Scalable Parallel Minimum Spanning Forest Computation

Sadegh Nobari
National University of Singapore
snobari@nus.edu.sg

Thanh-Tung Cao
National University of Singapore
caothanh@nus.edu.sg

Panagiotis Karras
Rutgers University
karras@business.rutgers.edu

Stéphane Bressan
National University of Singapore
steph@nus.edu.sg

Abstract

The proliferation of data in graph form calls for the development of scalable graph algorithms that exploit parallel processing environments. One such problem is the computation of a graph's minimum spanning forest (MSF). Past research has proposed several parallel algorithms for this problem, yet none of them scales to large, high-density graphs. In this paper we propose a novel, scalable, parallel MSF algorithm for undirected weighted graphs. Our algorithm leverages Prim's algorithm in a parallel fashion, concurrently expanding several subsets of the computed MSF. Our effort focuses on minimizing the communication among different processors without constraining the local growth of a processor's computed subtree. In effect, we achieve a scalability that previous approaches lacked. We implement our algorithm in CUDA, running on a GPU and study its performance using real and synthetic, sparse as well as dense, structured and unstructured graph data. Our experimental study demonstrates that our algorithm outperforms the previous state-of-the-art GPU-based MSF algorithm, while being several order of magnitude faster than sequential CPU-based algorithms.

Categories and Subject Descriptors G.2.2 [*Discrete Mathematics*]: Graph Theory—Graph algorithms, Network problems; E.1 [*Data Structures*]: Graphs and networks

General Terms Algorithms, Experimentation, Performance

Keywords Parallel Graph Algorithms, Minimum Spanning Forest, GPU

1. Introduction

A spanning tree of a connected graph G is an acyclic subgraph of G that connects all vertices of G. The Minimum Spanning Tree (MST) problem calls to find a spanning tree of a weighted connected graph G having the minimum total weight [17]. In case the graph is not connected, i.e. consists of several connected components, the problem is generalized to finding the Minimum Spanning Forest (MSF), i.e. a subgraph containing an MST of each component.

MST computation finds applications in domains such as the optimization of message broadcasting in communication networks [10, 12, 22], biological data analysis [34], and image processing [27], while it forms a basis for clustering algorithms [32, 35]. For instance, assume a graph $G(V, E)$ where vertices stand for persons and weighted edges for the cost of communication among them, in which we wish to spread some news at the minimum cost in real time. We then need to efficiently compute an MST of G.

Past research has proposed several sequential MST algorithms, starting out with Borůvka's seminal work [9]. Highlights of this research are Kruskal's [18], Prim's [26], and the Reverse-Delete [20] algorithms. Other MST algorithms may have lower asymptotic complexity, but larger hidden constants [24]. While existing algorithms are reasonably efficient, modern applications call for algorithms that can scale well to very large and high-density graphs, including complete graphs, as in the case of computing the MST in Eucledian space, which finds applications in hierarchical clustering [23]. Parallel processing comes into play to achieve this objective. Thus, a large body of literature is devoted to parallel MST algorithms [14, 19]. Nevertheless, such works use specialized hardware that is not so readily available as low-cost and easily-programmable Graphics Processing Units (GPUs); GPUs are commonly installed on today's home computers, workstations, consoles, and gaming devices. Several pieces of work have exploited the GPU's ubiquity to suggest high-performance, general data processing algorithms therefor [15, 16, 21]. In a similar spirit, Vineet et al. have already offered a data parallel version of Borůvka's MST algorithm adopted for a GPU [31]. However, Borůvka's algorithm is ill-chosen if the objective is to solve the MST computation in a scalable manner for dense graphs[1], as its performance is known to deteriorate in comparison to Prim's algorithm as graph density grows [13]. To date, two parallel adaptations of Prim's algorithm have been proposed [8, 19]; however, out of these, [8] allows for limited parallelism, as it does not allow two growing subtrees to touch each other, while [19] necessitates costly inter-processor communication to merge subtrees when they do get in contact. Thus, there is still a need for a size-scalable and density-scalable GPU-based MST computation algorithm.

In this paper we respond to this need; we propose a novel Parallel MSF Algorithm (PMA) for undirected weighted graphs, which adapts Prim's algorithm while eschewing the drawbacks of [8] and

[1] By dense we refer to those graphs with a number of edges ten times larger than the number of vertices ($|E| > 10 \times |V|$).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
Copyright © 2012 ACM 978-1-4503-1160-1/12/02...$10.00

[19]; in contrast to [8], it allows for full-scale flexible parallelism; still, unlike [19] it raises a much lower communication overhead. We implement PMA on the GPU[2] using the CUDA programming framework [2]. Our experimental study on both real and synthetic graphs verifies that our algorithm outperforms previous GPU-based MSF algorithms.

2. Related Work

The MSF problem was first formulated and solved by Borůvka [9]. Subsequently, three altervative algorithms were proposed, namely Kruskal's [18], Prim's [26] and the Reverse-Delete [20] algorithm. All these algorithms exploit two properties of the MSF [30]; the *cycle property*, which maintains that the heaviest edge in a cycle does not belong to the MSF; and the *cut property*, which maintains that, for every subset of the graph's vertex set, $C \subset V$, the lightest edge with one vertex in C and the other vertex in $V \setminus C$ belongs to the MSF.

The Reverse-Delete algorithm, based on the cycle property, iteratively removes the heaviest edge that does not break any graph component's connectivity. Likewise, Kruskal's algorithm iteratively adds the lightest edge that does not introduce a cycle. Prim's algorithm selects an arbitrary vertex and iteratively inserts the lightest edge from the current subtree to an unvisited vertex, based on the Cut property. Borůvka's algorithm differs from Prim's algorithm in starting from all vertices at once, and expanding all running subtrees at each iteration.

Several theoretical results highlight potential parallelism in MST computation, most of them do not lead to efficient practical algorithms as they incur large constant factors [7]. Thus, most practical parallel MST algorithms are merely parallelized versions of classical sequential algorithms, adapted for specific hardware architectures or programming models. As Borůvka's algorithm is naturally prone to parallelization, most of these works adapt that algorithm, sometimes in combination with Kruskal's or Prim's algorithm [7]. Chung and Condon [11] propose a parallel version of Borůvka's algorithm for asynchronous, distributed-memory machines. Borůvka's algorithm is divided into five steps and parallelize each step. Dehne and Götz [14] propose the Borůvka Mixed Merge (BMM) algorithm, which lets each processing unit find a local MST for its stored edges sequentially, and then prunes and merges the resulting partial MSTs at a single unit using a balanced D-ary tree.

