GPU ACCELERATED RECTILINEAR STEINER TREE CONSTRUCTION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

DİNÇER ÖZCAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

DECEMBER 2015

Approval of the thesis:

GPU ACCELERATED RECTILINEAR STEINER TREE CONSTRUCTION

submitted by **DİNÇER ÖZCAN** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Gönül Turhan Sayan _____
Head of Department, **Electrical and Electronics Engineering**

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı _____
Supervisor, **Electrical and Electronics Eng. Dept., METU**

Examining Committee Members:

Prof. Dr. Gözde B. Akar
Electrical and Electronics Engineering Department, METU _____

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Engineering Department, METU _____

Assoc. Prof. Dr. Ece Güran Schmidt
Electrical and Electronics Engineering Department, METU _____

Assoc. Prof. Dr. İlkay Ulusoy
Electrical and Electronics Engineering Department, METU _____

Assist. Prof. Dr. Kayhan İmre
Computer Engineering Departmentt, Hacettepe University _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: DİNÇER ÖZCAN

Signature :

ABSTRACT

GPU ACCELERATED RECTILINEAR STEINER TREE CONSTRUCTION

ÖZCAN, DİNÇER
M.S., Department of Electrical and Electronics Engineering
Supervisor : Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı

December 2015, 82 pages

The Rectilinear Steiner Tree (RST) problem is one of the fundamental problems in circuit design automation. The problem is to find the tree structure that connects all points in the input set such that total tree length is minimized. In order to reduce the total length, extra points, called Steiner points, are introduced to the tree. Since the RST problem is NP-complete, developing heuristic algorithms which can produce near optimal solutions is the primary approach to solve the problem. This thesis accelerates the Modified RST algorithm through parallelizing and using a state of art Graphics Processing Unit (GPU) platform. GPUs contain many computational units and they can provide massive computational power. However, it is not trivial to map an RST problem instance to GPU. In order to benefit from the resources of GPU, the problem has to be parallelized and suitably implemented.

In this study, we thoroughly investigate two recent rectilinear Steiner tree algorithms, RST and Modified RST, and identify parallel implementation opportunities. Modified RST algorithm is speed oriented and has better performance especially on large problem instances. Moreover, it is suitable for paralel implementation. Thus, we propose a paralel and scalable RST solution based on Modified RST algorithm, which paralellizes the whole algortihm and utilizes GPU resources more efficiently. Computational results for realistic applications and random generated benchmarks are presented to evaluate our implementation. Significant speed up is observed especially for large problem sizes.

Keywords: Graph Theory, Rectilinear Steiner Tree, Routing, GPGPU, CUDA

ÖZ

GPU HIZLANDIRMALI DOĞRULU STEINER AĞAÇ ÜRETİMİ

ÖZCAN, DİNÇER
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Cüneyt F. Bazlamaçcı

Aralık 2015 , 82 sayfa

Doğrulu Steiner Ağaç problemi elektriksel tasarım otomasyon alanının temel problemlerinden birisidir.Problem, verilen nokta kümesindeki noktaları kullanarak en kısa uzunluktaki ağaç yapısını bulmayı amaçlamaktadır. Bu amaçla ağaca Steiner noktası adı verilen yeni noktalar eklenmektedir. Doğrulu Steiner ağaç problemi NP-tam olduğu için yaklaşık çözüm veren algoritmalar ana çözüm yaklaşımı olarak görülmektedir. Bu tez değiştirilmiş RST algoritmasının GPU platformlarında paralelleştirilerek hızlandırılmasını sağlamaktadır. GPU platformları çok sayıda işlemci çekirdeği içermektedir ve GPU yüksek sayıda sayısal işlemi aynı anda gerçekleştirebilecek yeteneğe sahiptir. Ancak, GPU'nun kaynaklarının verimli kullanılabilmesi için doğrulu Steiner ağaç problemi uygun şekilde paralelleştirilmelidir.

Bu çalışmada, öncelikli olarak iki güncel Steiner ağaç üretim yaklaşımı olan RST ve Değiştirilmiş RST algoritmaları detaylı olarak incelenmiştir ve paralel uygulamaya uygun adımları tespit edilmiştir. Değiştirilmiş RST algoritması hız temellidir ve özellikle büyük problem boyutlarında daha iyi performans elde etmektedir. Ayrıca, Değiştirilmiş RST algoritması paralel uygulamalar için de uygundur. Bu sebeplerden dolayı Değiştirilmiş RST yöntemi temel alınarak yeni bir paralel ve ölçeklenebilir doğrulu Steiner ağaç algoritması önerilmiştir. Önerilen algoritma bütün çözüm adımlarını paralelleştirmektedir ve GPU kaynaklarını verimli kullanmaktadır. Son olarak, algoritmanın performansını değerlendirmek amacı ile gerçek uygulamalar ve rastlan-

tısal üretilmiş test kümeleri için simulasyon sonuçları sunulmuştur, özellikle büyük problem boyutları için önemli hızlanma değerleri gözlemlenmiştir.

Anahtar Kelimeler: Çizge Kuramı, Doğrulu Steiner Ağaç Problemi, GPGPU, CUDA

To my Family

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı for all his guidance and helpful advices throughout my study.

I would like to thank to my family for their love and support throughout my life. This thesis would not have been possible without their support and encouragements.

I would like to thank to my employer, ASELSAN, for supporting my study.

I also express my gratitude to TÜBİTAK BİDEB "National Scholarship Program for MSc Students"

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit
B1S	Batched 1-Steiner
BGA	Batched Greedy Algorithm
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FRST	Full Rectilinear Steiner Tree
FST	Full Steiner Tree
GPU	Graphics Processing Unit
HGP	Hierarchical Greedy Preprocessing
I1S	The Iterated 1-Steiner
LCA	Lease Common Ancestor
MRST	Minimum Rectilinear Steiner Tree
MST	Minimum Spanning Tree
OARSMT	Obstacle Avoiding Rectilinear Steiner Minimal Tree
RMST	Rectilinear Minimum Spanning Tree
RSG	Rectilinear Spanning Graph
RSMT	Rectilinear Steiner Minimum Tree
RST	Rectilinear Spanning Tree
SPR	Shortest Path Region
SMT	Steiner Minimal Tree
VLSI	Very Large Scale Integration

CHAPTER 1

INTRODUCTION

The Steiner tree problem constructs a tree with minimum cost that spans a set of nodes in the region. The set of input nodes are called vertices or terminals and newly added nodes to tree to minimize the total cost are called Steiner points, named after Jakob Steiner, a Swiss mathematician. The history of the Steiner tree problem starts with Fermat (1601-1665) and there is still an ongoing interest in modern heuristic solutions.

1.1. Overview

The Steiner tree problem has wide usage areas in telecommunication network design and printed circuit board design including integrated circuit design. Due to its importance in VLSI design, there is a significant attraction recently to the solution of the Steiner minimal tree problem. In VLSI design, following the placement process, routing is an important step. During routing, the paths for net on chip layout are constructed to interconnect the pins of the circuit blocks. Although IC manufacturing technology has introduced important abilities for net routing, there is still a minimum length constraint for critical nets. To be precise, minimum total trace capacitance for critical nets can be obtained and the occupied circuit area can be reduced only by the use of shorter net interconnections.

Only the vertical and horizontal lines can be used as wires between different pins in VLSI design. This special type of problem is called rectilinear Steiner tree problem (RST). The minimum length tree that spans all input terminals in rectilinear plane is called rectilinear minimum spanning tree (RMST). By introducing extra nodes, namely Steiner points, the total length of RMST can be reduced. Although there exist algorithms for finding the optimal solution of rectilinear Steiner minimal tree (RSMT) problem, it is NP-complete and there is no polynomial time solution. Hence,

there is much more interests in near-optimal solutions and several numbers of heuristic algorithms have been proposed for this problem.

Nowadays, performance requirements of applications are continuously increasing and hence the industry is moving towards parallel computing. This trend also affects researchers in the graph theory community. Since 2000, the interest on sequential RSMT solution shrunk considerably. On the contrary, there is more effort on parallel solutions of the problem. The state-of-the-art RSMT algorithms may benefit from data parallelism advantage of GPUs to accomplish more efficient algorithms.

1.2. Scope of the Thesis

In this study, we propose a parallel approach for the rectilinear Steiner tree problem. We start our study with a literature review. First, we analyze optimal and near-optimal RSMT solutions considering runtimes and solution qualities of the existing algorithms. Among existing work, we have chosen Modified RST algorithm [1] as the basis of our study. Modified RST algorithm is a combination of Rectilinear Steiner Tree by Zhou [2] and Batched Greedy Algorithm (BGA) by Kahng [3]. Modified RST is suitable for instances having thousands of terminals. Moreover, most of its steps are suitable for parallel implementation.

In our algorithm, as the initial phase of the algorithm, we keep Zhou's rectilinear sparse graph generation approach. Sweep operation can be computed for different nodes and different regions, simultaneously. The sweep operations are distributed among GPU execution units and active set maintenance method is improved for suitable GPU implementation. Hence, using the parallel sweep-line algorithm, sparse graph is generated. However, we replace Kruskal's [4] MST computation phase with Boruvka's algorithm [5] since it is more advantageous for parallel implementation. Finally, similar to Modified RST, Borah's edge substitution method [6] is employed for Steiner point computations and graph updates, which can be performed in parallel since each node-edge pair is independent from each other. A scalable parallel version of Borah's algorithm is implemented. As our implementation environment, Nvidia GPUs and CUDA software platform is selected. The RST solution operations are mainly based on integer arithmetic. Since RST execution does not contain any floating point operations, its parallel implementation is expected to have a high gain

on a GPU which include more hardware resources for integer operations than floating point operations.

1.3. Thesis Outline

The thesis has been organized as follows:

In Chapter 2, the problem definition of the Rectilinear Steiner Tree is stated and a brief history of it is given. Next, the exact and heuristic solutions are presented. The mainstream serial and parallel heuristic algorithms are discussed in this chapter. Furthermore, an overview of basic GPU structure and the use of a general purpose GPU are presented.

In Chapter 3, the RST and Modified RST algorithms are given in detail. The algorithmic steps are explained and the differences of these two algorithms are discussed.

In Chapter 4, details of our proposed parallel algorithm are explained. The amendments in Modified RST employed to make it more appropriate for our parallel implementation are clarified. Finally, our GPU implementation on CUDA platform is presented.

In Chapter 5, a performance evaluation study of the implementation is performed and speed up values are presented for various benchmark problems including randomly generated large size cases.

Chapter 6 summarizes and concludes the study indicating also possible future work.

CHAPTER 2

BACKGROUND AND STATE-OF-THE-ART

2.1. The Rectilinear Steiner Tree Problem

The Steiner tree problem, or Steiner minimal tree problem, is a combinatorial optimization problem that deals with finding the shortest interconnect for a given set of pins, terminals or objects. Given a set P of *n* nodes, or vertices, the Steiner Minimal Tree (SMT) define a set S of Steiner points such that the minimum spanning tree cost over P ∪ S is minimized where the total cost is sum of lengths of all edges [7]. The SMT problem has similar characteristics with minimum spanning tree (MST) problem. The difference between MST and SMT is that SMT introduces extra intermediate vertices and edges to tree in order to reduce the length of spanning tree. The new vertices introduced are known as Steiner points or Steiner vertices. One can find many Steiner trees for a given set of input nodes [8].

In the literature, many different variants and generalizations of the Steiner tree problem has been examined. Some significant types can be listed as follows [9]:

- Euclidean Steiner Tree Problem: Given a set P of *n* nodes in Euclidean plane, find a set S of Steiner points together with MST T for node set P ∪ S such that T has a minimum length in ℓ_2 – norm.
  For nodes a(x_1, y_1) and b(x_2, y_2), the distance between these two nodes in ℓ_2 – norm, can be calculated by d(a, b) = $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
- Rectilinear Steiner Tree Problem: Given a set P of *n* nodes in Euclidean plane, find a set S of Steiner points together with MST T for node set P ∪ S such that T has a minimum length in the ℓ_1 – norm.
  For nodes a(x_1, y_1) and b(x_2, y_2), the distance between these two nodes in ℓ_1 – norm, can be calculated by d(a, b) = $|x_1 - x_2| + |y_1 - y_2|$

- Directed Steiner tree problem: Given an initial directed graph G = (V, E), find a minimum cost tree in G that connects all terminal in V to a given root r.
- Weighted Steiner tree problem: Given an initial undirected and weighted graph G = (V, E), find a spanning tree T in V such that the total weight of edges and vertices in T is minimum.

2.1.1. History of Steiner Tree Problem

The Steiner tree problem is named after mathematician Jakob Steiner (1796–1863). Steiner studied a problem about joining three villages by a system of roads having minimum total length. Steiner provided a systematic solution to the problem while covering its general case [10]. Although Steiner's work on the solution is independent from his predecessors, he is not the first person who analyzed this problem. The work on this problem and its solutions goes much earlier than Steiner's analysis. Pierre de Fermat (1601–1665) proposed this problem to Toricelli (1608–1647). Toricelli solved this problem and passed it along his student Viviani (1622–1703) who published his own solution and Toricelli's solution in 1659. However, the earliest published discussion of Steiner tree problem can be found in a 1647 book by mathematician Cavallieri (1598–1647) [10].

2.1.2. The Rectilinear Steiner Tree Problem

The Rectilinear Steiner tree problem, minimum Rectilinear Steiner tree problem (MRST) or Rectilinear Steiner minimum tree problem (RSMT), is a variation of geometric Steiner tree problem in the plane. In RSMT distances are calculated in the ℓ_1 – norm. The formal definition of the problem can be stated as follow:

Given a set P of *n* nodes, or terminals, in the Euclidean plane, a Rectilinear Steiner tree of P, RST(P), is defined as a set of horizontal and vertical line segments which spans all nodes in P by using shortest network. A sample RSMT for a node set is given in Figure 2-1, the black nodes represent the input terminal set and white node represent the Steiner points added to decrease total length of the spanning tree.

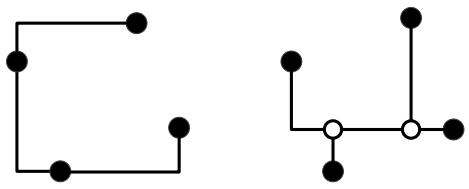

Figure 2-1: MST and RSMT in plane

The distance between pair of nodes a(x_1, y_1) and b(x_2, y_2) is calculated by the Manhattan or rectilinear distance:

$$\text{dist(a,b)} = \Delta x - \Delta y = |x_1 - x_2| + |y_1 - y_2|$$

In a rectilinear Steiner tree, Steiner points can be classified into three groups, specifically, corner point with degree 2, T-point with degree 3 and cross point with degree 4, where the degree is the number of vertical or horizontal line segments incident to a vertex.

2.1.3. Basic Definitions

A full rectilinear Steiner tree (FRST) is a RST and all of its terminal points are leaves. Every Steiner tree can be decomposed into some full Steiner trees and trees that are constructed after decomposition operation are called full components of the Steiner tree. By using two kinds of operation, namely, sliding and flipping, full components can be transformed into others [9]. An example of left slide operation is given in Figure 2-2

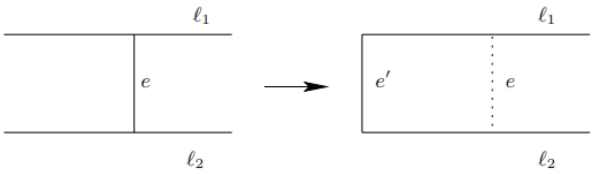

Figure 2-2: Sliding Operation [9]

Flipping operation is that two segments interconnecting at a corner are replaced with new ones where these two segments form a rectangle with new segments. Example flipping operation is shown in Figure 2-3.

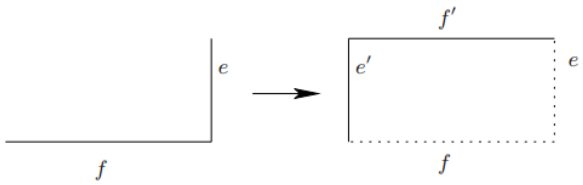

Figure 2-3: Flipping Operation [9]

Equilateral point concept is used for Steiner tree reduction operations. Considering an equilateral triangle with line segment z_0z_1 as one of its sides, the third corner of the triangle, staying at the right side when looking from z_0 towards z_1, is referred as equilateral point. An example is given in Figure 2-4 where e_1 denotes the equilateral point.

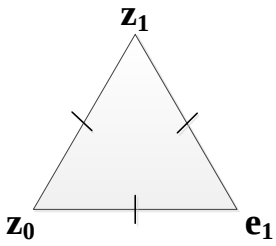

Figure 2-4: Equilateral Point

Depending on the Steiner point addition methodologies, RSMT algorithms can be classified into two basic classes; namely, two-point connection strategy and three-point connection strategy. Method that are based on two-point connection strategy use the principle of connecting a new Steiner point to the current tree via an essential path at each iteration. However, in a three-point connection strategy, at each iteration, a new terminal point is connected to two points already in the tree, which can be either terminal or Steiner points. Hence, for three point strategy, there is no need to maintain the connection from previous iteration since the connections between the two points already in the tree can be changed at the current iteration [16].

In this study, the minimum spanning tree over a node set P is denoted by *MST(P)* and the total cost of a minimum spanning tree is shown as *cost(MST(P))*. Given a set P of input nodes, a 1-Steiner point is defined as any node $x \in P$ such that $cost\big(MST(P \cup \{x\})\big)$ is minimized.

2.1.4. RSMT Algorithms

Many recent research works on RSMT problem are based on the result of previous main works. Hannan is the first who investigated Steiner tree problem in rectilinear distance and proposed a solution for it in 1966. In [11], Hannan presented that for an input node set P, a RSMT can be constructed by selecting Steiner points from Hannan grid. Hannan grid of a finite node set P, H(P), can be obtained from the intersections of all vertical and horizontal lines through each node *n* in P. Hannan grid of a sample node set is displayed in Figure 2-5. For every node, vertical and horizontal lines are drawn through nodes and Steiner points are selected from the intersections of these lines.

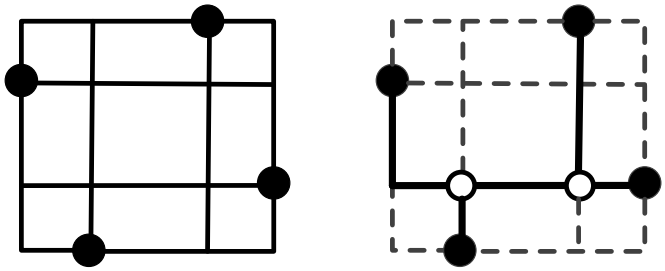

Figure 2-5: Hannan Grid of a Node Set

Although RSMT reduces the cost of the spanning tree, there is a relationship between the total length of the MST and RSMT. Hwang [12] proved that for any node set P, the performance ratio of SMT is limited by equation 2-1.

$$\boldsymbol{Performance\ Ratio = Steiner\ Ratio = \frac{cost(MST(P))}{cost(SMT(P))} \leq \frac{3}{2}} \qquad \textbf{(2-1)}$$

Hwang's outcomes indicate that any Steiner tree solution, which improves upon an initial MST has a performance ratio of, at most, $\frac{3}{2}$.

RSMT problem has received significant attention from researcher; however, despite restricting the Steiner points to stay on Hannan grid, Garey and Johnson [13] proved that RSMT problem is NP-complete and there is no known polynomial time algorithm. Optimal solution can be found only for a few special cases. For instance, there exists a linear time solution when all nodes in P lie on the boundary of a rectangle. Hence, many heuristic algorithms have been suggested in the literature.

2.1.4.1. Exact Algorithms of RSMT

Whenever an exact algorithm is proposed to obtain the optimal solution of an NP-complete problem, runtime is always exponential. However, if the problem size is small enough, some exact algorithms with exponential running time can be considered to be acceptable.

Melzak [14] proposed the first exact algorithm in Euclidean plane in 1961. In Melzak's algorithm, all full Steiner trees (FST) on *n* terminals are enumerated; afterwards, the positions of the Steiner points for each FST are optimized [15]. Reduction operation via equilateral points is applied for the optimization of the locations of Steiner points. Terminal pairs are replaced with an equilateral point where new equilateral point is considered as terminals and reduction operation is repeated. The runtime of Steiner points' location optimization algorithm is $O(2^n)$.

Winter [16] suggested an exact algorithm such that unlike Melzak's algorithm, it enumerates equilateral points instead of FSTs. The algorithm consists of two phases. Initially, full components are computed sufficiently and then using dynamic programming or Branch and Bound, Steiner minimum tree is generated. Salowe and Warme [17] propose another two phase exact algorithm similar to Winter. At the first step, it generates full components, which are then concatenated in an optimal tree.

The fastest exact algorithm in the literature is the GeoSteiner algorithm proposed by Warme, Winter and Zachariasen [16]. GeoSteiner algorithm follows the two phase approach of other exact Steiner minimal tree algorithms; however, it reduces the computations considerably by implicit instead of explicit enumeration of FSTs considering all subsets. At the early stages of the algorithm, groups of FSTs that do not satisfy necessary structural properties are eliminated.

