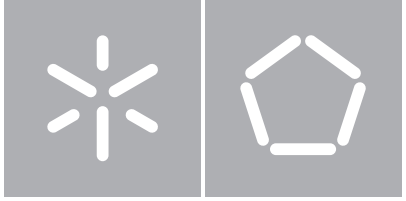
**Universidade do Minho**
Escola de Engenharia

Cristiano Rafael da Silva Sousa

# Efficient sequential and parallel versions of MST-solvers for multi-core CPU-chips and GPUs

Outubro de 2014



**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Cristiano Rafael da Silva Sousa

# Efficient sequential and parallel versions of MST-solvers for multi-core CPU-chips and GPUs

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

**Alberto José Proença**
**Artur Miguel Matos Mariano**

Outubro de 2014

# Acknowledgements

Ao meu orientador, Alberto Proença, que desde o meu primeiro ano da Licenciatura sempre me inspirou pelo seu rigor e brio académico, e pela orientação, tanto durante a dissertação como a nível pessoal. Ao meu coorientador, Artur Mariano, pela oportunidade desta dissertação, pelo seu acompanhamento, e por durante este ano ter sido muito mais para mim do que um coorientador, deixo um grande obrigado.

Aos Professores que mais me marcaram durante o meu percurso na Universidade do Minho: Alberto José Proença, José Nuno Oliveira, Jorge Sousa Pinto, Luís Paulo Santos, Manuel Alcino Cunha, António Luís Sousa, António Ramires Fernandes, Rui Mendes, Maria João Frade, Luís Silva Dias e Ana Cordeiro.

*I would also like to thank Rupesh Nasre, who was my advisor during my internship at the University of Texas at Austin, for his excellent guidance during the internship, for the availability and willingness to help even after the internship, and for helping me shape the next few years of my professional life.*

À Família que criei nestes cinco anos na Universidade deixo um abraço sentido. Os nomes são em demasia para serem enumerados, mas eles sabem quem são, e sabem que guardam todos um lugar especial no meu coração. Tudo que eu hoje sou, devo a eles.

Quero também deixar um agradecimento especial à Catarina Azevedo, que me apoiou e acompanhou de perto durantes estes últimos meses difíceis. Sem ela, esta dissertação não teria sido possível.

Acima de tudo, quero agradecer aos meus pais, irmã e irmão, que, apesar de estarem muito longe de mim, sempre me apoiaram e motivaram, e a quem eu sempre quis fazer orgulhosos. Para o meu pequeno irmão espero ter sido uma inspiração.

Agradeço a todos os meus amigos e familiares pela compreensão dada à minha ausência durante este ano. Dedico esta tese aos meus Irmãos e familiares que partiram deste mundo.

Em memória do meu tio e padrinho

*In memory of my uncle and godfather*

José António Oliveira de Sousa

05/12/1963 - 03/07/2013

Em memória da minha avó

*In memory of my grandmother*

Olívia de Brito Mota

27/06/1936 - 22/01/2015

Em memória do meu tio

*In memory of my uncle*

Joaquim Pinto Penha Cerqueira

04/08/1959 - 06/10/2014

E, em memória dos meus Irmãos

*And, in memory of my Brothers*

João Pedro Gonçalves Rei de Abreu Vieira

17/10/1995 - 23/04/2014

Vasco Alexandre Ferreira Rodrigues

12/03/1995 - 23/04/2014

Nuno Miguel de Araújo Caldeira Ramalho

01/03/1993 - 23/04/2014

## Abstract

The Minimum Spanning Tree (MST) problem is a well known graph problem that plays an important role in various scientific fields. Graph algorithms are known to be irregular, namely due to the unpredictable memory access patterns and workload distribution among the graph vertices. These problems place additional challenges on novel parallel computing systems, namely those that resort to a mix of shared and distributed memory paradigms and to heterogeneous computing environment with attached computing accelerators, such as many-core CPU-chips and GPU devices. This dissertation addresses these challenges, as to develop efficient implementations of MST-solvers.

Sequential implementations of the three key MST algorithms - Borůvka's, Kruskal's and Prim's algorithm - have been implemented, tested and compared in terms of performance. Parallel implementations of Prim's and Borůvka's algorithms were also implemented and tested using the same suite of widely used road-network graphs. This comparative analysis included a first-hand comprehensive empirical comparison of several disclosed state of the art third-party CPU and GPU implementations of MST-solvers.

A parallel algorithmic variant of Borůvka's algorithm was devised, which exhibited speedups that outperformed all other tested competitors. The functionality of this variant is shown to be easily ported to other shared and distributed memory systems, including heterogeneous systems with GPU devices, without placing constraints on the graph size or hurting performance.

**Resumo**

# Versões sequenciais e paralelas eficientes de algoritmos de árvores de extensão mínima para CPUs e GPUs

O problema da árvore de extensão mínima (MST) é um problema de grafos muito conhecido e tem um papel importante em várias áreas científicas. Os algoritmos de grafos são conhecidos por serem irregulares, nomeadamente devido aos padrões de acesso à memória imprevisíveis, e à distribuição de trabalho pelos vértices do grafo. Estes problemas impoem um maior desafio nas novas plataformas de computação paralela, em particular aquelas que recorrem à mistura de memoria partilhada e distribuída, e a ambientes heterogéneos com aceleradores, como os *many-core* CPUs e dispositivos GPU. Esta dissertação aborda estes desafios, a fim de desenvolver implementações eficientes de MST-*solvers*.

Os três principais algoritmos de MST - Borůvka, Kruskal e Prim- foram implementados em sequencial, testadas e comparadas em termos de performance. Implementações paralelas dos algoritmos de Prim e Borůvka também foram implementadas e testadas usando o mesmo conjunto de grafos de estradas rodoviárias. Esta análise comparativa inclui uma comparação empírica de primeira-mão de vários MST-*solvers* de terceiros.

Foi desenvolvida uma variante algorítmica paralela do algoritmo de Borůvka, que mostrou ganhos de desempenho que superam a concorrência. É mostrado que a funcionalidade desta variante é facilmente portada, sem restringir o tamanho dos grafos ou prejudicar o desempenho, para outros sistemas de memória partilhada e distribuída, aonde se incluem sistemas heterogéneos com dispositivos GPU.

# Contents

# Acronyms

**API** Application Programming Interface

**BGL** Boost Graph Library

**CPU** Central Processing Unit

**CSR** Compressed Sparse Row

**CUDA** Compute Unified Device Architecture

**CUDPP** CUDA Data Parallel Primitives Library

**EREW** Exclusive Read Exclusive Write

**FLOPS/s** Floating Point Operations per Second

**FPGA** Field-programmable gate array

**FSB** Front-Side Bus

**GPGPU** General Purpose Graphics Processing Unit

**GPU** Graphics Processing Unit

**HPC** High Performance Computing

**HT** HyperTransport

**MGPU** ModernGPU

**MIC** Many Integrated Core

**MPI** Message Passing Interface

**MST** Minimum Spanning Tree

**NUMA** Non-Uniform Memory Access

**OpenCL** Open Computing Language

**PAPI** Performance Application Programming Interface

**PCIe** PCI express

**PRAM** Parallel Random Access Machine

**QPI** QuickPath Interconnect

**R-MAT** Recursive Matrix

**SIMD** Single Instruction Multiple Data

**SIMT** Single Instruction Multiple Thread

**SM** Streaming Multiprocessor

**SMP** Symmetric multiprocessing

**SPMD** Single Program Multiple Data

**STM** Software Transactional Memory

**TBB** Threading Building Blocks

**UMA** Uniform Memory Access

**VLSI** Very-Large-Scale Integration

# List of Figures

# List of Tables

# List of Algorithms

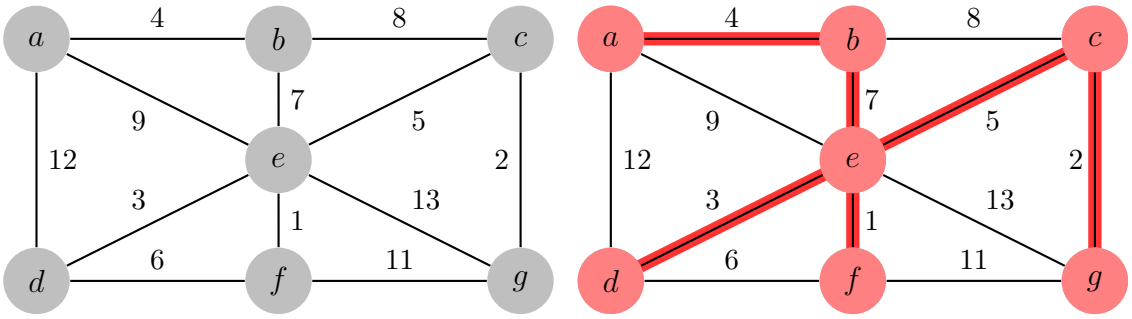
# Chapter 1

# Introduction

## 1.1 The Minimum Spanning Tree problem

Given a connected, undirected, weighted graph $G(V, E)$, where $V$ is the set of vertices and $E$ the set of edges, the Minimum Spanning Tree (MST) of $G$ is the sub-graph $T$ that spans all vertices of $G$ and has $|V| - 1$ edges, such that no cycles are formed and the total weight is minimized. If all edge weights are distinct then the graph's MST is unique, otherwise several MSTs are possible. An example graph and its corresponding MST is shown in Figure 1.1.

The MST problem is a well known graph problem and plays an important role in various scientific fields, such as in the Very-Large-Scale Integration (VLSI) circuit layout, in road, electrical and computer networks and in the approximation of the traveling salesman problem. Since 1926, when it was seen firsthand, the




(a) Example graph.

(b) MST of the example graph.

Figure 1.1: Example graph.

MST problem had undergone an extensive study on both sequential and parallel implementations.

This class of algorithms is known as irregular, i.e., both the workload and memory access patterns cannot be predicted at compile time, as they depend on the graph structure. Efforts are nowadays reserved to map these irregular algorithms onto parallel computing platforms.

## 1.2 Challenges and Motivation

Algorithms with regular memory accesses, such as matrix multiplication and linear system solvers, have undergone extensive studies. Highly efficient implementations have been developed, for various computing platforms and for a wide variety of these algorithms. This class of algorithms no longer poses a real challenge for researches.

Graphs are increasingly being used to represent social-, road- and computer-networks, but also in the medical field with genome sequencing and electroencephalographies. Algorithms that process graphs are much more complex and challenging to parallelize due to their inherent irregularity. As such, graph algorithms are becoming an attractive option not only to solve real-life problems, but also to benchmark the quality, performance, and productivity of the development frameworks to deploy, efficiently, portability of applications across distinct heterogeneous computing platforms.

With the advent of new computing platforms - such as Graphic Processing Units (GPUs) - and programming paradigms - such as shared and distributed memory - the algorithms no longer are the sole focus of research, when trying to extract performance from their implementations. Each paradigm has their own characteristics, and programmers are forced to understand the underlying architecture and technical details, if they are to develop efficient implementations.

Aside from figuring out ways to parallelize the algorithm, the partitioning of the workload and associated data must be taken into consideration. How are the parallel tasks and associated data distributed among the threads, processes and computing devices? How does one leverage this with inter-task communications, data locality in shared memory systems, and the data transfer overhead in distributed memory systems? When the application is efficiently deployed for one platform, is the same algorithm, and corresponding parallel code, efficiently distributed among the available computing resources, on other computing platforms, without further tuning? These are some of the challenges that are nowadays reserved for parallelizing algorithms.

## 1.3 Goals and Contributions

This dissertation aims to research and extend the state of the art of MST-solvers, while also assessing both sequential and parallel versions of MST-solvers, for multi-core Central Processing Unit (CPU)-chips and GPUs. The goal is to better understand MST-solver suitability to parallel architectures, and find ways to improve it. In particular, efficient parallel version for both CPUs and GPUs, should be implemented. As a final goal of this dissertation work, a new parallel version of an MST-solver is expected to be devised.

Throughout this dissertation, various existing and novel MST-solver algorithms are devised and implemented, from which a generic Borůvka's algorithm stands out for its portability across computing devices and high performance. A compilation of the existing state of the art of MST-solvers, and a comprehensive comparison with these solvers and the solvers developed in the context of this dissertation, are included.

The work on the generic Borůvka's algorithm resulted in a scientific paper that was submitted and accepted at an top tier conference: the 23rd Parallel, Distributed and Network-based Processing (PDP 2015). The accepted paper can be found in Appendix B.

## 1.4 Dissertation Structure

The rest of this dissertation is presented with the following chapter structure:

### Ch.2 Heterogeneous Computing Platforms

This chapter overviews the state of the art of computing platforms, focusing on multi-core CPU-chips in both shared and distributed memory programming models and on many-core devices, with a special emphasis on GPU devices. It also presents some relevant libraries used on these programming platforms, namely those used in the context of this dissertation.

## Ch.3 Minimum Spanning Tree Solvers

This chapter introduces the key algorithms and implementations that have been developed so far to compute the MST. It presents and details three seminal MST-solvers: Borůvka's, Kruskal's and Prim's algorithms. The last section overviews the literature on existing sequential and parallel implementations.

## Ch.4 Parallel Algorithms and Implementations

This chapter presents the new parallel MST-solver implementations developed in the context of this dissertation, and introduces the data structures used to represent graphs, and which are used for representation in the parallel implementations. It also presents parallelization and implementations details of the multiple instance Prim's algorithm, as described in Chapter 3, addressing collision resolution strategies, and partitioning approaches. Lastly, a parallel, platform-independent, algorithmic variant of Borůvka's algorithm is presented, addressing the key issues to a platform independent variant.

## Ch.5 Performance Evaluation

This chapter describes the experimental environment, including a detailed description of the computing platforms, external libraries used and graphs that were used for testing purposes. It presents the experimental study conducted in the context of this dissertation, including a comparative analysis of all the implementations that were developed in the context of this dissertation, and a critical analysis between the best implementation developed in this dissertation, with third-party MST-solvers

## Ch.6 Conclusions & Future Work

This chapter concludes the dissertation, presenting an overview of the results obtained with the developed implementations and suggesting lines of research to further investigate the more relevant outcome of this work.

# Chapter 2

# Heterogeneous Computing Platforms

*This chapter overviews the state of the art of computing platforms, focusing on multi-core CPU-chips in both shared and distributed memory programming models and on many-core devices, with a special emphasis on GPU devices. It also presents some relevant libraries used on these programming platforms, namely those used in the context of this dissertation.*

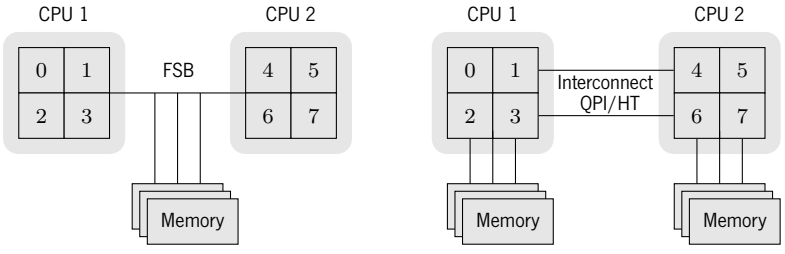
With scientific problems of the most diverse areas expanding their research, more people are resorting to computer applications to solve their problems. However, the time needed to solve these problems put a strain on the size of the data and the complexity of the application. To meet the consumers' demand, CPU chips became faster and more complex. However, the ever growing search for more computing power eventually faced power consumption and heat dissipation problems. The strategy adopted by major hardware companies was to simplify the cores, and pack multiple cores on a single chip. Previously, programmers did not need to worry with parallelism. Existing implicit parallelism such as pipeline supersclarity and out-of-order execution is handled by the compiler and the chip hardware, and multi-threaded execution involved only the parallel execution of different processes. To take advantage of the multi-core processor devices, a new programming model surfaced that allows explicit parallel code to be written within the application.

Algorithms that exhibited regular memory access patterns became a popular target for vectorization (Single Instruction Multiple Data (SIMD)), allowing the same instruction to be applied to a set of data. To address the growing amount of data, devices that target massive data parallelism were developed. The general term associated to these devices is *accelerators*.

The term *heterogeneous computing platform* has been thrown around frequently with different meanings. In this dissertation we will assume that a heterogeneous computing platform is a single computing node with one or more multi-core CPU-chips with attached accelerator devices. In turn, multiple heterogeneous nodes can be interconnected to form a computer cluster.

## 2.1 Multi-core architectures

Up until recently, most of the focus on parallel algorithms has been targeted to multi-core CPU-chips. The shared memory model, where different threads share the same memory space, has been the primary target for parallel implementations. In the shared memory model, the Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA) memory paradigms can be distinguished. UMA is more common, all threads have an uniform access time to the memory banks, while NUMA refers to a memory layout where memory access latency depends on the location of the running thread and the memory region it is trying to access. The NUMA paradigm is currently present in single computing nodes with more than one CPU-chip, and the non-uniform latency is due to the fact that current CPU-chips include on-chip the memory controllers. As a result, different memory banks are connected to distinct CPUs. Cross memory bank access is possible due to proprietary interconnects between the CPU-chips, such as the Intel QuickPath Interconnect (QPI) or AMD HyperTransport (HT). This interconnect replaces the Front-Side Bus (FSB) used in UMA systems. Time is being spent to figure out ways to take advantage of these systems, and how thread and memory affinity affects the application performance. However, performance scalability is limited to the number of cores and CPUs a single computing node can harbor. Figure 2.1 illustrates the difference between UMA and NUMA systems.




(a) UMA System. (b) NUMA System.

Figure 2.1: Multi-core architectures.