The work that most related to ours, [8], stands between Prim's and Borůvka's algorithms. This MST-BC algorithm lets each processing unit run Prim's algorithm starting from different vertices simultaneously, marking the vertices in its own MST and coloring all neighbors of marked vertices. These local MSTs grow until a conflict occurs, i.e. one unit reaches a vertex marked or colored by another unit. Then, the conflicting unit starts building a new MST from another unvisited vertex. The algorithm terminates when all vertices are either colored or marked and merges the resulting local MSTs. MST-BC degenerates to Borůvka's algorithm for P processing units equal to the number of vertices n, and to Prim's algorithm for one only processing unit. The algorithm in [19] is a relaxed version of MST-BC for the transactional memory model, differing therefrom in the way it treats conflicts. When a conflict occurs, the two parties involved switch to a Merge state to resolve the conflict (see state transition diagram in Figure 1), with one unit appending the other's MST to its own, while the other starts building a new tree from another unvisited vertex; on a connected graph, this algorithm ends up having only one thread working on the final MST, hence reduces parallelism. Last, Vineet et al. [31] recently adapted Borůvka's algorithm for the GPU, using parallel primitives. This algorithm provides the current state of the art for GPU-based MSF computation in terms of efficiency; however, it cannot scale to large numbers of edges.

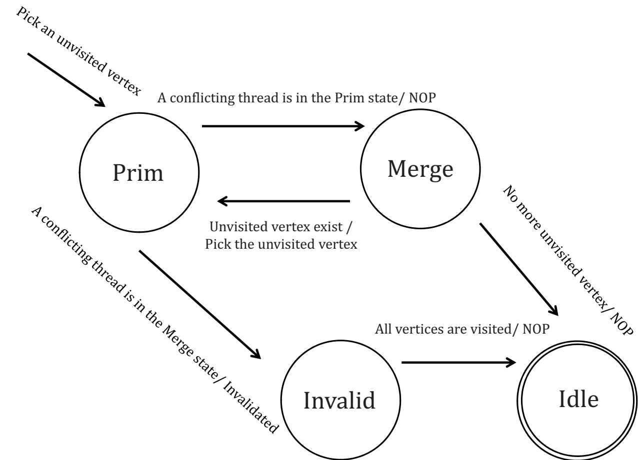


Figure 1. The state transition diagram of Kang and Bader's algorithm [19].

3. Motivation

We observe that most existing approaches to parallel MSF computation share a similar intuition: They build different trees in parallel, and, when conflicts occur (i.e. different trees run into each other), they merge the components and start over. However, this strategy is not equally efficient on all types of graphs; MST-BC [8] fares well with sparse graphs, its relaxation [19] manages well graphs with large diameter (thanks to a heuristic that may result in less conflicts), [14] does well only with sufficiently dense graphs, and [31], being an adaptation of Borůvka's algorithm, is challenged by high graph density; besides, the question of finding the MSF of a non-connected graph is mostly ignored. Most significantly, these approaches tend to be cautiously conservative when expanding their trees, as they have to be alert to potential conflicts, invoking too many redundant iterations that deteriorate their performance. Unfortunately, the attempt to alleviate this conservatism in [19], by merging the trees when a conflict occurs, ends up substituting that problem with another, as it raises the inter-processor communication cost and results into an unbalanced load as progressively fewer processors are left building fewer MSTs, forsaking the benefits of parallelism. Starting out from the next section, we present an alternative, elegant solution that eschews the conservatism of such methods, takes full advantage of parallelism, and keeps communication cost low.

4. Parallel MSF Algorithm

Algorithm 1 shows the overall design of our PMA algorithm, while Figure 2 depicts its state transition diagram. Given an undirected weighted graph $G = (V, E)$, PMA first performs a tailored version of Prim's algorithm, Partial Prim (PP), in parallel on P processors. Then it *unifies* each connected set of subtrees produced by PP into a single vertex and removes all self-loops. This process is repeated until no more edges are left. The union of the sets of edges returned by all PP executions is the desired MSF.

[2] Since each thread of PMA runs the same set of instructions on multiple data thus meeting the requirements of the Single Instruction Multiple Thread architecture of GPU. GPU allows running thousands of threads concurrently with low overhead.

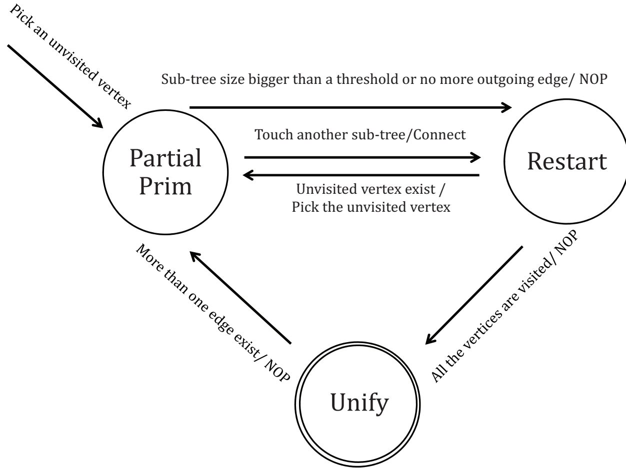


Figure 2. The state transition diagram of the PMA algorithm.

Algorithm 1: PMA algorithm

Input: $G(V, E)$: An undirected weighted graph; P : Number of processors;
Output: The MSF of graph G
1 **while** $(|E| > 0)$ **do**
2   Initialize the $successor$ arrays ;
3   Perform Partial Prim on P processors;
4   Compact each connected component into a vertex;

4.1 Partial Prim

Algorithm 2 outlines our Partial Prim (PP) algorithm. PP finds an unvisited vertex s, and builds a tree from it as in Prim's algorithm, iteratively finding the lightest outgoing edge from the running subtree. Our implementation keeps the list of vertices in the current subtree in Q, and, at each iteration, goes through Q and finds the lightest outgoing edge to expand the subtree with. A vertex's $successor$ value represents the root of the subtree that contains it, or -1 if it is unvisited. During the parallel execution of PP, two different processors may try to visit the same vertex at the same time; thus, we use the atomic test-and-set instruction on the $successor$ array to avoid conflicts. Another conflict can occur when a processor p building a subtree rooted at s visits a vertex v that has already been visited; then p connects its subtree with that of v by setting $successor[s] = successor[v]$, stops building that tree, and picks another unvisited vertex to build a tree from. Section 4.3 proves the correctness of the algorithm in connection to this step.

