Tree Contraction, Connected Components, Minimum Spanning Trees: a GPU Path to Vertex Fitting

Raul H. C. Lopes[1], *Ivan D. Reid*[1], *Peter R. Hobson*[1]

[1]Department of Electronic & Computer Engineering, Brunel University London, Kingston Lane, Uxbridge, UB8 3PH, United Kingdom

DOI: http://dx.doi.org/10.3204/DESY-PROC-2014-05/5

Standard parallel computing operations are considered in the context of algorithms for solving 3D graph problems which have applications, e.g., in vertex finding in HEP. Exploiting GPUs for tree-accumulation and graph algorithms is challenging: GPUs offer extreme computational power and high memory-access bandwidth, combined with a model of fine-grained parallelism perhaps not suiting the irregular distribution of linked representations of graph data structures. Achieving data-race free computations may demand serialization through atomic transactions, inevitably producing poor parallel performance. A Minimum Spanning Tree algorithm for GPUs is presented, its implementation discussed, and its efficiency evaluated on GPU and multicore architectures.

1 Introduction

We are concerned with the problem of finding parallel algorithms to compute a Minimum Spanning Tree for a weighted undirected graph. We introduce a parallel algorithm for computing minimum spanning trees over weighted undirected graphs.

An **undirected graph** G is a pair (V, E), where V is a finite nonempty set of vertices and E is a set of unordered pairs of vertices in V, called the set of edges of G. A **path** in G is a sequence of edges $e_0, e_1, \ldots, e_n$, with e_i and e_{i+1}, for $i \in \{0..n-1\}$, sharing exactly one vertex. A **cycle** is path $e_0, e_1, \ldots, e_n$, where e_0 and e_n share one vertex. A graph G is **connected** if there is a path between any pair of its vertices. A **tree** is connected graph containing no cycles. A **spanning tree** for a graph G is a tree containing all vertices in G. A **spanning forest** for a graph G that is not connected is a set of spanning trees, one for each connected component of G. A graph G is weighted when a function w assigns weights to each edge of G. A graph's weight is the sum of weights of its edges. A Minimum Spanning Tree (MST) of a graph G is a tree of minimal weight among all spanning trees of G. The concept extends to that of **Minimum Spanning Forest** (MSF) of G as a spanning forest of minimum weight.

Throughout this paper we will use n to denote number of vertices in a graph G, and m to denote its number of edges. An MST in $\mathcal{R}^2$ can be computed in $O(n \lg n)$ using Delaunay triangulations. However, the work demanded to compute an MST in $\mathcal{R}^d$ is limited from below by $\Omega(m)$ when $d > 2$, see [1]. Even the representation in memory of the edges of a 2^{20} vertices graph could be a challenge.

Minimum spanning trees have applications, for example, in computer networks, water supply networks, and electrical grids. The computation of MSTs is considered a fundamental challenge in Combinatorial Optimization [2] and is of interest for the HEP community.

In the next sections, we discuss first sequential algorithms for computing connected and spanning trees, and the difficulties involved in parallelizing them. Then we present an algorithm for computing minimum spanning trees on GPU architectures and discuss its performance on both GPU and multi-core architectures.

2 Sequential Algorithms

Minimum spanning tree computations make use of the following property to choose edges from the underlying graph to add to its MST. See [3] for a proof of its soundness.

Cut property: The lightest edge connecting any nonempty subset X of $V(G)$ to $V(G) - X$ belongs to $MSF(G)$.

The Borůvka algorithm is historically the first and possibly most general MSF algorithm. It uses the cut property to maintain the following invariant: each tree in the forest is an MST for the vertices in it, and each vertex in the initial graph belongs to exactly one tree. It starts with a set of forests, each containing exactly one vertex from the given graph G. Borůvka's is essentially a non-deterministic algorithm in which, at any time, all lightest edges that can expand any of its trees can be added. It's important to notice, however, that each edge addition joins two trees and this must be reflected in the data structures used in its implementation. Also, its non-deterministic nature appeals to parallel implementations, but introduces possible data-race conditions.