2.1.4.2. Heuristic Algorithms

Since the Rectilinear Steiner tree problem is NP-complete, many heuristic algorithms have been proposed for the problem. This chapter is divided into three sections. In the first section, historically early algorithms have been explained because the results of these studies affect modern algorithms. Next, more recent ones are described; these algorithms are more efficient compared to basic algorithms. The first two sections deal with sequential algorithms; finally, parallel RST algorithms are reviewed in the last section.

- **Basic RSMT Algorithms**

Hannan [11] is the first approximation algorithm proposed as a solution to RSMT problem. It starts with sorting the input terminals in ascending order according to their *x*-coordinates and these points are included in the tree in this order. During the addition operation, a two-point connection strategy is used. Lee, Bose and Hwang [18] is the first heuristic based on three point connection strategy, which can be implemented in $O(n^2)$ runtime. At the beginning of this algorithm, all three terminal points are formed and then shortest RSMT is generated using these points. It has an iterative approach and the closest terminal that is not in is added to the tree at each iteration. Hwang's algorithm [19] also deploys the three point strategy. However, compared to [18], it improves runtime complexity significantly to $O(n \log n)$. As was mentioned earlier, there exists a relationship between the quality of the solutions

of heuristic RSMT and MST algorithms. Therefore, Hwang starts the computation by constructing the MST in rectilinear distance. Then, it uses depth first search or breath first search to label the points in the MST. During labeling operation, nodes in P are ordered and then terminals are added to the tree in this order. The algorithm takes advantage of all benefits of the three-point connection strategy. Beasley's algorithm [20] is based on three-point connection and four-point connection strategies. The algorithm has an iterative approach. Each iteration begins with constructing a rectilinear MST using Prim's algorithm [21]. Steiner points are then computed using three-point or four-point connection strategies, afterwards, these Steiner points are included in the MST and the algorithm repeats itself.

- **Recent RSMT Algorithms**

The algorithms explained in Basic RSMT Algorithms chapter are historically earlier solutions to RSMT. Compared to recent algorithms, they have higher complexities and lower efficiencies.

- **The Iterated 1-Steiner (I1S) Approach**

Kahng and Robins [22] suggested one of the best performing heuristics in literature. I1S starts with computing the MST of a terminal set P and it searches for 1-Steiner points S such that:

$$\Delta MST = cost\big(MST(P)\big) - cost\big(MST(P \cup S)\big) > 0$$

As shown in Figure 2-5, intersection points of all horizontal and vertical lines through nodes in P are found; in other words, the Hannan grid is constructed. After that, all Steiner point candidates are found. Let Steiner point candidate set for P is denoted by *H(P)*, 1-Steiner point with respect to P is a node $s \in H(P)$ that maximizes $\Delta MST(P, s) > 0$. I1S starts with input node set P and $S(P) = \emptyset$ of Steiner points added to tree, it recurrently finds a 1-Steiner point *x* for $P \cup S$ and sets $S \leftarrow S \cup \{x\}$ as long as addition of *x* decreases the cost of the tree. The algorithm terminates when there is no 1-Steiner point *x* such that $\Delta MST(P, x) > 0$ [7]. Figure 2-6 illustrates an example of I1S algorithm.

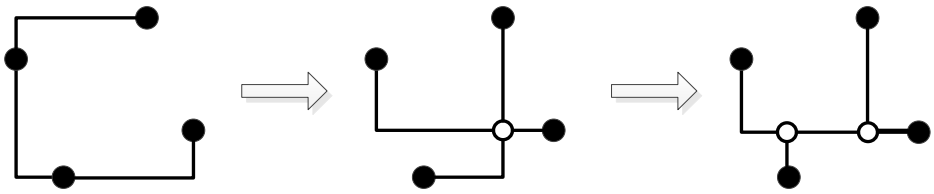

Figure 2-6: Execution of Iterated 1-Steiner Algorithm

In order to find a 1-Steiner point, it is required to generate an MST over $|P \cup S| + 1$ nodes for each of the $O(n^2)$ members of Steiner point candidates and the minimum cost candidate is selected. Next, the MST is computed in $O(n \log n)$ time [7]. Hence, runtime is $O(n^3 \log n)$ to find a single 1-Steiner point. Therefore, I1S algorithm cannot be used for instances having thousands of terminals; on the contrary, it is provably optimal for 4 or less nodes [23].

- **Batched 1-Steiner (B1S) Algorithm**

Since computational expenses of finding a 1-Steiner point I1S is high, a batched variant of this algorithm is proposed [22]. In this algorithm, at every round of computation, as many independent 1-Steiner points as possible are added to the tree. Steiner point candidates *x* and *y* are independent from each other, if including a new Steiner point in the tree does not reduce the potential gain of the other candidate. This definition can be formulized as follows:

$$\Delta MST(P \cup \{x\}) + \Delta MST(P \cup \{y\}) \leq \Delta MST(P \cup \{x\} \cup \{y\})$$

where $\Delta MST(P \cup S) = \max(0, MST(P) - MST(P \cup S))$ and a node x can be Steiner point candidate only if $\Delta MST(P \cup \{x\}) > 0$.

B1S greedily adds an independent 1-Steiner point to the tree at every round and it terminates when no 1-Steiner point can be added. B1S requires $O(n^2 \log n)$ runtime for each round.

Both I1S and B1S algorithms are suitable for parallelization since each compute unit in a parallel hardware can compute cost reductions in MST for different Steiner point candidates, and hence, a better performance can be obtained using a parallel implementation.

- **Zelikovsky Heuristic Algorithm**

The quality of an approximation algorithm is measured by its performance ratio, which is defined as the ratio of the length of RSMT found over optimal length. Hwang [12] has shown this ratio to be at most $\frac{3}{2}$ if RSMT found is the MST. Zelikovsky [24] worked on the quality of the RSMT and proposed a new algorithm with better performance. Zelikovsky's algorithm achieves $\frac{11}{8}$ RSMT to MST performance ratio obtained in $O(n^3)$ time.

The algorithm starts with MST and computes optimal Steiner trees for small subsets of the terminals iteratively. Then, these small subtrees are added to the current tree. Three terminals are included in one subset and it is called a triple. The algorithm iteratively introduces new points as part of a triple and then constructs the new MST for this new set of points.

- **Borah's Edge Substitution Algorithm**

Borah [6] proposed an edge based heuristic that has better runtime complexity than the previous algorithms. Using sophisticated data structures, $O(n \log n)$ runtime has been achieved.

The algorithm connects a node to nearest point on the rectilinear tree edge and it removes the longest edge formed in the tree. At the beginning of the algorithm, MST is computed on input terminal set using Prim's algorithm [21] in $O(n^2)$ time. Next, for each edge, the nodes that can be connected are searched and node-edge pairs are generated in recursive manner. During this operation, all nodes that are visible from the edge under computation are found and connection of each such node to the edge under study is considered where an edge is visible from a node if the line segment that joins them does not intersect any obstacles like another edge. The longest edge in formed cycle is searched and gain is calculated in this way for every pair. An example is given in Figure 2-7. For edge e_1, node n_4 is visible. Node n_4 is then considered to be connected to the nearest point on e_1, which is s_1. As a result of this, a cycle is formed from edges e_1, e_2, e_3 and e_x. Assume that the longest edge in the cycle is e_2, the gain for (n_4, e_1) pair is then calculated as follows:

$$gain(n_4, e_1) = |e_2| - |e_x|$$

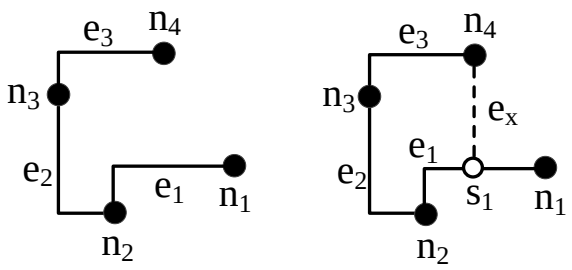

Figure 2-7: Borah's Edge Substitution Method

To obtain visibility information, a special sweep-line method is used. All visible nodes from an edge are computed and then the node with the largest gain is selected. Thus, a node-edge pair is formed. Finally, node-edge pairs are sorted with respect to their gains and starting from the largest, graph updates are applied to the tree.

- **Batched Greedy Algorithm**

Kahng, Mandoiu and Zelikovsky [3] suggested a highly scalable approximation algorithm for RSMT problem. Previous algorithms such as Iterated 1-Steiner algorithm are not suitable to be used with thousands of terminals. Kahng [3] in 2003 stated that Batched Greedy Algorithm (BGA) were able to generate rectilinear Steiner tree from 34k terminals in less than 25 seconds where Borah's algorithm [6] required more than 86 minutes. BGA has $O(n\,log^2\,n)$ runtime complexity for n terminals. Moreover, although, BGA improves runtime significantly, the quality of the solution is still comparable with previous algorithms.

Scalable BGA algorithm combines Zelikovsky's greedy triple contraction algorithm [24] with Batched 1-Steiner algorithm. BGA applies the batched approach of B1S [22], yet it relaxes the greedy rule used in computing triples in order to reduce runtime. It proposes a divide-and-conquer method to compute triples and a new linear data structure to search the longest edge in the formed cycles.

To begin with, BGA constructs a spanning graph on the terminal set. For each node, the region is divided into 8 partitions and the node is connected to the nearest neighbor in each partition as shown in Figure 2-8. Guibas-Stolfi's [25] north-east nearest neighbor method is used to obtain nearest points in each octant.

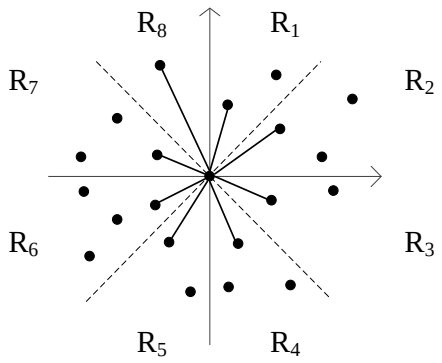

Figure 2-8: BGA Sparse Graph

After the generation of sparse graph, MST is computed on this sparse graph using Kruskal's algorithm [4]. Next, triples are generated for the MST edges where each triple consists of three terminals and contains one Steiner point candidate. When a Steiner point is included in the MST, a cycle is formed as explained for other algorithms. BGA constructs two arrays called parent and edge, which are used to detect the longest edge in the cycle. The generation of these arrays is embedded in the MST computations. According to the longest edges, the gain for each triple is calculated and starting from the largest gain, each Steiner point update is applied to the tree one by one.

- **Rectilinear Steiner Tree (RST) Algorithm**

Zhou's RST algorithm [25] is similar to BGA algorithm but it aims to obtain a better runtime performance than the Iterated 1-Steiner algorithm without sacrificing solution quality. RST is based on Borah's edge substitution method and Borah's [6] method is improved with Zhou's spanning graph algorithm [26]. RST has $O(n \log n)$ runtime complexity and requires $O(n)$ memory space.

RST starts with the construction of spanning graph for the input terminal set similar to BGA. Zhou's [26] sweep line algorithm is employed to compute a sparse graph. The area is divided into 8 regions for each terminal node and, at each octant, a special sweep process is applied. Each terminal is connected to its nearest neighbor in every octant following the sweep line operation. Afterwards, the MST is constructed on the sparse graph using Kruskal's algorithm. On the MST, Borah's edge substitution method is applied. Since geometrical proximity information is obtained during spanning graph construction operation, no further computation is required for visibility detection. The longest edges in the cycles are found with

Tarjan's offline search algorithm [27]. A binary search tree is used for Tarjan's offline search queries and it is constructed during the MST computations.

- **Parallel RSMT Algorithms**

Since 2000, the amount of interest in sequential solution of the RSMT problem has decreased significantly. However, there is now much more effort in the parallel solution of the problem. With a parallel implementation, the computing power of hardware can be exploited more efficiently. In this chapter, an overview of parallel approaches to rectilinear Steiner tree problem will be given.

- **Parallel Iterated 1-Steiner Algorithm**

I1S algorithm is based on Hannan grid, which initially computes MST and finds the best possible Steiner point from Hannan grid and then includes it in the tree. The operation terminates when there is no further improvement. I1S algorithm is very suitable for parallelization and Barrere [28] suggested an efficient parallel implementation of it. The gain of each Steiner point can be calculated independently. Therefore, $O(n^2)$ Steiner points are distributed among p processors. Each processor computes the gains of Steiner points in its own set and sends the best candidate to a master processor. The master processor sorts these best candidates and applies the one with the highest gain. This operation continues until no improving candidate can be found.

- **Parallel Steiner Tree Heuristic for Macro Cell Routing**

Fober and Grewal [29] proposed a two phase parallel algorithm for quick construction of a high quality Steiner tree to route multi-terminal nets. At the first phase of the algorithm, a single Steiner tree is constructed by using a special heuristic, called Shrubbery. Shrubbery simultaneously grows individual shortest path trees rooted at every terminal vertex to construct an initial Steiner tree. A modified version of Dijkstra's shortest path algorithm is used to grow the tree. In the second phase, multiple instances of local search are applied in parallel to create a pool of dissimilar and high quality trees.

Parallel implementation in [29] achieved near-linear speed up in relation to the number of processors, where the communication overhead for parallel work is insignificant to computational time required. Moreover, highly dissimilar trees can be generated in small runtimes.

- **Distributed Modified RST Algorithm**

RST algorithm has parallel steps in nature; hence, distributed modified RST algorithm [1] aims at taking advantage of these parallel steps. The algorithm is proposed for distributed memory systems.

The algorithm is called modified RST because it improves RST runtime by changing longest edge detection phase with BGA algorithm, since Tarjan's offline search algorithm used in RST has an iterative approach. Modified RST algorithm keeps the general flow of RST algorithm; however, during MST computations, instead of using a binary search tree as the main data structure, parent-edge arrays are generated and longest edge queries are computed using these arrays.

The algorithm applies master-slave model where there is a master processor that distributes computations uniformly to slave processors. Assuming that there are *n* slave processors, the region is divided into *n* sub-regions and each slave processor makes the nearest neighbor connections for the terminals lying in its own sub-region. Next, the MST is computed on the generated sparse graph sequentially using Kruskal's algorithm. Lastly, least common ancestor queries for node-edge pair generations are divided into *n* and each slave works on the queries independently. Average results obtained in [1] are shown in Table 2-1.

Table 2-1: Average Results for Distributed RST [1]

	p = 1		p = 2		p = 4		p =8	
Input Size ($x10^3$)	Imp. (%)	Time (s)	Imp. (%)	Time (s)	Imp. (%)	Time (s)	Imp. (%)	Time (s)
100	9,39	2,7	9,39	3,6	9,39	2,8	9,39	2,5
500	9,39	17,9	9,39	20,6	9,39	15,1	9,39	14,2
1.000	9,40	302,0	9,40	43,1	9,40	32,0	9,40	29,8

*p represents number of slave processor

Analyzing Table 2-1, we can state that the meaningful speed-up is achieved for the 1000k terminals; however, although algorithm runs in O(*n log n*) time and speedup is achieved for more processors, the algorithm is not scalable.

- **Parallel Obstacle Avoiding Rectilinear Steiner Tree Construction**

Chow, Li, Young and Sham [30] presented a heuristic based on maze routing to solve obstacle avoiding rectilinear Steiner minimal tree (OARSMT) problem. In [30], a parallel implementation of the algorithm is also presented. The algorithm is applied to GPUs using CUDA software design platform.

Chow's algorithm starts with defining shortest path regions between pin pairs. The *shortest path region* (SPR) is defined as the union of all candidate points of the shortest path between two pins. After obtaining SPR information, the algorithm develops a maze routing method over the escape graphs proposed by Ganly and Cohoon [31].

Chow practices a different propagation and backtracking approach than traditional maze routing based methods. On the Hannan grid, the distances of Steiner point candidates from each pin are calculated. This operation is suitable for parallel implementation and this computation is done on GPU. Moreover, after parallel propagation and backtracking, shortest paths between pins are computed in parallel.

2.2. Graphics Processing Unit (GPU)

In recent years, the computation needs of most applications are increasing constantly and hence industry is shifting towards parallel computing. GPUs are playing a significant role in the transition to parallel computing.

Originally GPU is an electronic hardware designed to manipulate and alter the memory to accelerate the creation of images. The term GPU was popularized by Nvidia in 1999 [32]. Today, GPUs are used in many areas, such as personal computers, mobile phones, workstations and embedded systems. Modern GPUs can efficiently handle computer graphics and perform high performance image processing. A GPU has a highly parallel structure compared to a central processing unit. A performance comparison of CPU and GPU is given in Figure 2-9.

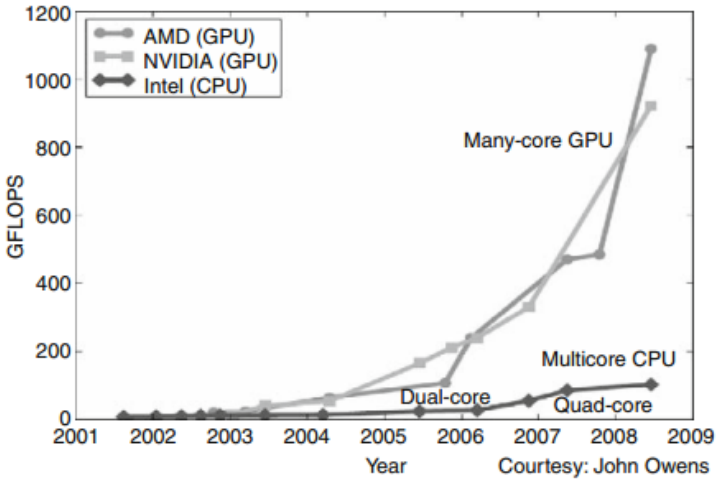

Figure 2-9: Performance Gap between CPU and GPU [30]

The general purpose CPU hardware is optimized for sequential operation. A CPU has a sophisticated control logic, which allows instructions of the same thread to execute in parallel or out of order. In the meantime, it maintains the sequential execution appearance. Moreover, CPUs have large volumes of cache memories to reduce data access time. Both of these features do not affect the peak speed given in Figure 2-9. GPUs have small caches and simpler control logic compared to CPU; nevertheless, they contain larger number of smaller cores. The CPU and GPU architecture comparison is illustrated in Figure 2-10.

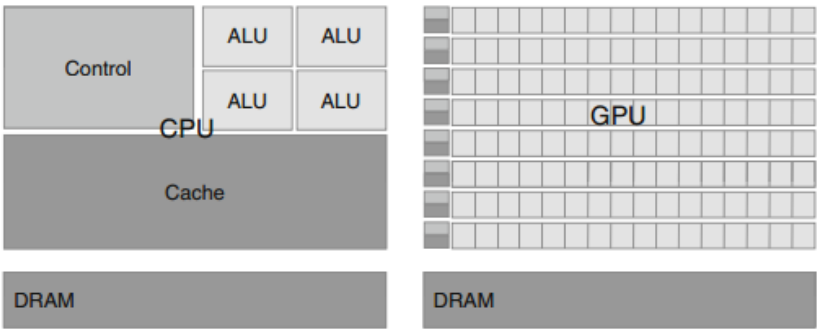

Figure 2-10: CPU and GPU Architecture [30]

2.2.1. Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture is a parallel computing platform and application programming interface for GPUs developed by Nvidia. CUDA allows developers to use appropriate graphic processing units for general purpose computing; it is a C based programming language with some extensions. The CUDA provides a software structure that gives direct access to GPU's parallel computational elements [32].

A CUDA program contains some number of phases, which are executed on either CPU, called host or on GPU, called device. The program phases that do not contain data parallelism are executed on host, CPU, and phases with data parallelism are executed on device, GPU. The data parallelism is the simultaneous execution of multiple cores on different parts of the data.

Typical execution of a CUDA program is given in Figure 2-11. Execution starts on CPU. When a GPU function called Kernel is launched, execution moves to device.

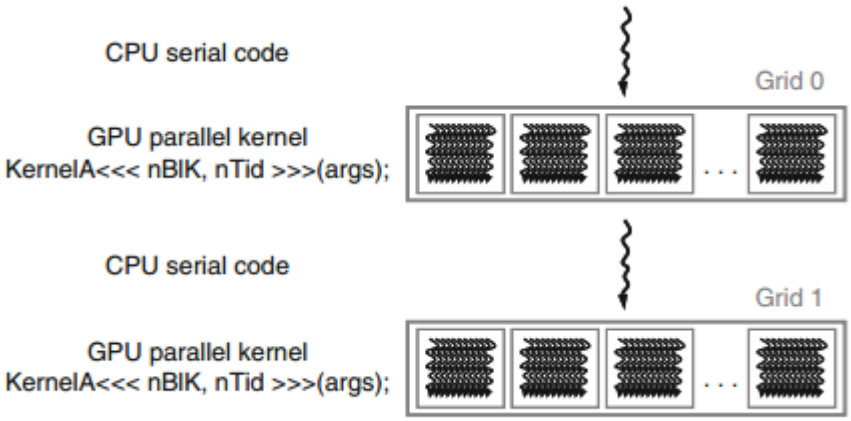

Figure 2-11: Execution of CUDA Program [30]

In a GPU kernel, a large number of threads are generated to benefit from data parallelism. All threads generated by a kernel after a launch is called grid. In Figure 2-11, executing two grids of threads are shown and the grid structure is given in Figure 2-12. Each thread shown in Figure 2-12 can execute in any order relative to other threads. All threads in a grid execute the same kernel function.