In the distributed memory model, each process works in their own private memory space. The private memory spaces is due to either the programming model employed, or by being physically separated, which is

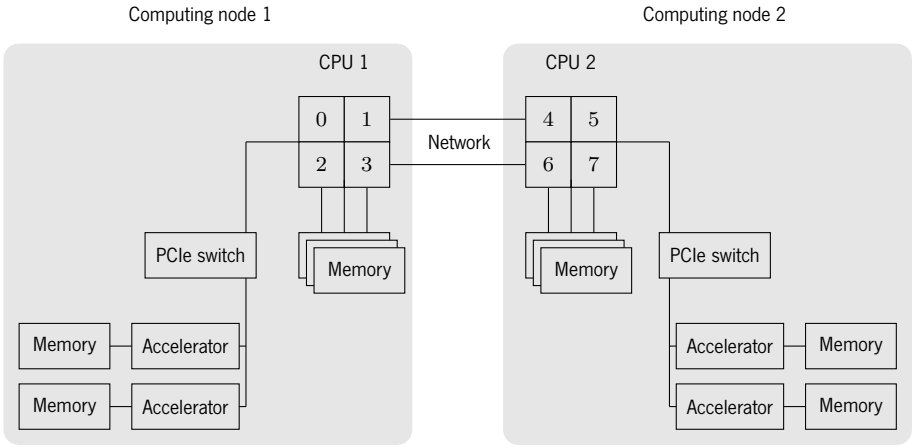



Figure 2.2: Distributed memory system.

the case when working with accelerators or in a multi-node environment. Note that each of these computing nodes are heterogeneous platforms, i.e., they can have multiple CPUs with attached accelerators. The sharing of data is achieved by inter-process communication, which has an added overhead. Scalability is limited to the the amount of communication. Figure 2.2 shows a distributed system with two UMA computing nodes, each with two accelerators. The interfaces that connects the CPU with its memory, the accelerator with its memory, PCI express (PCIe), which connects the CPU to the accelerator, and the network, which interconnects the computing nodes, all operate at different transfer rates. The various memory regions are not shared, which can create bottlenecks if communication is not dealt with properly.

### 2.1.1 Libraries

In the context of multi-core architectures, several libraries have been developed to aid the development of efficient parallel applications. The libraries used in this dissertation are overviewed below.

#### OpenMP

OpenMP is a high level Application Programming Interface (API) for developing parallel application on shared memory systems, offering a simple way to parallelize applications, using compiler directives for work sharing constructs, synchronization and reduction to a single variable. Furthermore, various schedulers are available to assign loop iterations to each thread.

### Intel Threading Building Blocks (TBB)

Intel Threading Building Blocks (TBB) is a task based, C++ template library and is somewhat similar to OpenMP, in the sense that it offers various work sharing constructs such as for, reduce, and scan. However, TBB is more low level, in comparison to OpenMP, as parallelization is achieved using template classes, and offering concurrent containers, pipelines, parallel sort and mutexes. Scheduling is done at a task level, which are dynamically allocated to each core, and work stealing is used to ensure efficient use of the available resources.

### pThreads

When more control is need in the development of parallel applications, one must resort to a more low level library. pThreads is the POSIX standard for threads, and offers full control on the creation and joining of threads. Furthermore, it also implements low level synchronization primitives such as mutexes, condition variables and barriers, leaving the responsibility of thread management entirely to the user. When using pThreads, complications, such as deadlocks, may arise. However, it also offers much more control, which is often necessary to parallelize complex algorithms.

### OpenMPI

OpenMPI is an open source Message Passing Interface (MPI) library for developing distributed memory applications and offers an API with a wide range of optimized primitives, such as barriers, broadcast, gather, scatter, scan and reduce, all of which implement various variants, such as synchronous, asynchronous and variable length message sizes. OpenMPI works over ethernet protocols, but also over proprietary interconnect networks such as Myrinet and Infiniband, while transparently working over shared memory when the processes are running on the same computing node.

Together with OpenMP, this can be used to develop hybrid applications to run on clusters, using OpenMPI to parallelize the application among computing nodes, and, in turn, OpenMP to parallelize each process, effectively obtaining a hybrid shared and distributed memory application.

#### Performance Application Programming Interface (PAPI)

The Performance Application Programming Interface (PAPI) is a portable interface that allows the user to access and collect low level performance counters. The available counters depends on the underlying CPU. Using PAPI, an application's miss rates for all cache levels, load balance, floating point operations per second, operational intensity, estimate bandwidth used, and many other metrics can be measured. This allows one to identify and address bottlenecks, and make grounded assertions on the limited performance of applications.

## 2.2 Many-core architectures

The many-core architecture devices, or *accelerators*, was the industry response to massive data parallel algorithms. These accelerators are off-chip devices with tens to thousands of cores and their own memory space. Active memory transfer between the main memory and the accelerator's memory is possible. However, the latency is high and transfer times may have a significant impact on performance, so these must be taken into account when building efficient applications for these devices. Furthermore, the private memory space is rather small compared to the main memory, limiting the possible input size and forcing frequent memory transfers. Often programmers stray away from accelerators, namely when their algorithms do not map well onto them and performance lacks due to memory transfers, completely overtaking execution time.

Several devices are currently available and popular, the most notable being the General Purpose Graphics Processing Unit (GPGPU), the Field-programmable gate array (FPGA) and the Intel Many Integrated Core (MIC) family, currently represented by the Xeon Phi.

### 2.2.1 Graphic Processing Units (GPUs)

Previously, GPUs were devices dedicated to processing and creating images to be shown on a graphics display. Since 2006, the GPU is becoming a more general computing device, targeted to massive data parallelism. These new GPGPUs have hundreds, to thousands of very simple cores. The new programming model employed forced programmers to adapt their algorithms and understand the underlying architecture.

Two well known frameworks for programming on the GPUs is the Open Computing Language (OpenCL) and NVIDIA's Compute Unified Device Architecture (CUDA), the largest open and proprietary frameworks, respectively. In this dissertation, the latter is used.

At the hardware level, the GPU is composed by several Streaming Multiprocessors (SMs) (changed in to SMX with the Kepler architecture), which, in turn, are composed by several execution units, known as CUDA cores. In the Kepler architecture, each SMX has 192 CUDA cores, each basically containing one single precision floating point execution unit, and an integer execution unit. Furthermore, the SMXs also contain additional functional units: double precision floating point units and load/store units

A CUDA kernel is a piece of code that every thread is going to execute, as a parallel task, on the GPU. The threads are grouped into blocks of user-specified size. The collection of all blocks of a kernel is called a grid Each block is assigned to an SM and broken down into warps of 32 threads for execution. All threads inside the same warp execute the same instruction at a given time. From this, it is clear that the CUDA model uses both the SIMD and Single Instruction Multiple Thread (SIMT) models at warp level, and Single Program Multiple Data (SPMD) at kernel level. The number of blocks, threads and warps that can reside in each SM is limited and depends on the specific architecture.

Each thread has access to a set of registers that is managed by the hardware, and an L1 and L2 cache with 64 KB and 1.5 MB, respectively, for the Kepler architecture. All threads in the same block also have access to a fast shared memory region that is managed by the user. The total size available for L1 cache and shared memory is distributed among them, but can be configured by the user in predefined combination of sizes. Lastly, all blocks have access to the global memory and texture memory. Although the global memory is much larger in size, it is relatively slow compared to the shared memory, while the texture memory is read-only, optimized for 2D spatial locality.

Unlike on the CPU, where long memory accesses are hidden by the use of a large memory hierarchy, the GPU overcomes the long memory accesses by scheduling other warps while data is accessed from memory. This is possible because context switching on the GPU is very cheap compared to the CPU.

Figure 2.3 shows the architecture of a GPU from the Kepler family, which has a massive amount of CUDA cores. Each new architecture does not just increase in the number of CUDA cores and SMs, but also offer new functionalities, such as support for double precision floating point operations, full IEEE754 compliant, introduced in 2010 with the Fermi architecture, Dynamic Parallelism, i.e., nested kernel calls, with the Kepler architecture in 2012, Unified Memory in 2014, with the Maxwell architecture, abstracting


Figure 2.3: Representation of NVIDIA GK110 Kepler (courtesy of NVIDIA)

the view of host and device memory for the user, and stacked DRAM, which is planned for the future with the Volta architecture.

### Libraries

With the wide spread use of GPUs as computing devices, new opportunities opened up regarding parallel algorithms and data structures. CUDA Data Parallel Primitives Library (CUDPP)[1] and ModernGPU (MGPU)[2] are examples of libraries that implement a series of data-parallel primitives, such as scan, reduce, bulk insert and remove, sort, and their segmented variants, which are used as building blocks for many parallel algorithms for the GPU. Furthermore, many other libraries are available, such as cuBLAS, CUDA implementation of the BLAS library, cuFFT, for Fast Fourier transforms, and Thrust, which implements various templates similar to the C++ Standard Template Library, while also offering some data-parallel primitives.

While previously the kernels implemented by these libraries could only be launched by the CPU, with the support for Dynamic Parallelism the GPU, threads can now launch kernels without the intervention of the CPU, which opens open a wide variety for new features.

[1] http://cudpp.github.io/
[2] http://nvlabs.github.io/moderngpu/

# Chapter 3

# Minimum Spanning Tree Solvers

*This chapter introduces the key algorithms and implementations that have been developed so far to compute the MST. It presents and details three seminal MST-solvers: Borůvka's, Kruskal's and Prim's algorithms. The last section overviews the literature on existing sequential and parallel implementations.*

## 3.1 Sequential Minimum Spanning Tree Algorithms

Given a connected, undirected, weighted graph $G(V, E)$, where $V$ is the set of vertices and $E$ the set of edges, the MST of $G$ is the sub-graph $T$ that spans all vertices of $G$ and has $|V| - 1$ edges, such that no cycles are formed and the total weight is minimized. If all edge weights are distinct then the graph's MST is unique, otherwise several MSTs are possible. An example graph and its corresponding MST is shown in Figure 1.1.

The first known algorithm to solve the MST problem is given by Borůvka [Borůvka, 1926a]. In this paper a long and mathematically complex definition of the algorithm is given. Although being the oldest algorithm, it is also the most interesting from a High Performance Computing (HPC) point of view: being inherently parallel, it has been the focus of substantial research and parallel implementations. The original paper was published before the appearance of graph theory, hence the complex definition of the proposed algorithm. In [Borůvka, 1926b] Borůvka gives a more contemporary definition[1].

[1] Both papers are written in Czech, please refer to [Nešetřil et al., 2001] for translations

In contrast, Kruskal's algorithm [Kruskal, 1956] is inherently sequential, and thus is not seen as often in the literature regarding parallel implementations. In addition to the original algorithm, Kruskal presents another algorithm that can be seen as a dual to the original, and is later rediscovered as mentioned in [Graham and Hell, 1985].

The third well known MST algorithm is usually credited to Prim [Prim, 1957], even though the same algorithm was discovered decades before [Jarník, 1930]. Both Prim's and Kruskal's algorithm are considered to be a particular case of a more generic algorithm that Kruskal himself presents, which will be discussed in Section 3.1.2.

Sequential implementations of these three algorithms are quite straightforward, and most of the analysis done is either on the data structure that stores the graph, a different interpretation of the algorithm using intermediate data structures to store relevant information, or both. In [Moret and Shapiro, 1994] an extensive empirical analysis is done using various sorting algorithms and priority queues for Kruskal's and Prim's, respectively. The authors state that asymptotic worst case analysis is inadequate since the input graphs need to have a very uncommon structure to actually hit the worst case bound, and in practice, it is not likely that this would happen often. Furthermore, the authors state that asymptotically worse algorithms algorithms perform better in practice.

In a parallel setting, Prim's algorithm is more suited for parallelization when compared to Kruskal's algorithm. However, the inherent sequential growing behavior of Prim's algorithm imposes limitations that require overly complex procedures, which require heavy use of fine-grained synchronization that substantially reduces the possible speedups, as detailed in Section 3.2.

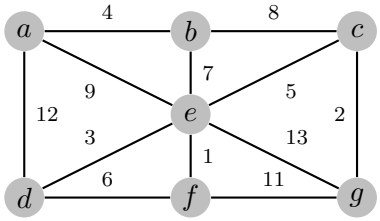
### 3.1.1 Borůvka's Algorithm

In the 1920s, Borůvka was asked to find the most economic solution to construct an electrical power grid. The proposed algorithm, described in [Borůvka, 1926a], 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

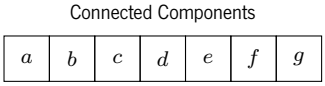
of Borůvka's algorithm can be found in Figure 3.1.

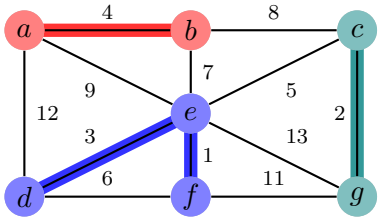
Efficient implementations can be obtained using a disjoint-set structure. A disjoint-set allows to keep track of different elements (vertices) across non-overlapping subsets (connected components). The pseudo-code for this algorithm is shown in Algorithm 1.

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, as shown in Figure 3.1. 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. This behavior is shown in Figure 3.1c.

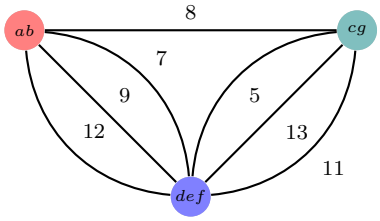



(a) Initial graph.

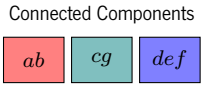


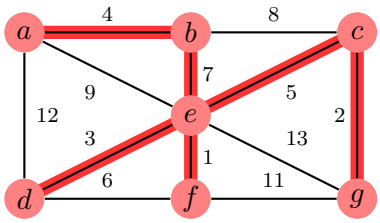




(b) Borůvka components iteration 1.




(c) Borůvka contraction iteration 1.




(d) Borůvka components iteration 2.


(e) Borůvka contraction iteration 2.

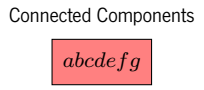



Figure 3.1: Borůvka's algorithm.

**Algorithm 1** Borůvka's algorithm

**Input:** Undirected, connected and weighted graph $G(V, E)$
**Output:** $T$ is MST of $G$

$T := G(V, \varnothing)$

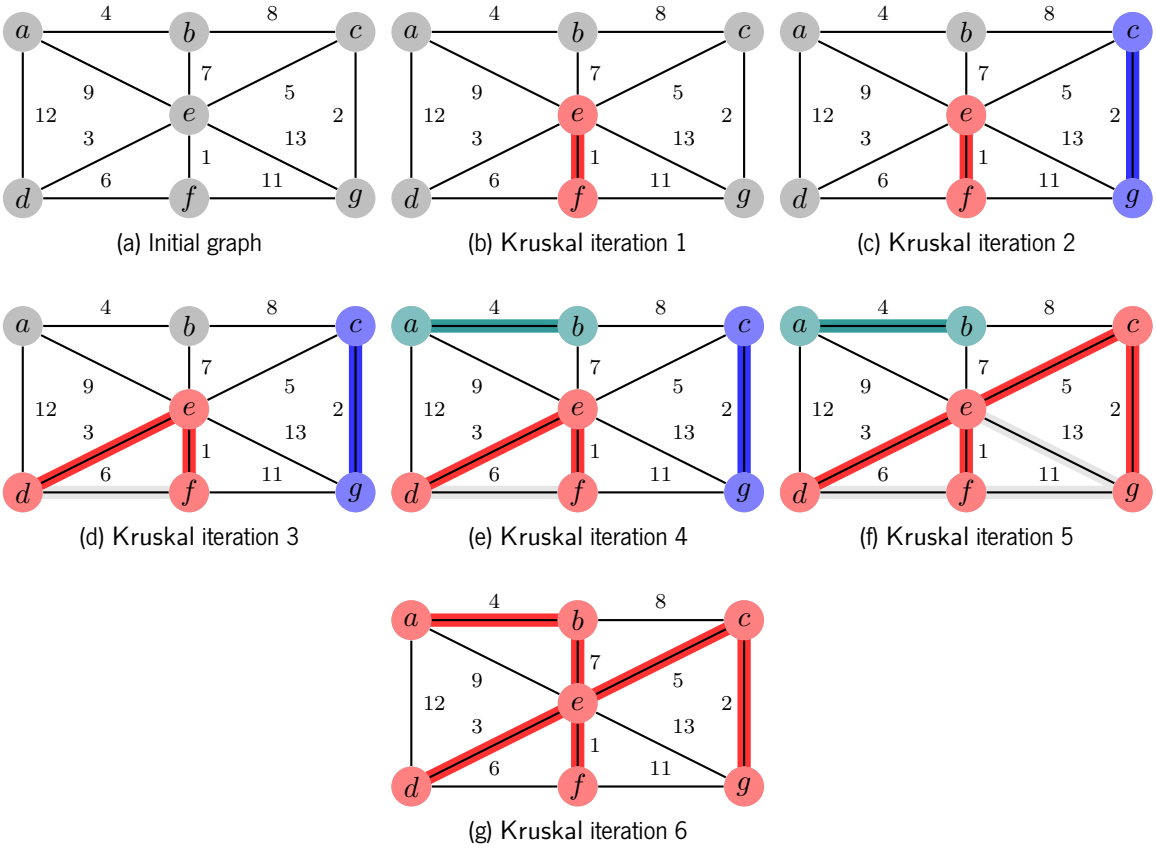
**while** $T$ has more than one component **do**
  $S := \varnothing$
  **for each** connected component $C \in T$ **do**
    **let** $e(u, v) \in E_G$ and $\notin E_T$ be the lightest edge such that $u \in C$ and $v \notin C$
    $S := S \cup \{(u, v)\}$
  $T := T \cup S$

### 3.1.2 **Kruskal**'s Algorithm

While Borůvka's algorithm is inherently parallel, Kruskal's algorithm, shown in Algorithm 2, is sequential: all vertices are initialized as a component with a single element. The list of edges is then sorted by increasing weight and processed one by one. If an edge connects two different components, the edge is added to the MST and the components are merged, otherwise the edge is discarded. The algorithm processes all edges of $G$. However, if the graph is connected, the algorithm can stop as soon as one component remains or $|V| - 1$ edges are added to the MST. A graphical description of Kruskal's algorithm can be found in Figure 3.2.

