Contents

4.4 Minimum Spanning Tree

The problem of finding the minimum spanning tree of a graph is typical of combinatorial problems which exhibit fine-grained irregular parallelism. Researchers have been frustrated in their attempts to exploit this parallelism and for many types of graph the fastest known algorithms are serial. To evaluate our check pointing technique we measure the performance of a concurrent implementation of a minimum spanning tree algorithm that uses entangled Immutable Data Structures. We find that our concurrent implementation does not perform as well as a serial implementation.

The problem is to determine the minimum spanning tree of a connected undirected graph with weighted edges. The minimum spanning tree of a graph is an acyclic sub-graph which connects all of the vertices and has the minimum weight.

Sedgewick explains the problem in detail and describes a number of serial algorithms for computing the minimum spanning tree [Sed02]. The minimum spanning tree problem is one of the most important in combinatorics. Ahuja describes how many problems in network routing and linear programming are related to the problem of finding the minimum spanning tree [AMO93].

A minimum spanning tree is the tree of edges $T \in G(V, E)$ with minimal weight:

$$W(T) = \sum_{(u,v) \in T} W((u, v))$$

where $W((u, v))$ is the weight of an edge (u, v).

The main contribution of this section is the evaluation of an algorithm which uses entangled Immutable Data Structures to facilitate the check pointing of speculative execution. This section focuses on comparing the times taken to determine the minimum spanning tree of an undirected planar graph.

4.4.1 Experiment

Prim describes an algorithm to determine the minimum spanning tree of a graph [Pri57]. To evaluate our check pointing technique we compare the performance of a concurrent implementation that uses entangled Immutable Data Structures with a concurrent implementation that uses Software Transactional Memory. We also compare these concurrent implementations with their serial counterparts.

Prim's algorithm is typically implemented using a mutable adjacency list to represent the graph and its minimum spanning tree and a priority queue from which minimally weighted edges are chosen. The implementation records whether edges belong to the minimum spanning tree by storing a value, which is usually referred to as a colour, as an edge property in the adjacency list. We call this a *Serial Graph Colouring Implementation* of Prim's algorithm. We use the adjacency list and the Serial Graph Colouring Implementation of Prim's algorithm from the Boost graph library. Siek describes the format of the adjacency list in detail [SLL01].

Section 4.4.3 describes the experimental set up.

Section 4.4.4 describes the Serial Graph Colouring Implementation of Prim's algorithm.

We develop an implementation of Prim's algorithm that uses a set, instead of graph colouring, to represent the minimum spanning tree. We call this a *Serial No-Colouring Implementation* of Prim's algorithm. This serial implementation is used to measure the effect that maintaining the minimum spanning tree in a set, rather than in the adjacency list, has on the execution time of the algorithm. We use a data structure from the C++ standard template library to implement the set of edges representing the minimum spanning tree and we also use a priority queue from the standard library [Jos99]. The graph is implemented by an immutable adjacency list from the Boost library.

Section 4.4.5 describes the Serial No-Colouring Implementation of Prim's algorithm.

A Concurrent Graph Colouring Implementation of Prim's algorithm must ensure the correctness of concurrent accesses to the edge colours.

Section 4.4.6 explains why a Concurrent Graph Colouring Implementation of Prim's algorithm that executes efficiently on a Chip Multi-Processor is difficult to construct.

Kang developed a concurrent implementation of Prim's algorithm using Software Transactional Memory [KB09]. We call this a *Concurrent Graph Colouring Implementation* of Prim's algorithm. The implementation allows some speculative execution by lazily detecting conflicting accesses to the graph colours.

Section 4.4.7 describes Kang's Concurrent Graph Colouring Implementation of Prim's algorithm.