4.2 Unification step

After the termination of PP on all processors, every vertex is visited and belongs to exactly one subtree. PMA then unifies all vertices in the same subtree into one vertex, by tracing the $successor$ array. In the resulting graph, it removes all self-loops and all redundant parallel edges, keeping only the lightest ones. After these operations, it is possible to be left with connected, non-island vertices (i.e., with non-eliminated edges); this is due to the fact that, after two subtrees mutually touch each other, both of them stop growing prematurely. Thus, even after a unification step, PMA (Algorithm 1) repeats its basic iteration, until all edges have been eliminated; only then is the MSF construction finished and the algorithm terminates.

Algorithm 2: Partial Prim algorithm

Input: $G(V, E)$: An undirected weighted graph;
$successor$: The successor array, shared among all processors;
γ: Maximum size that a sub-tree can grow ($\gamma \geq 2$)
Output: A part of the MSF of G
1 **while** *there is an unvisited vertex* s **do**
2   **Atomic** $successor[s] = s$ if it is not set;
3   **if** $successor[s]$ *was set by another processor* **then**
4   **Continue**;
5     $Q = \{s\}$;
6   **while** $|Q| < \gamma$ **do**
7   Find the lightest edge $e = (u, v)$ such that $u \in Q$ and $v \notin Q$;
8   **if** *no more edge* e **then**
9   **Break**;
10   Include e in the global MSF;
11   **if** $successor[v]$ *not set* **then**
12   **Atomic** $successor[v] = s$;
13             $Q = Q \bigcup \{v\}$;
14   **else**
15             $successor[s] = successor[v]$;
16   **Break**;

4.3 Proof of Correctness

The correctness of our PMA algorithm emanates from the *Cut property* of the Minimum Spanning Forest.

Lemma 1 (Soundness)**.** *Each subtree constructed by each processor with the PP algorithm is a subset of the MSF of G.*

Proof. Under the assumption that edge weights are distinct, regardless of the vertex at which we start Prim's algorithm, we will build exactly the same MST in a connected component. The only difference between our PP algorithm and the standard Prim algorithm is that PP stops expanding the tree under certain conditions; thus, any constructed subtree is a subset of the MSF of G. □

Lemma 2 (Completeness)**.** *The unification step in PP does not remove any edge belonging to the MSF of G.*

Proof. We can think of the unification step as repeatedly unifying two vertices at a time. We unify two vertices if and only if we have selected an edge between them in the MSF of G. It follows that any other parallel edges between these two vertices cannot be in the MSF of G, thus removing self-loop edges of the unified vertex is safe. After unifying all possible vertices, we only remove parallel edges that are not the lightest ones between two new vertices; these edges cannot be in the MSF of G either. □

Theorem 1. *PMA constructs the MSF of the input graph G.*

Proof. The proof follows from Lemma 1 and Lemma 2. □

Theorem 1 proves the correctness of our PMA algorithm. However, it is still unclear why there are no cycles in the resulting MSF while we allow subtrees to touch each other independently. The following property provides further intuition on this question.

Property 1 (Touch property)**.** *If subtree T_1, constructed by PMA, touches subtree T_2 at edge e, then it stops growing. Subtree T_2 can continue to grow until it either reaches its desirable component size γ, touches another subtree T_3, or touches T_1. In the last case,*

assuming edge weights are distinct, the edge leading from T_2 to T_1 is the same edge e.

Proof. Since e is chosen by PP, it is the lightest edge between T_1 and $V \setminus T_1$. In turn, when T_2 touches T_1, as it is also expanded by PP, it uses the lightest edge between T_2 and $V \setminus T_2$, say e'. Assume $e' \neq e$. Then the weight of e' should be lighter than that of e, since e' was chosen by PP for expanding T_2 while e was also available on its boundary. However, e had been also chosen by PP for expanding T_1 while e' was available on its boundary as well; thus, the weight of e should be lighter than that of e'. By reductio ad absurdum, and the distinct weight assumption, it follows that $e' = e$. □

4.4 Complexity Analysis

We now analyze the complexity of PMA. We start with estimating the number of iterations of the loop in Algorithm 1. We ignore vertices with degree 0, as they do not affect the MSF of G. With all processors running PP, each vertex is visited exactly once and is then compacted into a single vertex with at least one more vertex either in the same subtree or in another subtree. Thus, each iteration of the loop reduces the number of vertices by at least half, hence the loop runs at most $log_2(|V|)$ times. Besides, each edge in G is checked by at most two processors (each visiting one of the two end points of that edge), at most γ times. Thus, the total work of Algorithm 2 is $O(2\gamma|E|)$ = $O(|E|)$ when γ is a small constant. Putting it all together, the total work of PMA is $O(|E|\ log_2(|V|))$. While having the same complexity as other parallel MSF algorithms, PMA allows each processor to grow its subtree without incurring costly merge and synchronization work. Thus, by increasing γ, we can increase the subtree size, and hence decrease the iterations of Algorithm 1 at the only cost of increasing the computation in line 7 of Algorithm 2.

5. Implementation

We now discuss some implementation details of PMA. We assume that the input graph is represented as an adjacency list. This data structure consists of a vertex list (an array of vertices that stores for each vertex, its index and a pointer to its adjacent vertices in the edge list) and an edge list (that stores for each edge, its weight and the index of the endpoint vertex of the edge). Our implementation assumes that the graph, adjacency list, fits into the GPU memory.