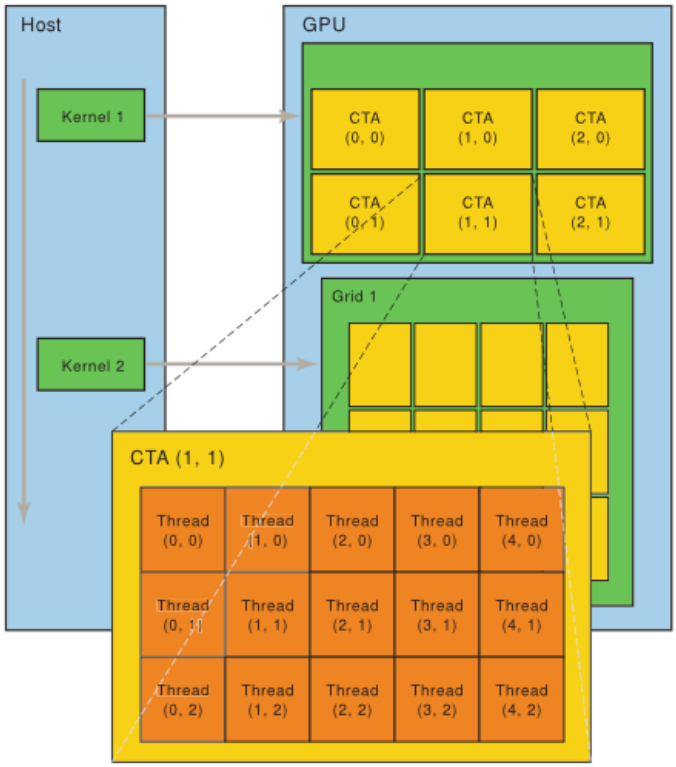

Figure 2-12: Threads in Grid for a Kernel [30]

CHAPTER 3

MODIFIED RST ALGORITHM

3.1. The Rectilinear Steiner Tree (RST) Algorithm

Zhou's RST algorithm [2] is a minimum spanning tree based heuristic approach. It uses Borah's edge substitution method [6] as basis and improves it with Zhou's minimum spanning tree solution [26]. The complexity of this algorithm is O(*n log n*) and the storage requirement is O(*n*) where *n* is the number of nodes. The implementation of the algorithm is easier compared to other heuristic algorithms. Although the RST algorithm only focuses on the rectilinear distances, the idea behind the RST is valid for Euclidean distances, too. The basic algorithm flow is given in Figure 3-1.

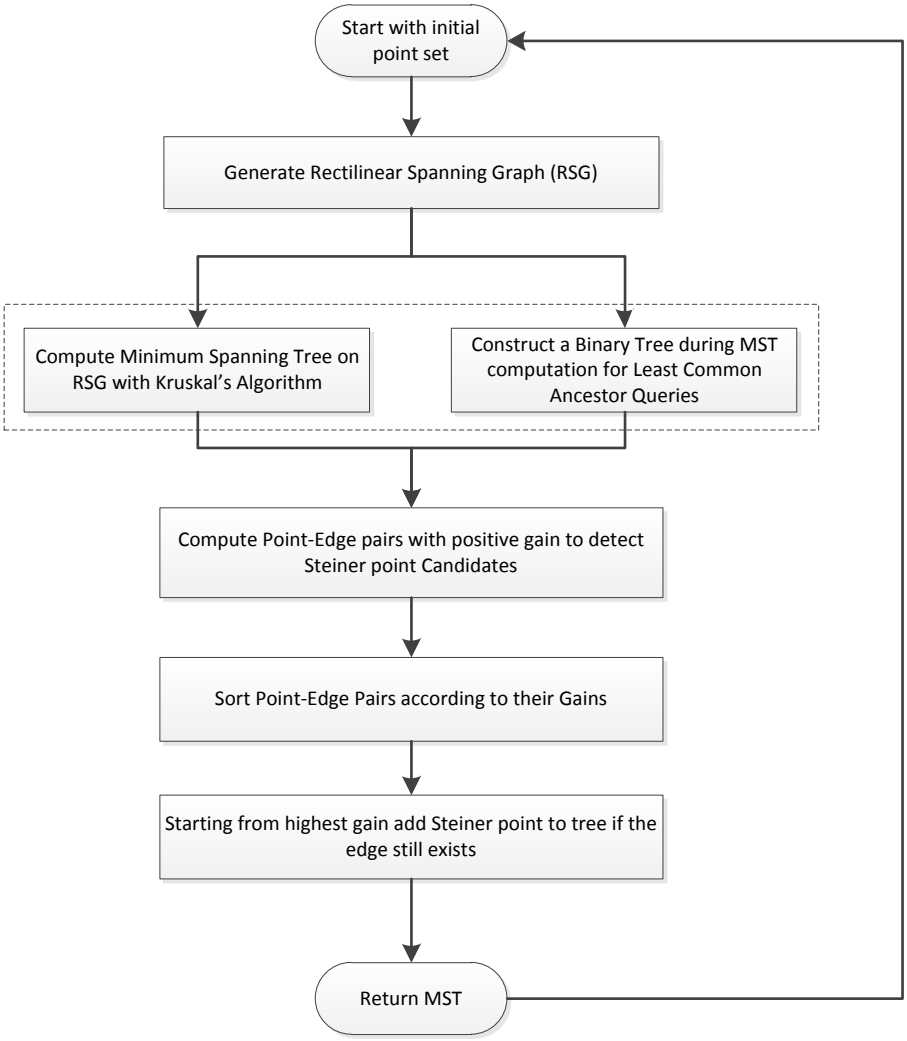

Figure 3-1: RST Algorithm Flow

3.1.1. Sparse Graph Construction

The Sparse Graph is used to compute MST, which is the backbone of RST algorithm, the MST algorithm works on the sparse graph constructed at the first step. In order to be able to form the sparse graph Zhou's spanning graph algorithm [26] was chosen.

3.1.1.1. Spanning Graph

Spanning graph is a connected graph; every node is reachable from every other node, with no cycles. According to Zhou, a graph with V number of nodes and E number of edges is called spanning graph if it contains a spanning tree [26]. A graph can have many possible spanning trees, yet all spanning trees have V vertices and V-1 edges. An example of a connected graph and its possible two spanning trees are given in Figure 3-2. The minimum spanning tree may not be unique for a given set of nodes. Zhou in [26] defines a new class of spanning tree called strong spanning tree.

That is, if spanning tree includes all minimum spanning trees for the given set of nodes, it is called strong spanning tree.

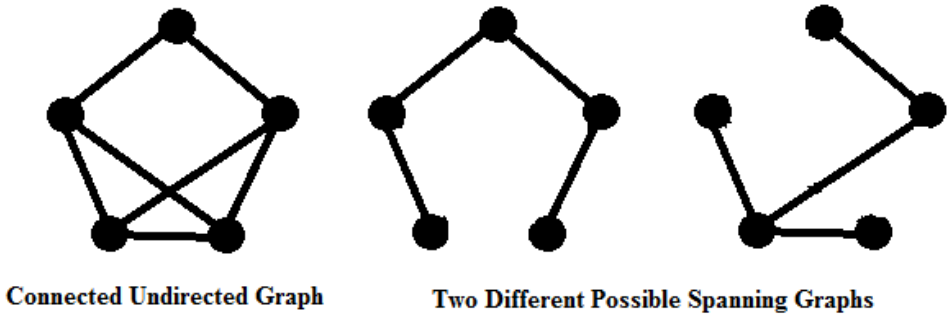

Figure 3-2: Connected Graph and Its Spanning Graphs

Minimum spanning tree algorithms such as Prim [21] and Kruskal [4] usually applies two properties, namely, cut property and cycle property in order to decide to include or to exclude an edge in the minimum spanning tree.

Cut property states that for a minimum spanning tree T in a connected graph G(V,E), let X ⊆ T and S ⊂ V, there is no edge in X that crosses between S and (V − S). Let *e* be a minimum weight edge among the edges crossing between S and (V − S), then X ∪ {*e*} ⊆ T where T is a MST in G(V,E). An example of cut property is given in Figure 3-3.

The second property, cycle property, states that for any cycle in the graph, if the weight of edge e is larger than the weights of all other edges in the cycle, then edge *e* cannot belong to MST.

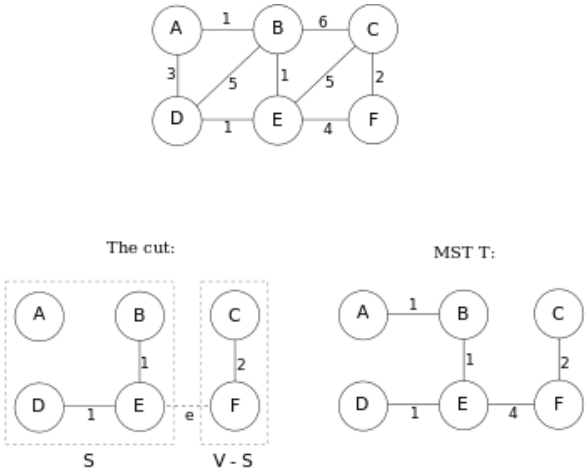

Figure 3-3: The Cut Property for Minimum Spanning Problem [33]

3.1.1.2. Rectilinear Spanning Graph Construction

Zhou's rectilinear spanning graph construction algorithm starts with the division of the plane to equal regions around each node; that is, from each node *p*, the plane is partitioned into eight octal regions as given in Figure 3-4.

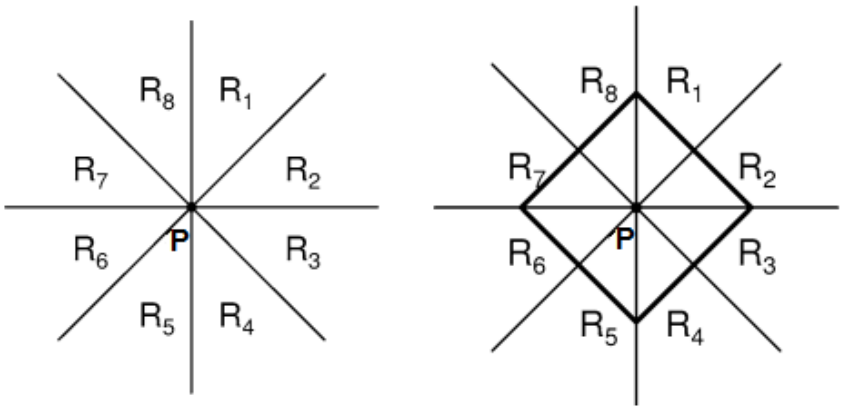

Figure 3-4: Octal Partition and Equal Distance Nodes [26]

Using the terminology given in Robins [34], the uniqueness property is defined as follows: For a given node *p*, a region partition R has the uniqueness property with respect to node *p* if for every pair of nodes x, y ∈ R, $||xy|| < \max(||px||, ||py||)$.

For a node *p* in the given node set, each octal partition has uniqueness property with respect to node *s*. Example case for R_1 partition is given in Figure 3-5.

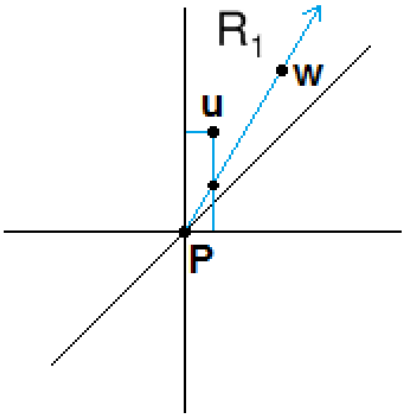

Figure 3-5: Uniqueness Property for R_1

The points, which have *p* in their R_1, must obey the following inequalities:

$$x \leq x_p \qquad (3\text{-}1)$$

$$x - y > x_p - y_p$$

Considering Figure 3-5, there are two points *u* and *w* in R_1 where $x_w \geq x_u$ and $y_w \geq y_u$. Then the rectilinear distance between points is

$$||pw|| = ||pu|| + ||uw|| > ||uw||$$

For the case, $x_w \geq x_u$ and $y_w \leq y_u$:

$$\begin{aligned} ||uw|| &= |x_u - x_w| + |y_u - y_w| \\ &= x_w - x_u + y_u - y_w \\ &= (x_w - y_w) + y_u - x_u \\ &< (x_p - y_p) + y_u - x_p \\ &= y_u - y_p \\ &\leq x_u - x_p + y_u - y_p \\ &= ||pw|| \end{aligned}$$

Thus, we can state that R_1 partition has uniqueness property with respect to point *p*. Since all eight partitions are symmetric, all partitions given in Figure 3-4 have uniqueness property with respect to point *p*.

Considering the cycle for the point set *p*, *u* and *w* given in Figure 3-5, depending on the cycle property, only the point with the minimum rectilinear distance from *p* is connected to *p*. In addition, the uniqueness property states that $||uw|| < \max(||up||, ||wp||)$. Therefore, we only need to consider edges from a point *p* to its closest neighbors in each octant. Sparse graph is going to be constructed based on this fact, each point given in the initial set will be connected to its closest neighbor in each octant and MST algorithm will compute on this sparse graph.

Octal partitioning approach has another property that is worth pointing out. The contour of equidistant points from *p* forms a line segment in each region. In first and third quadrant, R_1, R_2, R_5 and R_6, these line segments are described by an equation in the form of $x + y = c$; in second and fourth quadrants, R_3, R_4, R_7 and R_8, they are captured by an equation in the form of $x - y = c$.

Zhou's algorithm [3] starts with finding all nearest neighbor candidate points in a specified octant for a given point p. A sweep-line algorithm is constructed on all points according to their non-decreasing $x + y$ or $x - y$ values. Sweep operation for the right half of the plane is given in Figure 3-6.

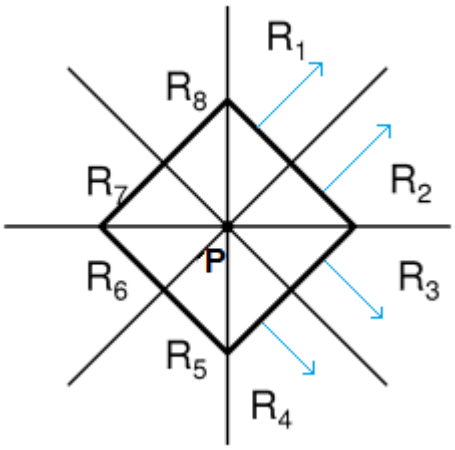

Figure 3-6: Sweep Operation [26]

For R_1 region, a sweep-line algorithm is applied according to (x + y) values of the points. During the sweep operation, an active set is maintained. This active set consists of points whose nearest neighbor in R_1 region still to be discovered. When a point p is processed, entire set of nearest point candidates, which have p in their R_1 regions, are found. Let's assume a point *w* is such a point, since the points in the active set are in non-decreasing (x + y) order, we can say that *p* is the nearest point in R_1 for w. Hence, edge *pw* is added to spanning graph and point *s* is deleted from active set. After computing those points, *p* is added to active set at the end. Each point given in the initial set is added and deleted at most once from the active set. The fundamental aim of the sweep-line algorithm is to find, for a point *p*, the subset of active points such that p is in their R_1 regions.

To find the subset of points in the active set, which have *p* in their R_1 partition, we can start with finding the point with the largest x value since $x \leq x_p$, then advance in the search until x – y becomes smaller than $x_p - y_p$. In order to manage the active set, a priority search tree is organized. For this data structure, the deletion and insertion operations take *O(log n)* time and the query operation takes *O(log n + k)* time where k is the number of objects within the search range. Thus, the total time for the whole sweep operation is *O(n log n).* Since all eight octants are symmetric, and they can be processed in the similar way as in R_1, the running time of the algorithm is *O(n log n).* During the sweep operation, a point is deleted from the active set if a point in R_1 region is found; therefore, no point in the active set can be in R_1 region of another point in the set. This feature of the algorithm makes it easy to implement, since,

priority search tree depends on managing a balanced structure for fast query times and rebalancing the tree is challenging for a dynamic input set.

For R_2 region, sweep-line algorithm works for non-decreasing x + y values; however, R_2 neighbor candidates are evaluated according to the following inequalities:

$$y < y_p \qquad (3\text{-}2)$$
$$y - x \geq y_p - x_p$$

According to equation (3-2), active set can be sorted with respect to y values of the given points, and point can be computed by decreasing order y until y – x becomes smaller than $y_p - x_p$.

For R_3 and R_4 regions, active set is constructed according to non-decreasing order on x – y. In order to decide R_3 neighbors, inequalities given in equation (3-3) are used:

$$y \geq y_p \qquad (3\text{-}3)$$
$$x + y < x_p + y_p$$

While processing R_3 neighbors, we can start from the point with the smallest y value considering equation (3-3) and continuing the search operation until x + y value gets smaller than $x_p + y_p$ value.

The points in the active set should obey inequalities given in equation (3-4) to become R_4 neighbor of p:

$$x < x_p \qquad (3\text{-}4)$$
$$x + y \geq x_p + y_p$$

As stated in equation (3-4), we can order active set in non-decreasing order of *x* and start from the point with the largest *x* value and extend the R_4 region nearest point search until x + y becomes smaller than $x_p + y_p$.

When we analyze the results of the sweep-line algorithm, we state that for given two points *p* and *u* in the plane, point *p* is in R_1 region of point u if and only if *u* is in the R_5 region of *p*. Based on this observation, we can conclude that after completing the sweep process, there is no need to run the algorithm for R_5 region, since all edges needed in this phase are already connected. The example case is shown in Figure 3-7, while searching for points that have *p* in their R_1 regions, we find two points in the active set, namely, *w* and *u*. Therefore, *w* and *u* are in the R_5 region of *p* where $||wu|| < \max(||pu||, ||pw||)$ and *p* is connected to the nearest active point.

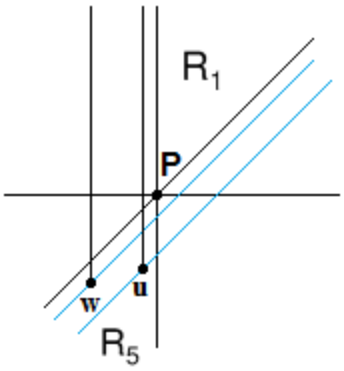

Figure 3-7: The Nearest Point in R_5 Region

3.1.2. Minimum Spanning Tree Construction

A minimum spanning tree can be defined as a connected weighted spanning tree which has the minimum weight among all spanning trees. There are two classic MST algorithms, Prim's [21] and Kruskal's [4]. With the use of disjoint set operations in Kruskal [4], the minimum spanning tree computations and longest edge computations for each node-edge pair can be unified. Hence, because of this ability to unify two computations, Kruskal's algorithm has been chosen to construct MST instead of Prim's in this thesis. While constructing the MST for the given node set, a binary search tree should be organized for the least common ancestor queries.

Kruskal's algorithm builds the MST in the form of a forest. At the beginning of the algorithm, each vertex is considered to be a separate tree in the forest. The algorithm sorts the edges according to their weights and then starting with the edge with the least cost, connects any two trees in the forest if both end points of the edge does not belong to same tree. Kruskal is a greedy algorithm. At each step of the computation, edge with larger cost is added to graph.

Given V (number of vertices) and E (number of edges), the running time of Kruskal is *O(E log E)*, or equivalently *O(E log V)*.

3.1.3. Edge Based Heuristic

In order to achieve a fast running time heuristic algorithm, Zhou [2] selected the edge substitution method of Borah [6] as the basis. Borah's algorithm can achieve $O(n^2)$ running time with simple implementation using conventional data structures, it has *O(n log n)* asymptotic run time complexity when it is implemented with sophisticated data structures.