We develop a concurrent implementation of Prim's algorithm which uses entangled Immutable Data Structure to allow check pointing, backtracking and rollback to a previous state of the algorithm. We call this a *Concurrent No-Colouring Implementation* of Prim's algorithm. The implementation uses an immutable set, to represent the minimum spanning tree, and an immutable priority queue, from which minimally weighted edges are chosen. Both of these data structure are specialisations of the Canonical Binary Tree. The data structures are entangled to facilitate check pointing. The graph is implemented by an immutable adjacency list from the Boost library.

Section 4.4.9 describes the Concurrent No-Colouring Implementation of Prim's algorithm.

4.4.2 Results

Our experiment shows that the Concurrent No-Colouring Implementation of Prim's algorithm takes longer to determine the minimum spanning tree of a graph than either the Serial Graph Colouring Implementation or the Serial No-Colouring Implementation for all graph sizes. The Serial No-Colouring Implementation of the algorithm takes about twice as long as the Serial Graph Colouring Implementation for all graph sizes.

Figure 4.1 illustrates a comparison of the elapsed time taken to determine the minimum spanning tree of a graph.

The Concurrent No-Colouring Implementation does not return the memory used by the Immutable Data Structures because they are persistent. Only 32 GB of memory are available to contain the persistent data structures on the evaluation hardware and this limited the maximum size of the graph whose minimum spanning tree could be determined to 2^{19} vertices.

The topology of the graphs representing the road maps of urban states differs from those of more rural states. This accounts for some of the variation in elapsed time taken to calculate the minimum spanning tree of states with similar numbers of vertices.

This thesis does not make any claims about the absolute performance of Immutable Data Structures. However, even when using 8 hardware threads the Concurrent No-Colouring Implementation takes longer to calculate the minimum spanning tree than either serial algorithm.

Section 4.4.10 describes how the performance of the Concurrent No-Colouring

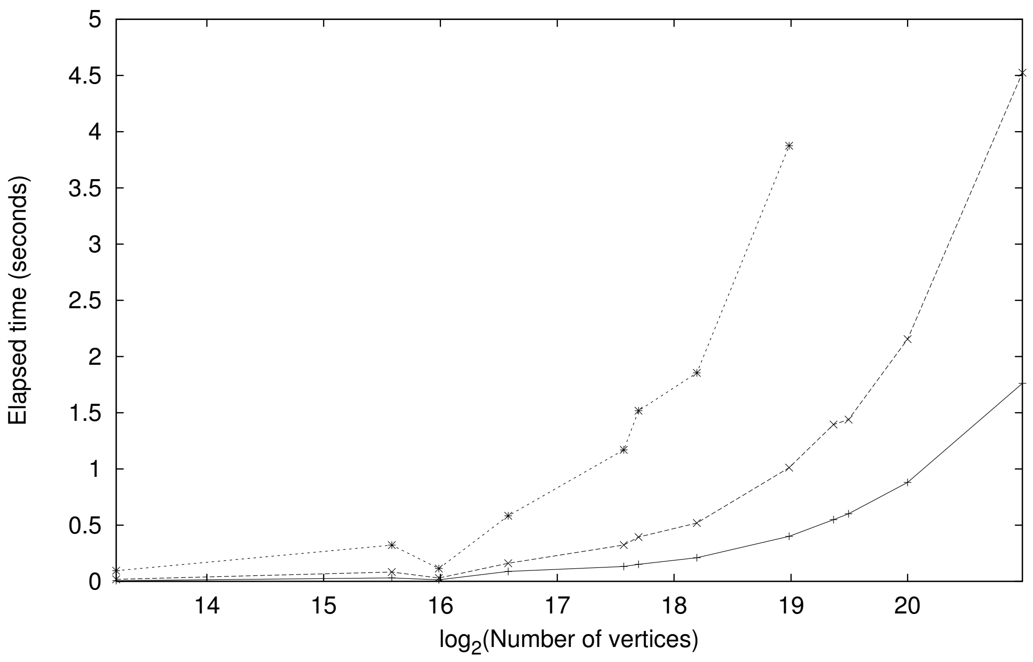

Figure 4.1: **Comparison of the elapsed time taken to calculate the minimum spanning tree** of planar undirected graphs representing road maps of US states.
The elapsed time taken by the Serial Graph Colouring Implementation (+), the Serial No-Colouring Implementation (x) and the Concurrent No-Colouring Implementation (*) is plotted against varying graph sizes. We uses a log scale to represent the number of vertices in the graph.
Eight hardware threads participate in the concurrent execution. Each hardware thread executes on a dedicated processor. Figures given are the mean of 10 measurements.