Most theoretically efficient MST algorithms use Borůvka functions, as for example [4], and the deterministic algorithm given by [1] and [5]. These algorithms, however, seem to lack efficient practical implementations or parallel versions. The most successful implementations of sequential MST algorithms, the **Prim-Dijkstra** and **Kruskal** algorithms evaluated in [6], can be seen as specializations of the Borůvka algorithm and they also maintain an invariant, see [7], where at any moment in the computation the edges already selected by the algorithm form an MST of the subgraph in question and any new edge added satisfies the *cut property.* In particular the Kruskal algorithm [3] is a sequential version of the Borůvka algorithm where edges are processed in increasing order of weights.

In the next section, we discuss the challenges found in parallelizing the Borůvka algorithm and the fundamental requirements for parallelization of a minimum spanning tree computation.

3 Parallel Algorithms

The nondeterministic nature of Borůvka's algorithm clearly invites the introduction of asynchronous parallelism in its implementation. We would have an algorithm that performs a sequence of parallel steps, where each parallel step joins all possible pairs of trees, using the lightest edges connecting them. However, care must be taken in that: more than two trees may be joined in the same parallel step; given two trees t_0 and t_1 joined in a parallel step, all edges connecting vertices in t_0 to vertices in t_1 must be discarded.

Those problems result from the fact that many trees may be expanded simultaneously. The Prim-Dijkstra algorithm expands exactly one tree and thus avoids complications of performing simultaneous unions of vertices and edges by being strictly sequential. The **Kruskal** algorithm excludes the possibility of joining more than two trees simultaneously by processing the edges in increasing order of weight. It still keeps track of more than one tree being expanded at any time, but each expansion and join is serialized. Its first obvious disadvantage is in processing the edges in increasing weight order, which demands ordering all edges before the real construction of the tree starts, which can be very expensive or even prohibitive when processing, for example, Euclidean graphs where the number of edges can approach n^2. A graph with 2^{20} vertices might demand memory to maintain close to 2^{40} edges, maybe two to four terabytes just to store edge weights.

The Kruskal algorithm, or any other derived from the Borůvka algorithm, must still take care of the joining of trees and the union of sets of respective vertices and edges. In practice this is performed using efficient implementations of the disjoint-set data structure, see [8]. A disjoint-set data structure offers fast union of two sets by labeling the elements of two sets with a common identifier that defines that new common set that they belong to. It offers fast set membership tests by performing a fast search from the vertex to the label of the structure representing the set it belongs to. In a sequential implementation this is achieving by making vertices the leaves of trees where each node points to its parent, with the root of the tree pointing to itself and being taken as the label of the tree. Set union is trivially performed by making the root of one tree to point to the root of another tree. However, this is an operation that must be synchronized. Synchronization may also be needed if the search for a tree's root is performed concurrently with a join operation.

Nondeterminism has been forcefully defended by authors like Dijkstra [9] and Lamport [10] as a powerful tool in the design of computer algorithms and concurrent systems. More recently, Steele [11] has made a strong point for the integration of asynchronous computations in synchronous parallel algorithms, given that asynchronous programs tend to be more flexible and efficient when processing conditionals. In addition many authors have pointed out asynchronous and non-deterministic parallelism as the root of success of many programs based on the MapReduce model [12].

An ideal implementation of Borůvka's algorithm would start as many asynchronous threads as possible, each expanding a different minimum spanning tree. Synchronization, however, would be needed, for example, if a tree t_0 must be joined to a tree t_1 while, simultaneously, t_1 is being joined to t_2 and t_2 is being joined with t_0. That sort of synchronism is a fundamental requirement of any parallel implementation of Borůvka's algorithm.

The objective of this paper is to present a GPU algorithm. Its design must take into account that GPU architectures perform at their peak when running in SIMD (Single-Instruction-Multiple-Data) fashion. Performance is lost when atomic transactions, or conditions that send threads into different paths, are present.