In his original paper, Kruskal presented three constructions of his algorithm, which are shown next:

> *Construction A. Perform the following step as many times as possible: among the edges of $G$ not yet chosen, choose the shortest edge which does not form any loops with those edges already chosen. Clearly the set of edges eventually chosen must form a spanning tree of $G$, and in fact it forms a shortest spanning tree.*
>
> *Construction B. Let $V$ be an arbitrary but fixed (nonempty) subset of the vertices of $G$. Then perform the following step as many times as possible: Among the edges of $G$ which are not yet chosen but which are connected either to a vertex of $V$ or to an edge already chosen, pick the shortest edge which does not form any loops with the edges already chosen. Clearly the set of edges eventually chosen forms a spanning tree of $G$, and in fact it forms a shortest spanning tree. In case $V$ is the set of all vertices of $G$, then Construction B reduces to Construction A.*
>
> *Construction A'. This method is in some sense dual to A. Perform the following step as many times as possible: Among the edges not yet chosen, choose the longest edge whose removal will not disconnect them. Clearly the set of edges not eventually chosen forms a spanning tree of $G$, and in fact it forms a shortest spanning tree.*




(a) Initial graph (b) Kruskal iteration 1 (c) Kruskal iteration 2

(d) Kruskal iteration 3 (e) Kruskal iteration 4 (f) Kruskal iteration 5

(g) Kruskal iteration 6

Figure 3.2: Kruskal's algorithm.

**Algorithm 2** Kruskal's algorithm

**Input:** Undirected, connected and weighted graph $G(V, E)$
**Output:** $T$ is MST of $G$

1: $T := \varnothing$
2: $E' := E_G$ sorted by ascending weight
3:
4: **for each** vertex $v \in V$ **do**
5: $\quad makeSet(v)$
6:
7: **for each** $e(v, w) \in E'$ **do**
8: $\quad setV := findSet(v)$
9: $\quad setW := findSet(w)$
10: $\quad$ **if** $setV \neq setW$ **then**
11: $\quad\quad T := T \cup \{(e, v)\}$
12: $\quad\quad unionSet(setV, setW)$

It is clear that construction A is in fact Kruskal's algorithm. Furthermore, Kruskal points out that construction A is derived from construction B when $V$ is the set of all vertices of $G$. On the other hand, when $V$ is the set that only contains a single vertex of $G$, construction B reduces down to Prim's algorithm. The third construction is also known as the reverse-delete algorithm.

The reverse-delete algorithm considers the graph as a single connected component, and sorts the list of edges by decreasing weight. The edges are then processed one by one: if removing the the edge would disconnect the graph (i.e., the connected component would be split in two), then the edge belongs to the MST, otherwise it can be discarded. Instead of needing to keep track of connected components, this algorithm resorts to graph connectivity checking.

Parallel implementations of Kruskal's algorithm focus on:

- The sorting of edges, either by parallelization, or by partially sorting the edges, leaving heavier edges to be processed later [Rostrup et al., 2011];
- Divide and conquer approach [Loncar et al., 2013], assigning consecutive vertices to each process and having each one computing the MST of their assigned vertices.

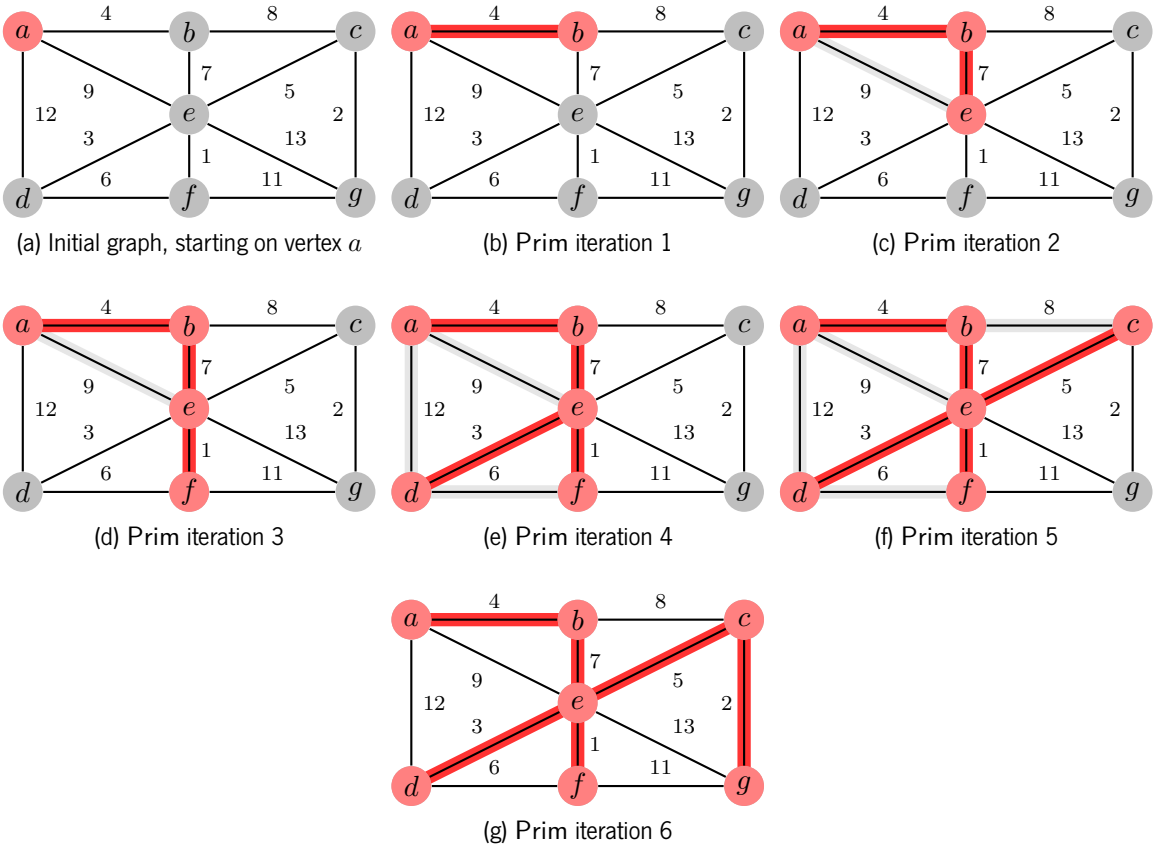
### 3.1.3 **Prim**'s Algorithm

Prim's algorithm starts on a random vertex and grows the tree from there, adding, on each step, the lightest edge that connects a vertex inside the tree to a vertex outside to the MST. A graphical description of Prim's algorithm can be found in Figure 3.3.

The original algorithm, described in Algorithm 3, has a large cost of finding, on each iteration, the lightest edge that connects a vertex inside the tree to a vertex outside the tree. This cost can be reduced by using a priority queue to keep track of the candidate edges. This is the algorithm proposed by [Fredman and Tarjan, 1987] using a Fibonacci heap. The heap needs to implement the following operations:

- `insert(k, v)` - inserts the key `k` with the value `v` into the heap
- `decreasekey(k, v)` - decreases the existing value of `k` to `v` and updates the internals of the heap
- `deletemin` - returns the smallest value and deletes it from the heap

An auxiliary array is needed to store the minimum distance from the growing tree, to each vertex. If this distance is zero, then the vertex is already part of the MST. The array is initialized with a distance of infinity




(a) Initial graph, starting on vertex $a$

(b) Prim iteration 1

(c) Prim iteration 2

(d) Prim iteration 3

(e) Prim iteration 4

(f) Prim iteration 5

(g) Prim iteration 6

Figure 3.3: Prim's algorithm.

(with exception of the starting vertex), to denote that all vertices are currently unreachable. The algorithm starts on a random vertex and either inserts each edge into the queue, or decreases it, if the end-point of the edge was previously unreachable or a new minimum weight is found, respectively. The heap internally reorders the elements to keep weights sorted in increasing order. The next edge to be added is the edge that is stored on the top of the heap. This process is repeated until all vertices are visited.

[Moret and Shapiro, 1994] perform an empirical analysis using various heaps, including Fibonacci heaps, but obtain better performance using pairing heaps. A drawback of this algorithm is that if the graph is dense, the queue can grow quickly, and the computational cost to keep the heap ordered becomes too high. The pseudo-code for Prim's algorithm using a priority queue can be found in Figure 4. Parallel implementations of Prim's algorithm focus on (described in further detail in Section 4.3):

- Parallelizing the searching and updating of the lightest edge [Wang et al., 2011, Loncar et al., 2013, Mariano et al., 2013];

**Algorithm 3** Prim's algorithm

**Input:** Undirected, connected and weighted graph $G(V, E)$
**Output:** $T(V, E)$ is MST of $G$

$T := \varnothing$
let $w \in V_G$ be a random starting vertex
$V_T := V_T \cup w$

while $V_T \neq V_G$ do
    let $e(u, v) \in E_G$ be the lightest edge such that $u \in V_T$ and $v \notin V_T$
    $T := T \cup \{(u, v)\}$

**Algorithm 4** Prim's algorithm using a priority queue

**Input:** Undirected, connected and weighted graph $G(V, E)$
**Output:** $T(V, E)$ is MST of $G$

$T := \varnothing$
let $w \in V_G$ be a random starting vertex
$V_T = V_T \cup w$

for each vertex $v \in V_G$ do
    $dist[v] := \infty$
$dist[w] := 0$

for each edge $e(w, t) \in E_G$ and $t \in V_G$ and $t \notin V_T$ do
    if $weight(e) < dist[t]$ then
        if $dist[t] == \infty$ then
            $insert(t, weight(e))$
        else
            $decrease(t, weight(e))$
        $dist[t] := weight(e)$

while queue not empty do
    $e(v, w) := deletemin()$

    if $dist[w] \neq 0$ then
        $T := T \cup (v, w)$
        $dist[w] := 0$
        for each edge $e(w, t) \in E_G$ and $t \in V_G$ and $t \notin V_T$ do
            if $weight(e) < dist[t]$ then
                if $dist[t] == \infty$ then
                    $insert(t, weight(e))$
                else
                    $decrease(t, weight(e))$
                $dist[t] := weight(e)$

- Running multiple instances of Prim with different starting vertices [Bader and Cong, 2004, Kang and Bader, 2009, Setia et al., 2009].

## 3.2 State of the Art of Sequential and Parallel Implementations

The state of the art of both sequential and parallel implementations of the various MST-solvers is quite extensive. This next section overviews, organized by architecture, the literature and existing parallel algorithms. Previous literature compilations of sequential algorithms can be found in [Graham and Hell, 1985, Mareš, 2008]. Some of the implementations presented in this section are used in the comparative analysis performed in Section 5.5, as such, these implementations are assigned an unique name for future reference.

### 3.2.1 SMP Systems and Multi-Core CPU-Chips

A parallel Borůvka implementation for shared memory Symmetric multiprocessing (SMP) systems was presented in [Bader and Cong, 2004]. The authors implemented Borůvka's graph contraction variant, and experimented with several adjacency list representations. They also present a new data structure, the flexible adjacency list, that is more suited for graph contraction on the CPU. Furthermore, a new parallel algorithm is presented as a combination of Prim's and Borůvka's. This algorithm grows multiple concurrent instances of Prim's algorithm from different starting vertices. When one Prim instance collides with 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 collisions, thus incurring additional, excessive overhead.

The same authors presented in [Cong and Bader, 2005] an algorithmic variant of Borůvka's 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 cannot be used for comparisons purposes, and one with *pThread* mutexes.

In contrast to [Bader and Cong, 2004], [Setia et al., 2009] handles the collisions by merging the pair of collided threads, having one starting from a new initial vertex and the other continuing the work of the

merged MSTs, making it a pure Prim implementation. This is achieved using POSIX signals. Although, the approach seems interesting, little experimentation is done, and the data set size is in the order of $10^3$, while several data sets, whose sizes are several orders of magnitude higher, can be found in practice.

A similar approach is presented in [Kang and Bader, 2009], implemented on a Software Transactional Memory (STM) system. However, instead of relying on signals or locking mechanisms, the underlying STM system is expected to handle all data races. In addition to the high overhead incurred by STM systems, they are recent and not widely available.

The *Galois* framework, presented in [Pingali et al., 2011], 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.

### 3.2.2 Distributed Memory Systems

A parallel Kruskal implementation on a distributed memory system using MPI is described in [Loncar et al., 2013]. The authors use an adjacency matrix to represent the graph, allowing them to easily assign consecutive sets of vertices to each process. Each process computes the local MST, and then each pair of processes merges their local MSTs by applying Kruskal on the union of the two MSTs. This process is repeated until only one process remains. Furthermore, the authors also present a distributed memory algorithm for Prim that uses the same partitioning strategy to assign vertices to processes. In this approach, each process selects the minimum weight edge that connects a vertex that is assigned to the process, to the MST, followed by a global min reduction to the root process, that selects and adds the global minimum edge to the MST. This process is repeated until all vertices are in the MST. Due to the high memory usage of the adjacency matrix representation, the analysis is limited to graphs with up to $10^5$ vertices. Nevertheless, further analysis is done using different graph densities with up to $10^9$ edges.

### 3.2.3 GPUs

While reviewing the literature, the first parallel implementations of MST-solvers, for the GPU, appear in [Harish et al., 2009] and [Vineet et al., 2009], implementing Borůvka's color and explicit graph contraction approaches, respectively. Both of these implementations are implemented using the CUDA programming

model.

[Harish et al., 2009], from here on referred to as *Harish2009*, is based on the Exclusive Read Exclusive Write (EREW) Parallel Random Access Machine (PRAM) algorithm presented in [Johnson and Metaxas, 1992], using color propagation to identify connected components and explicitly removing cycles. Speedups were obtained in comparison to a CPU version presented in the paper.

The algorithm presented in [Vineet et al., 2009] is implemented as a stack of parallel primitives using CUDPP. While the authors report it 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. Furthermore, it was not possible to reproduce these results, and these restrictions limit the comparison against all the other implementations. As such, this implementation is not included in the analysis performed in Section 5.5.

A parallel variant of Kruskal's algorithm was proposed in [Rostrup et al., 2011], focusing on the memory usage on the GPU. The variant proposed 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, it was not possible to obtain access to this implementation. Furthermore, since their most efficient implementation also employs a bit-packing mechanism similar to [Vineet et al., 2009] it is not include it the critical analysis, for the same reason described above.

Another GPU implementation of a parallel variant of Prim's algorithm was presented in [Wang et al., 2011]. 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)[2]. However, beside the fact that it was not possible to obtain this implementation, 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 [Mariano et al., 2013], but comparisons with these specialized devices are out of the scope of this paper.

A similar implementation to *Harish2009* was presented in [Nasre et al., 2013], from here on referred

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

to as *Nasre2013*. This implementation uses disjoint sets to denote connected components. The authors obtained no speedup in comparison with *Galois*.

### 3.2.4 Conclusions

It stands out from the literature that there is a clear trade-off between the effort that is required to implement the parallel algorithms, and the actual performance obtained. The best performing algorithms are considered hard to implement. Furthermore, several implementations introduce limitations in order to boost performance (e.g. reserved bits in [Vineet et al., 2009, Rostrup et al., 2011]). An exception is the implementation presented in [Kang and Bader, 2009], where super-linear speedups on a STM system are achieved with little effort.

Due to its importance, several parallel algorithms were devised to work on PRAM abstract shared memory machines [Chong et al., 2001, Johnson and Metaxas, 1992, Pettie and Ramachandran, 2002]. However, this dissertation focuses on the empirical assessment of MST-solvers and, thereby, these algorithms are out of the scope. Yet, some of the implementations presented in the papers cited in this section are based on PRAM algorithmic descriptions.

# Chapter 4

# Parallel Algorithms and Implementations

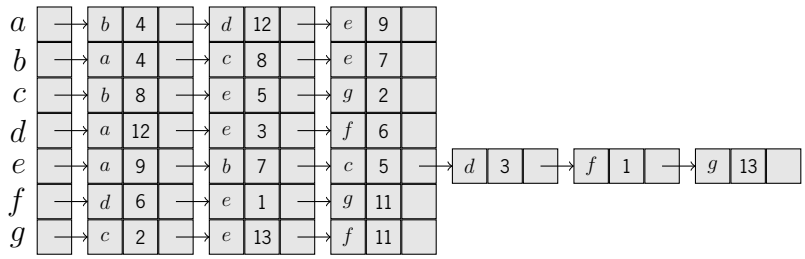
*This chapter presents the new parallel MST-solver implementations developed in the context of this dissertation, and introduces the data structures used to represent graphs, and which are used for representation in the parallel implementations. It also presents parallelization and implementations details of the multiple instance Prim's algorithm, as described in Chapter 3, addressing collision resolution strategies, and partitioning approaches. Lastly, a parallel, platform-independent, algorithmic variant of Borůvka's algorithm is presented, addressing the key issues to a platform independent variant.*

## 4.1 Graph representation

The choice of data structures for graph representation is very important, as it has a direct impact on performance and can influence algorithm and implementation decisions.