Implementation can be improved.

Kang provided results for a Concurrent Graph Colouring Implementation which uses Software Transactional Memory [KB09]. Kang measured the elapsed time taken to calculate the minimum spanning tree of a planar graph with 2^{22} edges. Unfortunately, we were not able to calculate the minimum spanning tree of a graph of this size so we cannot make a direct comparison with Kang's result.

When a single hardware thread was used the elapsed time taken to determine the minimum spanning tree was 1143 seconds. When using 8 hardware threads, on the same core, a 14X speed-up was achieved. Kang attributed this super-linear speed-up to the sharing of cache by the hardware threads. When using 64 hardware threads, on 8 cores, a speed-up of 61.5X was achieved.

Kang attributed 98.5% of the execution time of the Concurrent Graph Colouring Implementation to the overhead of Software Transactional Memory. To overcome this overhead 64 hardware threads were applied to the problem. Kang concluded that "even with this level of scalability, our parallel algorithm runs only at the comparable speed to the single-threaded case which does not incur the Software Transactional Memory overhead" [KB09].

This thesis claims that the use of Immutable Data Structures can make concurrent programming easier. Kang described the difficulty of ensuring the correctness of the Concurrent Graph Colouring Implementation which uses locks to ensure serialisable access to the graph colours "this [acquiring locks] can lead to many complex scenarios that can cause race conditions, deadlocks, or other complications, and it is far from trivial to write correct and scalable code" [KB09]. Our Concurrent No-Colouring Implementation of Prim's algorithm requires no synchronisation. It is not prone to race conditions, because all shared data is immutable and it is not prone to deadlock because it does not block. Our Concurrent No-Colouring Implementation of Prim's algorithm was simpler to develop than the Concurrent Graph Colouring Implementation described by Kang.

4.4.3 Method

Demetrescu describes a set of planar graphs, representing road maps, which were used during the DIMACS implementation challenge competition [DGJe09]. These graphs are widely accepted as benchmarks for evaluating minimum spanning tree algorithms. Each graph node represents an intersection between roads and each graph edge is weighted with the distance between intersections. The graphs have an average of 2.7 edges per vertex.

We evaluate our algorithm using a Sun Ultra Sparc T2 server [vRV+09]. The server contains a single Niagara Chip Multi-Processor which implements simultaneous multi-threading, it has 64 hardware threads and 8 physical processors.

Both the hardware and the graph data sets used in our evaluation are identical to those used by Kang [KB09].

We use an immutable priority queue implemented by a Directed min-tree specialisation of the Canonical Binary Tree. The Canonical Binary Tree is balanced but none of the optimisations, suggested in section **??** are implemented. We also use an immutable set implemented by an interval tree specialisation of the Canonical Binary Tree.

4.4.4 Serial Graph Colouring Implementation

Prim's algorithm is based on an observation known as the graph cut property [Pri57]. A graph cut partitions the vertices of a graph into two disjoint sets. Given a cut in the graph, any edge between the two sets which has a minimum weight belongs to some minimum spanning tree of the graph.

Prim's algorithm grows a minimum spanning tree iteratively from a single graph edge. The algorithm maintains a set of crossing edges, called the fringe, which is the set of edges with one vertex in the growing minimum spanning tree and one outside it.