5.1 Graphic Processing Units

Although the GPUs are designed for graphics processing tasks, their performance, availability and ease of use render them an excellent platform for executing general-purpose algorithms. A modern GPU consists of several multiprocessors, each designed to execute hundreds of computing threads in parallel efficiently, with a zero-overhead thread switching capability and a small amount of high-performance on-chip shared memory. With a design of each multiprocessor similar to SIMD architecture, the GPU works best with fine-grain parallelism. To fully utilize the computing power of graphics hardware, it is desirable for a program to have a very high level of parallelism, in the order of tens of thousand of threads. At the same time, the communication between the GPU and the CPU memory, going through the slow PCI-Express bus, should be minimized. The main memory residing on the graphics card is shared among all processors. As such, the GPU can be treated as a shared memory architecture. To facilitate synchronization among different threads on the GPU, atomic operations such as the test-and-set instruction can be used.

We use the CUDA programming model [2] to program the GPUs. In order to exploit the advance features of GPU, we break our algorithm to different building blocks to leverage the parallel primitive algorithms, namely prefix sum, stream compaction, and sorting, as intermediate components of PMA. These primitives have been implemented and optimized for CUDA [3, 28, 29] by coalescing the memory accesses, using shared memory, avoiding bank conflicts, memory paddings and alignments and other GPU dedicated optimizations like unrolling loops. We use these parallel primitive algorithms in the PMA implementation to better utilize the graphics hardware.

In the following, we refer to a CUDA thread as a processor.

5.2 Partial Prim implementation

Each processor running Partial Prim needs to pick one unvisited vertex and grow a tree. When a growing tree is terminated, another unvisited vertex is picked. To reduce the conflict among P processors in picking vertices we divide the set of vertices into P partitions of equal size. Each processor only picks vertices in its own partition to start growing a tree. This operation is still done atomically since a processor growing its tree might visit a vertex in another's partition. The most important step in PP is to pick the lightest outgoing edge to grow from those in the list Q containing the vertices in a processor's current tree. We examine different ways for doing so.

Algorithm 3: MinPMA algorithm

Input: $G(V, E)$: An undirected weighted graph; Q : the list of vertices in the current sub-tree;
Output: The lightest edge $e(u, v)$ such that $u \in Q$ and $v \notin Q$

```
1 minW = ∞ ;
2 for Each u in Q do
3     for Each edge e(u, v) of u do
4         if successor[u] ≠ successor[v] and
          e.weight < minW then
5             minE = e;
6             minW = e.weight;
7 Return minE;
```

5.2.1 MinPMA algorithm

The first approach (Algorithm 3) is to go through the list of vertices in Q, and, for each vertex u, through its adjacent edges, and pick the lightest one with a destination $v \notin Q$. To check if a vertex v is in Q or not, we check whether $successor[v]$ is the same as the root of the current tree or not. By this approach, we have to go through the list of adjacent edges of each vertex in Q up to $|Q|$ times, thus its complexity is $O(\gamma|E|)$.

5.2.2 SortPMA algorithm

Our second approach (Algorithm 4) tries to alleviate the disadvantage of MinPMA. We first sort the list of adjacent edges of each vertex by increasing weight[3]. Then, whenever we look for the lightest edge going out from vertex u, we pick the first edge that does not have a destination in Q. By recording the previously chosen edges for each vertex u in Q, we only have to go through the list of adjacent edges of each vertex at most once. Thus, the cost of this step becomes $O(|E|)$, at the cost of sorting the edges, which takes $O(|E|\ log(|E|))$. The latter cost can render SortPMA more costly than MinPMA for small $|Q|$ or large $|E|$. Thus, for very sparse

[3] PMASort uses the parallel radix sort proposed in [28]. However, PMASort needs only adjacent edges of every vertex to be sorted in ascending order of their weights, not the whole edge list. Therefore, In order to have the edge list partially sorted, we first sort the whole edge list in ascending order by the weights then we sort the result in ascending order of the starting vertices.

Algorithm 4: SortPMA algorithm

Input: $G(V, E)$: An undirected weighted graph, with the list of adjacent edges of each vertex sorted by weight; $Last$: Last minimum outgoing edge found for each vertex. Q : the list of vertices in the current sub-tree;
Output: The lightest edge $e(u, v)$ such that $u \in Q$ and $v \notin Q$

```
1  minW = ∞ ;
2  for Each u in Q do
3      for Each edge e(u, v) of u starting from Last[u] do
4          if successor[u] ≠ successor[v] then
5              if e.weight < minW then
6                  minE = e;
7                  minW = e.weight;
8              Last[u] = e ;
9              Break;
10 Return minE;
```

graphs, MinPMA fares better than SortPMA. On the other hand, for dense graphs, PP has a low level of parallelism, and, as a result, SortPMA becomes faster as most of the work is done in the sorting step, which runs efficiently on the whole edge list.

5.2.3 HybridPMA algorithm

As MinPMA is efficient for certain graphs and SortPMA for others, we combine the strengths of both in a hybrid approach, HybridPMA, where we use MinPMA for the first iteration and SortPMA for the rest. The rationale for this choice is that, in the later iterations, the graph gets a lot denser as multiple vertices get unified into one.

In order to pick the lightest outgoing edge, one can argue in favour of using a heap. We empirically observed that using heap for the purpose of finding the lightest outgoing edge in the list Q is much slower than the MinPMA algorithm. For instance, assume that the maximum subtree size (γ) is 10, then the MinPMA algorithm goes through the list of outgoing edges for each vertex at most 10 times. On the other hand, when we use heap, each time we find the lightest outgoing edge for a vertex, we need several rounds of min extraction to overlook the non-outgoing edges (i.e. edges that were visited). In addition, the overhead of creating multiple heaps for each vertex and the need of going through more heaps by a thread when its queue grows bigger, make using heap more costly than just going through all the edges and finding min, as in the MinPMA algorithm.