Borah's edge based heuristic algorithm considers connecting a vertex to the nearest point on the closest rectilinear tree edge and removing the longest edge on the formed cycle. Consider the tree given in Figure 3-8 in which the edges are shown, for the sake of clarity, in Euclidian form although all the distances are in ℓ_1 – norm (rectilinear metric). Point *a* is connected to closest edge which is e_1, and hence a loop is forms between points *a*, *b*, *c* and *d*. Assuming that e_2 is the longest edge in the formed cycle, the following modifications in tree should be make:

I. Add Steiner point *p* to tree
II. Remove edge e_1 and longest edge in loop e_2
III. Add edge connecting *p* to *a*
IV. Add edge connecting *p* to *h*
V. Add edge connecting *h* to *d*

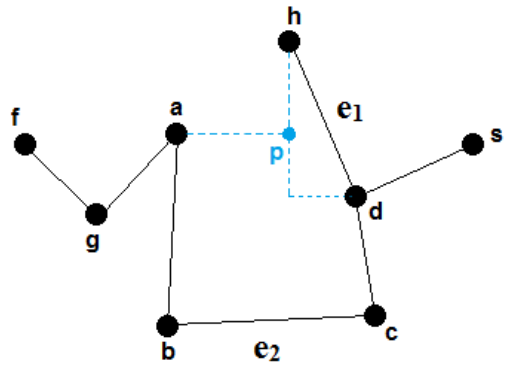

Figure 3-8: Borah's Edge-Based Graph Update

The new node added to the tree is called Steiner point and the addition of this Steiner point requires a modification in the existing tree edges. Following the modification, the initial graph becomes a spanning tree with an extra node. The reduction in the cost of whole spanning tree, called gain, is given in equation (3-5).

$$\text{Gain} = \text{length}(e_2) - \text{length}(a, p) \qquad (3\text{-}5)$$

$$= \|bc\| - \|ap\|$$

During the edge substitution process, Borah's algorithm computes all possible node-edge pairs with positive gain and applies as many edge modifications as possible to initial tree. The main flow of the algorithm is given in Figure 3-9. Once an edge is participated an update operation, it is replaced and no longer exists for the algorithm. In Figure 3-8, edge e_1 may be connected to several Steiner points other than *p*; however, only one case can be applied to graph, since after graph update edge e_1

does not exist anymore. Hence, the pair with maximum gain for given edges should be computed and reported at the end.

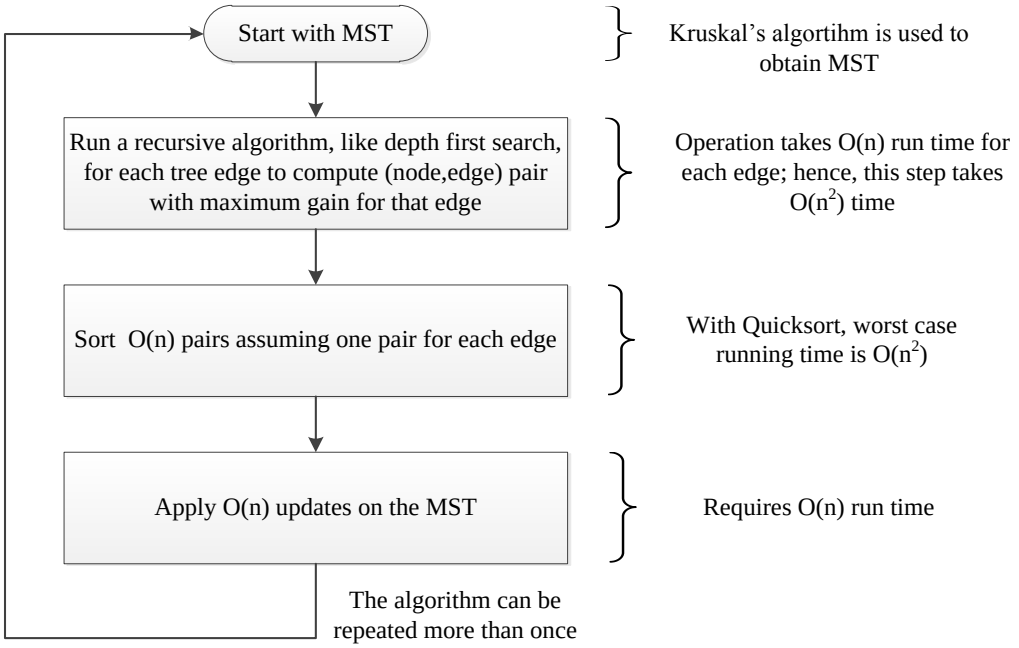

Figure 3-9: Borah's Edge-Based Heuristic Flow

In one iteration, Borah's Edge Based Heuristic algorithm computes all possible node-edge pairs with positive gain before applying the modifications to tree, the graph is fixed for this step; hence, this algorithm works in batched mode. For the next step, before applying updates, the both edges going to be updated are checked; if both of the edges do not exist in the tree, update is not applied.

The algorithm can be repeated more than once and at each time further improvements can be applied; however, according to [6], three iterations are adequate in most cases. Rarely, fourth iteration may be required. The experiment results of Borah are given in Table 3-1.

Table 3-1: Improvements for Different Number of Iterations of Algorithm [6]

size	Passes Required			Avg. improv./pass (%)				Max. improv./pass (%)			
	Avg	Max	Min	1st	2nd	3rd	4th	1st	2nd	3rd	4th
6	1.41	3	1	8.88	.99	.02	0	19.69	11.21	5.19	0
10	1.7	3	1	9.15	.97	.01	0	15.04	4.65	.47	0
50	2.3	4	1	9.08	1.31	.06	.01	12.69	2.87	.61	.07
100	2.5	5	2	9.28	1.40	.07	.002	11.26	3.12	.49	.18
200	2.7	4	2	9.48	1.36	.07	.001	11.05	2.24	.37	.04
500	3.3	4	3	9.46	1.41	.05	.002	10.39	1.79	.15	.02

The Edge Substitution Heuristic given in Figure 3-9 has $O(n^2)$ running time; however, in [6], possible implementations to reach $O(n \log n)$ running time has been discussed. These implementations are based on the observation that every node doesn't need to be considered for every edge as pair.

When we examine Figure 3-10, we observe that nodes *a* and *b* can be connected to edge e_1; nonetheless, nodes *g* and *k* cannot be connected to e_1 due to edges e_2 and e_3. Edge e_2 is blocking node *k* and edge e_3 is blocking node *g*. By considering this tree structure, we can state that e_1 is visible from *a* and *b*, similarly e_3 is visible from *h*. In the meantime, e_1 is not visible from *g* and *k*. As a result, it is clear that a node can be connected to an edge during node-edge pair updates only if it is visible to that edge.

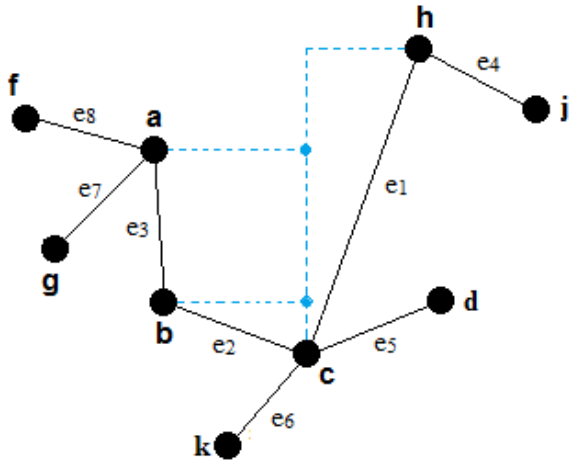

Figure 3-10: Node Visibility

In Figure 3-10, edge e_1 is visible to nodes *a* and *b* from its left and *d* and *j* from its right. Moreover, e_2, e_3, e_4 and e_5 are neighbor edges of e_1. The nodes other than the ones connected to neighbor edges are blocked by the neighbor edges, for instance, *g* and *k*. Based on this fact, we can state that the visible nodes from an edge should be connected to its neighbor edges. Since a point cannot be connected to a non-visible

edge, there is no need to consider those edges during computation. In [35], it is proved that the degree of any vertex in a rectilinear minimum spanning tree is at most 8; moreover, at most 6 of these edges are non-degenerate which means that the end points of the edges do not lie on the same horizontal or vertical line. Only non-degenerate edges are meaningful for Steiner point computations; hence, edge-pair couples can be computed in *O(n)* time for a node and *O(n)* run time for whole graph.

Borah [6] suggested a sweep-line algorithm to overcome edge visibility problem. However, the first step of RST algorithm is spanning graph construction and since each node is connected to its nearest neighbors, the geometrical proximity information about the nodes and edge is embedded in this spanning graph. In [2], it is specified that, if a point is connected to one end of the MST edge in the spanning graph, then the edge is generally visible to that point in MST. In Figure 3-11, MST contains edge ab, it is clear that region R_a does not contain any nodes since the edge is included in MST and if node p is connected to a in the spanning graph, it can be concluded that there are no nodes in R_p and hence ab is visible to node p. Consequently, Zhou's algorithm uses spanning graph to generate node-edge candidate pairs and there is no need to apply Borah's sweep-line algorithm to obtain node-edge visibility information.

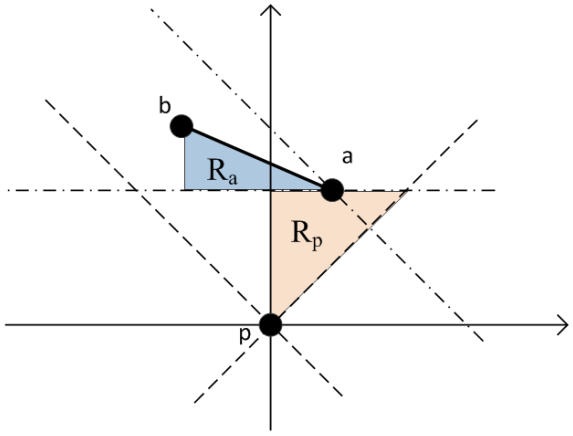

Figure 3-11: Visibility Information

There is no direct relation between spanning graph connection information and visibility. For some cases, even though node is not connected to MST edge end point, the edge is still visible from that node. An example is given in Figure 3-12; even though node *p* is not connected to node *a* in spanning graph due to node *c*, the edge ab is visible from *p* and can be included in node-edge pair computations.

For Zhou's algorithm, for each edge in the MST, only the neighbors of end points are computed to form node-edge pairs with that edge. Although, this can cause degradation in the quality of final Steiner tree, the results are still better than the similar algorithms and O(*n*) run time has been achieved.

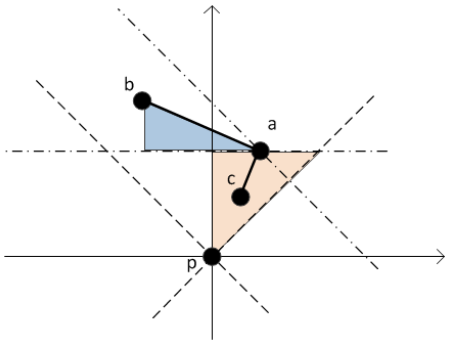

Figure 3-12: Visibility of an Edge from a Non-Neighbor Point

3.1.4. Longest Edge Detection

When a Steiner point is added to MST, a cycle is formed in the MST. For instance, in Figure 3-8, if point *p* is added to tree, the cycle shown in Figure 3-13 is formed and hence the longest edge in this cycle should be deleted. In RST, longest edge detection task is computed in two steps, that is, binary tree construction and least common ancestor queries. At the beginning of the computation, a binary tree is constructed respecting the weights of edges and longest edge search operation is applied on the binary tree.

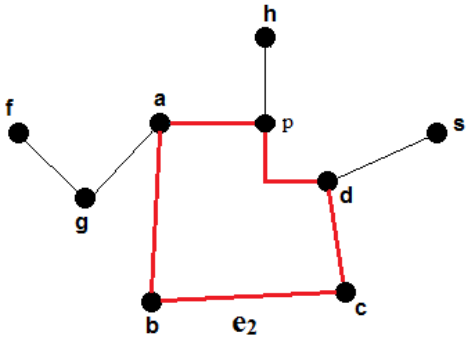

Figure 3-13: Loop Formed After Steiner Point Addition

3.1.4.1. Merging Binary Tree

The binary tree constructed is used for longest edge detection queries. The connection information of tree is converted into a binary tree where the points are represented with leaves and the edges are represented with internal nodes. An example MST and its binary search tree are given in Figure 3-14. The binary tree is generated considering the weights of edge. The least common ancestor (LCA) of two tree nodes is defined as the ancestor in tree that is located farthest from the root. In RST, LCA of two points in binary tree is going to be the longest edge between them. For example, the longest edge between point *a* and *e* is *bd*; in the meantime, *bd* is the least common ancestor node in binary tree for these two points.

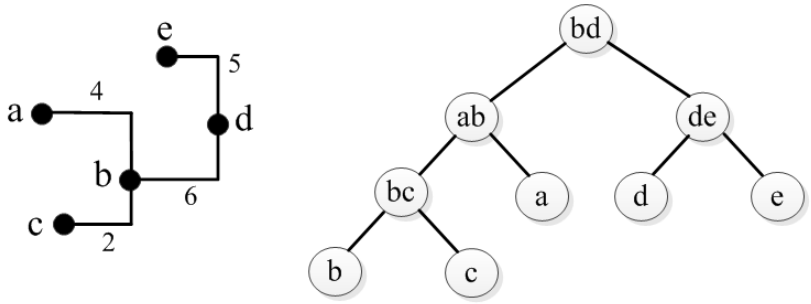

Figure 3-14: Binary Search Tree for the Given Tree

In RST, the MST construction from spanning tree is achieved using Kruskal's algorithm. This algorithm sorts the edges in line with their weights and processes each edge individually. The edge is included in the tree if both end points belong to different disjoint sets otherwise; the edge is left out from the MST. So as to accomplish efficient longest edge detection operation for each node-edge pair, binary search tree generation process is embedded into Kruskal's algorithm. During the MST computations, when an edge is included in the MST, a node with two children is created in binary tree. The node represents the edge created and the children of it represent the nodes connected by that edge. In RST, for the longest edge search, Tarjan's offline search algorithm is applied.

3.1.4.2. LCA Algorithm

Having the merging binary tree at hand, the next thing to do is searching longest edges in cycles formed. Tarjan's Offline Least Common Ancestor [27] algorithm is selected for this purpose. Using Tarjan's algorithm longest edge for each node-edge pair is computed and then the gain is calculated. Unlike other LCA data structures, in

Tarjan's algorithm, all pairs of binary tree nodes for which LCA is desired must be specified in advance; that is, prior to providing the first answer, the entire request sequence can be seen. Hence, the problem is offline [27].

Post order traversal is performed for the algorithm. The pair list is examined to decide whether any ancestor computations are to be performed before returning from processing a node. For instance, assuming *u* is the current node and (*u*, *v*) is in the pair list and recursive call to node *v* is completed, then there is adequate information to determine LCA of node *u* and *v*.

In Figure 3-15, a binary tree is given where a recursive call to node D is about to finish. The nodes enclosed by dashed lines have been visited and those node share the same group name and anchor where anchor is the node on the current access path that is closest to visited node. For example, node p's anchor is A and q's anchor is B. The nodes like *r* that are not marked are unanchored. When all its descendants are processed, a node is marked. Shaded nodes represent the nodes visited by a recursive call. For Figure 3-15, all recursive calls other than for the node on the path to D have been completed.

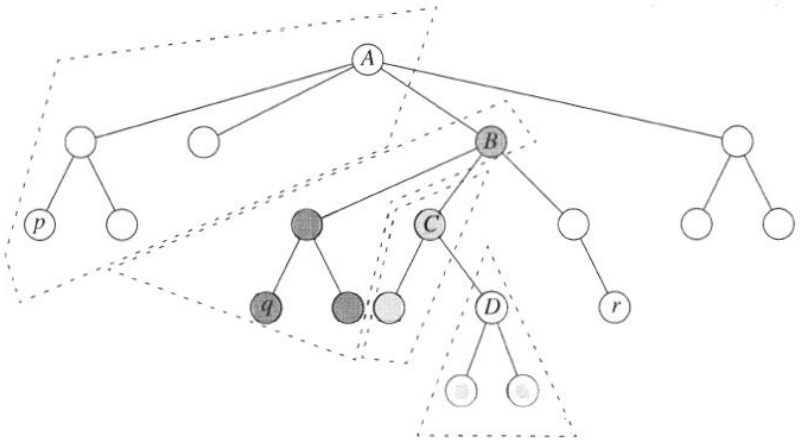

Figure 3-15: The Nearest Common Ancestor Problem [10]

Considering Figure 3-15, suppose the (*D*, *v*) is in the pair list, there can be three cases which are:

a) If *v* is unmarked, the information to compute LCA (*D*, *v*) is missing and hence it cannot be computed.
b) *v* is marked but not in D's subtree, so LCA (*D*, *v*) is *v*'s archor
c) *v* is in D's subtree, hence LCA (*D*, *v*) is D

The basic version of Tarjan's algorithm applies the union-find data structure. Therefore, unlike other LCA algorithms, its run time can be more than constant time per operation if the number of pairs of nodes is comparable with the number of nodes.

3.2. Modified RST Algorithm

Modified RST algorithm [12] is based on Zhou's RST algorithm. It enhances RST by replacing some parts of it with Kahng's BGA algorithm [3]. BGA and RST algorithms have similar approaches for the solution of Rectilinear Steiner tree construction problem. Both algorithms start with spanning graph and apply MST algorithm on it. The comparison of these two algorithms is given in Figure 3-16.

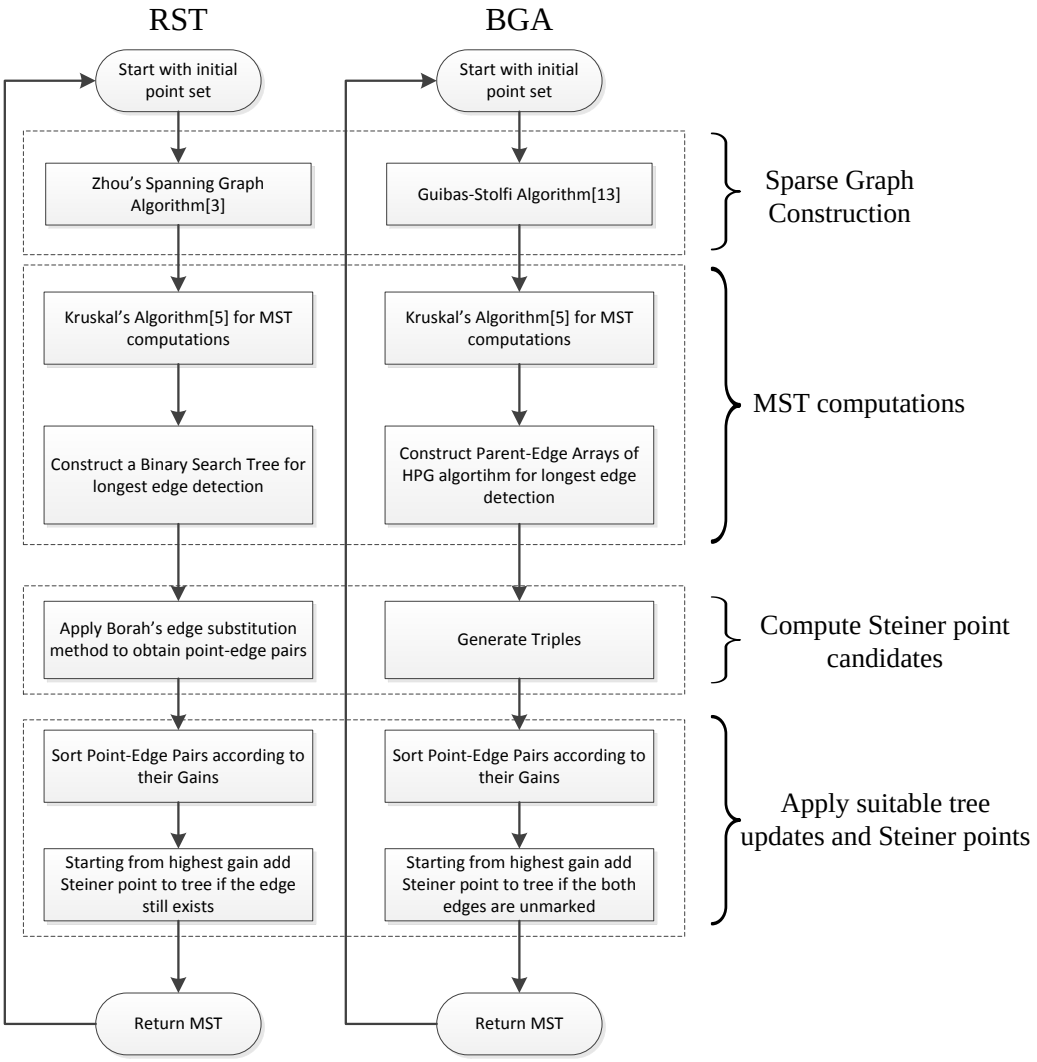

Figure 3-16: The Comparison of BGA and RST algorithms

3.2.1. Batched Greedy Algorithm

3.2.1.1. Sparse Graph Construction

Like RST algorithm, BGA starts the operation with sparse graph construction; however, it applies to Guibas-Stolfi [25] algorithm. Guibas-Stolfi divide region to eight partitions and each point is connected to its nearest R_1 neighbor, north-east neighbor. The nodes in each octant are mapped to R_1 partition and they are computed for this partition.