Initially, both the minimum spanning tree and the fringe are empty. A single vertex is added to the minimum spanning tree and all of the edges from this vertex are added to the fringe. At each iteration the minimally weighted edge is removed from the fringe and added to the growing minimum spanning tree. This adds a new vertex to the minimum spanning tree and all of the edges from this vertex to vertices that are not already in the minimum spanning tree are then added to the fringe. The resulting fringe is processed by the next iteration. The algorithm completes when the set of vertices outside the minimum spanning tree is empty and edges in the minimum spanning tree connect all of the nodes in the graph.

Prim's algorithm can be implemented by using an adjacency list to represent the graph. Each graph edge has an associated weight. The minimum spanning tree is represented by a colour indicator associated with each edge. The fringe is represented by a priority queue which contains references to edges in the adjacency list. The priority associated with an edge is the weight of that edge.

Serial Graph Colouring Implementations of Prim's algorithm are among the fastest known. Typically, high performance Serial Graph Colouring Implementations focus on improving the performance of the priority queue.

4.4.5 Serial No-Colouring Implementation

Prim's algorithm can also be implemented by using a set to maintain the growing minimum spanning tree instead of colouring graph edges. We call such an implementation a Serial No-Colouring Implementation because the adjacency list does not have any mutable properties. The set contains references to edges in the adjacency list and is used to determine whether an edge is part of the minimum spanning tree.

The Serial No-Colouring Implementation of Prim's algorithm uses three data structures. The graph is represented by a constant adjacency list with weighted edges. The fringe is represented by a priority queue which orders edges by weight. The minimum spanning tree is represented by a set of edges. In our implementation the priority queue is implemented using a vector from the standard library and the set is implemented using the standard map [Jos99].

The algorithm starts from a single node and adds edges to the minimum spanning tree iteratively. The edge with the minimum weight is removed from the priority queue and added to the set to indicate that it is part of the minimum spanning tree. This adds a new vertex to the minimum spanning tree. The edges including this vertex, which are not already present in the set, are added to the priority queue to complete the iteration.

To determine whether an edge is part of the minimum spanning tree a set look-up is performed. Typically, set look-up takes $O(log(n))$ time whereas accessing a mutable graph edge takes $O(1)$ time, so there is a significant overhead associated with looking up edges in a set. Consequently, it is not common to implement Prim's algorithm in this way. Our Concurrent No-Colouring Implementation maintains the minimum spanning tree an immutable set and uses a constant adjacency list to avoid sharing mutable data. We implement the Serial No-Colouring Implementation algorithm so that we can measure the effect of maintaining the growing minimum spanning tree in a set.

4.4.6 The concurrent implementation of Prim's algorithm

A concurrent implementation of Prim's algorithm may attempt to combine the minimum spanning trees of sub-graphs, produced by multiple processors, to form a larger minimum spanning tree.

Two sub-graphs are disjoint if they do not have any vertices in common. Two sub-graphs are adjacent if they are disjoint and there is at least one edge joining vertices which belong to different sub-graphs. The minimum spanning trees of adjacent sub-graphs can be combined by including the minimally weighted joining edge in the graph formed by their union. The minimum spanning trees of disjoint sub-graphs can be combined easily only by growing them until they are adjacent. The minimum spanning trees of overlapping sub-graphs are difficult to combine. Ideally, the minimum spanning trees of adjacent sub-graphs should be identified and combined.

The graph cannot be decomposed into disjoint sub-graphs before the algorithm starts because this problem is more difficult than the minimum spanning tree problem itself. Dor proves that the problem of decomposing a graph into disjoint sub-graphs with no common edges is NP-Complete [DT92].

To permit the combination of minimum spanning trees created concurrently an algorithm can check that their sub-graphs are disjoint each time a vertex is added. This imposes a synchronisation overhead on the concurrent implementation. The implementation can speculate that sub-graphs are disjoint to reduce the synchronisation overhead. However, it must be prepared to roll-back the algorithm to the point at which adjacency first occurs.