Algorithm 5: Unifying algorithm

Input: $G(V, E)$: An undirected weighted graph; $successor$: The successor array, shared among all processors;
Output: The simplified graph G with each connected component unified into one vertex

1. Find the root of the component for each vertex ;
2. Compute the new vertex indices ;
3. Update the starting and ending vertices of the edges ;
4. Remove self-loop edges ;
5. **Return** minE;

5.3 Unification implementation

The unification step is common to most parallel MSF algorithms. We opt to sacrifice the total work complexity a little, so as to achieve better parallelism and better utilize the GPU power. Our implementation is inspired from the merging algorithm of Vineet et al. [31]. The unification process with total work $O(|E|log(|V|))$ is explained in Algorithm 5. First, we use the distance doubling technique on the $successor$ array to find the root of the component for each vertex in the graph. We also have to remove cycles in the $successor$ array, created two subtrees touching each other at the same time, as in [31]. Having done that, we mark the root vertices as 1 and other vertices as 0, and perform a parallel prefix sum to compute the new vertex index for each connected component, and the total number of components. We then update the start and end vertex of all edges with the new component indices. Self-loop edges are removed. We then sort the edge list by start vertex so as to bring edges with the same starting point together. Thus, all edges that need to be removed are pushed to the end of the edge list, so we can easily remove them and compute the new number of edges. In case when the graph is dense, even if we can reduce the vertices by half, we still end up with almost the same number of edges, most of them being parallel edges between the same vertices (see Figure 3). In such a case, it is worth removing the parallel edges. To do that, we sort edges first by starting vertex, then by ending vertex, and finally by weight. Then we can easily identify the non-minimal parallel edges, and remove them using the stream compaction primitive.

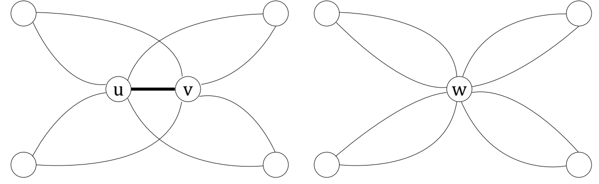


Figure 3. The graph (a) before and (b) after unifying u and v.

5.4 Implementation notes

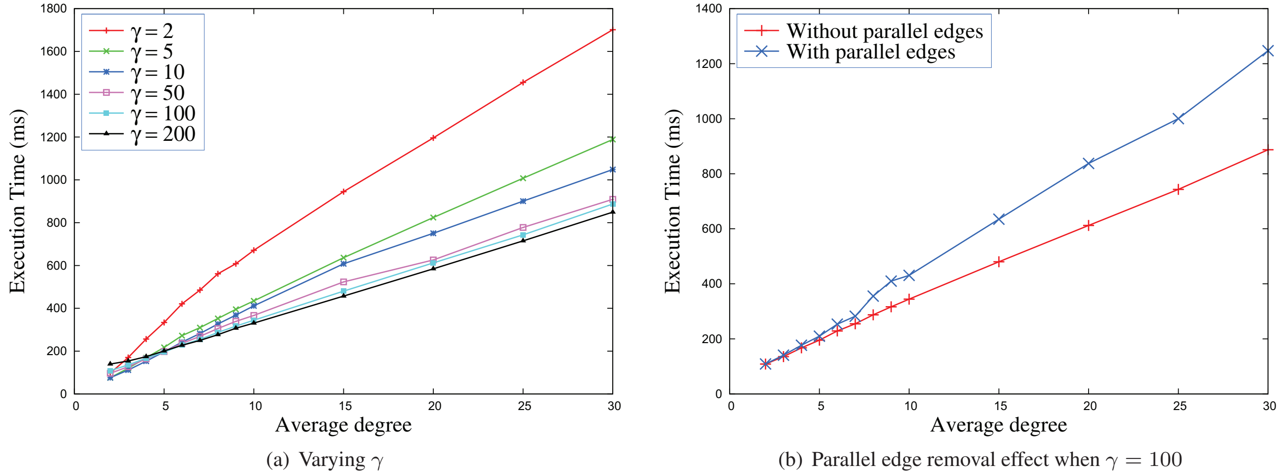
We note that vertices of degree 0 contribute nothing in the MSF, so we should better remove them all before running the algorithm. We incorporate this logic in the Unification step too, removing any new vertices with no outgoing edge.

In MinPMA each time we find the minimum outgoing edge from a vertex u in Q to a vertex outside Q, we bring that edge to the top of the edge list of u. Next time when we need to scan the edge list of u, we first check its first edge; if it still goes out of Q, it is still the minimum outgoing edge.

6. Performance Evaluation

We now compare different implementations of PMA to Vineet et al's algorithm (Vineet) [31] and Bader and Cong's algorithm (BC) [7], the two state-of-the-art parallel algorithms tuned for GPU and CPU respectively. All runtimes are reported on an Intel Xeon 5420 $2.4Ghz$ workstation with $8GB$ of memory. The GPU algorithms are executed using CUDA Toolkit 3.1 running on the NVIDIA Tesla S1070 server with a single Tesla card. When measuring the execution time, for all algorithms, we exclude the time to copy the input graph to the GPU memory and the resulting MSF back to main memory (since this time is negligible[4] and independent of the algorithm). For the sequential algorithms, we use the Boost library [1].

[4] For instance, the time to copy the input graph to the GPU memory and the resulting MSF back to main memory for the New York graph(0.7 million edges) is 0.42 milliseconds and for the USA-West(15 million edges) graph is 6.45 milliseconds.



(a) Varying γ

(b) Parallel edge removal effect when $\gamma = 100$

Figure 4. Execution time of HybridPMA on Erdős-Rényi graphs, varying average degree.

(a) Erdős-Rényi graphs

(b) R-MAT graphs

(c) SSC#2 graphs

(d) WS graphs

Figure 5. Experiments on varying average degree for four types of graph, $|V| = 1M$.

6.1 Datasets

We use both real and synthetic graphs. Our real data are the DIMACS USA road networks data [4] (also used in [31]) and large network data from the SNAP library [6]; for our synthetic data, we use the Georgia Tech graph generator suite [5] to generate the R-

Graph	Vertices	Edges	Execution Time (ms)						
			CPU			GPU	GPU		
			Prim	Kruskal	Boruvka	Vineet	MinPMA	SortPMA	HybridPMA
New York	264K	733K	183	216	541	29	18	27	18
San Francisco	321K	800K	207	255	477	30	19	27	19
Colorado	435K	1M	290	359	592	38	25	37	24
Florida	1.07M	2.7M	780	987	2139	79	51	81	50
Northwest USA	1.2M	2.8M	835	1111	1576	83	59	86	58
Northeast USA	1.52M	3.8M	1120	1571	2811	112	81	118	80
California	1.8M	4.6M	1451	2003	3246	137	103	144	102
Great Lakes	2.7M	6.8M	2076	3004	4832	193	158	213	157
USA-East	3.5M	8.7M	2752	3944	6964	242	195	267	193
USA-West	6.2M	15M	4852	7158	12052	430	346	469	343
USA-Central	14M	33.9M	- [1]	-	52306	-	912	1170	900
USA-Full	23.9M	57.7M	-	-	46890	-	1378	1909	1361
Arxiv Astro Physics	19K	7.9M	93	347	354	39	32	30	30
Penn Road network	1M	6.17M	1603	5522	3315	180	108	169	106
Amazon co-purchasing	410K	6.7M	1457	5923	24047	274	175	218	164
Google graph	876K	10.2M	2616	9730	59671	334	240	268	231
Internet topology	1.7M	22.2M	7000	24629	1408810	934	839	799	797
US Patents Citation	3.8M	33M	-	-	97175	2434	1482	1494	1221

Table 1. Runtime of different CPU and GPU algorithms on real-world networks.