Guibas-Stolfi algorithm computes north-east nearest neighbors for a node set in O(*n log n*) run time. The algorithm starts with sorting the nodes according to their x coordinates and the x value that divides nodes into left and right half is found. For nearest neighbor computations, tree pointers are used, specifically, left, right and min. Pointer left moves down in the sorted list of left half nodes in decreasing y values. The other two pointers move down at the right half plane in decreasing y again. Pointers move in only one direction, they do not back up. During the downward movement, pointer right always stays at nodes with larger y values than pointer left and pointer min keeps the nearest node to left in the right half.

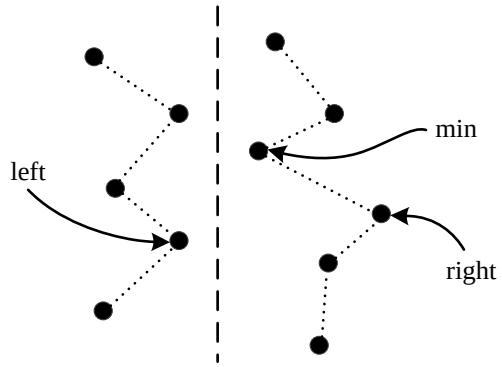

Figure 3-17: Guibas-Stolfi Pointer Position

During the movement of pointer right, three possibilities can occur, these are:

i. The node pointed by right has higher y value than left but further than min, pointers are not altered
ii. The node pointed by right has higher y value than left and closer than min, min is equalized to right
iii. The node pointed by right has smaller y value than left, for this condition, the min is set as nearest north-east neighbor of left and left is moved down

By following the steps given, nearest neighbors for R_1 regions are found and same procedure is applied for each octant.

3.2.1.2. MST Construction

Kruskal's algorithm [4] is used for MST construction. The detail of Kruskal's algorithm is given in chapter 3.1.2. In RST algorithm, binary search tree construction for least common ancestor queries is embedded in this computation; however, BGA algorithm does not use binary search tree for longest edge detection, instead, it applies hierarchical greedy pre-processing algorithm for this purpose. HGP algorithm creates two arrays, namely, parent and edge in at most 2n-1 steps where n is the number of terminals. These two arrays are used to obtain the longest edge in a cycle formed after addition of a Steiner point.

HGP algorithm has a recursive approach; initially, it searches the cheapest outgoing edge for each node. For a node *u*, the cheapest edge is directed away from *u* and its index is saved in edge(*u*). After this operation, some of the edges become bidirected or unidirected and some of the edges remain undirected. An example is given in Figure 3-18, in which edges *ab*, *de* and *hg* become bidirected, edge *bc* becomes unidirected and *be* and *eg* remain undirected, after the first iteration of HGP,.

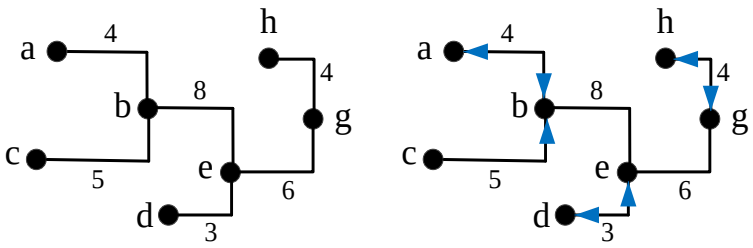

Figure 3-18: Edge Directions after First Run of HGP

In this directed graph, nodes connected to bidirected and unidirected edges are attached to each other and form subgraphs, called component K. HGP collapses these components into a single node *q*; then parent(*u*) is set for every u ∈ K. There can be at most n/2 nodes created, since each subgraph has at least one bidirected edge. The node collapsing process is repeated until all trees collapse to a single node. The result of the first iteration of the tree given in Figure 3-18 is shown in Figure 3-19. After HGP iteration, the connected nodes collapses to a single node, that is, nodes *a*, *b* and *c* collapse to node *x*, node *c* and *d* collapses to *y*, lastly, node *g* and *h* collapses to *z*.

The parent and edge arrays of tree in Figure 3-19 is given below:

edge = { ab, ab, bc, de, de, gh, gh, NIL, NIL, NIL, NIL, NIL, NIL}

parent = { x, x, x, y, y, z, z, NIL, NIL, NIL, NIL, NIL, NIL}

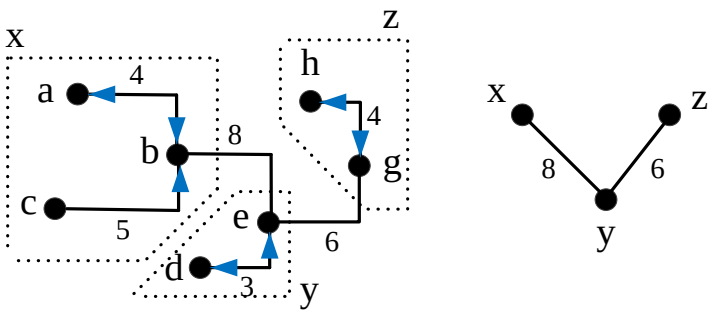

Figure 3-19: The Node Collapsing After HGP Iteration

Analyzing the details of HGP, we can state that the edge cost between components increases after each iteration. Moreover, if two nodes *u* and *v* are in the same component, the edge with maximum cost can be found from max{edge(*u*), edge(*v*)}. For instance, in Figure 3-19, nodes *a* and *c* are in component x and the longest edge between them can be calculated by max{edge(*a*), edge(*c*)} where edge(*a*) = ab and edge(*c*) = bc. If two nodes *u* and *v* are in different components, namely K and K', in order to compute the longest edge between *u* and *v*, the maximum cost edge of component T should be found, where T is the component K and K' components collapses to a single node.

3.2.1.3. Triple Generation

Triple is defined as the optimal Steiner tree for a set of three nodes. Repeatedly, all possible triples are found and for the triple with the largest positive gain, the edges of the triple are discarded and nodes are connected to Steiner point.

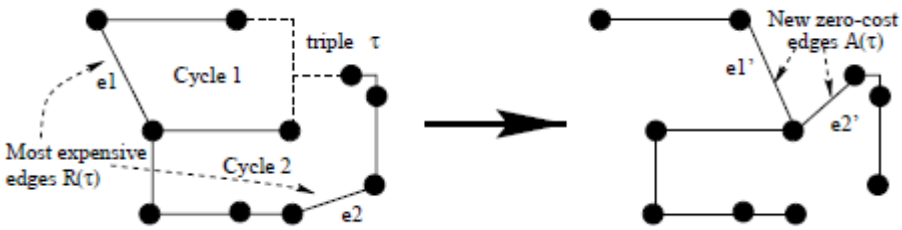

Figure 3-20: Triple Contradiction [3]

The triples are classified according to their position in region. There are four types of triples which are north-west, north-east, south-west and south-east. A triple can be classified as north-west if the diagonal terminal is in north-west quadrant of center. After triple classification, nodes are sorted according to their (x+y) or (x-y) values, and an algorithm similar to Guibas-Stolfi is applied to determine Steiner points.

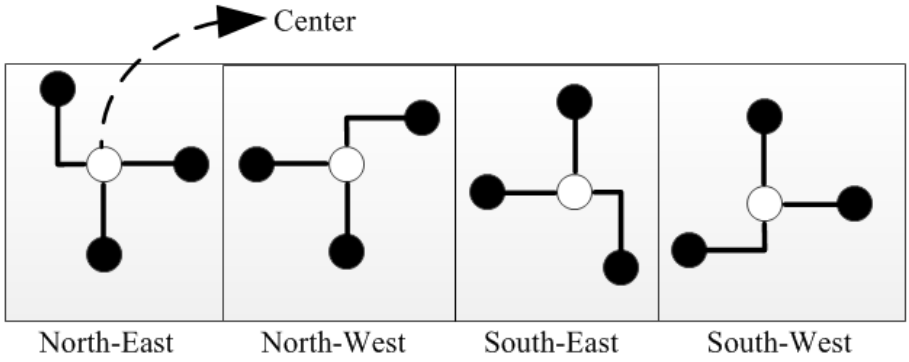

Figure 3-21: Triple Types

3.2.1.4. Graph Update

Once the all possible triples are computed, the next step is the graph updates. Starting from the triple with highest gain, the Steiner point is added to tree and the longest edges in the cycles formed are discarded from MST.

At the beginning of the operation, all edges are unmarked. If both edges in a triple are unmarked, this update can be applied to tree and those edges are deleted. Moreover, the longest edge in the cycle is computed through parent and edge arrays.

3.2.2. Modified RST Algorithm Flow

Modified RST algorithm focuses on the total runtime of the Rectilinear Steiner Tree construction solution and hence it tries to increase the speed of the algorithm. BGA and RST algorithms consist of phases with iterative and recursive approaches. Recursive operations require storing the data from previous step and reloading it, moreover, they may cause overflow problems especially for large data sets. Thus, recursive algorithms are not desired for speed oriented solutions.

The Modifies RST algorithm flow is given in Figure 3-22. In the initial phase of the algorithm, RST generates sparse graph using special sweep method iteratively; however, BGA implements a recursive approach; therefore, RST is considered as advantageous to use. The longest edge detection algorithm of RST is based on Tarjan's offline search algorithm, for this purpose a binary search tree is constructed

during MST computations. Unlike HGP used in BGA, Tarjan's offline search algorithm is recursive and hence it is not preferred, in Modified RST algorithm, HGP algorithm is used to detect longest edges in formed cycles. Last of all, for Steiner point candidates, triples are formed and added to MST in BGA with a recursive attitude. In RST, neighbors in spanning graph is computed as node-edge pair, that's why it is considered more effective and practiced in Modified RST algorithm.

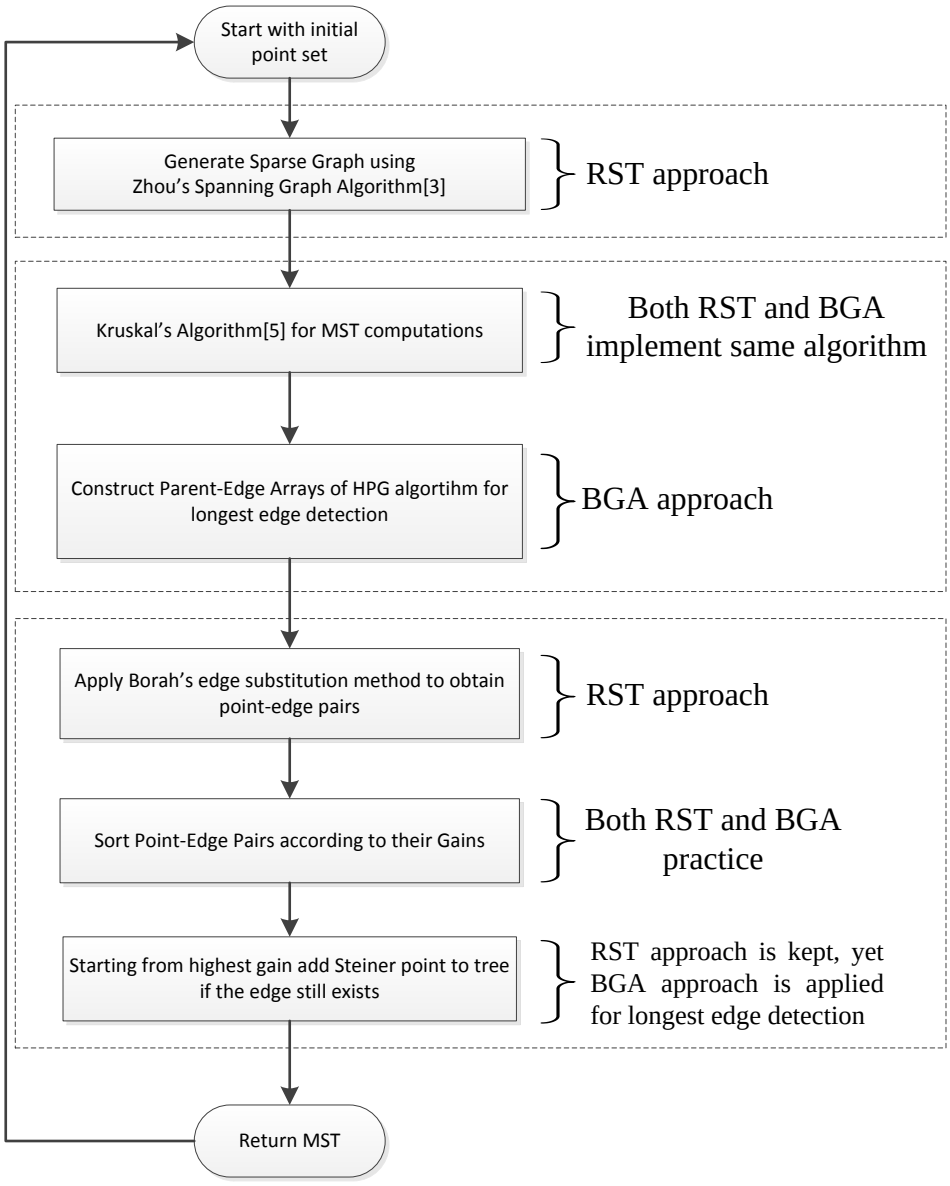

Figure 3-22: Modified RST Algorithm Flow

CHAPTER 4

GPU IMPLEMENTATION OF MODIFIED RST ALGORITHM

Modern IC designs contain billions of transistors and their complexities increases continuously. Practical industrial methods require rectilinear routing of nets with tens of thousands of terminals; moreover, several hundreds of thousands of electrically equivalent points are connected to each other. Such a problem size consumes significant routing resource in the layout and minimizing the total length becomes an important problem [3]. Although most of the electrical design automation tools implements sequential algorithms [30], there is a growing interest in parallel approaches.

In this thesis, we have chosen Modified RST algorithm to work on because it has a speed-oriented approach and has better performance compared to most Steiner tree algorithms. There is a distributed version of Modified RST [1], which works on clusters of computers having large inter-communication costs. The implementation in [1] is not scalable and it is only partially parallel. This thesis aims to parallelize the whole algorithm phases and aims to obtain a scalable algorithm having a better runtime.

Even though some steps are needed to be replaced, most of the Modified RST algorithm phases are suitable for parallel implementation. The revised version of the Modified RST algorithm is shown in Figure 4-1. For the first phase of the algorithm, i.e., sparse graph generation, Zhou's spanning graph algorithm is preferred because sweep operation can be applied to each octant independently and active set can be partitioned into small subsets using geometrical proximity information. The next step is to obtain MST on the sparse graph. Modified RST algorithm applied Kruskal's algorithm [4] for this purpose; however, Kruskal needs a sequential operation since each time minimum cost edge has to be searched in the whole tree to be included in

the MST. We, therefore, suggest using Boruvka's MST algorithm [5] because it consists of independent disjoint set *find* and *union* operations. Moreover, the longest edge detection solution is kept since the parent-edge array generation method of Modified RST is very similar to Boruvka's algorithm and hence it becomes easier to compute these arrays by modifying this phase. Finally, Borah's edge substitution method is to be practiced in the parallel approach similar to the sequential one. Since Borah considers only neighbor points as candidates, the region can be partitioned and the algorithm can be applied to each partition at the same time.

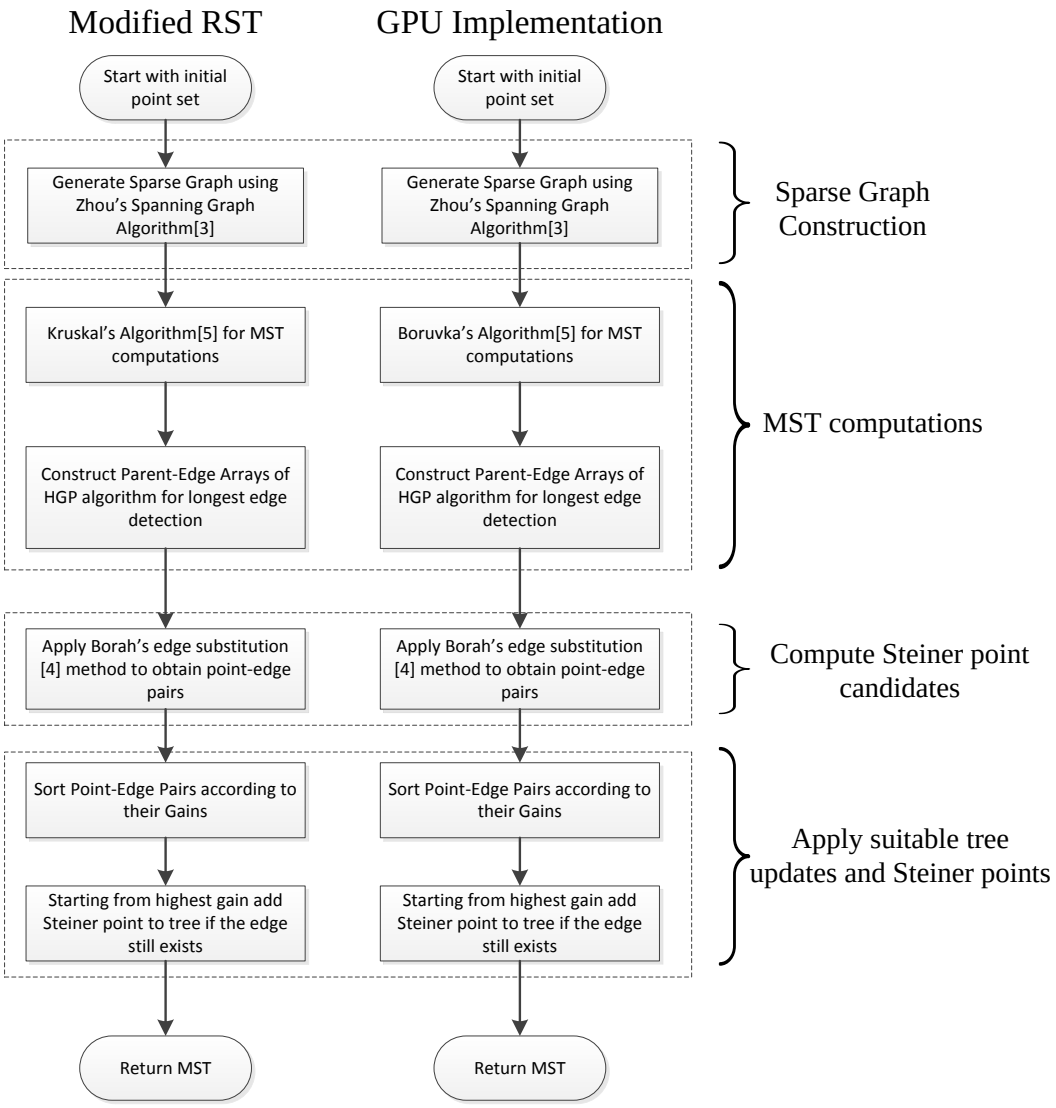

Figure 4-1: GPU Implementation of Modified RST Algorithm

4.1. Sparse Graph Construction

The details of Sparse Graph generation algorithm are given in Chapter 3.1.1. Zhou's algorithm divides the region into 8 partitions and sweeps each octant till finding the nearest neighbor. Since running the algorithm for the first four partitions is enough, sweep process is not going to be applied on R_5, R_6, R_7 and R_8.

The nearest neighbor search algorithm runs independently for each octant and hence the search operations in R_1, R_2, R_3 and R_4 regions will run simultaneously as long as there is available hardware resource.

The search computations in R_1 and R_2 partitions start with sorting the input data set according to nodes' (x +y) values. For parallel implementation of sort computations, segmented scan primitives given in [36] is to be used. The input array is divided into segments and sort operation runs on each segment concurrently. Segmented scan is suitable for quicksort since communication between threads in the algorithm is within a single segment only. The algorithm starts with choosing the first element of each segment as pivot and then the elements in the segment are compared with the pivot. While comparing the input with the pivot, both "greater than pivot" and "greater than or equal to pivot" conditions can be used. If the input is smaller than the pivot, it is moved to the head of the segment and if it is larger than the pivot, it is moved to the end of the segment. At this step each segment is divided into two subsets and the operation continues until the output is sorted. It is checked against a global reduction after each step. A small example of GPU quicksort implementation is given in Figure 4-2.