A concurrent algorithm has the potential to demonstrate speed-up provided it is faster to check and combine the minimum spanning tree of sub-graphs with the growing minimum spanning tree than to grow the minimum spanning tree by the corresponding amount. Once checked, the merging of minimum spanning trees is straightforward. The problem is to create minimum spanning trees in such a way that they can be combined without incurring a significant synchronisation overhead.

4.4.7 Concurrent Graph Colouring Implementation

Kang describes a Concurrent Graph Colouring Implementation of Prim's algorithm which uses Software Transactional Memory [KB09]. Kang's implementation is an application which implements Memory Transactions rather than an application developed within an existing Software Transactional Memory framework.

Memory operations acting on the graph colours can cause race conditions, so simultaneous access must be restricted. A naïve implementation might serialise every access to the colours but this effectively serialises the entire algorithm, negating any benefit from concurrent execution. Kang uses Software Transactional Memory to support speculation by buffering memory operations and detecting conflicts. Conflicting accesses to the graph colours are rare so it can be beneficial to speculate that a conflict did not occur.

Kang's Concurrent Graph Colouring Implementation relies on the semantics of memory transactions to avoid data races. Each thread colours the vertices of its own minimum spanning tree and also colours all of the neighbours of the marked vertices with a unique colour. The process of picking one vertex and then applying an operation to its neighbours is encapsulated in a Memory Transaction. Conflicts are detected by checking the colour of vertices in graph to determine whether another processor has included the node in its minimum spanning tree.

Our experimental results are directly comparable to those of Kang because we use identical graph data sets and identical hardware. Unfortunately, Kang was not able to report concurrent speed-up because the overheads associated with Software Transactional Memory exceed the benefits of concurrent execution.

4.4.8 Previous work

Borůvka described a concurrent algorithm to determine the minimum spanning tree of a graph nearly a century ago [NMN01]. However, realising speed-up from concurrent execution has proved difficult. Chazelle describes an algorithm which has the minimal amortised time [Cha00]. Vineet describes an algorithm that makes use of a graphics processing unit. This speeds-up the calculation of some very large minimum spanning trees by an order of magnitude when compared with a serial implementation [VHPN09]. In practice, the fastest methods for finding the minimum spanning tree of a dense graph are based on a serial implementation of Prim's algorithm. Bazlamacci presents a survey of high performance minimum spanning tree algorithms [BH01]. In the fastest, the fringe is represented by a priority queue based on a Fibonacci heap. Weiss describes the implementation of a Fibonacci heap data structure in detail [Wei93].

Dice describes a Concurrent Graph Colouring Implementation of Prim's algorithm which uses Hardware Transactional Memory [DLMN09]. Dice uses a form of Hardware Transactional Memory known as speculative lock elision, which permits speculative access to the graph colours, while relying on hardware to detect conflicting accesses [RG01]. Dice implements this algorithm on the Sun ROCK processor [CCE+09]. The implementation difficulty and the modest speed-up observed may have been factors contributing the cancellation of the ROCK processor, which we described in section **??**.

4.4.9 Concurrent No-Colouring Implementation

We develop a Concurrent No-Colouring Implementation of Prim's algorithm. One processor is designated as the main processor and it grows the minimum spanning tree of the entire graph. The other processors are designated as helper processors and they build the minimum spanning trees of sub-graphs. The main processor occasionally checks whether the minimum spanning trees of these sub-graphs overlap with the growing minimum spanning tree. When overlap is detected the sub-graphs produced by the helper processors are rolled-back to the state they were when they were adjacent to the growing minimum spanning tree. They are then combined with the growing minimum spanning tree and their fringes are added to the fringe of the growing minimum spanning tree. The helper processors contribute to reducing the elapsed time taken to calculate the minimum spanning tree.

We use the Serial No-Colouring Implementation of Prim's algorithm on the main processor because it makes the merging of the sub-graphs built by the helper processors easier. However, we could have chosen a Serial Graph Colouring Implementation, in which case only the main processor would access the graph colours.