MAT and SSCA#2 graphs and the techniques proposed in [25] to produce Erdős-Rényi (ER) and also Watts and Strogatz (WS) random graphs. WS model [33] generates random graphs with small-world properties in two steps. WS first draws a ring lattice, n vertices each connected to its k nearest neighbors. Then, in an ER fashion selects an edge, i.e. sampling the edges with probability p, from the so-called ring and rewires the edge.

6.2 Maximum subtree size (γ)

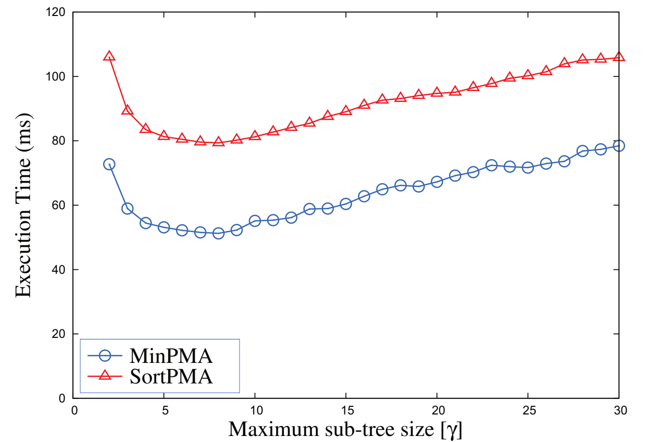
The maximum subtree size is the maximum number of vertices of the subtree that a thread is allowed to grow. This maximum allowed subtree size plays a crucial role in PMA. If $\gamma = 2$, PMA behaves like Vineet, while if $\gamma = |V|$, PMA degenerates to Prim's algorithm running sequentially. This is so because, to ensure that each processor has a fair chance to grow its subtree up to γ, we set the P (number of CUDA threads) to be $\frac{|V|}{\gamma}$.

Figure 7(a) shows the execution time of MinPMA and SortPMA with varying γ, processing the Florida road network, a very sparse graph with $|V| \approx 1M$ and $|E| \approx 2.7M$. Both implementations of PMA speed up as γ increases, reaching their peak performance with$\gamma = 8$. This is so because, as the subtree gets bigger, finding the lightest outgoing edge becomes more costly. We also note that MinPMA outperforms SortPMA for very sparse graphs.

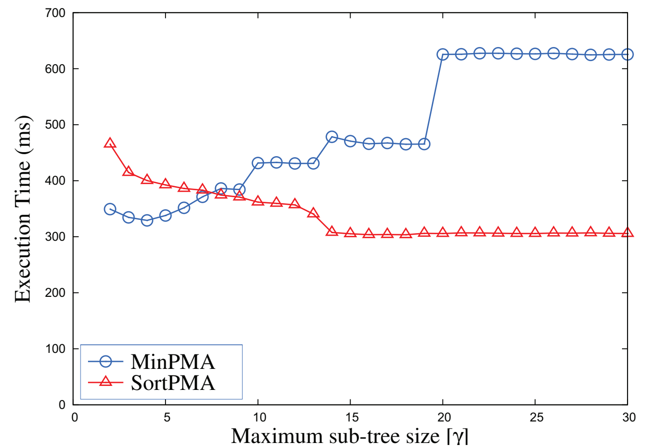
Figure 7(b) examines the behavior of PMA on very dense graph, an R-MAT graph with 10K vertices and approximately 11M edges. For such a graph, the chance for two processors to conflict is very high. The performance of SortPMA peaks at $\gamma = 14$, and is unchanged after that, most likely because no processor can grow a subtree bigger than 14. On the other hand, due to the high cost of finding the minimum outgoing edge, the performance of MinPMA degrades as γ grows. MinPMA continues to slow down as γ grows because, as we increase γ, we decrease P. SortPMA is not much affected by this because sorting dominates the execution time. These results suggest that for sparse graphs, a small value of γ yields good performance, while for dense graphs, a bigger γ might be better.

To verify this trend, we run HybridPMA, on Erdős-Rényi Random graphs with varying average degree and $|V|$ fixed to $1M$ (Fig-