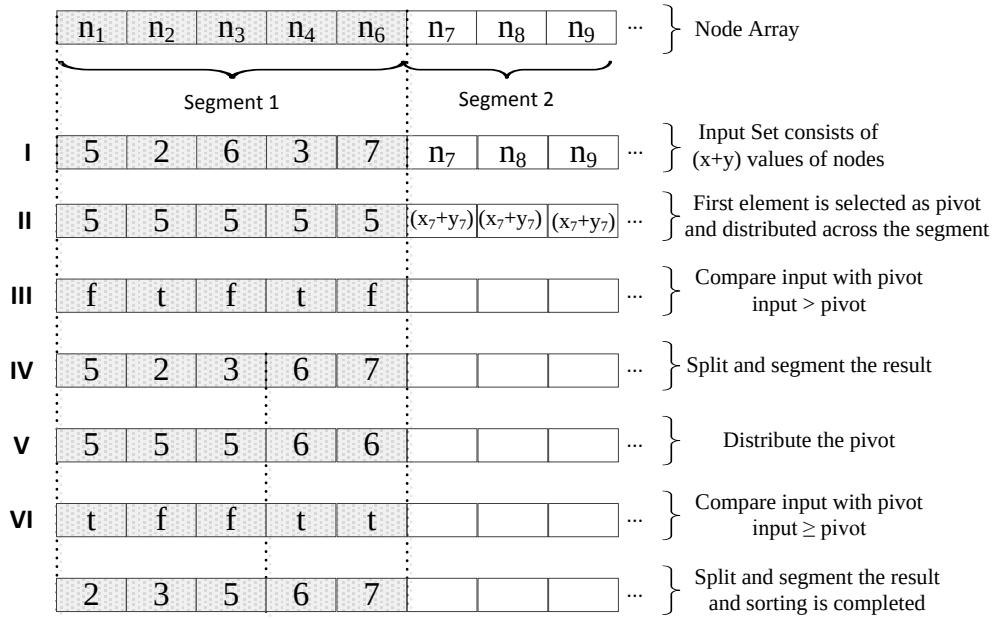

Figure 4-2: Parallel Quicksort Implementation Example

The next step of Zhou's sparse graph algorithm is to run the sweep line algorithm. For this purpose, space is divided into equal partitions and computations are done on each partition separately. A sample partitioning of space into five regions is given in Figure 4-3. Since left-most and right-most nodes are known, starting from these nodes, the plane is to be divided into equal partitions. During this partitioning of the plane, it is assumed that each partition contains the same number of nodes. Trial and error method will be used to determine how many partitions are going to be used during simulations.

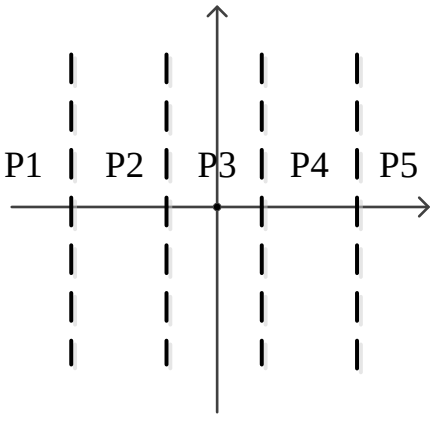

Figure 4-3: Partitioning the Space for Sweepline Algorithm

After dividing the whole plane into partitions, each CUDA core maintains an active set and computes the R_1 neighbors of the nodes contained in that partition. In order to decide whether a node is in R_1 region or not, equation given in equation (4-1) is

applied. For R_2 neighbors, the same algorithmic steps are followed; however, to classify a node whether it is in R_2 region or not, inequalities given in equation (4-2) are applied.

R_3 and R_4 nearest neighbor search operations begin with sorting input vertex set according to (x-y) and sweep operation is then computed on this sorted array. For sorting, parallel quicksort algorithm is used. Afterwards, the plane is divided into sub-regions and each CUDA core computes R_3 and R_4 neighbors for its own sub-region.

The search operation for R_1, R_2, R_3 and R_4 is to be computed simultaneously as long as there is available hardware resource. In our system, we have 384 CUDA cores, which mean 384 computations can run at the same time. The usage of GPU with more cores should increase performance considerably without changing any algorithmic steps in the implementation.

4.2. Minimum Spanning Tree Construction

Modified RST algorithm implements Kruskal's algorithm to be able to obtain MST from the sparse graph. Kruskal starts with sorting edges according to their weights and then adds the edge with the least cost to MST if both end points of the edge do not belong to the same tree. This behavior of Kruskal requires a sequential operation, i.e., for each iteration, the forest structure should be examined and then after the addition of edge, the forest should be updated. Furthermore, parent-edge array construction steps are embedded in the generation of MST; however, this kind of array construction can cause long query times. Hence, because of this sequential behavior, we do not prefer Kruskal in our MST computations.

MST problem has an irregular structure; thus, it is suitable for sequential implementations. The existing sequential algorithms such as Prim or Kruskal are efficient to use; however, industrial applications require more scalable solutions because input terminal sets are getting bigger and bigger every day. Therefore, there is a growing interest in the parallel solution for the MST problem. There are a couple of parallel algorithms for the MST such as [37], [38], [39] and [40]. Most of these algorithms are based on Boruvka [5] due to its parallel nature.

In our GPU implementation of RSMT algorithm, we propose Boruvka's MST solution instead of Kruskal and our implementation becomes similar to [38].

4.2.1. Boruvka's MST Algorithm

Otokar Boruvka suggested its MST algorithm to construct an efficient electricity network in 1926. This is a greedy algorithm based on union and find operations. The basic flow of Boruvka's algorithm is given in Figure 4-4. At the beginning of the algorithm, each node in the spanning graph forms a disjoint set and then "find" operation is applied, where the minimum cost edges outgoing from nodes are searched. The minimum cost edge is directed from the node to its neighbor. After this operation, bidirected, unidirected and undirected edges are generated in the graph. The nodes of bidirected and unidirected edges are merged into a single component. This search and merge operation continues till spanning graph collapses to a single component, this final component representing the MST of input data set. Unlike Kruskal's algorithm, Boruvka's algorithm proceeds in an unordered pattern.

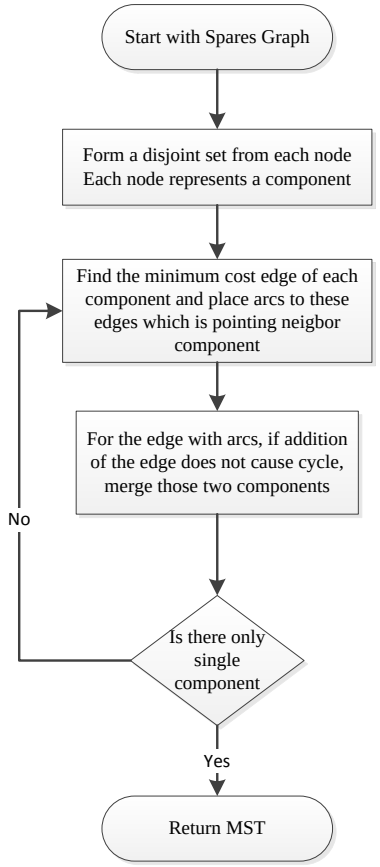

Figure 4-4: Boruvka's Algorithm Flow

A MST construction process on sample spanning graph is given in Figure 4-5. Initially, each node is defined as a separate disjoint set. Afterwards, the minimum cost edges for these disjoint sets are found and arcs placed on these edges. As a result of first iteration, there exists 2 bidirected, 3 unidirected and 5 undirected edges.

Nodes at the end points of these edges are merged into a single component, node *a*, *b*, *e* and *g* forms a component and nodes *d*, *e* and *f* forms another component. Then, the algorithm repeats itself by searching outgoing edges of new components.

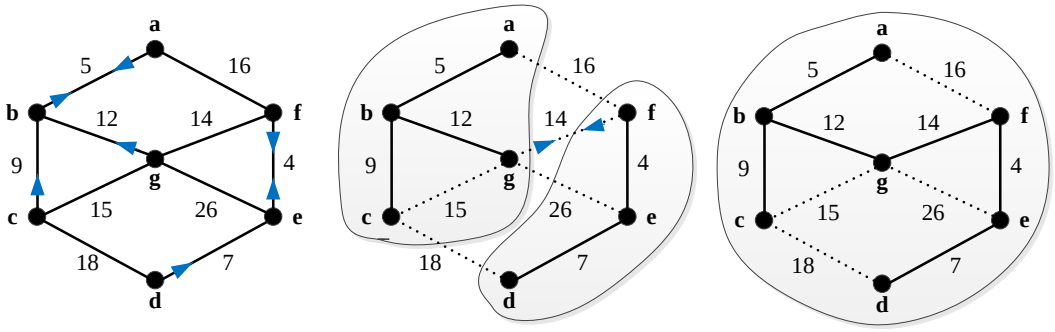

Figure 4-5: Boruvka Algorithm on an Example

4.2.2. Parallel Boruvka Implementation

Our parallel Boruvka implementation is similar to [38]. In addition to MST computations; we generate parent-edge arrays for the longest edge detection problem during this operation.

The algorithm starts with defining disjoint sets from each node; these disjoint sets could be called as components. A unique component id is assigned to each node. The nodes and their weights are stored in one array and the edges generated after sparse graph construction is stored in another array (Figure 4-6). In addition to node indexes and their weights, node array contains pointers to edge array, which shows the partition of elements the node is connected [38]. For the minimum cost edge search operation the segmented scan given in [36] is to be used and hence a flag array is needed to be able to specify the segments of the edge array. In edge array, each segment represents edge connections of a single node. For instance, in Figure 4-6, node n_1 is connected to edge e_1, e_2 and e_3, node n_2 is connected to edge e_2 and e_6. In the edge array, e_1, e_2 and e_3 is the segment of node n_1; in addition, edges e_2 and e_6 is the segment of node n_2. To be able to specify each segment, the first element of the segment in the flag array is set to '1' and the rest of the segment is set to '0'.

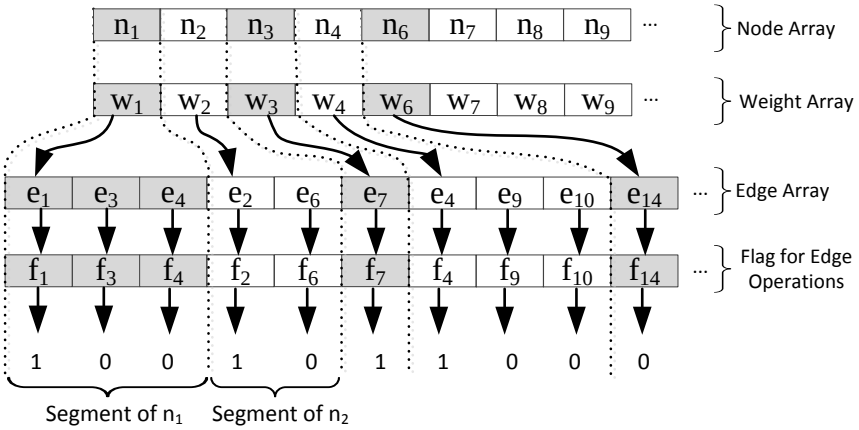

Figure 4-6: The Graph Structure

The minimum cost edge of each node is computed using segmented scan algorithm and the result is stored in a new array, called NWE; nevertheless, only the edge that does not cause cycles will be included in the MST. In other words, the end points of the edge are controlled, if both end points belong to the same component, the edge is not included in the tree. For cycle detection, a successor array is generated to hold the outgoing vertices from node *v* to node *u*. The NWE array is used to construct the successor array. In the successor array, the nodes that obey S(S(*u*)) = *u* condition forms cycles and related edges are discarded from MST. Successor array creation for the graph in Figure 4-5 is illustrated in Figure 4-7. For the next iteration, the first element in the node array is selected as representative for each subset and successor of each node is updated according to this information.

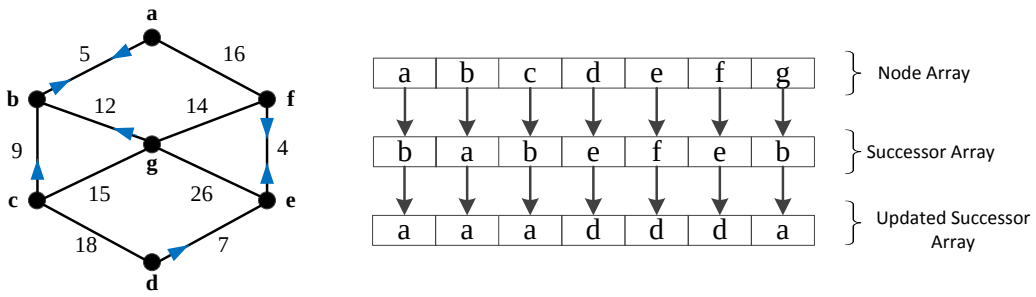

Figure 4-7: Successor Array Structure

Once an appropriate edge is found, it is added to MST and two disjoints sets (or components) at the end points of this edge are merged. The components are split according to their component ids and the new component ids assigned to each node that participates in merge operation. In the meantime, edge array is sorted with respect to new component numbers and flag array is reorganized to reflect new boundaries of segments.

In order to ease computations of the next iteration, some edge list is arranged during the merge operation. The edge list is reduced such that the edges that lie between same components are discarded from the list. Hence, search computation does not deal with these edges. Moreover, the end points of the edges are updated considering the new component ids and the successor array is updated and set to $S(u) = S(S(u))$.

As was stated earlier, parent-edge array for the longest edge detection problem is constructed in this step. The array generation of BGA algorithm is very similar to Boruvka's algorithm, thus, these arrays are generated without any overhead. The smallest weight node obtained for each component at the beginning of this step can be used as edge(u) and the new component id after merge operation can be used as parent for that node, parent(u). For instance, in Figure 4-5, after the first iteration, parent and edge arrays become as follows:

nodes = { a, b, c, d, e, f, g}

edge = { ab, ab, bc, de, ef, ef, bg, NIL, NIL, NIL, NIL, NIL, NIL}

parent = { x, x, x, y, y, y, x, NIL, NIL, NIL, NIL, NIL, NIL}

4.3. Edge Substitution

In the Modified RST algorithm, Borah's edge substitution method is used in order to generate Steiner point candidates. For each edge, the neighbors of the edge end points are considered as possible node-edge pairs and their gains are calculated if a corresponding cycle does not occur for that node-edge pair. Since node-edge pairs are computed only for neighbor nodes of the edge's end points, we can use the geometrical proximity information obtained from sparse graph operations and compute gains in parallel.

As a result of the MST computations, the set of edges, which are included in MST is obtained and they are stored in an array. This array is segmented among CUDA cores of GPU uniformly and each core computes the node-edge pairs for its segment. Each core has access to parent-edge arrays constructed during MST computations, and these arrays are used to detect longest edges in the cycles formed. A sparse graph and its MST tree are given in Figure 4-8. Assuming that there are four GPU cores available, the edge array is divided into 4 segments and each core computes node-edge pairs, separately. For example, $core_1$ works on $segment_1$ and it searches available Steiner points for edges "ab" and "bc". The candidate nodes are selected

from sparse graph neighbors. For edge ab, nodes *c*, *d*, *e*, *f*, *g* and *h*; for edge "bc", nodes *a*, *d*, *e*, *f*, *g* and *h* are candidates. In the meantime, $core_2$ computes node-edge pairs for $segment_2$, that is, edge "bd" and edge "de". Nodes *a*, *c*, *e*, *f* and *g* are computed for edge "bd".

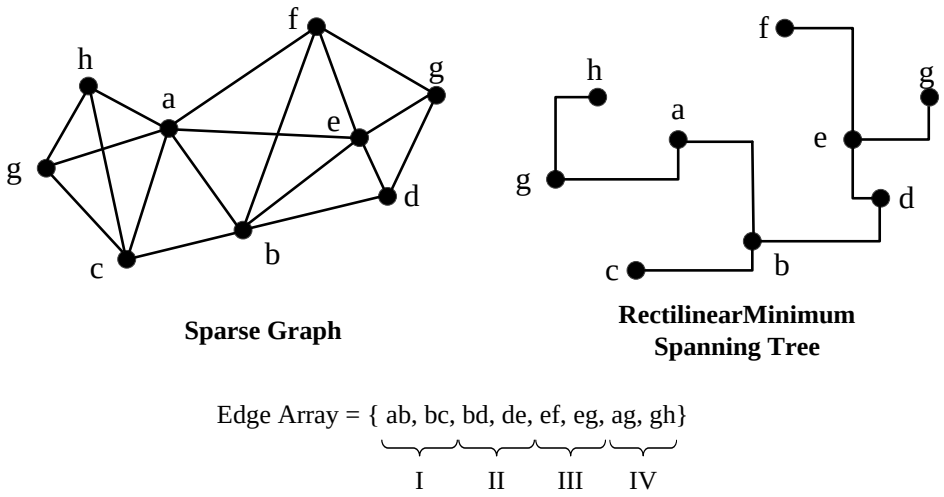

Figure 4-8: Segmented Borah's Edge Substitution

Afterwards, gain is calculated for each candidate point and the point with the highest gain is chosen for the target edge. An example can be depicted in Figure 4-9, where, for edge ag, two possible Steiner points are discovered.

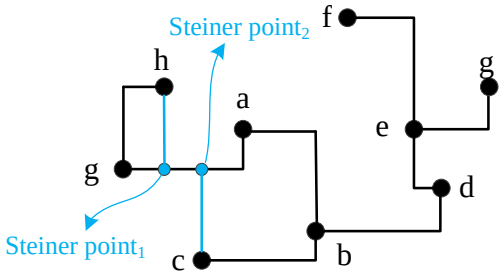

Figure 4-9: Steiner Point Candidates

Gain calculations begin with the detection of the longest edge in the cycle. For Steiner $point_2$, a cycle is created between nodes *a*, *b*, *c* and sp_2. Using parent-edge arrays, a search operation is conducted and longest edge in the loop is accessed. Subsequently, the gains for node-edge pairs are calculated. Assuming that the longest edge is edge gh for $loop_1$ and edge bc for $loop_2$, the calculations for Figure 4-9 is given in equation (4-1) and (4-2).

$$gain(\text{Steiner point}_1) = ||g,h|| - ||h, sp1|| \qquad (4\text{-}2)$$

$$gain(\text{Steiner point}_2) = ||b,c|| - ||c, sp2|| \qquad (4\text{-}3)$$

Following these calculations, the point with the highest gain is selected and the corresponding node-edge pair is generated. The node-edge pair, longest edge in cycle and their gain are kept in arrays as shown in Figure 4-10. Each core updates the related part of these arrays after its computations and hence all data is stored in a single array. If a point with positive gain for an edge is not found, it is marked as NIL.

I II III IV

Edge Array = { ab, bc, bd, de, ef, eg, ag, gh}

Pair Point Array = { e, a, e, NIL, g, d, c, NIL}

Longest Edge = { bd, ab, de, NIL, eg, de, bc, NIL }

Gain Array = { g_1, g_2, g_3, NIL, g_5, g_6, g_7, NIL }

Figure 4-10: Point-Edge Pairs and Their Gains

During simulations, edge array will be divided into 64, 128, 256 and 512 segments and the performance will be evaluated. Although, for each division, significant run time improvement is expected, the best choice for our case is going to be obtained after some experiments.

4.4. Graph Update

The last step of the implementation is the amendment of the graph. The gains obtained from Borah's edge substitution method are stored unorderly in the gain array. This array is sorted with parallel quicksort algorithm in descending order and starting from highest gain the Steiner points are added to the tree sequentially.

Meanwhile, for each edge, a flag is maintained to decide whether it is involved in an update operation or not. The flags are set to true at the beginning of the update action and when an edge is deleted during the graph update, the edge's flag is complemented to false. If the edge's flag is false, the Steiner point for this point-edge pair is skipped and is not included in the tree.

CHAPTER 5

PERFORMANCE ANALYSIS

In this chapter, the GPU implementation and analysis results of the Rectilinear Steiner Tree problem solution is presented. Time measurements for sequential and parallel solutions are explained and the achieved speed up values for each input terminal set is discussed.

5.1. Implementation Environment

Our sequential algorithm is implemented for CPU in C language. For sequential implementation, Microsoft Visual Studio 2010 on Windows 8.1 is selected as software development environment. The brief specification information of CPU hardware is given in Table 5-1.

Table 5-1: CPU Hardware Specifications

Processor Model	Intel i7 U Series
Number of Cores	2
Number of Threads	4
Processor Base Frequency	2 GHz
Max Turbo Frequency	3.1 GHz
Installed Memory	8 GB
Instruction Set	64 Bit
Passmark Benchmark Result	3942

The parallel algorithm is implemented using C language with CUDA extensions. CUDA toolkit 6.0 is used for parallel code development environment. The GPU hardware specifications are given in Table 5-2.

Table 5-2: GPU Hardware Specifications

Device Model	Nvidia GeForce 840M
GPU Architecture	Maxwell
CUDA Capability	5.0
Number of CUDA Cores	384
Number of Multiprocessors	3
Total Amount of Global Memory	4 GB
GPU Clock Rate	1.12 GHz
Memory Clock Frequency	900 MHz
Memory Bandwidth	16 GB/s

5.1.1. CPU and GPU Performance Comparison

Our implementation environment consists of two processing elements. The first one is Central Processing Unit, which is usually used for general purpose applications. CPU is optimized to improve the performance of serial applications. A CPU core executes the instructions of the same thread in parallel or out of order. It contains complex control logic such as branch prediction, data prefetching and out of order execution and large caches. However, there are some limitations about CPU performance, for instance, its operation clock frequency is fixed around 3 GHz due to power consumption issues and its memory bandwidth is low. Therefore, recent CPUs have multiple cores to execute multiple threads at the same time.