The helper processors create minimum spanning trees in such a way that the execution of their algorithm can be rolled back to a previous state. The algorithm executing on the helper processors is also a Serial No-Colouring Implementation of Prim's algorithm. It uses an immutable set to identify edges in the minimum spanning tree and an immutable priority queue to represent the fringe. Both of these Immutable Data Structures are specialisations of the Canonical Binary Tree. The immutable set and the priority queue are entangled so that they can both be rolled-back to a mutually consistent state.

The Immutable Data Structures are entangled by storing the root of the immutable priority queue in the leaves of the immutable set. Each leaf contains an edge and a reference to the root of the past version of the priority queue from which it was removed. The leaf also contains the corresponding root of the past version of the set. This entanglement check points the state of the algorithm at the start of each iteration. The check point allows the set and the priority queue to be restored to a consistent state in which the set represents a minimum spanning tree and the priority queue its fringe.

At some moment in time the minimum spanning trees of sub-graphs built in isolation are checked by the main processor and possibly combined with the growing minimum spanning tree. The frequency at which the minimum spanning trees of sub-graphs are checked is a heuristic of the algorithm which does not affect its correctness.

If necessary, the main processor examines the set produced by a helper processor and backtracks through past versions by traversing the leaves of the set. The algorithm must backtrack to the moment in time when the first common edge was added. An ordinal number is used to determine the first common edge. Each iteration of Prim's algorithm performed by the helper processor causes a process-unique ordinal number to be incremented. This ordinal number is stored in a leaf of the set. The first common edge is the common edge with the lowest ordinal number.

The main processor does not block the execution of the helper processors while backtracking. The data structures produced by the helper processors are immutable and can be examined by the main processor without requiring synchronisation.

When backtracking detects overlap the state of the algorithm is rolled-back to the point at which the first common edge was added. The traversal finds the leaf containing a common edge with the lowest ordinal number. This leaf contains a reference to the past version of the set which represents the minimum spanning tree of a sub-graph which is adjacent to the growing minimum spanning tree. This leaf also contains a reference to the version of the priority queue which represents the fringe of the minimum spanning tree of the sub-graph.

A merge process combines the past version of the minimum spanning tree of the sub-graph with the growing minimum spanning tree and the past version of the fringe of the sub-graph with the fringe of the growing minimum spanning tree. The helper processor is stopped after the merge to reduce contention in the path to memory.

4.4.10 The performance of the Concurrent No-Colouring Implementation

The performance of a concurrent implementation of Prim's algorithm is dependent on both the topology of the graph and the choice of starting vertices. Heuristics can guide the choice of starting vertices used by the helper processors. Our concurrent implementation chooses the starting vertices for each processor at random. We have focused exclusively on planar graphs, so the performance of the Concurrent No-Colouring Implementation when applied to other graph topologies remains to be investigated.

The elapsed time taken by the Concurrent No-Colouring Implementation of Prim's algorithm is dependent on heuristics such as the frequency at which the main processor checks whether the minimum spanning trees of sub-graphs, produced by the helpers, overlap with the growing minimum spanning tree. The checking process is performed by the main processor. There is a trade off between the frequency of checking and the benefit from merging a minimum spanning tree produced by a helper. We found that the best results were obtained when the main processor checked for overlap infrequently. Our implementation adds 2^{10} nodes to the growing minimum spanning tree between checks. There is little chance of overlap when the minimum spanning trees are small and little to be gained from merging sub-graphs when the growing minimum spanning tree is near completion. Checking for overlap is probably most advantageous when about one quarter of the planar graph is covered by the minimum spanning trees of sub-graphs. However, we did not attempt to find the optimum interval because it is dependent on the topology of the graph.