The most common representation seen in the literature is the adjacency list. The graph is represented as an array where each vertex is mapped to an index. Each entry of the array points to a list of destination vertex and weight pairs, representing the edges. This list is usually implemented as a linked-list, allowing the easy manipulation of the graph structure. Alternatively, by using arrays, a more cache friendly approach can be obtained, in detriment of easy of manipulation.

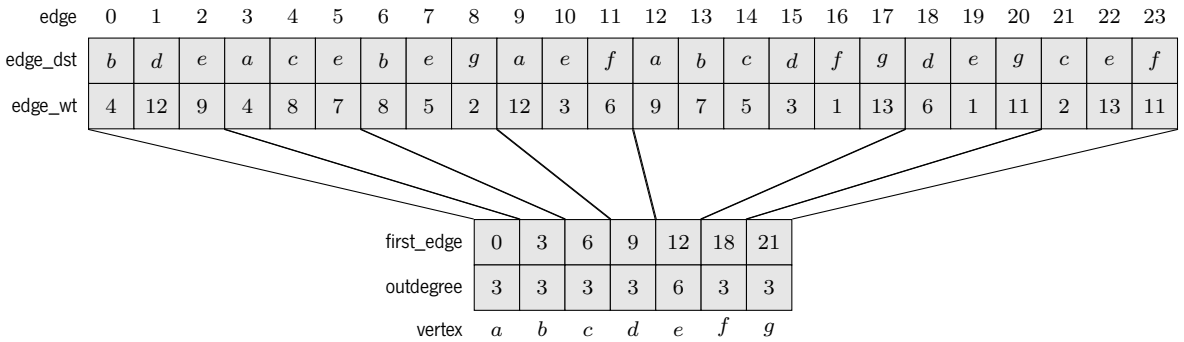
[Bader and Cong, 2004] introduces an extension to the adjacency list representation specifically for Borůvka's graph contraction algorithm on the CPU: each index can point to multiple lists of incidents




(a) Adjacency list representation.

| | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ | $g$ |
|---|---|---|---|---|---|---|---|
| $a$ | 0 | 4 | 0 | 12 | 9 | 0 | 0 |
| $b$ | 4 | 0 | 8 | 0 | 7 | 0 | 0 |
| $c$ | 0 | 8 | 0 | 0 | 5 | 0 | 2 |
| $d$ | 12 | 0 | 0 | 0 | 3 | 6 | 0 |
| $e$ | 9 | 7 | 5 | 3 | 0 | 1 | 13 |
| $f$ | 0 | 0 | 0 | 6 | 1 | 0 | 11 |
| $g$ | 0 | 0 | 2 | 0 | 13 | 11 | 0 |

(b) Adjacency matrix representation.




(c) CSR representation.

Figure 4.1: Representations of the example graph in Figure 1.1.

edges, making it much easier to merge vertice's edge lists.

The traditional adjacency matrix representation is still widely used. The graph is represented by a matrix of $|V| \times |V|$, where a value greater than zero in a position $(i, j)$ represents an edge from $i$ to $j$ with the specified weight. The downside of the adjacency matrix is the large memory requirement. Furthermore, the adjacency matrix stores the information of non-existing edges, making it cumbersome to iterate on the edges, especially for sparse graphs.

A compromise between the adjacency list and adjacency matrix is the CSR format and is often seen in the literature as the representation of choice for graph algorithms on the GPU. In this format the graph is represented by four arrays:

- `edge_dst` - an array of size $|E|$, which maps each edge to its destination;
- `edge_wt` - 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. In the case of the adjacency matrix, the matrix will be symmetric. Figure 4.1 shows the representations for these three data structures of the example graph in Figure 1.1.

In algorithms where the graph structure might change, the usage of adjacency lists is the more attractive approach, 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 4.4 a technique is presented that allows the CSR format to be used in Borůvka's graph contraction variant.

In this dissertation, the CSR format will be used to represent the graph, as a way to ensure fairness in the comparison of CPU and GPU implementations, and to increase any potential portability of cross-platform algorithms.

## 4.2 Lock-Free Adjacency List

In some of the algorithms that will be presented in this chapter, there is the need for an efficient way to build the MST. With Prim's algorithm, one usually resorts an father array: given an arbitrary vertex `i`, `father[i]` is the predecessor of `i`. The predecessor of the starting vertex is usually the null vertex id, such as `-1`. Such an array, in a multi-threaded setting, shared among all threads, is not possible, as each vertex can only have a single father, and multiple threads could change the father of a given vertex, resulting in lost edges.

Since the adjacency list is more adequate for building graphs, it can be here instead, as long as adjustments are made to make it capable of safe concurrent access. In essence, an adjacency list is an array of linked lists. Implementing a concurrent adjacency list without locks is not possible. However, the only concurrent operation that is needed for this particular adjacency list is insertion at the head of the list, which can be achieved using a single atomic operation.

GNU GCC provides a built-in atomic exchange function, as shown in Listing 4.1, which writes the contents of `*val` into `*ptr`, and to original value of `*ptr` into `*ret`.

```
void __atomic_exchange(type *ptr, type *val, type *ret, int memmodel);
```

Listing 4.1: GNU built-in atomic exchange function

Using this function, the insertion of an edge to the head of a linked list can be easily achieved, as shown in Listing 4.2. The `Edge` structure is a linked list of edges, and each entry has 3 fields: destination vertex id, edge weight, and a pointer to the next element in the list. The `Adj_List*` is an array of the type described above. The operation described in Listing 4.2 sets the head of the linked list of the specified `source` vertex to the newly created edge, while setting the next element of this new edge to the old head of the linked list.

```
void insert_edge(Adj_List *list, unsigned src, unsigned dst, unsigned wt){
  Edge *new_edge = (Edge*)malloc(sizeof(Edge));
  new_edge->dst = dst;
  new_edge->wt = wt;

  // Performs the equivalent to this, atomically:
  // new_edge->next = list[src];
  // list[src] = new_edge;
  __atomic_exchange(&(list[src]), &new_edge, &(new_edge->next), __ATOMIC_SEQ_CST);
}
```

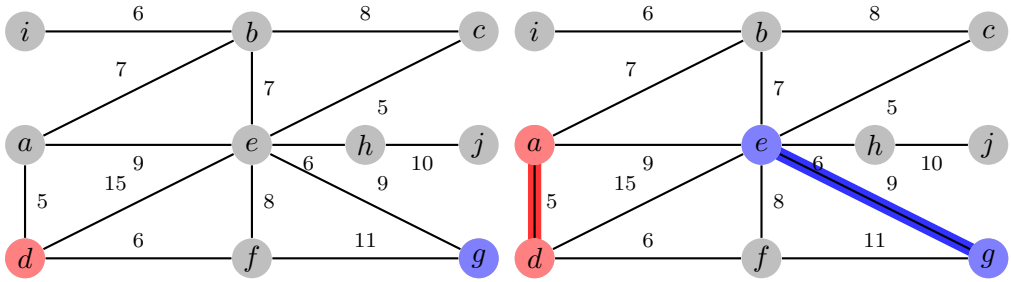
Listing 4.2: Atomic insertion of an edge into the linked list.

## 4.3 Multiple Instance **Prim**

Subsection 3.1.3 introduced two distinct approaches for parallelizing Prim's algorithm. In this section, a more in-depth analysis is presented on the multiple instance Prim approach, exploring opportunities for parallelism and identifying limitations. Furthermore, a novel approach, which relies on graph partitioning to assign vertices to each thread, is presented.

Prim's algorithm behaves sequentially from its starting vertex, making it possible to run multiple instances, in parallel, of Prim growing from different starting vertices [Bader and Cong, 2004, Kang and Bader, 2009, Setia et al., 2009], as illustrated in Figure 4.2. As long as the growing trees never touch one other (otherwise known as collisions), they can proceed without interruption. It is state in [Bader and Cong, 2004], that if an instance of Prim is started on every vertex, the algorithm behaves like Borůvka.

In order to maintain the correctness of the MST, whenever a Prim instance tries to add a vertex that belongs to another instance, a collision will occur, and must be handled accordingly. In the provided example, the blue tree will try to add vertex $b$ to its own tree. However, this vertex is marked as belonging to the red tree. The way collisions are treated has a major impact on performance and code complexity.




(a) Initial graph, starting on vertices $d$ and $g$

(b) Prim iteration 1

(c) Prim iteration 2

(d) Prim iteration 3

(e) Prim iteration 4 with collision

Figure 4.2: Example of multi instanced Prim

### 4.3.1 Collision Treatment

This strategy detects and resolves any collision as soon as they are found, not allowing the algorithm to continue without before solving all collisions. The technique presented is seen in the following three publications, each one addressing this problem in a distinct way:

- [Bader and Cong, 2004] - the thread stops growing its tree if it collides with another active tree or with a tree that has already stopped growing, and restarts from a different vertex. When all vertices are visited, it identifies connected components and runs one iteration of Borůvka, considering that each connected component is a (super-)vertex;

- [Kang and Bader, 2009] - the thread steals MST information from the other thread. Relies on the semantics of the underlying transactional memory system to avoid data races;

- [Setia et al., 2009] - the thread steals MST information from the other thread. Relies on `POSIX` signals and fine-grained locking.

In [Kang and Bader, 2009] and [Setia et al., 2009], after the collision is resolved, one thread will continue the work of the union of the two collisioned MSTs.

Assuming the graph in Figure 4.2 extends beyond the represented vertices, there would be a number of major problems depending on the collision resolution strategy:

- [Bader and Cong, 2004] - the algorithm considers an unvisited vertex, but colored by a thread, a collision. Each thread colors the neighbors of the current vertex with an unique color, as a way to lock them to the thread. If one of these vertices is colored by another thread, the algorithm will consider it a collision. This vertex might not lead to a true collision. Yet, to avoid possible race conditions, and due to the lock-free nature of the algorithm, these situations are treated as collisions. This problem becomes more frequent with the decreasing of diameter of the graph, resulting in the Prim stage of the algorithm doing no useful work;

- [Kang and Bader, 2009, Setia et al., 2009] - if the graph is dense, the algorithm will spend most of its time stealing and merging MST data, and the progress between each collision would be very small;

- [Bader and Cong, 2004, Kang and Bader, 2009, Setia et al., 2009] - the newly selected starting vertex might be in the proximity of an already growing MST. If this hold , chances are that a new collision will quickly take place, thereby reducing the amount of work done.

Even if the reported situations do not occur frequently, the algorithm must be able to handle these situations, adding additional complexity to the implementation. An exception to this is the algorithm presented in [Kang and Bader, 2009], as the underlying transactional memory system handles all data races. However, these systems are only now becoming common on mainstream, convenience processors.

### 4.3.2 Partitioned

This novel approach is based on the premise that, if the graph is partitioned beforehand, multiple instances of a MST-solver can be run on each partition, allowing them to compute a MST without interacting with the other partitions. Instead of treating collisions as they occur, they can be postponed by ignoring edges that cross partitions. This sort of strategy is often seen in the parallelization of finite volume methods, such as heat transfer in solids, where the input problem is broken down into smaller problems, and the heat flow of the boundary of each partitions is computed at regular intervals during execution, or at the end.

It should be noted that there is a conceptual difference between scheduling and partitioning. While scheduling assigns work-units to threads or processes, taking into account factors such as load-balancing, partitioning selects the work-units based on some criteria. In this case, the work-units are vertices, and all vertices that belong to a partition should form a connected component, as to minimize interference with other partitions, and the partitions should not overlap. Furthermore, the partitions could benefit from having the following:

- The number of vertices is equally distributed;
- The number of edges is equally distributed;
- The number of edges that cross partitions should be minimized.

The algorithm receives as input the graph and an array that maps each vertex to its partition. Each thread or process is assigned one partition and grows the MST from one, random of its vertices. The MSTs computed for each partition, from here on referred to as *local* MSTs, do not connect one other, since the edges that connect them were previously ignored. However, some of these edges could be part of the optimal, or *global*, MST, and since these edges were not considered, other edges that, are not supposed to be part of the *global* MST, were added to the *local* MSTs. Figure 4.3 illustrates this: the red and blue areas represent the different partitions, while the teal edges represent edges that cross partitions. It is clear that the edge $(a, b)$ should not be part of the MST, while the edges $(a, c)$ and $(b, c)$ should.

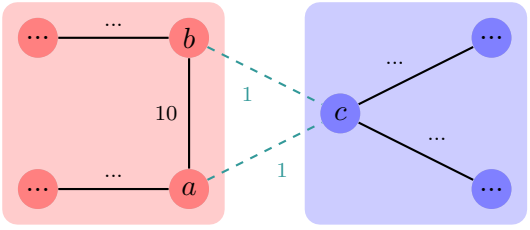



Figure 4.3: Simple example of an incorrectly added edge.

In order to obtain the *global* MST, the following strategies can be considered. Each of these strategies runs multiple instances of sequential Prim's algorithm, in parallel.

### Union of Local MSTs

In [Loncar et al., 2013], the *global* MST is obtained by running another MST-solver on the union of the *local* MSTs. The same strategy can be applied here: another MST-solver can be run on the union of the *local* MSTs, but this time including the edges that cross partitions. However, running a full-fledged MST-solver on a graph that almost is a MST, just to add and remove a couple of edges, might be a waste of computational effort. Therefore, the need to find a intelligent way to compute the *global* MST is crucial. Taking into account the properties of the union of the *local* MSTs (is a sparse graph and an approximation of the *global* MST), a targeted strategy can be devised to obtain the *global* MST.

The main reason for finding a targeted algorithm for this specific problem is reducing the number of vertices and edges that are processed. Since most of the edges added to the *local* MSTs are in fact part of the *global* MST, there is no need to process them again.

### Reverse Delete Algorithm

Section 3.1.2 described Kruskal's algorithm and its dual, the reverse delete algorithm. This particular algorithm may be useful in the following situation: the graph is a close approximation to the *global* MST and the number of edges that need to be removed is small, reducing the amount of computation done by the algorithm.

This strategy is similar to the previous one. The *local* MSTs need to be joined together with the edges that cross partitions. But in this particular case, the MST-solver selected to compute the *global* MST is the reverse delete algorithm. The downside of this strategy is the need to check graph connectivity on each iteration. Graph connectivity checking could be reduced down to a breadth-first search or a similar algorithm, searching for a path between the vertices that have been disconnected by the removed edge. However, this solution may not be adequate, as the number of edges processed is only small if the edges that need to be removed have a high weight, which may not be the case.

Parallelizing the reverse delete algorithm is hard, and graph connectivity checking is a problem on its own. Therefore, this strategy is out of the scope for this dissertation.

### Boundary Marked Vertices

In order to devise a targeted algorithm to handle the incorrectly added edges, some strategy must be adopted to identify potential vertices or edges that need to be looked at.

Given the graph and its partitions, edges cross partitions can easily be identified, as well as boundary vertices, i.e., vertices that have at least one edge that connects to a vertex belonging to another partition. Since the boundary vertices and its incident edges should be the only ones affected by the partitioned approach, we only need to process these specific vertices after the union of the *local* MSTs.

However, analysis shows that this strategy would only work correctly for the situations described in Figure 4.3, where incorrectly added edges belong to a boundary vertex. Incorrectly added edges that lie in the interior of the partition would never be removed, resulting in an non-minimum, yet a close approximation, spanning tree. Figure 4.4 show this problem: it is clear that the edges $(a, c)$, $(a, b)$ and $(b, h)$ should be part of the MST while the edge $(f, g)$ should not.

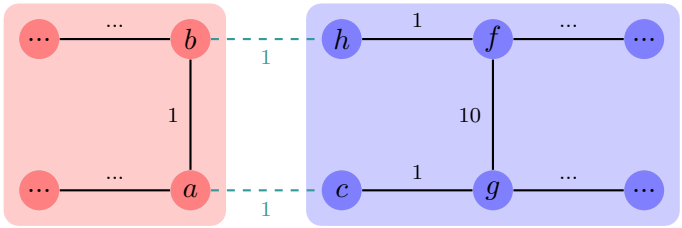



Figure 4.4: Complex example of an incorrectly added edge.

The magnitude of these problems increase with the number of partitions, along with the capability to mentally visualize and understand the various different situations that may lead to an incorrectly built MST. For example, Figure 4.5 illustrates a situation were cycles could be introduced. Each partition has no knowledge of which vertices have been visited by the other partitions. In this case, the edges $(b, c)$, $(d, e)$ and $(a, f)$ would be added, forming a cycle.

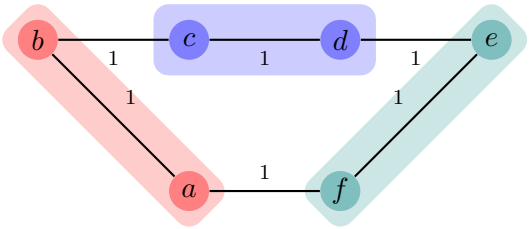



Figure 4.5: Cycle creation with 3 partitions.

An implementation that returns a non-minimum spanning tree may be useful, as long as the relaxation in quality results in a significant performance increase.

#### Connected Components Identification

Whatever strategy that is used to mark possible incorrectly added edges, it all boils down to the fact that each Prim instance grows in a sequential manner, and the MST's validity is compromised as soon as it tries to add an edge whose endpoint belongs to another Prim instance. In other words, whenever a collision occurs, the current Prim instance cannot continue to grow.

