *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 12–25, 2011.

[14] M. Mareš, “The saga of minimum spanning trees,” *Computer Science Review*, vol. 2, no. 3, pp. 165–221, 2008.

[15] S. Rostrup *et al.*, “Fast and memory-efficient minimum spanning tree on the GPU,” *IJCSE*, vol. 8, no. 1, pp. 21–33, 2011.

[16] P. J. Mucci *et al.*, “PAPI: A portable interface to hardware performance counters,” in *Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[17] R. Nasre *et al.*, “Data-driven versus topology-driven irregular computations on GPUs,” in *IPDPS*, 2013, pp. 463–474.

[9]https://github.com/beatgodes/BoruvkaUMinho