[1] Crashed, e.g. out of memory

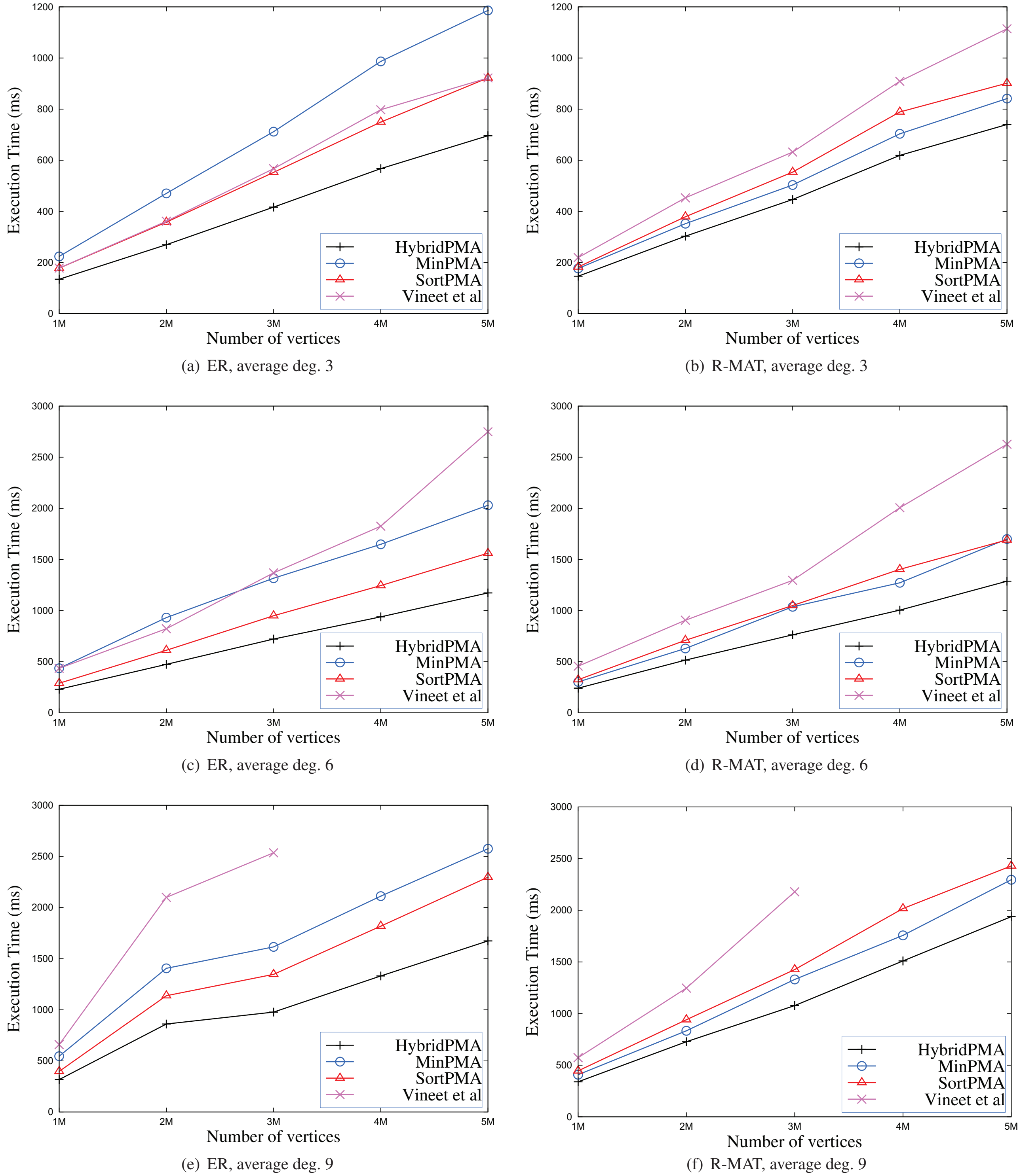


(a) Florida graph



(b) R-MAT graph

Figure 7. Execution time of PMA with varying γ.



(a) ER, average deg. 3

(b) R-MAT, average deg. 3

(c) ER, average deg. 6

(d) R-MAT, average deg. 6

(e) ER, average deg. 9

(f) R-MAT, average deg. 9

Figure 6. Experiments on varying the number of vertices.

ure 4(a)). We observe that, for small average degree, a small γ of 5 or 6 gives the best performance, whereas for bigger average degree, a γ as big as 200 has an advantage.

However in the whole set of our experiments, we observe that very small γ like $\gamma = 2$ is always not efficient for any kind of graph. The reason is if $\gamma = 2$ we just find one edge for each vertex, while if we increase the γ we can find more edges (maximum $\gamma - 1$ for each Partial Prim). On the other hand, when we have a relatively large γ like $\gamma = 1000$ we need to pay the cost of iterating through the neighbors of γ vertices. Therefore both small (2) and (relatively)

large (1000) γ do not yield good performance. We observe that the optimal maximum subtree size depends on the average degree of the graph and the graph structure, such as average shortest path length, average degree and the centralities.

6.3 Removing parallel edges

We also examine the effect of removing parallel edges to the performance of PMA. As discussed, after each unification step, we might end up with a lot of parallel edges. Figure 4(b) shows the execution time of HybridPMA with and without parallel edge removal. It is clear that removing parallel edges significantly improves the performance and, the denser the graph, the more significant is the performance boost. For the rest of the experiments, we always run PMA with parallel edge removal.

6.4 Reduction rate

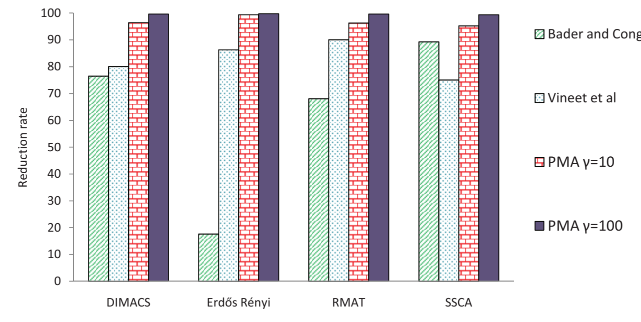


Figure 8. Reduction rate of different algorithms

The three compared parallel algorithms, namely BC, Vineet, and PMA, all have a similar workflow: first, some subsets of the MST are computed, then the connected components are unified into one vertex; if any edges remain the algorithm iterates. However, PMA is less conservative than Bader and Cong's algorithm, as it allows the subtrees it constructs to touch each other, and thus has much fewer components in the next iteration. To demonstrate this effect, Figure 8 shows our measurements of the *reduction rate*, i.e., the rate of vertices unified after the first iteration, with four different types of graphs: a DIMACS graph (Florida), an Erdős-Rényi graph ($|V| = 1M$, $|E| = 30M$), an R-MAT graph ($|V| = 1M$, $|E| = 10M$) and an SSCA graph ($|V| = 1M$, $|E| = 8M$). We have implemented Bader and Cong's algorithm in CUDA, using atomic operations instead of the coloring technique, thus allowing the subtrees to grow a little bit further. The number of processors used is very small to further reduce conflicts. Still, the observed reduction rate of Bader and Cong's algorithm is much lower than that of PMA, especially in dense graphs where the chance of conflict is high. Vineet also exhibits lower reduction rate than PMA, sometimes even lower than Bader and Cong's algorithm. In PMA, larger γ (100 vs 10) improves the rate. Since Bader and Cong's algorithm has very low reduction rate, especially when the number of processors is large, it becomes very inefficient on a massively multithreaded architecture like the GPUs. Thus, we do not include it in the rest of our evaluation.