Instead, the Prim instance can stop the current tree, and start from a new vertex inside its partition, with the added condition that it needs to stop whenever it encounters one of its own stopped trees. After all the vertices are visited, a series of connected components can be identified inside each partition. One only needs to run another MST-solver, such as Kruskal's or Borůvka's algorithm, which work with components, to obtain the *global* MST.

Essentially, without the partitioning, this strategy is similar to the one presented in [Bader and Cong, 2004]. Initial empirical analysis shows that constantly restarting Prim instances is causing a large overhead with respect to initialization. Further tuning would be necessary to obtain competitive results. Therefore, this approach will not be included in the analysis presented in Chapter 5.

### 4.3.3 Conclusions

The algorithms presented in this section present overly complex collision resolution strategies, high usage of fine-grained locking and atomic operations, and limited speedups. Furthermore, the partitioned approach requires close attention to particular situations that can lead to an incorrectly built MST.

Another important aspect to note is that these algorithms can not be ported to the GPU, as each sequential Prim's algorithm uses, internally, a priority queue, which is inefficient on the GPU. Alternatively, Prim's algorithm could be implemented using the searching and updating of the lightest edge approach, as explained in Section 3.1.3. However, it would not be possible to run multiple instances of Prim's algorithm, and [Wang et al., 2011] reports limited speedups using this approach.

## 4.4 A Generic **Borůvka**'s Algorithm

The previous section presented an approach for parallelizing Prim's algorithm. However, complications arise due to the heavy need for fine-grained synchronization and overly complex collision resolution strategies, and the reduced opportunity for parallelism, for the multiple instanced Prim approach and the GPU implementation, respectively.

In this section, a novel parallel variant of Borůvka's algorithm is presented, focusing on its genericness, i.e., its portability across computing devices, and performance. This variant is based on Borůvka's graph contraction algorithm (shown in Subsection 3.1.1).

### 4.4.1 Algorithm

This algorithmic variant comprises a series of simple kernels, as shown in Algorithm 5[1]. 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. The algorithms referred in this section can all be found in Appendix 7. This sequence of kernels is repeated until only one super-vertex remains:

**Algorithm 5** 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
```

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

### Find minimum edge per vertex

The algorithm starts 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 the `vertex_minedge` array. Figure 4.6 shows the selected edges for each vertex of the example graph in the initial state (Figure 3.1a). Algorithm 7 describes in further detail this kernel.

This kernel does not resort to a segmented parallel primitive, as it would not be possible 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.

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

Figure 4.6: Find minimum edge per vertex.

### 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, as described in Algorithm 8. E.g. in Figure 4.7, one of the mirrored edges is the pair $(e, g)$ and $(g, e)$, since $g > e$, the edge $(e, g)$, selected by vertex $e$, is removed while the edge $(g, e)$, selected by vertex $g$, is maintained. The edges that remain in the `vertex_minedge` array are marked to be part of the MST.

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

Figure 4.7: Remove mirrored edges.

### 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, as shown in Algorithm 9. 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 4.8: $e$ sets its successor to himself since it has no selected edge, while $d$ and $f$ selected the edges $(d, e)$ and $(f, e)$, respectively, set their successor both to $e$. In this particular case, no propagation takes place as the successors converge immediately. Algorithm 10 is repeated for all vertices until it converges.

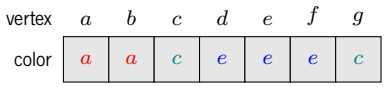



Figure 4.8: Initiaize and propagate colors.

### 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`, as shown in Algorithm 11. An *exclusive prefix sum* is then computed on the flag array, assigning new vertex ids for the contracted graph. In Figure 4.9, $a$, $c$ and $e$ are the representative vertices. After computing the prefix sum, these vertices are assigned the new vertex ids $0$, $1$ and $2$, respectively.

The prefix sum ensures 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 preexisting proximity between a vertex id and the id of its neighbors, all of which improve locality.

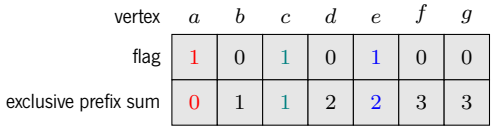



Figure 4.9: Create new vertex ids.

### 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

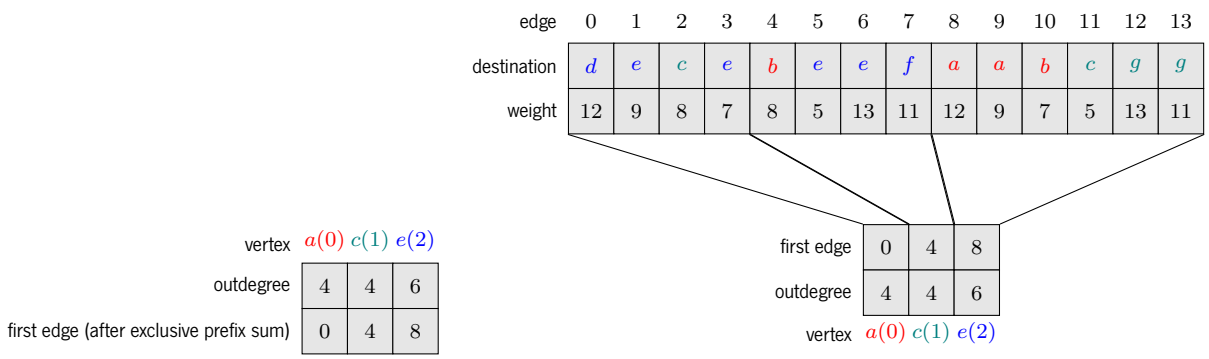
kernel that counts the number of edges that cross the component for each vertex, and adds it to `outdegree` array of its corresponding super-vertex. Since multiple vertices may belong to the same super-vertex, an atomic function for the operations on the `outdegree` array has to be used, as detailed in Algorithm 12.

An exclusive prefix sum is then computed 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. A copy of the `first_edge` array is made first, 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. Algorithm 13 further details this kernel.

Intra-component edges are discarded by comparing the colors of the two end-points of each edge. However, duplicate edges between pairs of super-vertices are not removed, as the benefit of doing this does not outweigh the incurred computational cost. Figure 4.10a shows the number of neighbors for each super-vertex. E.g. $a(0)$ has four: $(a, d)$, $(a, e)$, $(b, e)$ and $(b, c)$. Even though the first three connect the same super-vertices, they are still added.

Figure 4.10b 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.




(a) Count and assign edges.

(b) Inserting edges and contracted graph.

Figure 4.10: Count, assign, and insert new edges.

### 4.4.2 Implementation Details

#### Shared Memory and GPU Implementations

For the CPU shared memory and the GPU approach, a topological approach is adopted, having each thread operate on a set of vertices (one at a time), for the CPU, and one vertex per thread, on the GPU.

#### Distributed Memory Implementation

As a proof of concept, this algorithm was modified to work on distributed memory systems. The kernels that compose Algorithm 5 remain the same. The adjustments necessary are between kernel calls, since all processes need to have a global vision of the graph. Each process is assigned a set of contiguous vertex ids (the same as the shared memory approach), and after each kernel, broadcast all necessary information to all other processes. Algorithm 6 describes in further detail all the required communication steps.

Further workaround was necessary. In particular, since each process will propagate the colors of the vertices assigned to it, there might be processes that converge while others do not. As such, an integer variable is used to check for convergence, which is set to 0 when it converges, and to 1 when it does not. By performing a reduction, using the sum operator, on this variable, it is easy to determine whether or not all processes have converged, by checking if the value is greater than 0.

Furthermore, the prefix sum also need to be adjusted: the first process computes the prefix sum of its vertices, and then sends the last value of the prefix sum to the subsequent process, which receives it and adds it to all values from its own prefix-sum. This process is repeated in a pipeline fashion for all processes.

There are two advantages to this approach:

- A hybrid shared/distributed memory implementation is easily achievable, since the kernels remain unchanged, the parallelization remains the same as in the shared memory approach.
- Having a working distributed memory approach and a GPU implementation opens up the possibility for an heterogeneous implementation. This can be done by applying the same communication steps as done in the distributed memory approach, but instead of inter-process communication, host-device communication is done[2].

[2] Due to the lack of time, it was not possible to implement this approach

**Algorithm 6** Parallel, distributed memory, 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:      Allgather vertex_minedge
4:      Remove mirrored edges
5:      Allgather vertex_minedge
6:      Initialize colors
7:      while not converged do
8:          Allgather color
9:          Propagate colors
10:         Allreduce converged
11:     Create new vertex ids
12:     Allgather exclusive prefix sum
13:     Count new edges
14:     Allreduce(+) outdegree
15:     Assign edge segments to new vertices
16:     Allgather first_edge
17:     Insert new edges
18:     Allgather edge_dst
19:     Allgather edge_wt
```

# Chapter 5

# Performance Evaluation

*This chapter describes the experimental environment, including a detailed description of the computing platforms, external libraries used and graphs that were used for testing purposes. It presents the experimental study conducted in the context of this dissertation, including a comparative analysis of all the implementations that were developed in the context of this dissertation, and a critical analysis between the best implementation developed in this dissertation, with third-party MST-solvers*

## 5.1 Experimental Environment

All tests were carried out on a dual-socket NUMA system, specified in Table 5.1. The CPU codes were compiled with `g++` 4.8.2, and GPU code with `nvcc` 5.5, both with `-O3` flag. 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 the measurements, the *k*-best measurement scheme was used, 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 each another, are selected. The best of the 3 is then used in the results.

For the CPU implementation, OpenMP is used, assigning chunks of vertices to each thread. For the use of parallel primitives, Intel TBB 4.2 was used, since OpenMP does not have have an exclusive prefix sum primitive.

| | CPU | GPU |
|---|---|---|
| #Devices | 2 | 1 |
| Manufacturer | Intel | NVIDIA |
| Model | E5-2670 v2 | K20m |
| Launch date | Q3'13 | Q1'13 |
| $\mu$Arch | 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 5.1: System characteristics.

For the GPU implementation, MGPU[1] 1.1 is used for the parallel primitives, and the implementation is extended, for a small performance boost, with the usage of texture memory, wherein the four arrays that represent the graph at a given iteration are stored.

The partitioned Prim based implementations require a pre-computed partition. The METIS application, presented in [Karypis and Kumar, 1995], includes a graph partitioner. The underlying algorithm is well established and has been focus of large amount of research by the scientific community.

## 5.2 Data sets

To evaluate the various implementations, a set of test graphs is needed. Ideally, it should include graphs that are used in the literature, in order to have a basis for comparison. Furthermore, the graphs should have various structures and sizes. It should also be noted that, with the computing power and memory capacity that is available today there is not much interest in running the algorithms with graphs that are small to the point of not being able to take advantage of the available hardware.

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

| 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 5.2: Road-network graphs used in benchmarks.

### 5.2.1 Real-life graphs

The graph collection supplied provided by 9th DIMACS Implementation Challenge[2] include sparse graphs that depict the United States road network. These graphs are seen frequently in the recent literature. Furthermore, the OpenStreetMap's[3] Portuguese road-network, provided by Geofabrik[4] is included. The used input graphs are described in Table 5.2.

### 5.2.2 Synthetic graphs

Synthetic graphs are computer generated graphs using a variety tools. These tools allow custom parameters to be set with the purpose of allowing graphs with different characteristics and sizes to be generated. A well known tool is GTgraph [Bader and Madduri, 2006] which includes three graph generators, the most notable being the Recursive Matrix (R-MAT) graph generator.

R-MAT graphs are small-world graphs that follow a power-law degree distribution, mimicking social and computer networks where each pair of nodes are separated by a relatively small number of hops. Furthermore, the graphs exhibit a community structure by usage of several vertices, known as *hubs*, that have a

[2] http://www.dis.uniroma1.it/~challenge9/
[3] http://www.openstreetmap.org/
[4] http://download.geofabrik.de/europe/portugal.html

very high degree [Chakrabarti et al., 2004].

GTgraph also includes two random graph generators and the SSCA#2 generator. Note that the graphs need to be connected and undirected. Therefore, adjustments to the graphs to guarantee this requirement are necessary.

For the sake of completeness, these graphs are mentioned here. However, they are not used in the comparative analysis. The reason for this is two-fold. First, the benchmarks reported in the Sections 5.4 and 5.5 can easily be replicated. Second, graph generators usually work with various parameters that can be tweaked to favor specific implementations, whose performance fluctuates for different graph properties.

## 5.3 Description of the Implementations

This section gives a short description on all the implementations that were used in the comparative analysis.

### 5.3.1 Dissertation Implementations

The following sequential implementations, as described in Chapter 3 were used:

- *prim_seq* - Prim's algorithm as described in Algorithm 4, using a Pairing heap form the Boost library;
- *kruskal_seq* - Kruskal's algorithm, using a disjoin-set implementation from Boost;
- *boruvka_seq* - Borůvka's algorithm using an own implementation of a disjoint-set to denote components.

The following parallel Prim based implementations, as described in Sections 4.3, were used:

- *tm_mst_pt* - with collision treatment, based on [Kang and Bader, 2009], but without stealing MST information from other threads;
- *prim_omp_union* - partitioned, using the union of *local* MSTs to obtain the *global* MSTs;
- *prim_omp_bmv* - partitioned, using the boundary marked vertices approach to solve collisions.

The following implementations are the OpenMP, GPU and OpenMPI implementations of the generic Borůvka algorithm presented in Section 4.4:

- *GenBoruvka OMP*
- *GenBoruvka GPU*
- *GenBoruvka MPI* - hybrid OpenMPI and OpenMP implementation.

### 5.3.2 Third-Party Implementations

The following, third-party, sequential implementation was used:

- *boost_kruskal* - Kruskal's algorithm, provided by Boost Graph Library[5] version 1.55.0.

The following, third-party, parallel implementations, which are described in Section 3.2, were used.

- *Cong2005*
- *Galois*
- *Nasre2009*
- *Harish2009*

## 5.4 Experimental Results

Figure 5.1 shows the execution times of sequential implementations for all road-network graphs. *prim_seq* clearly outperforms all others, while *boost_kruskal* is significantly slower than all other implementations. As mentioned in Chapter 3, the performance of the sequential implementations varies with the input graph and data structures used. In this particular case, *prim_seq* is faster, but depending on the graph structure and density, it could be easily outperformed.

Figure 5.2 shows scalability for all parallel implementations developed in the context of this dissertation, in comparison to the best sequential algorithm shown in Figure 5.1, for the largest road-network graph in the data set (USA graph). *prim_bmv* is the only Prim based implementation that outperforms the sequential implementation, albeit with limited speedups. However, recall, as pointed out in Section 4.3.2,

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

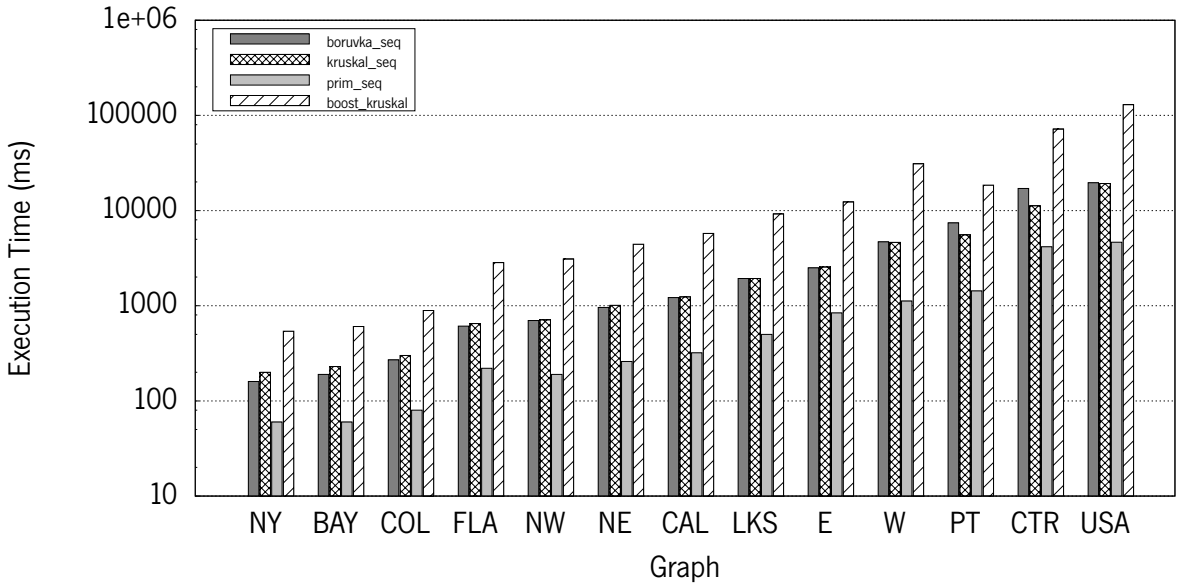



Figure 5.1: Measured execution times of the sequential implementations for all road-network graphs.