Correctness in parallel and concurrent computation can only be achieved by ensuring non-interference between threads. Blelloch et ali [13], see also [14], have argued that being data-race free is the minimum requirement in order to achieve correctness in the presence of concurrency. However, being data-race free at the cost of atomic operations may be too costly for SIMD computing. Internal determinism is the solution they introduce to obtain a fast multi-core algorithms by transforming the code to ensure that internal steps are always deterministic. Internal determinism, however, should not be introduced through the introduction of expensive atomic transactions or nesting of conditional structures and code. Possibly the fundamental mandate for SIMD (and GPU) architectures must be functional determinism, where the absence of side effects between threads is a guarantee of non-interference and absence of the need for atomic constructions as semaphores or even transactional memory. This should be achieved by program transformations to perform parallel computation through the composition of efficient GPU parallel constructions.

In the next section we present a parallelization of the Kruskal algorithm and discuss its performance on both GPU and multi-core architectures.

4 A Parallelization of the Kruskal Algorithm and its Performance Evaluation

We introduce a parallel version of the Kruskal algorithm. The algorithm is composed of two phases: parallel sorting of the edges of the given graph, followed by the construction of the minimum spanning tree. The sorting phase is performed by a standard parallel sorting function and is not described here. We describe the construction phase by introducing first its objects and the morphisms that transform them. The objects used by the algorithm follow.

- A set of weighted edges. Represented by a vector of triples (v, v, w), where u and v are the edge vertices and w is its associated weight, it is ordered by a parallel sorting function in the first phase of the algorithm.
- An ordered sequence of prioritized edges. Represented by a vector of triples (i, u, v), u and v being vertices and i the order of the edge (u, v) in the sorted set, it is generated by a parallel zip of the index i and an edge from the sequence of ordered edges fused with a parallel map to form triple.
- A set of vertex reservations. Represented by a vector that at any moment assigns a vertex to an edge, where $\mathit{reserved}[u] = i$, states the fact that edge i owns the vertex u.
- A set of delayed edges. A vector of booleans indexed by edges, which determines if, in a given parallel step, the addition of an edge to the tree will be suspended and delayed to be performed in a future step.
- A disjoint set of trees. Represented by a vector SV of vertices to trees, it assigns to a vertex v the partial minimum spanning tree it belongs, in the sense that if $SV[u]$ equals to v then v if the parent of u in disjoint-set structure, and they belong to same tree. We assume a function $\mathit{root}(u)$ which returns the root of u in SV.

The edges are processed in blocks of p edges. An edge (u, v) is considered for processing and consequent addition to a tree only if its vertices do not belong to the same tree, i.e., $\mathit{root}(u)$ and $\mathit{root}(v)$ are different. Each block is processed by the sequence of parallel transformations described below.

- Parallel reservation. $\mathit{reserve}(i, u, v)$. It is implemented by a parallel loop, where each parallel step takes a triple (i, u, v) from the sequence of prioritized edges described above and tries to mark u and v as reserved by i. It is important to notice that a data-race condition arises when two parallel steps try to acquire the same vertex for different edges.
- Parallel delay. A parallel loop where each step takes a prioritized edge (i, u, v) and the reserved set and marks the edge i as delayed if neither u nor v is reserved for i.

- Parallel commit. Another parallel loop where each step commits a prioritized edge by linking v to u when v is reserved by i, or linking u to v, when u is reserved by i. Linking u to v consists in assigning v to $SV[u]$.
- Parallel disjoint set compression. Using a pointer jumping technique as described in the rake and compress algorithm of Shiloach and Vishkin [15] applied to the disjoint set structure implemented by SV, it compresses the structure to make each leaf to point directly to its respective root.

Table 1 shows times for tests with an implementation of the algorithm on a machine with an *Intel Xeon E5620* and an *Nvidia Tesla C2070*. *Ubuntu 14.04* was the operating system and *nvcc*, the compiler. Each line shows the time to solve a problem using 1, 4, 8 threads, and the time on the GPU. The first line shows the problem of sorting 2^{25} single-precision floating-point numbers from a uniform distribution. It is important to notice that the *Intel Xeon E5620* has four cores, that can be used to run 2 hyperthreads (HT) each. The GPU sort produces close to 11.7 times acceleration compared to sorting on eight threads. In both cases, the tests used the standard sort function provided by the NVidia *thrust library*.