6.5 Performance comparison

We now compare the performance of different PMA implementations to Vineet [31]. As the implementation of Vineet we obtained cannot handle disconnected graphs, in some cases we add extra edges to render the graphs connected. Table 1 shows the execution time on real world networks. These are mostly very sparse graphs and thus Vineet using Borůvka's algorithm is quite efficient. Nevertheless, in all cases, PMA is faster, up to 2 times for large graphs like the network of citations among US Patents. Besides, HybridPMA is faster than both MinPMA and SortPMA. Different sequential algorithms on CPU using Boost C++ library [1] are included for the sake of illustration. As GPU implementations are orders of magnitude faster than CPU implementations, we omit the CPU algorithms from the rest of our presentation.

Figure 5 shows the runtime on different types of synthetic graphs when we fix the number of vertices at 10^6 and vary the average degree (or number of edges). For all four type of graphs, PMA outperforms Vineet, and the speed up increases as the graphs get denser. This result reconfirms the advantage of using Prim's algorithm over Borůvka's algorithm.

In Figure 6, we measure runtime when we fix the average degree and vary the number of vertices in the graph. For Erdős-Rényi (ER) graphs, when the degree is small, Vineet performs quite close to PMA. However, as the average degree grows, the gap between them increases too, with PMA taking the lead. In addition, Vineet's implementation has a limitation on the maximum number of edges it can handle, so it cannot run some of the very big graphs. Varying the number of vertices in R-MAT graphs presents a similar trend, while now PMA is substantially faster even on very small average degree, as there are still many vertices with high degree in R-MAT graphs.

7. Conclusions

This paper proposed the first, to our knowledge, size- and density-scalable parallel algorithm for Minimum Spanning Forest computation. Our PMA algorithm, based on Prim's algorithm, avoids the conservatism and inter-processor communication cost of earlier approaches, while it is tailored for implementation on a GPU. The key to this achievement is that we allow different processors to grow their partial MSTs unimpeded, while conflicts are handled smoothly without raising extra communication demands. Our algorithm, implemented on a GPU, outperforms the previous state-of-the-art GPU-resident parallel MSF algorithm, which is built on Borůvka's algorithm.

A crucial underlying assumption for our algorithm is that edge weights are distinct. Still, our algorithm can be easily adapted to handle duplicate edge weights, by appending a unique number to the end of each edge weight; such appending does not affect the total weight of the MSF.

References

[1] Boost C++ graph library. http://www.boost.org.

[2] CUDA Zone: Toolkit & SDK. http://developer.nvidia.com/what-cuda.

[3] CUDPP. http://cudpp.googlecode.com.

[4] The Ninth DIMACS challenge on shortest paths. http://www.dis.uniroma1.it/∼challenge9/.

[5] GTgraph - A suite of synthetic random graph generators. https://sdm.lbl.gov/ kamesh/software/GTgraph/.

[6] Stanford large network dataset collection. http://snap.stanford.edu/data.

[7] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In *IPDPS*, 2004.

[8] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *JPDC*, 66(11):1366–1378, 2006.

[9] O. Boruvka. O jistém problému minimálním (about a certain minimal problem). *Práce mor. prírodoved. spol. v Brne III*, pages 37–58, 1926.

[10] J. Carle and D. Simplot-Ryl. Energy-efficient area monitoring for sensor networks. *Computer*, 37(2):40–46, 2004.

[11] S. Chung and A. Condon. Parallel implementation of boruvka's minimum spanning tree algorithm. *Parallel Processing Symposium, International*, 0:302, 1996.

[12] J. Cong, L. He, C.-K. Koh, and P. H. Madden. Performance optimization of VLSI interconnect layout. 21(1-2):1–94, 1996.

[13] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. Introduction to algorithms. MIT Press, third edition, 2009. ISBN 0070131511.

[14] F. Dehne and S. Götz. Practical parallel algorithms for minimum spanning trees. In *SRDS*, 1998.

[15] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD*, 2004.

[16] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, 2006.

[17] E. Horowitz and S. Sahni. Fundamentals of computer algorithms. *Potomac, Md., Computer Science Press*, 1978.

[18] J. Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7:48–50, 1956.

[19] S. Kang and D. A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *PPoPP*, 2009.

[20] J. Kleinberg and Éva Tardos. The minimum spanning tree problem. In *Algorithm Design*. Pearson/Addison-Wesley, Boston, 2006.

[21] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing*, 2001.

[22] X.-Y. Li, Y. Wang, and W.-Z. Song. Applications of k-local mst for topology control and broadcasting in wireless ad hoc networks. *IEEE TPDS*, 15(12):1057–1069, 2004.

[23] W. B. March, P. Ram, and A. G. Gray. Fast euclidean minimum spanning tree: algorithm, analysis, and applications. In *KDD*, 2010.

[24] B. M. E. Moret and H. D. Shapiro. An empirical assessment of algorithms for constructing a minimal spanning tree. *In DIMACS Monographs*, 1994.

[25] S. Nobari, X. Lu, P. Karras, and S. Bressan. Fast random graph generation. In *EDBT*, 2011.

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

[27] P. W. Rafael C. Gonzalez. *Digital Image Processing*. Addison-Wesley, 1987.

[28] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS*, 2009.

[29] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware*, 2007.

[30] R. E. Tarjan. Data structures and network algorithms. *Society for Industrial and Applied Mathematics, Philadelphia*, 1983.

[31] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the gpu. In *HPG*, 2009.

[32] X. Wang, X. Wang, and D. Mitchell Wilkes. A divide-and-conquer approach for minimum spanning tree-based clustering. *IEEE TKDE*, 21 (7):945–958, 2009.

[33] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, (393):440–442, 1998.

[34] Y. Xu, V. Olman, and D. Xu. Clustering gene expression data using a graph-theoretic approach: an application of minimum spanning trees. *Bioinformatics*, 18(4):536–545(10), 2002.

[35] C. Zhong, D. Miao, and R. Wang. A graph-theoretical clustering method based on two rounds of minimum spanning trees. *Pattern Recogn.*, 43(3):752–766, 2010.