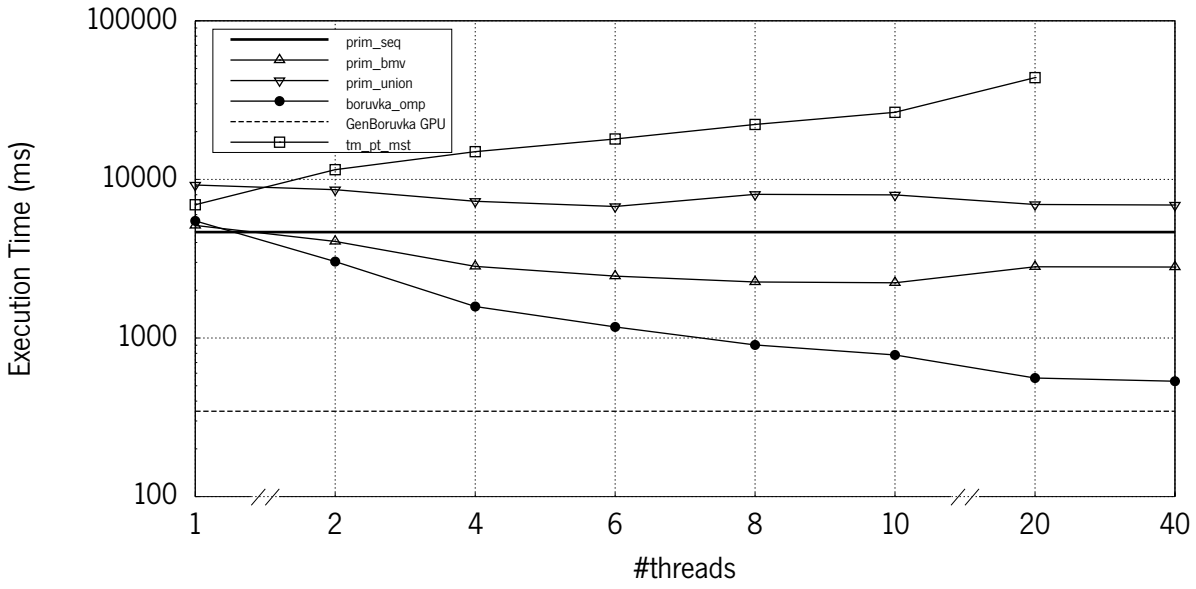



Figure 5.2: Scalability for USA road-network graph.

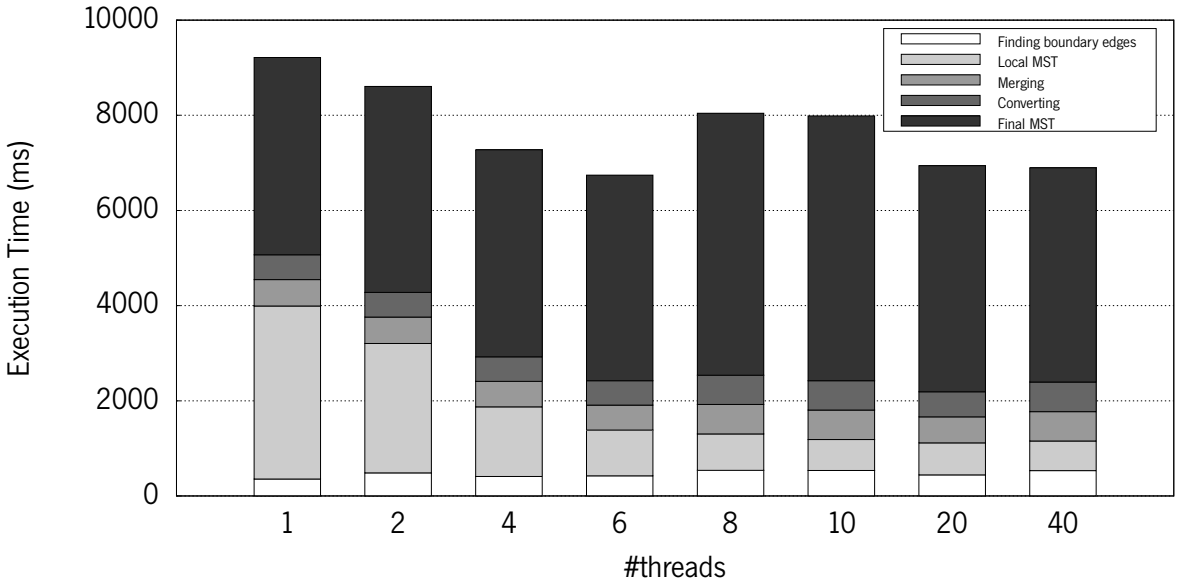



Figure 5.3: Execution time breakdown for *prim_union*.

that this particular implementation produces a non-minimum spanning tree. The remaining Prim based implementations all perform worse than the sequential Prim algorithm.

Figure 5.3 shows the execution time breakdown for the *prim_union* implementation. It is clear that the performance of this implementation is limited to the performance of the final MST stage. As suggested in Section 4.3.2, obtaining the *global* MST by re-visiting the edges added to the *local* MSTs and the boundary edges, would be inefficient, since most of the edges are in fact already part of the *global* MST.

The *tm_mst_pt* implementation has massive slowdowns, this is attributed to the large load imbalance caused by the collisions: threads that collide need to wait until all other threads have stopped before proceeding, the more time these threads need to stop, the larger the load imbalance will be. Since the graph is extremely sparse, the number of collisions is rather low, and remains low in executions with more threads. Figure 5.4 shows the percentage of vertices each thread processes between each collision, for execution with two threads on the USA graph. Just with two threads there is a very large amount of load imbalance, e.g., in the third collision, one of the threads processes over 50% of the vertices while the other thread barely 10%. This is due to the fact that the threads stop growing their MST when they collide with another thread. To avoid this, an extremely complex MST stealing would be needed, in order to allow a thread to stop and steal the MST that is being grown by another thread. However, for the same reasons as the strategy described in Section 4.3.2, the cost of constantly initializing Prim instances would be extremely high.

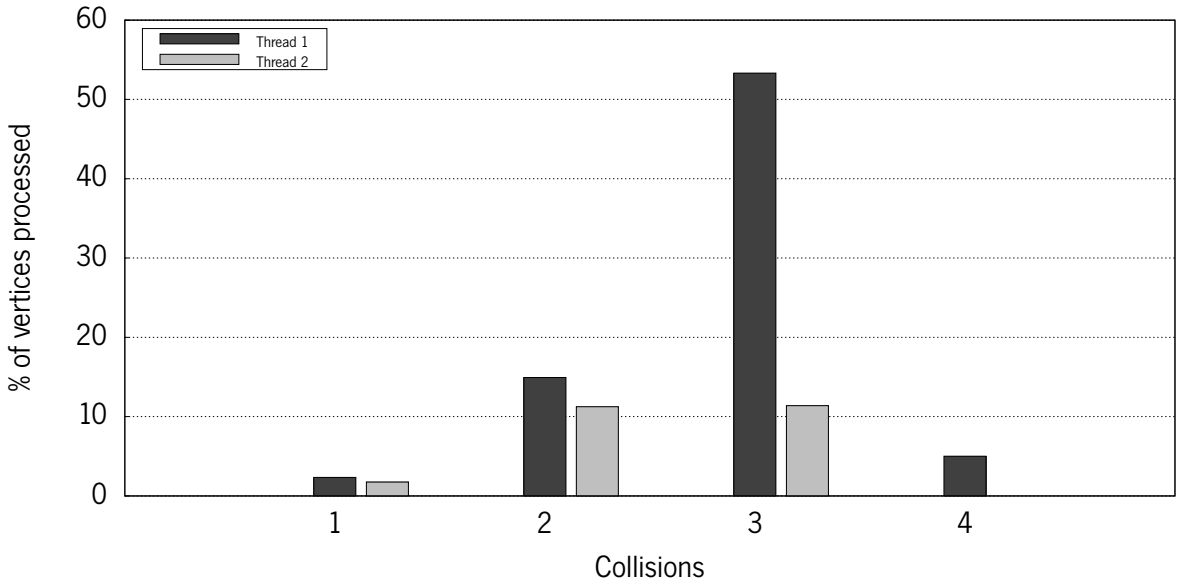



Figure 5.4: (%) vertices processed per thread by *tm_mst_pt* for execution with 2 threads on USA graph.

On the other hand, the generic, Borůvka based implementation exhibits good speedups, for the CPU version, and good, overall execution time for the GPU version. Table 5.3 summarizes the speedup and efficiency attained by *GenBoruvka OMP* and *GenBoruvka GPU*, with respect to the *GenBoruvka OMP* implementation running with a single thread for three 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 to 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 can be attributed to load imbalance. The load imbalance in *GenBoruvka OMP* 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. *GenBoruvka OMP* uses guided scheduling, which only partially corrects this problem.

Further ways of improving the scalability of *GenBoruvka OMP* implementation were experimented with. In particular, experimenting several combinations of thread affinity setups, even though none has shown to perform better than the others. In fact, since the edges that are read by one of the threads are never read by all the other thread, there is no optimal thread affinity setup.

Figure 5.5 shows the average percentage of load imbalance, in terms of edges processed per thread, for each iteration of the kernel described in Section 4.4.1, for different numbers of threads (2-40), when executing on the USA graph. Although all 11 iterations of the algorithm are shown, it should be noted that the

| | 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% |
| *GenBoruvka GPU* | 7.32x | | 16.21x | | 15.85x | |

Table 5.3: Speedup (S) and Efficiency (E) for *GenBoruvka OMP* and *GenBoruvka GPU* for 3 graphs with respect to *GenBoruvka OMP* implementation with a single thread.

first 7 are considerably more relevant than the others, since they represent a much larger chunk of the total execution time (>80%). In the figure, there is also plotted a line at 5%, which is 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 the generic Borůvka's algorithm. In particular, it would be necessary 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.

The next step would be building the system's roofline [Williams et al., 2009], and placing the application within it. The roofline model allows one to easily identify a specific system's maximum performance and the various optimization techniques (or, *performance ceilings*) that can be applied to an application in order to increase performance. This model relies on the system's peak bandwidth and obtainable bandwidth with optimizations such as unit stride accesses, software pre-fetching and memory affinity on NUMA systems, together with attainable performance, with optimizations such as SIMD and multi-threading, which depends on the architecture of the CPU.

The application is placed within the roofline based on its operation intensity, which is calculated as the number of floating point operations per byte accessed to main memory, and its performance in terms of Floating Point Operations per Second (FLOPS/s). However, *GenBoruvka OMP* does not have any floating point operation, since the integer data type is the only one that is used. At most, the weights associated to each edge could be represented as a floating point number, yet, it would not matter for much because the

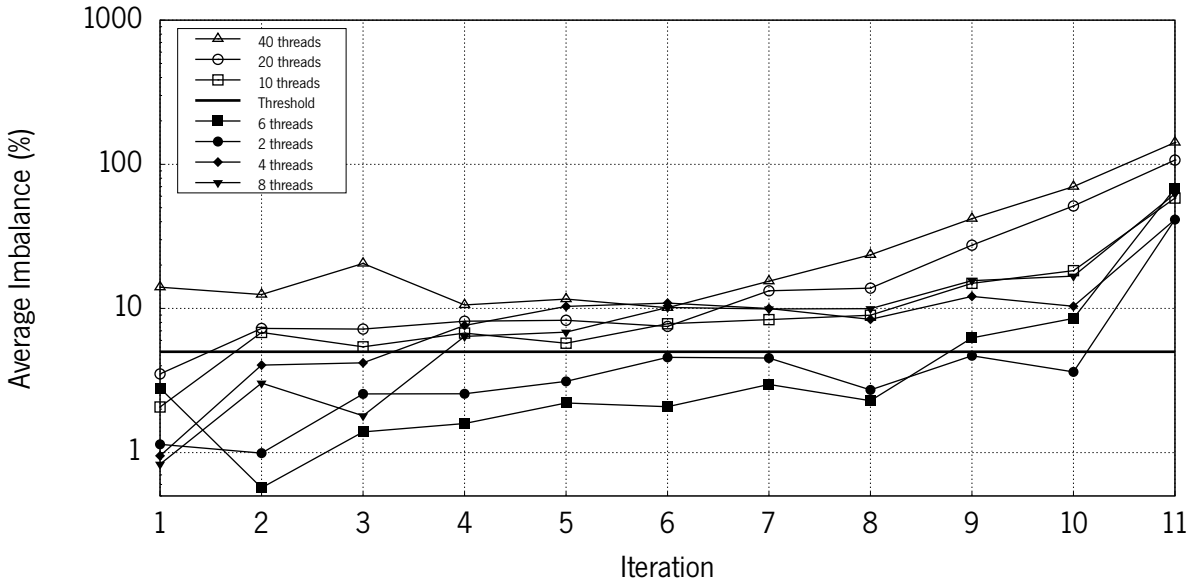



Figure 5.5: Average load imbalance per iteration for *GenBoruvka OMP* for the USA road-network graph.

most computationally demanding operations done on edge weights, are comparisons. This also happens to be true for the integer data, as most of the operations are typical instructions used for loop control.

While PAPI does support counters for floating point and integer instructions, only the former is usually supported. This is due to the fact that CPUs do not include native counters for counting integer instructions. However, since, in the generic Borůvka algorithm, memory access and copy instructions completely dominate the algorithm, it is safe to say that operational intensity is extremely low.

Alternatively, measuring the application's consumed bandwidth could be used to show that the application is limited due to the large amount of memory accesses that are saturating the available bandwidth. Measuring bandwidth on NUMA systems can be misleading, since memory affinity is a factor, the peak memory bandwidth can be reduced due to all memory traffic going over CPU interconnect. Using PAPI, one could measure the consumed bandwidth using the following counter:

- `PAPI_L3_TCM` - total L3 cache misses.

And the following formula, where the L3 line size is usually 64 bytes.:

$$2^{-30} \times \frac{\texttt{PAPI\_L3\_TCM} \times \text{L3 line size}}{\text{time(s)}} \text{ in (GB/s)} \tag{5.1}$$

However, this would only be an estimate of the actual consumed bandwidth, as this would only measure memory traffic generated by last level cache misses, while traffic from mechanisms such as prefetches and cache writebacks to memory are not accounted for, which often happens to be the largest contributors to total memory traffic.

Furthermore, it is debatable whether or not application bandwidth actually tells us any useful information. A low bandwidth could indicate that there are few last level cache misses, but it could also indicate the presence of bank conflicts, which reduces the amount of requests serviced at a time, effectively reducing the consumed bandwidth. On the other hand, a high bandwidth may indicate a high amount of last level cache misses, while also be a indicator of the application actually performing well, consuming the available resources and saturating memory bandwidth.

This tells us nothing about its actual performance and what can be done to improve it. It seems that the only real conclusion that can be made is that, when the application saturates the available bandwidth, it will not benefit from increasing the number of threads. Any other case of consumed bandwidth would be inconclusive. Interestingly, studies on the measurement memory bandwidth consumed by applications seems to be lacking in the literature, all of which begs the question: why bother measuring bandwidth if the same conclusions (and more) can be taken from measuring the last level cache miss rate.

Using PAPI, the L3 cache miss rate was measured, for each iteration of the main loop described in Algorithm 5, for different numbers of threads (1-40) on the USA graph. L3 cache miss rate was computed by using the following counters:

- `PAPI_L2_TCM` - total L2 cache misses, which is an alias for total number of accesses to L3 cache;
- `PAPI_L3_TCM` - total L3 cache misses.

And using the following formula to compute the last level cache miss rate:

$$100 \times \frac{\texttt{PAPI\_L3\_TCM}}{\texttt{PAPI\_L2\_TCM}} \text{ in (\%)} \tag{5.2}$$

As shown in Figure 5.6, for single-threaded execution, the L3 cache miss rate starts to drop drastically at iteration number 4, and remains 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

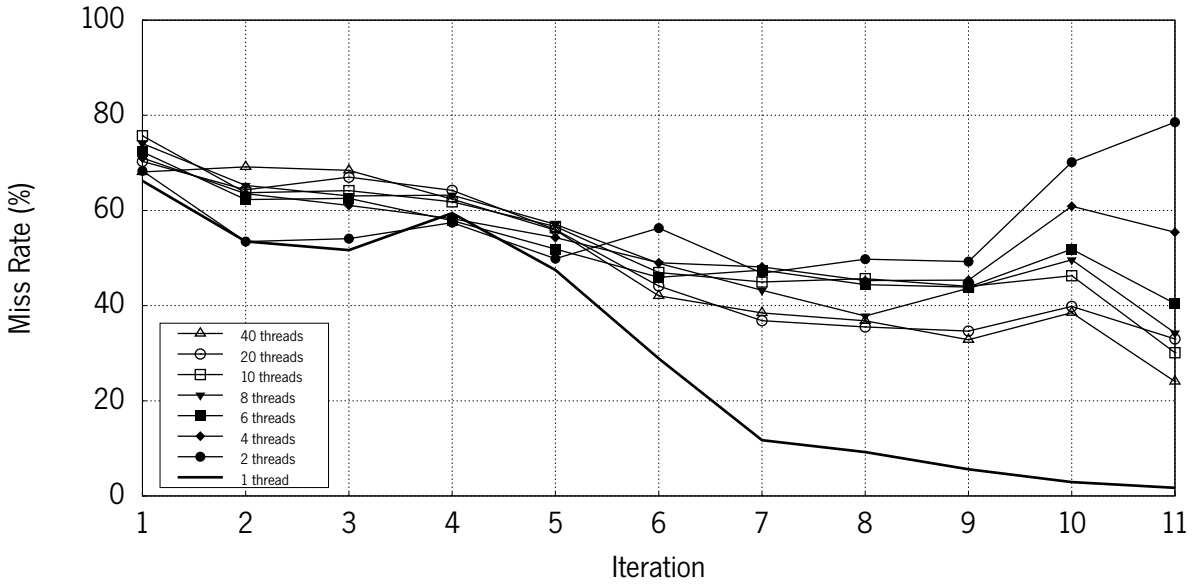



Figure 5.6: L3 cache miss rate per iteration for *GenBoruvka OMP* for USA road-network graph.

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 both limits the performance and the scalability of the application. This is 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 higher 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.