Problem	1 HT	4 HT	8 HT	GPU
sort	1.34	1.11	0.91	0.078
Uniform	6.31	4.05	4.12	0.74
PLaw	7.90	5.91	6.05	1.24

Table 1: Time (seconds) comparing sorting floats and MST construction

The second line shows the times to build a minimum spanning tree with 2^{20} uniformly distributed vertices, and 2^{23} edges on an Euclidean three dimensional space. The GPU gives around 5.56 times acceleration compared with running on 8 threads. The last line shows tests for random graphs with 2^{20} nodes and 2^{24} edges with a power law distribution, based on [16].

5 Conclusions and Further Work

The technical literature generally presents the construction of minimum spanning forests as an outstanding combinatorial optimization problem, that is challenging enough to be used in several benchmarks as shown in the recent [13]. Even if many theoretical parallel algorithms can be found, only a few seem to show effective implementations. Badder et ali [17] have studied parallelizations both based on the Borůvka algorithm and on the Prim-Dijkstra algorithm. The paper [17] shows some gain from the range of two to four processors, but the absence of numbers for one core tests doesn't allow for a clear evaluation of the acceleration obtained. Chong [18] presents a multi-core asynchronous parallel algorithm relying on atomic transactions, that would hardly be realizable on a GPU architecture. The algorithm in [19] seems to be an exception in targeting a GPU architecture. It is based on the Prim-Dijkstra algorithm and achieves less than 3 times acceleration when compared with a single core algorithm. Belloch [13] presents an algorithm for multi-core architectures that introduces the idea of vertex reservation used in this paper. It seems, however, to depend on atomic transactions to access a disjoint-set data structure. The algorithm presented in this paper borrows from Blelloch and from Steele's ideas on combining asynchronous and synchronous parallelism.

A clear limitation of any algorithm derived from Kruskal is the need to use an explicit representation of the edges. We have previously presented a parallel algorithm for k-d-tree construction [20] that could be used to avoid the huge allocation of space when a quadratic number of edges is present in an Euclidean graph.

6 Acknowledgments

Lopes, Reid and Hobson are members of the GridPP collaboration and wish to acknowledge funding from the Science and Technology Facilities Council, UK.

References

[1] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackerman type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[2] William J. Cook, William H. Cunningham, and William R. Pulleybank. *Combinatorial Optimization*. Willey-Blackwell, 1997.

[3] Thomas Cormen, C. Leiserson, Ron Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.

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

[5] Seth Petit and Vijaya Ramachandran. An optimal minimum spanning algorithm. *Journal of the ACM*, 49(1):16–34, 2002.

[6] Bernard M. E. Moret and Henry D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. *DIMACS Monographs 15*, pages 99–117, 1994.

[7] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Mathematics*. Springer, 2010.

[8] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[9] Edsger E. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[10] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison Wesley, 2002.

[11] Guy L. Steele. Making asynchronous parallelism safe for the world. *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 218–231, 1990.

[12] Derek G. Murray and Steven Hand. Non-deterministic-parallelism considered useful. *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, pages 19–19, 2011.

[13] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 181–192, 2012.

[14] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[15] Y. Shiloach and Uzi Vishkin. An $o(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[16] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*, pages 442–446, 2004.

[17] David A. Bader and Guojing Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *J. Parallel Distrib. Comput.*, 66(11):1366–1378, November 2006.

[18] Ka Wong Chong, Yijie Han, and Tak Wah Lam. Concurrent threads and optimal parallel minimum spanning trees algorithm. *Journal of the ACM*, 48(2):297–323, March 2001.

[19] Sadegh Nobari, Thanh-Tung Cao, Panagiotis Karras, and Stéphane Bressan. Scalable parallel minimum spanning forest computation. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 205–214, New York, NY, USA, 2012. ACM.

[20] Raul H C Lopes, Ivan D Reid, and Peter R Hobson. A well-separated pairs decomposition algorithm for k-d trees implemented on multi-core architectures. *Journal of Physics: Conference Series*, 513(052011), 2014.