The theoretical speed up and the number of processors relation is defined by Amdahl's law. Amdahl's law can be stated as Eq.5-1 where T(n) is the total runtime with n cores and S is the total runtime of serial portion of the application. The speed up of an algorithm improves with the increase in the number of cores and it is limited by the serial portion of the application.

$$T(n) = S + \frac{T(1) - S}{n} \qquad (5.1)$$

As shown in Figure 5-1, an application consists of a serial and a parallelizable portion. The parallelizable portion is distributed across some number of cores and speed up increases in accordance with the number of cores.

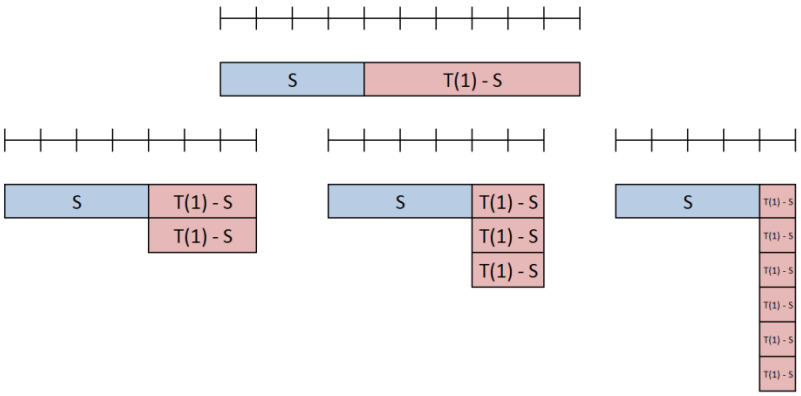

Figure 5-1: Serial and Parallel Portions of an Application

Although our system contains a CPU with two cores, we implemented an optimized single thread version of Rectilinear Steiner Tree problem. After observing the theoretical speed up values given in Figure 5-2, we can conclude that multi thread CPU implementation cannot obtain enough speed up values to overcome GPU implementation.

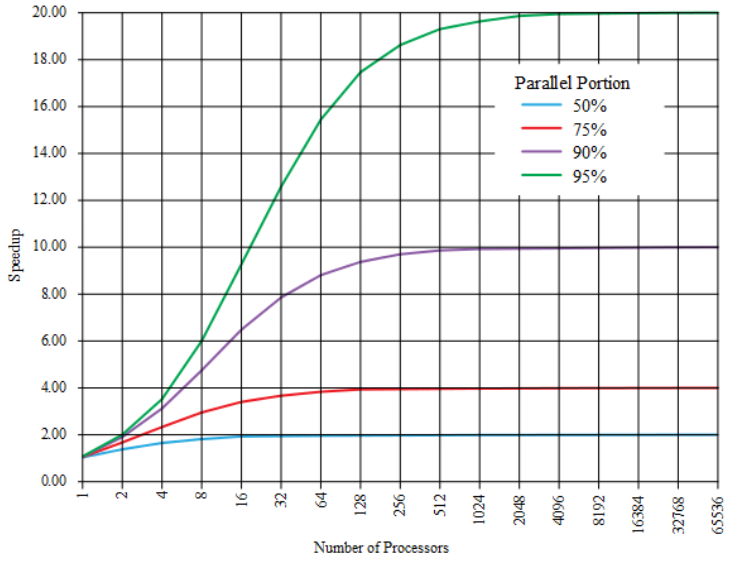

Figure 5-2: Theoretical Speed Up According to Amdahl's Law [48]

Amdahl's law neglects inter processor communication overhead and memory latencies. When we consider high memory bandwidth and memory latency hiding features of GPU, we can state that multi thread CPU implementation does not affect our speed up measurements significantly; hence, we decided to run our experiments for single thread CPU implementation.

5.2. Simulation Results

The test plan of our implementations consists of using two sets of problem instances. Initially, we generated random test points that are uniformly distributed in the region where the number of input terminals range from 1.000 to 1.000.000. Next, a set of VLSI test cases, which are extracted from recent industrial designs will be used [41]. There are 8 test instances in this set and the input size varies from 337 to 34.728. By using these VLSI benchmarks, we were able to observe the experimental results of our algorithm on realistic applications.

5.2.1. Random Test Points

5.2.1.1 Data set

To evaluate our algorithm, first, we implemented it sequentially for CPU and then in parallel for GPU. We used GeoSteiner 5.0 software, which can be found in [46], to generate the random node sets. The number of nodes for these benchmarks scales from 1.000 to 1.000.000. For each benchmark, ten different random node sets are generated and the average of the speed up values of these different node sets is reported in this study. The nodes are uniformly distributed in the region.

5.2.1.2 GPU Implementation Optimization

Performance tuning is an important issue in CUDA programming. A key decision for this purpose in CUDA is the thread structure. Thread size and shape of a kernel can affect the overall algorithm performance significantly.

One of the basic principles to achieve good performance is to keep all execution units in the device as busy as possible. If the algorithm implemented is poorly balanced across the existing multiprocessors, it delivers suboptimal performance. Hence, we tried to maximize hardware utilization by arranging the thread and block sizes in our algorithm efficiently. The most common and intuitive effort for this purpose is occupancy. Occupancy is defined as the ratio of the active warps per multiprocessor to the maximum number of possible active warps. Although higher occupancy does not always achieves higher performances, low occupancy causes memory latencies resulting in performance degradation.

After implementing our GPU algorithm, we applied CUDA occupancy calculator provided by Nvidia [49]. The impact of block size on the occupancy for R_1 neighbor search operation is given in Figure 5-3. Our time measurements are consistent with

Figure 5-3; however, we did not observe significant variations in kernel time between block sizes 256, 512 and 1024. Hence, we kept block size parameter as 512 in our simulations.

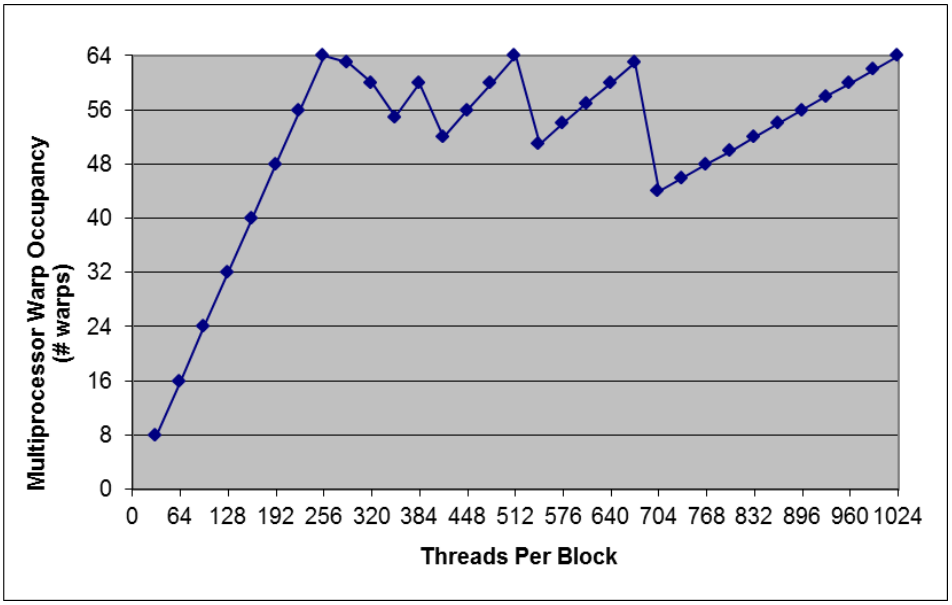

Figure 5-3: Impact of Varying Block Size on Occupancy

5.2.1.3 Experimental Results

RST simulations have been performed on ten test cases for each node set size and the average runtimes are given in Figure 5-4 and in Table 5-3. The runtime measurements have less than or equal to 5% error margin with 90% confidence interval.

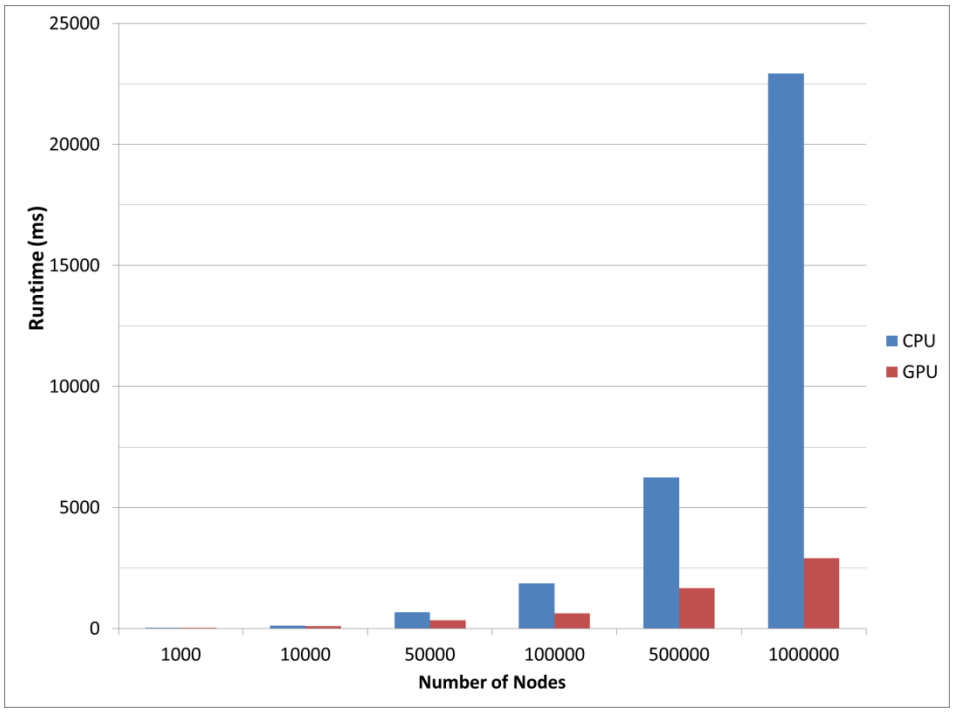

Figure 5-4: Total Runtime of RST Algorithm on CPU and GPU

Table 5-3: Total Runtime of the Algorithm

Number of Nodes	CPU time (ms)	GPU time (ms)
1.000	27.77	30.01
10.000	112.11	91.40
50.000	665.68	342.90
100.000	1855.10	620.09
500.000	6241.39	1662.38
1.000.000	22936.72	2905.40

The overall improvement in runtime can be seen in Figure 5-5. The Speed-Up values given in Figure 5-5 have less than or equal to 5% error margin with 90% confidence interval and they are obtained using Eq. 5-2.

$$\boldsymbol{Speed\ Up} = \frac{\boldsymbol{Total\ CPU\ Runtime}}{\boldsymbol{Total\ GPU\ Runtime}} \qquad (5\text{-}2)$$

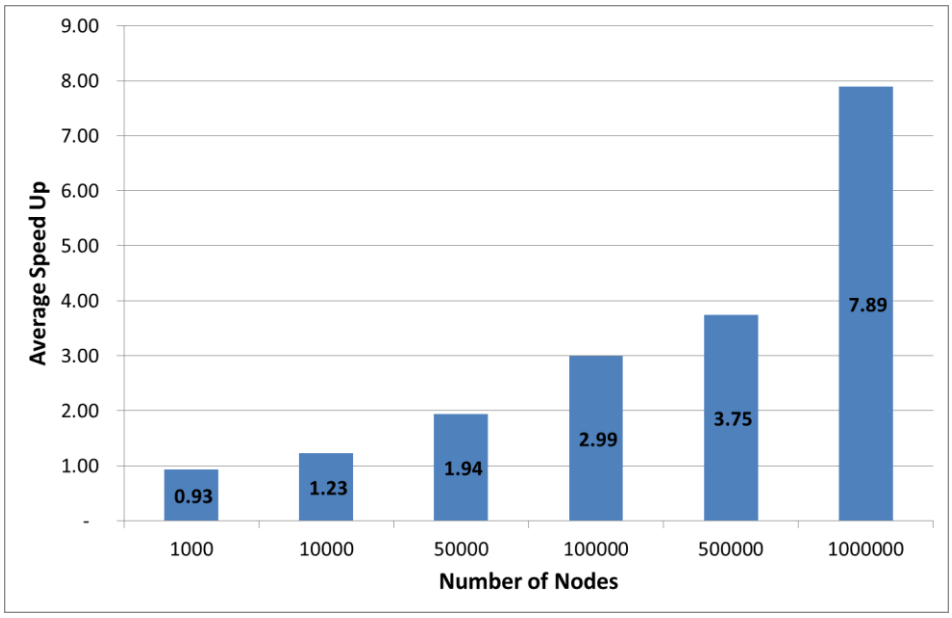

Figure 5-5: Average Speed-Up of the GPU implementation

Our GPU algorithm shows a reasonable improvement over the CPU implementation and a promising strong scaling. During our study, we proved that up to 7.89x performance increase in total algorithm time could be achieved with our GPU implementation compared to CPU implementation. We achieved a significant runtime improvement for all data set sizes except 1000 nodes. Working with larger data sets such as 1.000.000 nodes rather than smaller ones results in much better speed up.

Speed up of our parallel implementation increases with the input set size as shown in Figure 5-5. There are a couple of reasons why performance gap increases with the data size. First, GPU is composed of hundreds of execution units and it can handle thousands of threads simultaneously; as a result, it can perform the same operation over and over very quickly for huge data. However, CPU is composed of only a few cores; therefore, it can handle only a few application threads at a time. Therefore, when the total number of threads increases, due to the sequential nature of the CPU, GPU outperforms on our algorithm. Next, since a large number of blocks and threads run concurrently on GPU and it has a large memory bandwidth, it is more tolerant to memory latencies than CPU. Moreover, GPU contains on-chip shared memory, which is much faster than local or global memory. It is allocated for each thread block independently, so all the threads in the block can access the same shared memory. Use of this shared memory facilitates such a high performance by making neighboring node in the data array instantly available for computation, for instance,

in neighbor search or sorting. Consequently, GPU has a special memory architecture, which is suitable for parallel operations and performance gain increases on large data sets. Lastly, at the beginning of the execution, the input data set is copied from CPU memory to GPU memory, where copying data from/to GPU memory is a comparatively slow activity; meaning that, in addition to the application execution time, an extra memory overhead is added to the total runtime. This memory overhead increases proportionally with the data size; nevertheless, the CPU execution time has an exponential increase trend as was shown in Figure 5-4. Thus, the impact of memory overhead decreases for large data size and overall performance still increases with the increasing data size.

Speed up is not observed only for the first data set, which is the smallest of all. The first reason for this result is the memory overhead, data copying process to GPU local memory is a slow operation and hence it deteriorates the performance significantly especially for small data sets. Next, CPU cores are optimized for sequential serial processing, they run at higher frequencies compared to GPU and they are faster in terms of the instruction per cycle perspective. Although GPU has a large number of execution units, these units operate with a lower clock frequency and their hardware architecture is much simpler compared to CPU. Hence, for small data sizes, performance gain obtained from GPU parallel architecture is not sufficient to overcome CPU performance.

As was explained in Chapter 4, our improved version of Modified RST algorithm consists of three major blocks, namely, rectilinear sparse graph construction, minimum spanning tree construction and Steiner point operations. The average runtimes of these algorithm blocks for largest three data sizes are shown in Figure 5-6. Moreover, the ratio of the runtime of each sub-block to total runtime for these benchmark cases is given in Figure 5-7.

By analyzing Figure 5-6 and Figure 5-7, we can conclude that MST construction is approximately 25% of the total runtime for all data set sizes. Rectilinear sparse graph generation block takes more time compared to other blocks for small data sizes like 1.000 nodes. However, for large data sizes, the time spent for Steiner point operation such as Steiner candidate search and tree update is more dominant than the RSG and MST constructions.

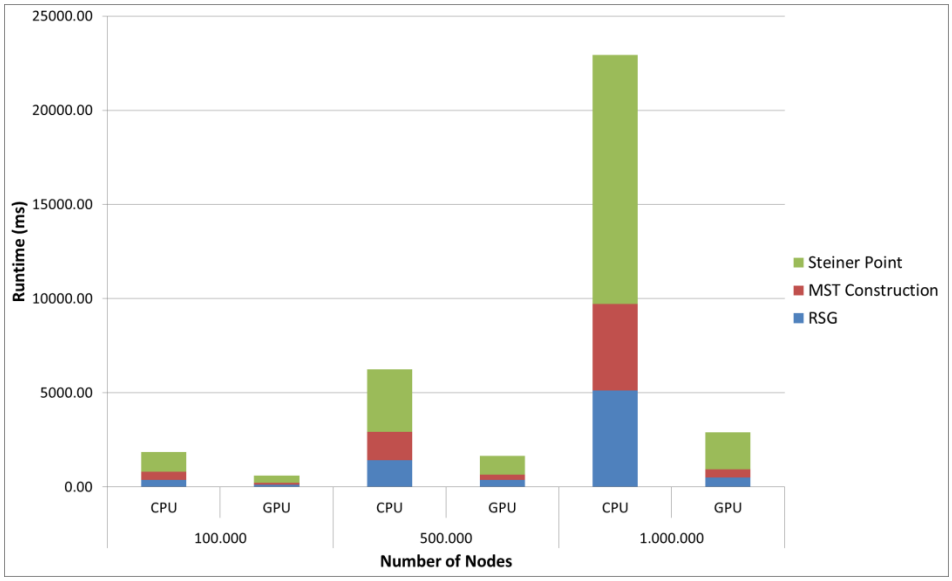

Figure 5-6: Average Runtimes of RST Algorithm Blocks

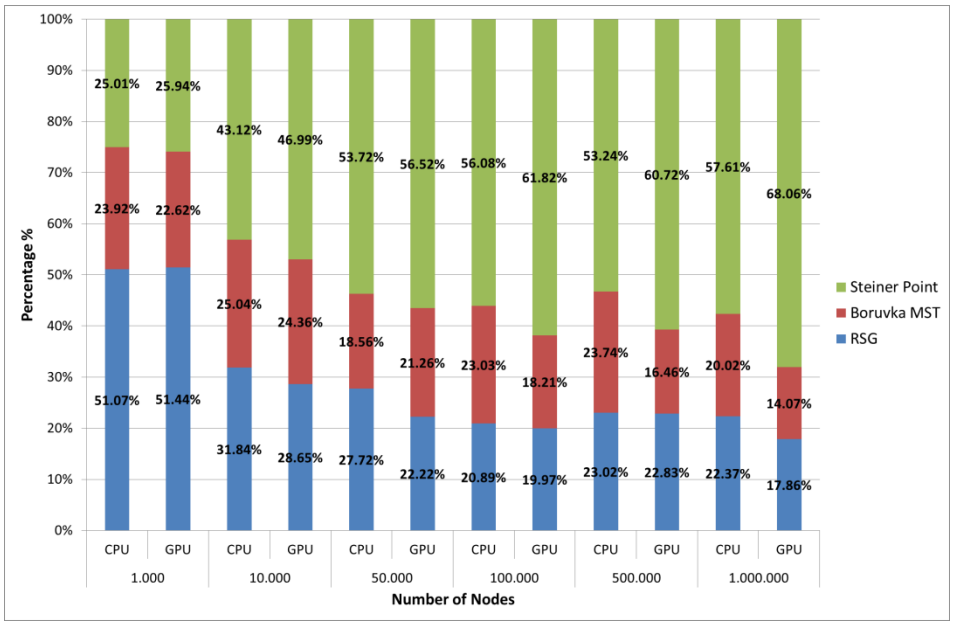

Figure 5-7: Runtime Ratio of RST Algorithm Blocks

Average speed up values for RST steps is illustrated in Figure 5-8. We improved total runtimes of each sub block for almost every data set. Only for the smallest data set, speed up is less than one.