Figure 5.7 shows the execution times for *GenBoruvka MPI* with various combinations of processes and threads for the USA graph. Running with a single process, the implementation has good performance, outperforming all other executions with more than one process. On the other hand, scalability is hurt when running with multiple processes, which is due to the inter-process communication. A heterogeneous or multi-GPU implementation could be seen as an execution with 2 processes, and as seen from the figure, the amount of overhead incurred by the communication is not as high as one would expect, which could motivate such an implementation.

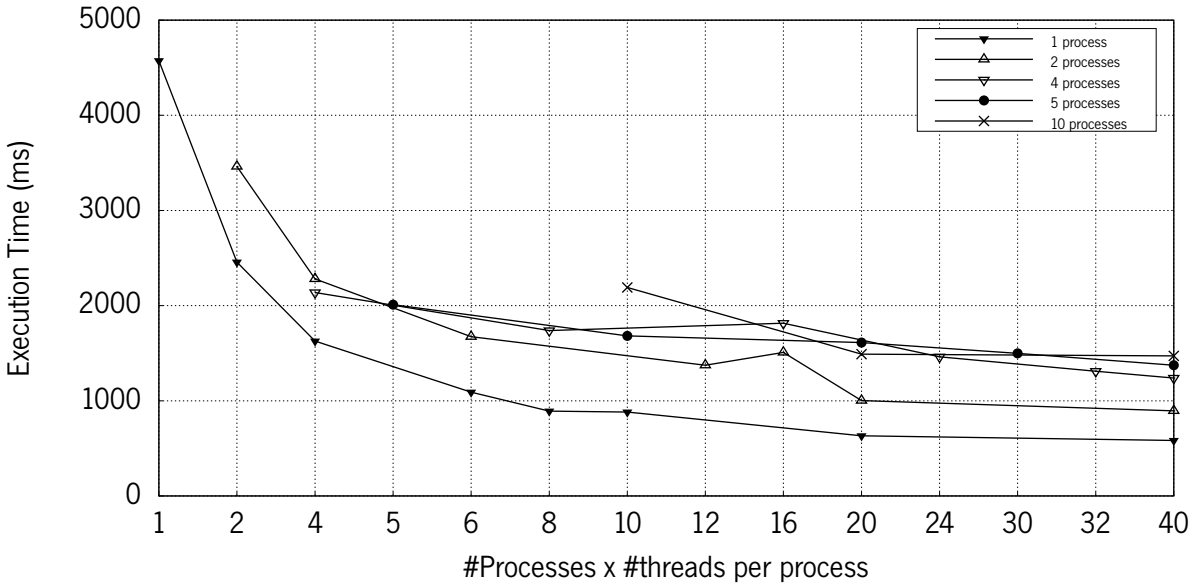



Figure 5.7: Results for hybrid *GenBoruvka MPI* execution for USA graph.

## 5.5 Critical Analysis

This section compares the best CPU and GPU implementations presented in Section 5.4 (*GenBoruvka OMP* and *GenBoruvka GPU*), with third-party implementations presented in the various publications cited in this dissertation.

Figure 5.8 shows the execution time of all the selected GPU and CPU implementations, with 1 and 10 threads, for the set of input graphs. 10 threads were used as to avoid NUMA, i.e., force all threads to run on the same CPU-chip, and avoid running multiple threads on the same CPU.

For *Cong2005*, it was only possible to compute graphs 1 to 6, since executions for the remainder did not terminate in a timely manner, apparently due to a live-lock in the color propagation procedure. As shown in the figure, *GenBoruvka OMP* outperforms both *Cong2005* and *Galois* for all input graphs, when running with a single thread. Both *GenBoruvka OMP* with 10 threads, and *GenBoruvka GPU*, 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. The results back up the superiority of the generic Borůvka implementations, which is a direct consequence of the suitability of the algorithmic variant for parallel architectures.

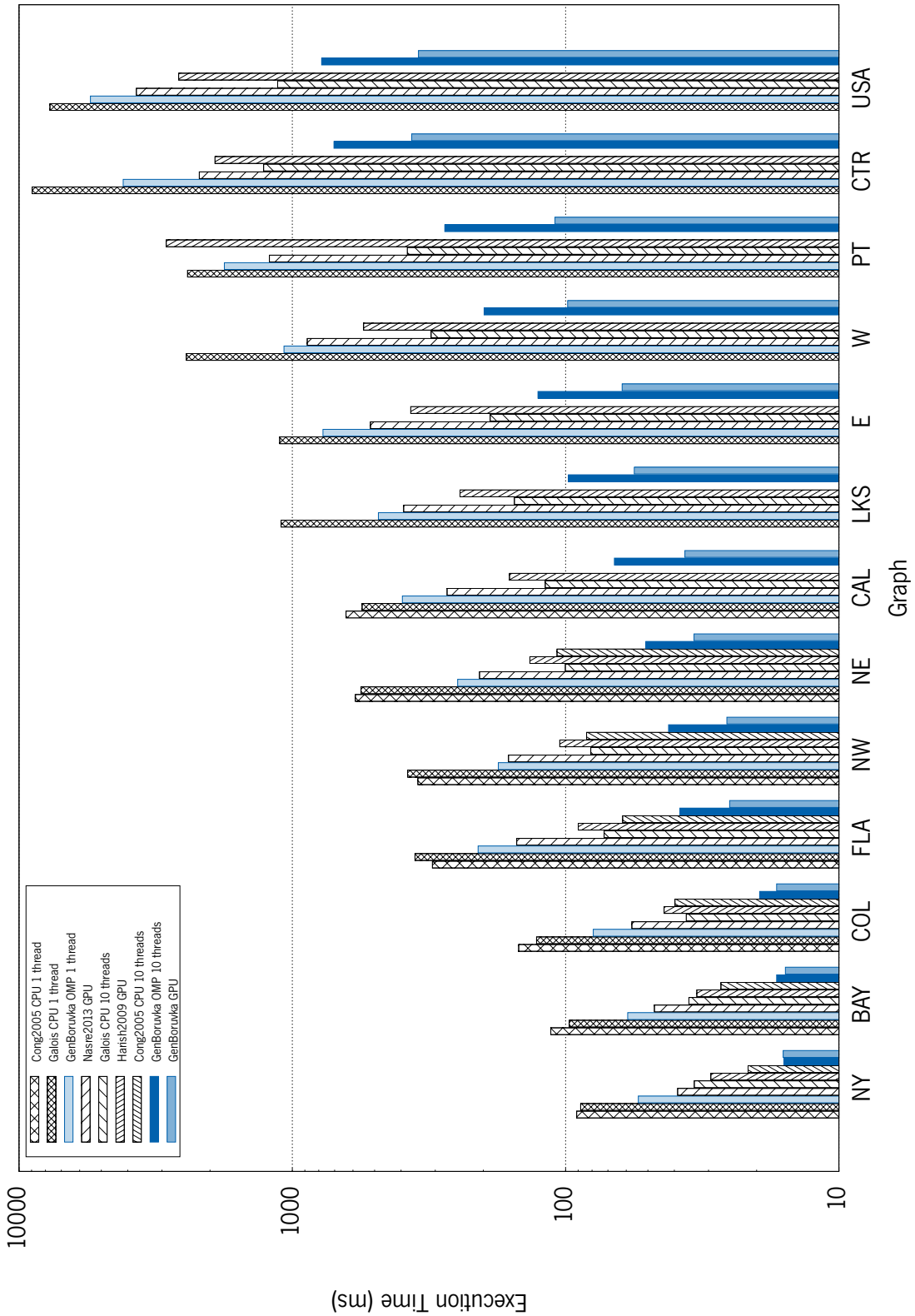



Figure 5.8: Measured execution times for all road-network graphs

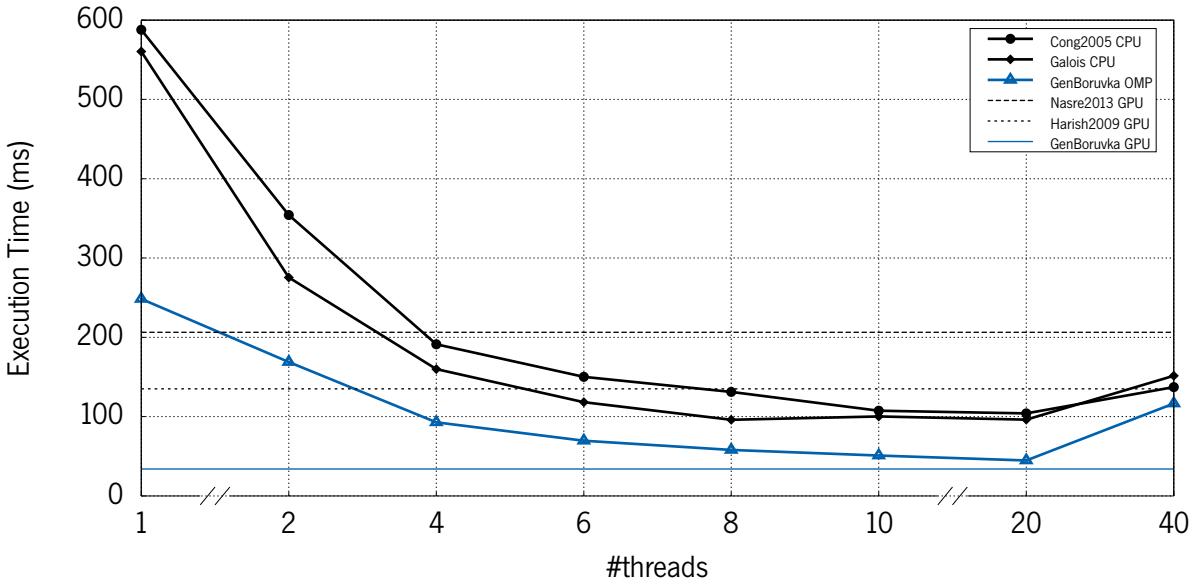



Figure 5.9: Measurements for NE road-network graph.

Figures 5.9, 5.10 and 5.11 show the scalability of the CPU implementations for three particular representative input graphs (NE, PT and USA), and compares them with GPU implementations. Figure 5.9 shows the results for the NE graph, the largest graph for which it was possible to execute all implementations and combinations of threads. Figure 5.10 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, each kernel of *Harish2009* was profiled, and concluded that the color propagation is inefficient on this particular graph. This is most likely due to the structure of the PT graph, since the degrees (number of neighbors) of the vertices of the graph vary considerably. Figure 5.11 shows the results for the largest graph in the data set. In all cases, the CPU versions outperform *Harish2009* and *Nasre2013* GPU implementations with relatively few threads.

It has to be noted that, across the various executions of *Cong2005*, the execution time was very irregular. 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.

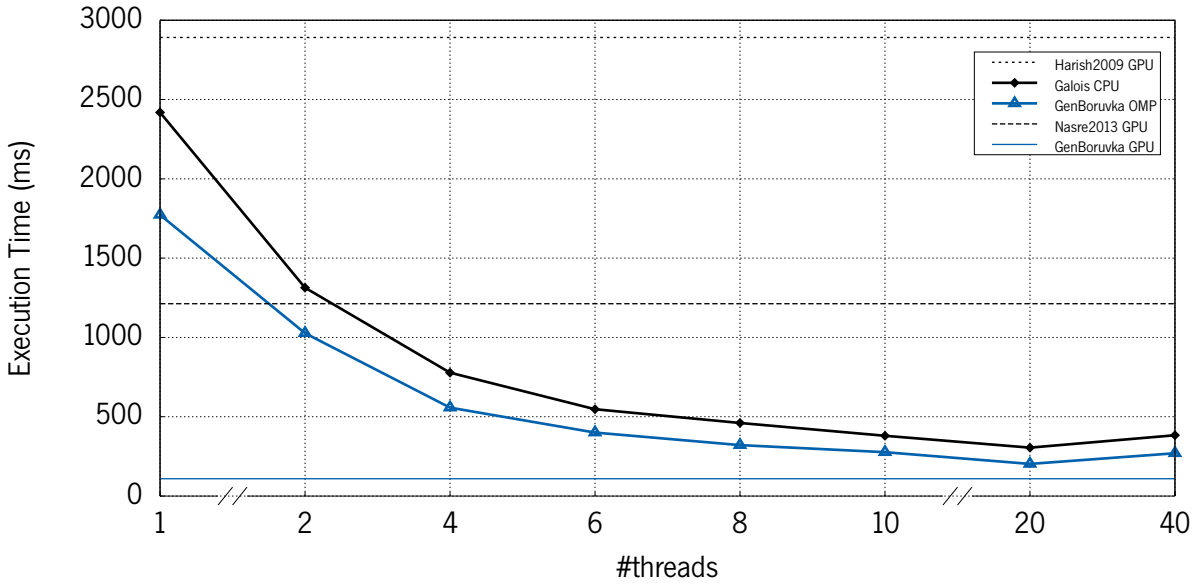



Figure 5.10: Measurements for PT road-network graph.

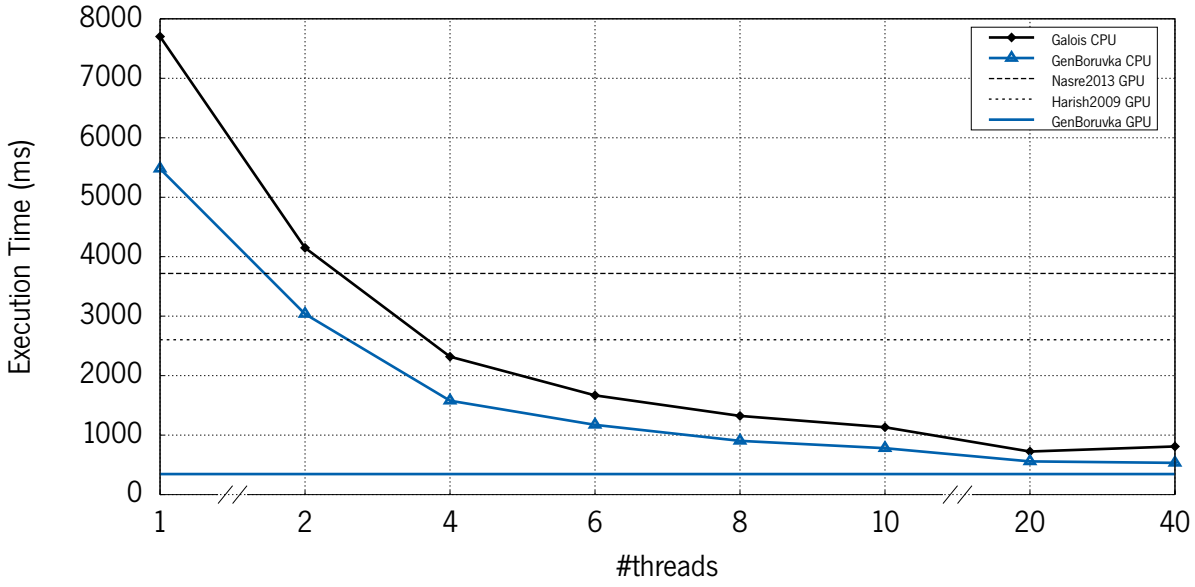



Figure 5.11: Measurements for USA road-network graph.

# Chapter 6

# Conclusions & Future Work

*This chapter concludes the dissertation, presenting an overview of the results obtained with the developed implementations and suggesting lines of research to further investigate the more relevant outcome of this work.*

## 6.1 Conclusions

This dissertation presented a comprehensive literature compilation on the state of the art of MST-solvers, along with various parallel implementations of MST-solvers based on Prim's and Borůvka's algorithms. The proposed implementations were tested using a suite of widely used road-network graphs, including a road-network graph that derived from OpenStreetMap, and compared with one other. Included in this analysis is a first-hand comprehensive empirical comparison of several disclosed state of the art third-party CPU and GPU implementations of MST-solvers.

In general, all Prim based implementations perform poorly, usually having worse performance than the best sequential algorithm. The core contribution of this dissertation is the generic Borůvka algorithmic variant, which exhibited considerable speedups, outperforming all disclosed MST-solver implementations, and being designed in such a way that it is implementable, with little effort and changes, on CPU-based shared and distributed memory systems, and on GPUs.

With regard to the question posed in Chapter 1, which raised the problems faced when trying to increase application performance, such as algorithmic design, distribution of tasks, and data locality, and how it

affects portability. This dissertation work showed that it is possible, with careful design of the algorithm and data structures, to develop portable implementations, without losing performance.

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. The generic Borůvka algorithm fills this gap by presenting implementations that are not only very efficient, but can also compute the MST for any graph size, without the need to adjust parameters.

## 6.2 Future Work

The obtained results motivate further research on irregular algorithms and, in particular, generic graph algorithms. Future lines of research may include, but not limited to:

- Heterogeneous (CPU + GPU) implementation of the generic Borůvka algorithm, based on the combination of the OpenMP, OpenMPI and CUDA implementations.
- The OpenMPI implementation of the generic Borůvka algorithm can be optimized as to possibly reduce communication and memory footprint.
- Extend the concept of the generic algorithm to other graph algorithms such as Breadth-first search (BFS), Depth-first search (DFS), partitioning and betweenness centrality.
- The generic Borůvka algorithm exhibited good results for the suite of road-network graphs. Extending the benchmarks to graphs with varying densities and structures would be adequate.
- The generic Borůvka implementation can be adjusted to support unconnected graphs.
- The 9th DIMACS challenge graphs are still widely used for benchmarks, but at the rate that real-life graphs are growing they will quickly become obsolete. This will force researchers to resort to synthetic graphs. There are multiple graph generators available, each capable of generating graphs with distinct characteristics and offering configurable parameters to fine tune the structure. However, it is not clear how these parameters affect the graphs, and how they relate to real-life scenarios. Therefore, it would be interesting to create a suite of graph benchmarks, representing a wide range of real-life scenarios, and with sizes that challenge the available memory on today's computing platforms.