When processing a large graph a high proportion of memory accesses result in cache misses. Our implementation uses only 8 hardware threads, one on each physical processor. We found that using additional hardware threads did not reduce the elapsed time taken to determine the minimum spanning tree. This indicates that the elapsed time is bound by the performance of the memory subsystem. Jacob describes how the Niagara processor in the Sun Ultra Sparc T2 server has four memory controllers and uses fully buffered DIMM memory to permit fast processing for frequent cache misses [Jac09]. However, we also carried out experiments using an Intel core i7 system. We found that for graphs of less than 2^{18} vertices the elapsed time taken by the Intel system was less than that obtained by processing the same graph on the Sun Ultra Sparc T2. The restricted memory available on the Intel system prevented a comparison for larger graphs.

During the execution of Prim's algorithm edges are checked for inclusion in the minimum spanning tree before being added to the fringe. However, when fringes are merged the resulting fringe can contain edges in common with the merged minimum spanning trees. These edges make the fringe larger than necessary and introduce redundant iterations of the algorithm. Periodic fringe compaction can improve the performance of the algorithm by removing these edges from the priority queue.

The immutable priority queue implementing the fringe can be compacted by creating a new version which does not contain any edges in common with the set implementing the minimum spanning tree. Compaction does not affect the Entanglement between the data structures, it makes subsequent versions of the priority queue smaller but it does not return memory. Our backtracking algorithm requires that all past versions of both the priority queue and the set are retained.

We chose not to compact the fringe of the minimum spanning tree grown by the main processor. Instead, we compact the fringes of minimum spanning trees grown by the helper processors. The immutable priority queue is compacted after adding 2^{10} nodes. We did not attempt to find the optimum interval between fringe compactions because it is dependent on the topology of the graph.

The Sun Ultra Sparc T2 server has 32 GB of main memory. Our algorithm does not return any memory so the size of graphs considered during the evaluation are restrained by the available memory. There are many ways that the memory restraint could be lifted. For example, the helper processors could return the memory occupied by a sub-tree after it is merged with the growing minimum spanning tree.

The size of the node used to implement the Canonical Binary Tree and the number of nodes accessed while balancing of the tree are important factors affecting the performance of the No-Colouring Implementations because they contribute to the effective memory bandwidth of the implementation.

Section **??** describes how node size affects the performance of algorithms which use specialisations of the Canonical Binary Tree.

Bibliography

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[BH01] Cuneyt F. Bazlamacci and Khalil S. Hindi. Minimum-weight spanning tree algorithms a survey and empirical study. *Computers & Operations Research*, 28(8):767 – 785, 2001.

[CCE$^+$09] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A high-performance Sparc CMT processor. *IEEE Micro*, 29(2):6–16, 2009.

[Cha00] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000.

[DGJe09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson (eds.). *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. American Mathematical Society, 2009. DIMACS Series in Discrete Mathematics and Theoretical Computer Science.

[DLMN09] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 157–168. ACM, March 2009.

[DT92] Dorit Dor and Michael Tarsi. Graph decomposition is npc - a complete proof of holyer's conjecture. In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 252–263, New York, NY, USA, 1992. ACM.

[Jac09] Bruce L. Jacob. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[Jos99] Nicolai M. Josuttis. *The C++ Standard Library: A tutorial and reference.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[KB09] Seunghwa Kang and David A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *PPOPP*, pages 15–24, 2009.

[NMN01] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar Borůvka on Minimum Spanning Tree Problem, translation of both the 1926 papers. *Discrete Math.*, 233:3–36, April 2001.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technology Journal*, 36:1389–1401, 1957.

[RG01] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[Sed02] Robert Sedgewick. *Algorithms in C++, part 5: graph algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[SLL01] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library User Guide and Reference Manual.* Addison-Wesley Professional, December 2001.

[VHPN09] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the gpu. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 167–171, New York, NY, USA, 2009. ACM.

[vRV$^+$09] Vladimir Čakarević, Petar Radojković, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. Characterizing the resource-sharing levels in the UltraSPARC T2 processor. In *Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 481–492, New York, NY, USA, 2009. ACM.

[Wei93] Mark Allen Weiss. *Data structures and algorithm analysis in C*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993.