# Bibliography

[Bader and Cong, 2004] Bader, D. A. and Cong, G. (2004). Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In *Parallel and Distributed Processing Symposium*, page 39. IEEE.

[Bader and Madduri, 2006] Bader, D. A. and Madduri, K. (2006). Gtgraph: A synthetic graph generator suite. *Atlanta, GA, February*.

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

[Borůvka, 1926b] Borůvka, O. (1926b). Příspěvek k řešení otázky ekonomické stavby elektrovodních sítí. *Elektrotechnick obzor*, 15:153–154.

[Chakrabarti et al., 2004] Chakrabarti, D., Zhan, Y., and Faloutsos, C. (2004). R-mat: A recursive model for graph mining. *Computer Science Department*, page 541.

[Chong et al., 2001] Chong, K. W., Han, Y., and Lam, T. W. (2001). Concurrent threads and optimal parallel minimum spanning trees algorithm. *Journal of the ACM (JACM)*, 48(2):297–323.

[Cong and Bader, 2005] Cong, G. and Bader, D. (2005). Lock-free parallel algorithms: An experimental study. In *HiPC 2004*, pages 516–527. Springer.

[Fredman and Tarjan, 1987] Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615.

[Graham and Hell, 1985] Graham, R. L. and Hell, P. (1985). On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57.

[Harish et al., 2009] Harish, P., Vineet, V., and Narayanan, P. (2009). Large graph algorithms for massively multithreaded architectures. *Centre for Visual Information Technology, I. Institute of Information Technology, Hyderabad, India, Tech. Rep. IIIT/TR/2009/74*.

[Jarník, 1930] Jarník, V. (1930). O jistém problému minimálním. *Práca Moravské Prírodovedecké Spolecnosti*, 6:57–63.

[Johnson and Metaxas, 1992] Johnson, D. B. and Metaxas, P. (1992). A parallel algorithm for computing minimum spanning trees. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 363–372. ACM.

[Kang and Bader, 2009] Kang, S. and Bader, D. A. (2009). An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. *ACM Sigplan Notices*, 44(4):15–24.

[Karypis and Kumar, 1995] Karypis, G. and Kumar, V. (1995). Multilevel graph partitioning schemes. In *ICPP (3)*, pages 113–122.

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

[Loncar et al., 2013] Loncar, V., Škrbic, S., and Balaz, A. (2013). Distributed memory parallel algorithms for minimum spanning trees. In *Proceedings of the World Congress on Engineering*, volume 2.

[Mareš, 2008] Mareš, M. (2008). The saga of minimum spanning trees. *Computer Science Review*, 2(3):165–221.

[Mariano et al., 2013] Mariano, A., Lee, D., Gerstlauer, A., and Chiou, D. (2013). Hardware and software implementations of prim's algorithm for efficient minimum spanning tree computation. In *Embedded Systems: Design, Analysis and Verification*, pages 151–158. Springer.

[Moret and Shapiro, 1994] Moret, B. M. E. and Shapiro, H. D. (1994). An empirical assessment of algorithms for constructing a minimum spanning tree. *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science: Computational Support for Discrete Mathematics*, pages 99–117.

[Nasre et al., 2013] Nasre, R., Burtscher, M., and Pingali, K. (2013). Morph algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*,

pages 147–156. ACM.

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

[Pettie and Ramachandran, 2002] Pettie, S. and Ramachandran, V. (2002). A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM Journal on Computing*, 31(6):1879–1895.

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

[Prim, 1957] Prim, R. C. (1957). Shortest connection networks and some generalizations. *Bell system technical journal*, 36(6):1389–1401.

[Rostrup et al., 2011] Rostrup, S., Srivastava, S., and Singhal, K. (2011). Fast and memory–efficient minimum spanning tree on the GPU. *International Journal of Computational Science and Engineering*, 8(1):21–33.

[Setia et al., 2009] Setia, R., Nedunchezhian, A., and Balachandran, S. (2009). A new parallel algorithm for minimum spanning tree problem. In *Proc. International Conference on High Performance Computing (HiPC)*, pages 1–5.

[Vineet et al., 2009] Vineet, V., Harish, P., Patidar, S., and Narayanan, P. (2009). Fast minimum spanning tree for large graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 167–171. ACM.

[Wang et al., 2011] Wang, W., Huang, Y., and Guo, S. (2011). Design and implementation of GPU-based prim's algorithm. *International Journal of Modern Education and Computer Science (IJMECS)*, 3(4):55.

[Williams et al., 2009] Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76.

# Appendix A

# Generic **Borůvka**'s Algorithm Pseudo-Code

- `vertex_minedge[i]` is the minimum selected edge for vertex `i`;
- `getDestination(vertex_minedge[i])` is the successor of `i`.

**Algorithm 7** Find minimum edge per vertex

**Input:** Vertex id, $vertex\_minedge$ array

1: $min\_weight :=$ max weight
2: $min\_edge :=$ null edge
3: $min\_dst :=$ null vertex
4:
5: **for each** neighbor $e(id, w)$ of $id$ **do**
6: | **if** $weight(e) < min\_weight \vee (weight(e) = min\_weight \wedge w < min\_dst)$ **then**
7: | | $min\_weight := weight(e)$
8: | | $min\_edge := e$
9: | | $min\_dst := w$
10:
11: $vertex\_minedge[id] = min\_edge$

**Algorithm 8** Remove mirrored edges kernel

**Input:** Vertex id, $vertex\_minedge$ array

```
succ = getDestination(vertex_minedge[id])
succ_succ = getDestination(vertex_minedge[succ])

if id = succ_succ then
    if id < dst then
        remove vertex_minedge[id]
    else
        remove vertex_minedge[dst]
```

**Algorithm 9** Initialize colors

**Input:** Vertex id, $color$ array, $vertex\_minedge$ array

```
edge := vertex_minedge[id]
if edge = null edge then
    color[id] := id
else
    color[id] := destination(e)
```

**Algorithm 10** Propagate colors kernel

**Input:** Vertex id, $color$ array

```
succ := color[id]
succ_succ := color[succ]

if succ ≠ succ_succ then
    color[id] := succ_succ
```

**Algorithm 11** Create new vertex ids

**Input:** Vertex id, $color$ array, $flag$ array

```
if id = color[id] then
    flag[id] := 1
else
    flag[id] := 0
```

**Algorithm 12** Count new edges

**Input:** Contracted graph $G_{new}(V, E)$, vertex id, $color$ array, $new\_vertex$ exclusive prefix sum array

```
1: my_color := color[id]
2: new_edges := 0
3:
4: for each neighbor e(id, w) of id do
5:     if my_color ≠ color[w] then
6:         new_edges := new_edges + 1
7:
8: supervertex_id := new_vertex[my_color]
9: atomicAdd(G_new.outdegree[supervertex_id], new_edges)
```

**Algorithm 13** Insert new edges

**Input:** Contracted graph $G_{new}(V, E)$, vertex id, $color$ array, $new\_vertex$ exclusive prefix sum array, $topedge$ copy of $first\_edge$ array

```
1: my_color := color[id]
2: supervertex_id := new_vertex[my_color]
3:
4: for each neighbor e(id, w) of id do
5:     if my_color ≠ color[w] then
6:         top_edge := atomicInc(topedge[supervertex_id])
7:         G_new.edge_dst[top_edge] := new_vertex[color[w]]
8:         G_new.edge_wt[top_edge] := weight(e)
9:
```

# Appendix B

# Scientific Paper

The following scientific paper resulted from the work on the generic Borůvka's algorithm. This paper was submitted and accepted at an top tier conference: the 23rd Parallel, Distributed and Network-based Processing (PDP 2015).

# 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.

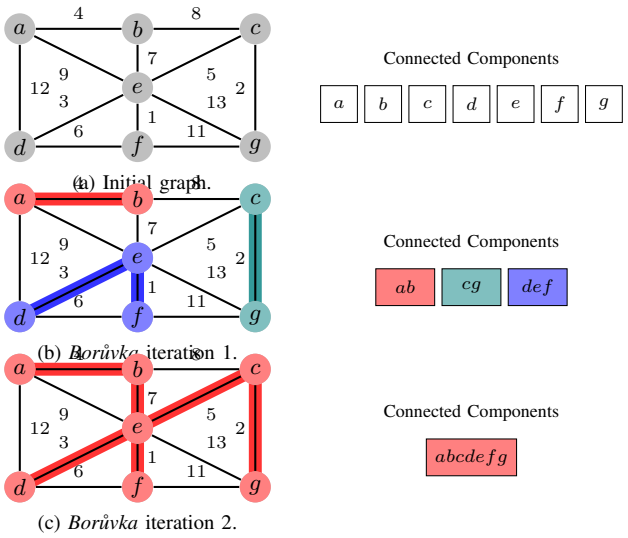
In this context, *Kruskal*'s algorithm is the least attractive candidate, due to its inherently sequential workflow. In contrast, *Prim*'s algorithm is more suited for parallelization but, it either breaks down to operations with reduced parallelization opportunities or ends up with overly complex parallel 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/

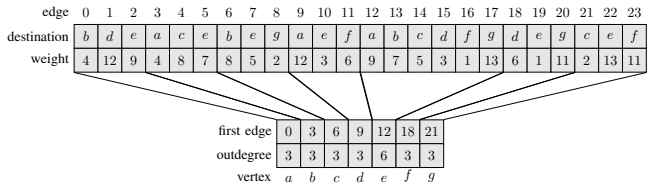



Fig. 2: CSR representation of the example graph in Figure1.

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.

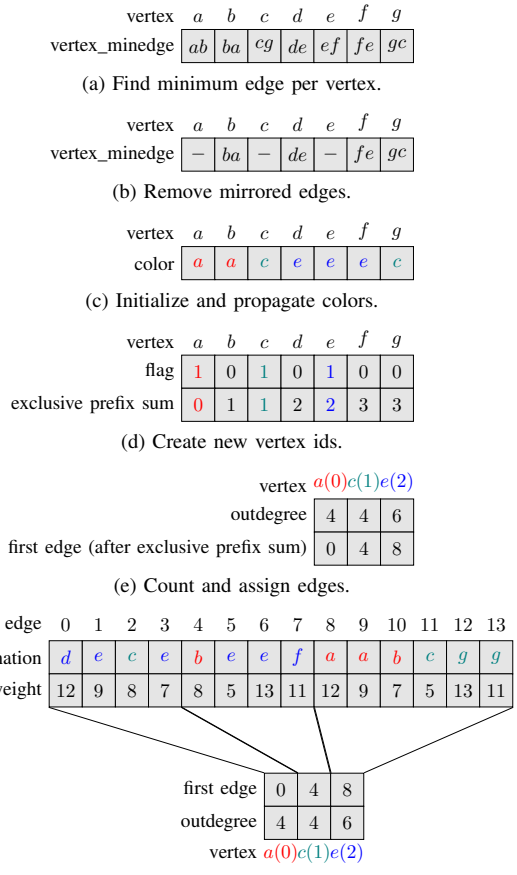
To represent undirected graphs, all edges are duplicated to cover both directions. Figure 2 shows the CSR representation of the example graph shown in Figure 1.

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 $G(V, E)$

```
while number of vertices > 1 do
    Find minimum edge per vertex
    Remove mirrored edges
    Initialize colors
    while not converged do
        Propagate colors
    Create new vertex ids
    Count new edges
    Assign edge segments to new vertices
    Insert new edges
```




(a) Find minimum edge per vertex.

(b) Remove mirrored edges.

(c) Initialize and propagate colors.

(d) Create new vertex ids.

(e) Count and assign edges.

(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 |
| $\mu$Arch | 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

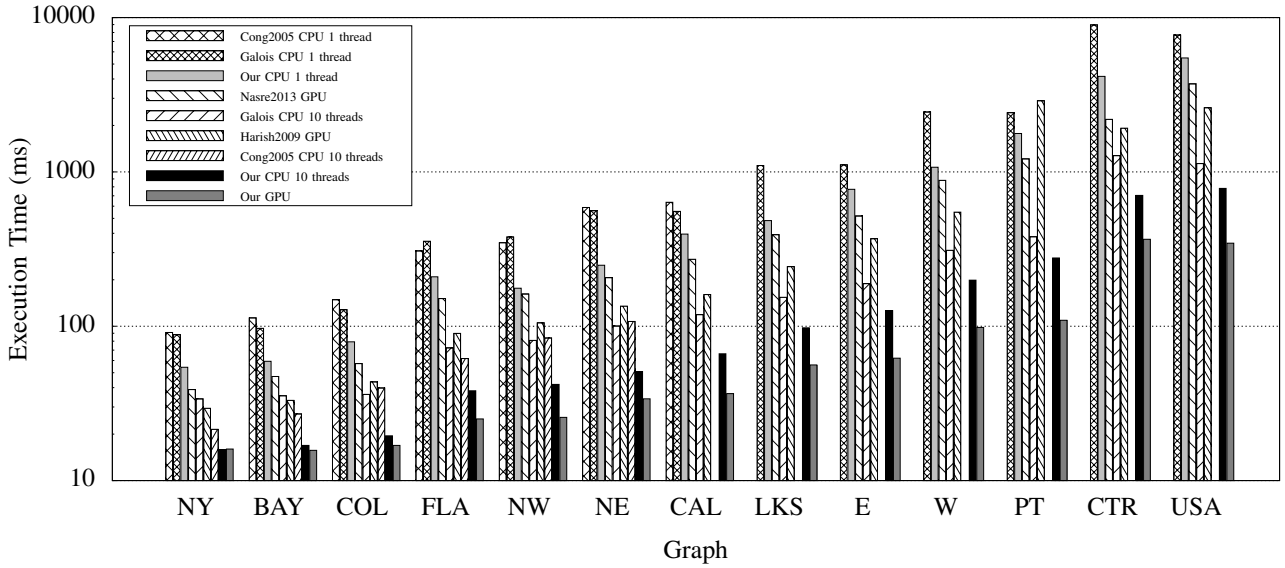
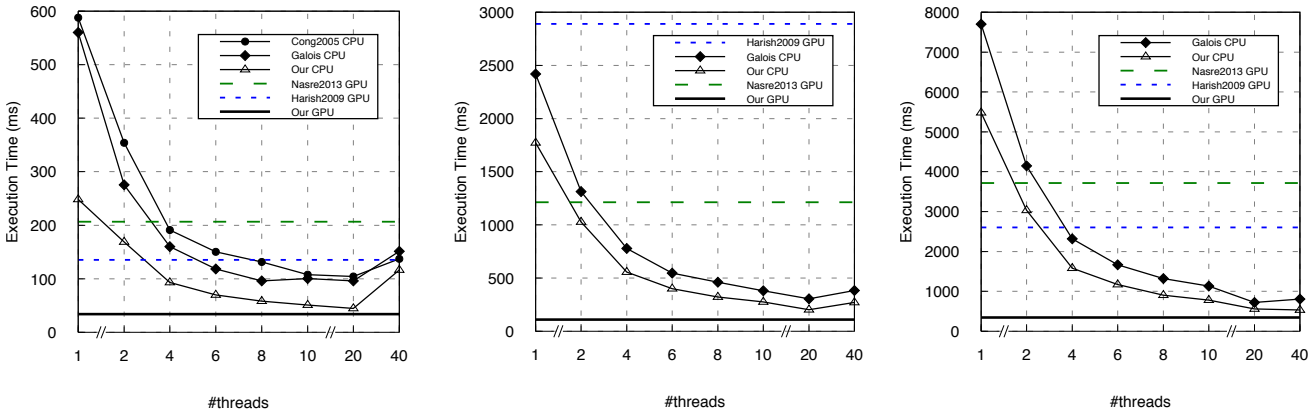
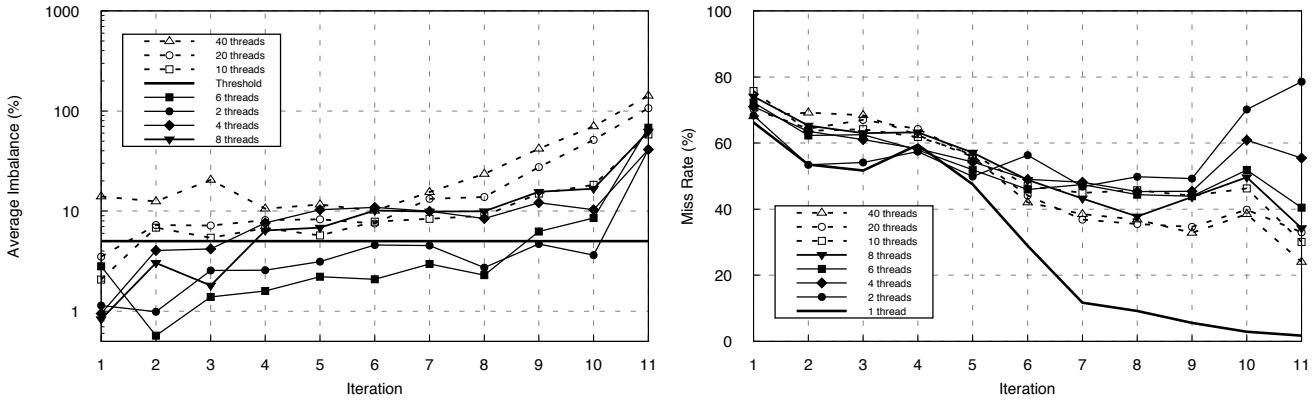



Fig. 4: Measured execution times for all road-network graphs




(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.

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


## 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. Prírodoved. 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.