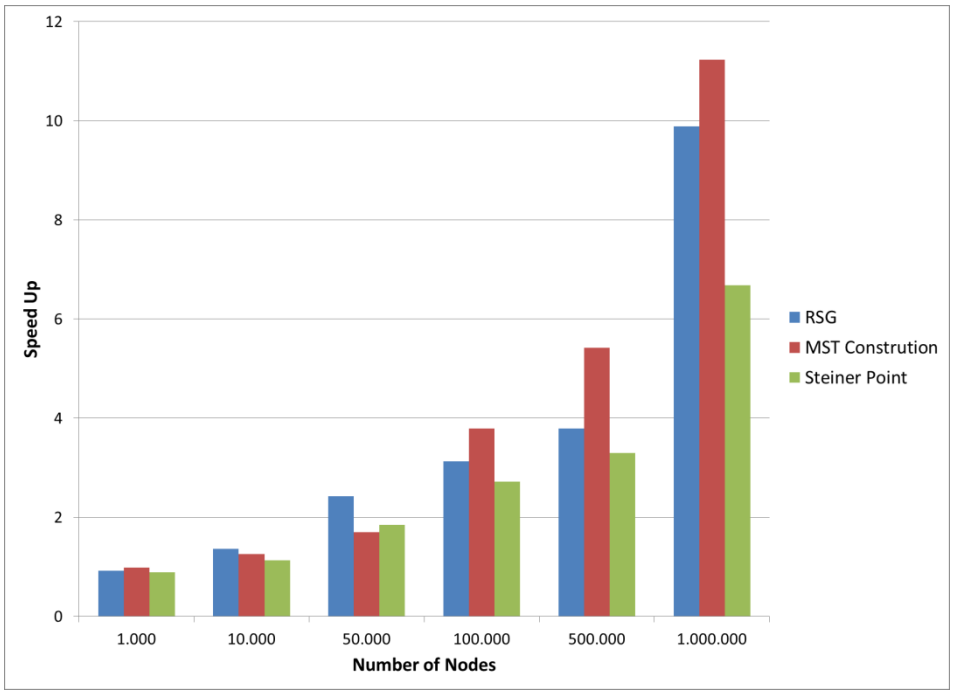

Figure 5-8: The Average Speed-Up of the RST Algorithm Blocks

The larger the data set size, the higher the achieved speedup as expected. The highest speed up of 11.2x is accomplished for the MST algorithm. Following Boruvka's MST construction algorithm, with parallel implementation of RSG construction algorithm, 9.8x speedup is obtained. Finally, up to 6.7x speed up is achieved for the Steiner point operations block. Since the Steiner point operation runtime is higher compared to other algorithm blocks, Steiner point block has more influence on the overall performance of the algorithm.

In order to decrease the total length of the Rectilinear Steiner Tree and increase the quality of the solution, the Steiner point search algorithm is repeated more than once and after each pass the tree is updated. This repetitive sequential structure of this block is one of the main reasons why higher speed up is not obtained in Steiner point operations in comparison to MST construction.

5.2.1.4 Algorithm Performance

The quality of the RST algorithm is measured as the total improvement by the algorithm in the length of the MST. Eq.5-3 is used improvement calculations.

$$\textbf{\textit{Total Lenght Improvement}}\ \% = \frac{\textbf{\textit{Lenght of MST}} - \textbf{\textit{Lenght of RST}}}{\textbf{\textit{Lenght of MST}}} \times \mathbf{100} \qquad (5\text{-}3)$$

MST length improvements by our methods are given in Figure 5-9.

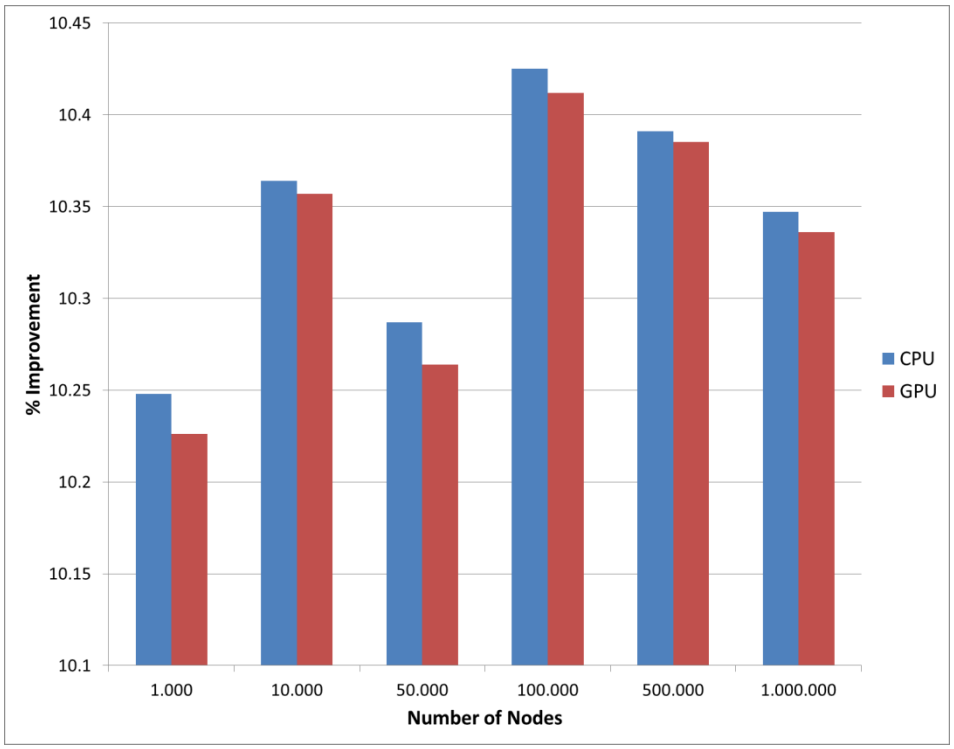

Figure 5-9: Steiner Tree Length Improvement over the MST

Even for small data sets, significant length improvements have been obtained. Since the nodes are distributed uniformly in the region, total length improvements are close to each other for all benchmarks.

Length improvement for GPU implementation is slightly lower than the sequential CPU implementation. In order to be able to take advantage of parallel features of GPU hardware, we have applied some optimizations in our code to increase memory access bandwidth and to improve thread parallelism; thus, we did not obtain exactly the same length reduction in our parallel implementation. However, since this difference is acceptably small, we did not attempt to increase the improvement level further for the GPU implementation.

Improved version of the modified RST algorithm proposed in this study achieves a similar performance in comparison to RST and Modified RST algorithms found in the literature. The comparison of our implementations, namely, sequential CPU and parallel GPU, with other algorithms is given in Table 5-4. Therefore, we show that using our algorithm significant speed up can be accomplished without deteriorating the algorithm's quality wise performance.

Table 5-4: Length Improvements in the Literature

Input Size	RST [1]	Modified RST [1]	CPU	GPU
10.000	10.43%	10.42%	10.36%	10.36%
50.000	10.43%	10.43%	10.29%	10.26%
100.000	10.45%	10.45%	10.43%	10.41%
500.000	-	10.44%	10.39%	10.38%
1.000.000	-	10.45%	10.35%	10.33%

5.2.2 VLSI Test Instances

Following our initial experiments, we evaluated our implementation on some real world problems, and for this purpose, used VLSI test instances. In order to make a fair performance comparison, we obtained the executable files of RST and Modified RST algorithm in [1] and obtained time measurements on the same platform on which we run our current implementations.

5.2.2.1 Data set

There are eight test instances in this set and the input size varies from 337 to 34.728 nodes. The input sizes of the instances and performance results of the algorithms in the literature can be seen from Table 5-5. The data sets of VLSI instances are taken from [3].

Table 5-5: Comparison of RSMT Algorithms on VLSI Instances [3]

Input Size	MST		Prim-Based		BOI		BGA		GeoSteiner	
	Len.(µm)	CPU (s)	%Imp.	CPU (s)	%Imp.	CPU (s)	%Imp.	CPU (s)	%Imp.	CPU (s)
337	247.7	0.0020	5.96	0.000	6.50	0.060	6.43	0.040	6.75	16.040
830	675.6	0.0055	3.10	0.010	3.19	0.320	3.20	0.080	3.26	9.480
1944	452.2	0.0165	6.86	0.040	7.77	3.640	7.85	0.400	8.15	1304.270
2437	578.8	0.0217	7.09	0.040	7.96	5.740	7.96	0.680	8.34	13425.310
2676	887.2	0.0235	8.07	0.040	8.99	5.340	8.93	0.770	9.38	430.800
12052	2652.7	0.1378	7.65	0.180	8.46	540.840	8.45	5.230	—	—
22373	13962.5	0.3419	8.99	0.480	9.83	2263.760	9.85	13.060	—	—
34728	9900.5	0.5455	8.16	0.690	9.01	5163.060	9.05	24.200	—	—

5.2.2.2 GPU Optimization

The shared memory usage, the register usage and the kernel structure for VLSI test cases are also the same as random cases. Consequently, the performance tuning solution explained in Chapter 5.2.1.2 is found to be valid for this case, too. We did not observe significant time measurement differences for block sizes 256, 512 and 1024 and hence we selected blocks size to be 256 threads for this benchmark set.

5.2.2.3 Experimental Results

Time measurements for VLSI test cases are illustrated in Figure 5-10. Results obtained from implementations given in [1] are represented as RST and Modified RST bars, our sequential and parallel implementations are represented as CPU and GPU, respectively.

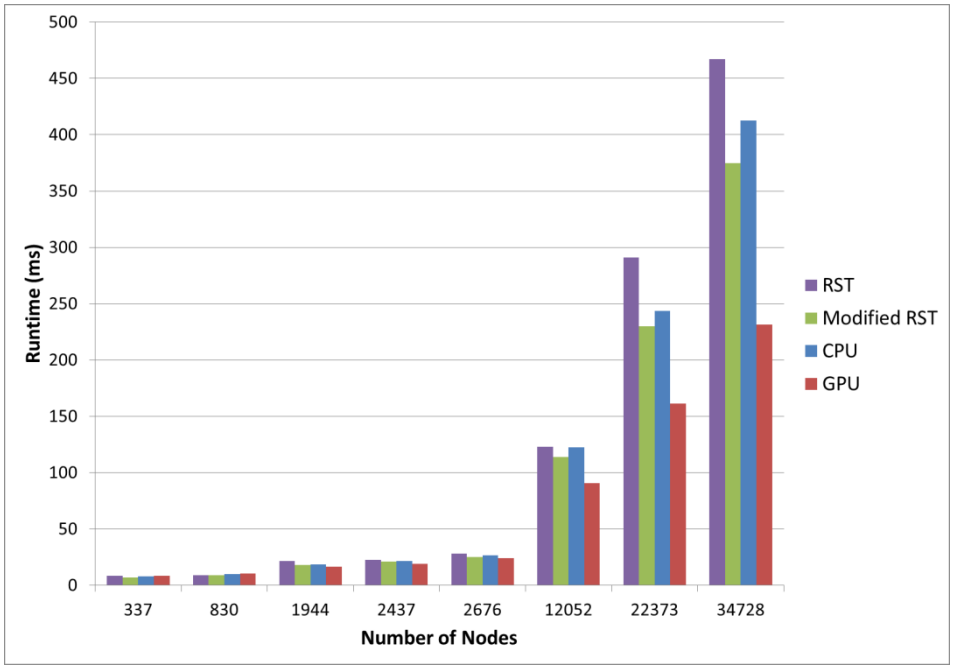

Figure 5-10: Total Runtime of RST Algorithm on CPU and GPU

Overall runtime of our sequential implementation is higher than the Modified RST algorithm. The difference between these two algorithms is mainly due to their MST computations. We have chosen Boruvka's algorithm as the MST construction solution instead of Kruskal's algorithm in this work. Although Boruvka's algorithm is very suitable for parallel operation, it deteriorates the sequential runtime performance.

The speed up obtained for VLSI cases are given in Figure 5-11. Our parallel algorithm accomplishes noteworthy speed up for all instances other than the first

two. Unavoidably, the performance of the CPU implementation is better than GPU implementation for small data set sizes. However, when data set size becomes large, GPU implementation outperforms CPU implementation. Starting from the 1944 node case, significant speed up values are observed.

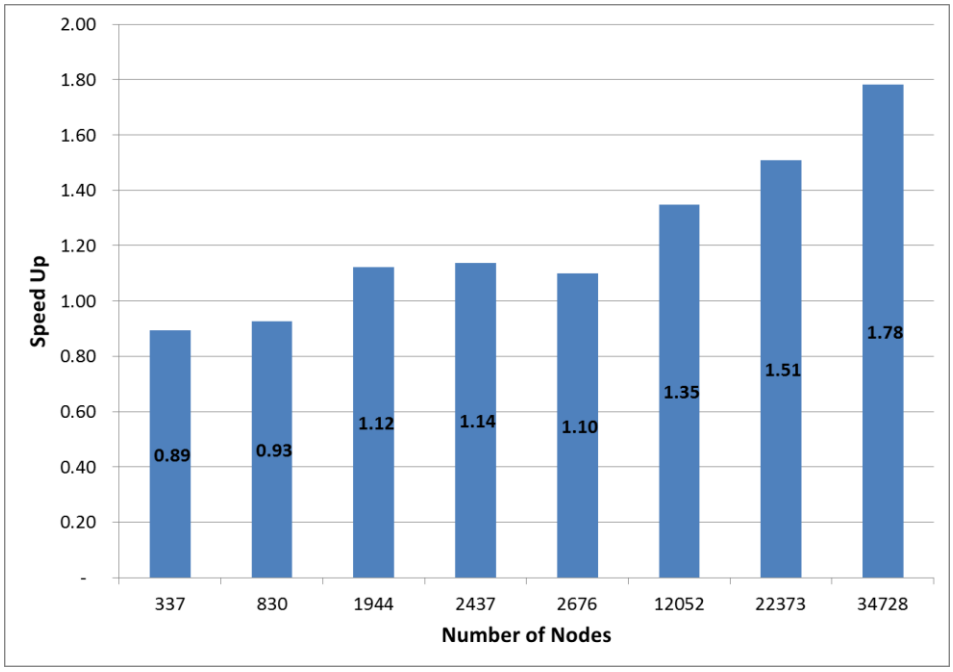

Figure 5-11: The Speed-Up of the GPU implementation

For the largest data set, highest speed up of 1.78x is achieved. Although this speed up is relatively smaller than the highest speed up for random cases, it is still not negligible. In real world applications, for example a CAD tool might be attempting to connect the terminals of a large number of nets such as ground, power or some other signals during the auto routing process. Therefore, more than one RST run corresponding to each net can be required to solve the auto routing problem.

The runtimes of RST blocks for eight VLSI test instances are shown in Figure 5-12. Moreover, the ratio of the runtime of each sub-block to total runtime for VLSI test cases are given in Figure 5-13 and Figure 5-14 and results similar to random cases have been observed. MST construction time is approximately 25% of the total runtime irrespective of the number of nodes in the benchmark. For small data sizes, this value is slightly higher than large data sets. As illustrated in Figure 5-13, RSG block consumes more time than the other two blocks for small data sizes. Nevertheless, Steiner point operations sub block becomes more dominant in overall runtime as seen in Figure 5-14.

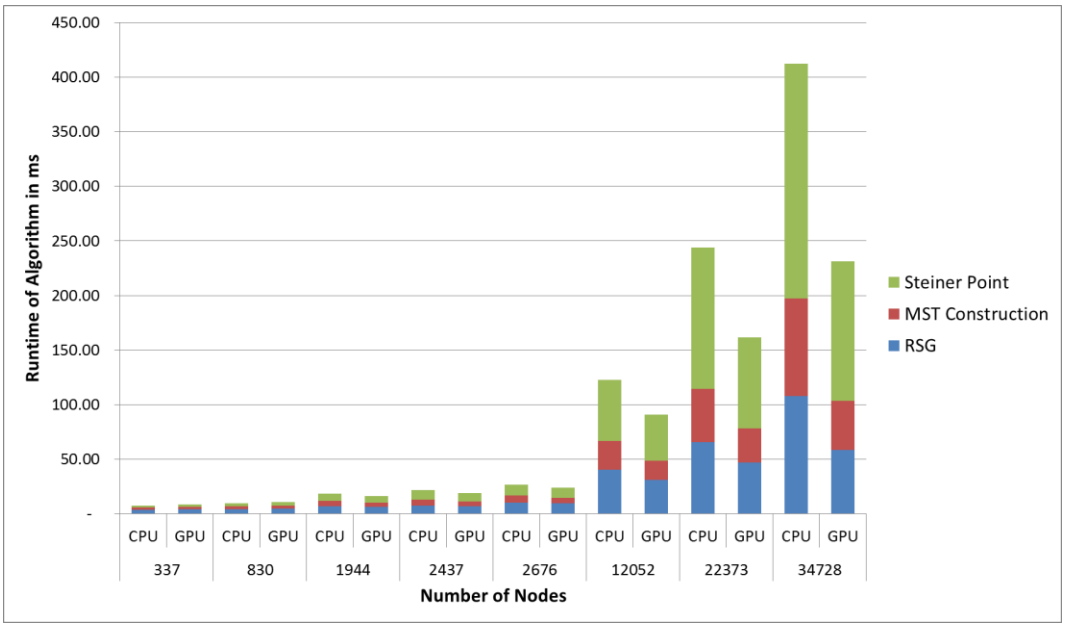

Figure 5-12: Runtimes of RST Algorithm Blocks on VLSI Test Cases

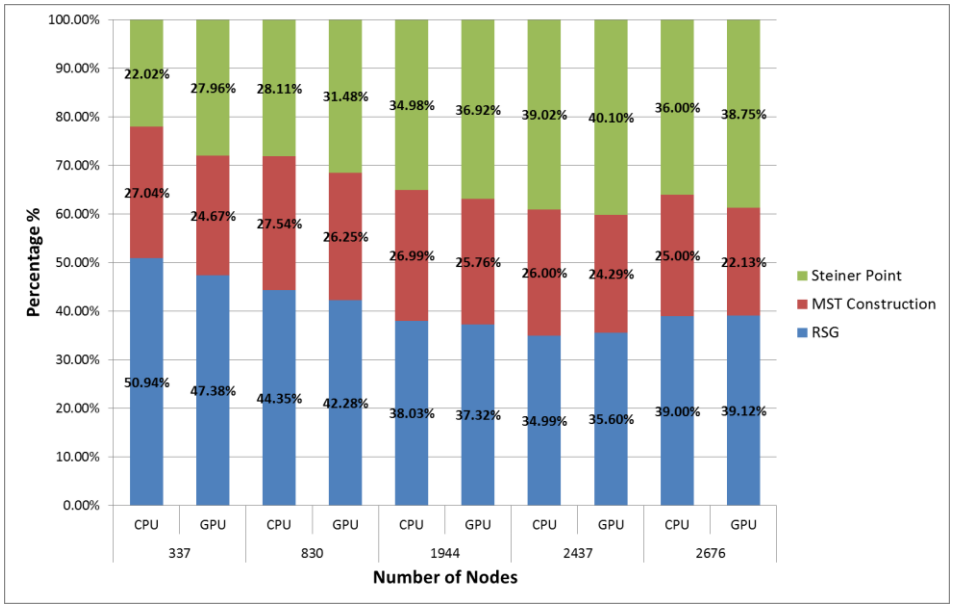

Figure 5-13: The Runtime Ratio for Small Data Sets

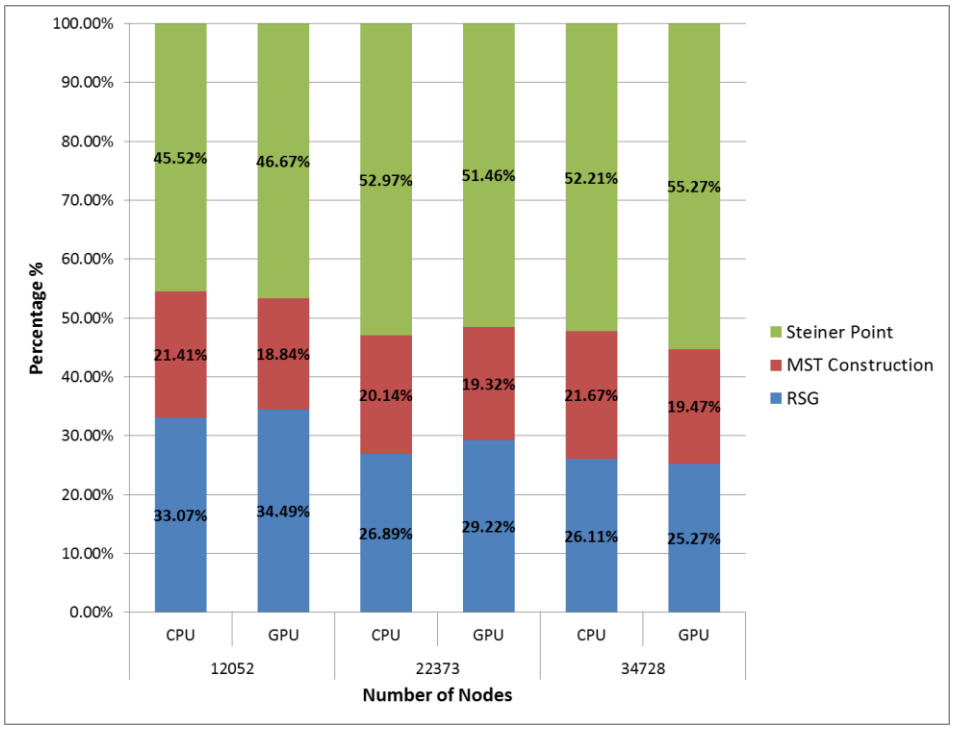

Figure 5-14: The Runtime Ratio for Large Data Sets

The speed ups measured for each sub block is shown in Figure 5-15. It is observed that speed up is achieved for each block apart from the first two smallest size benchmarks.

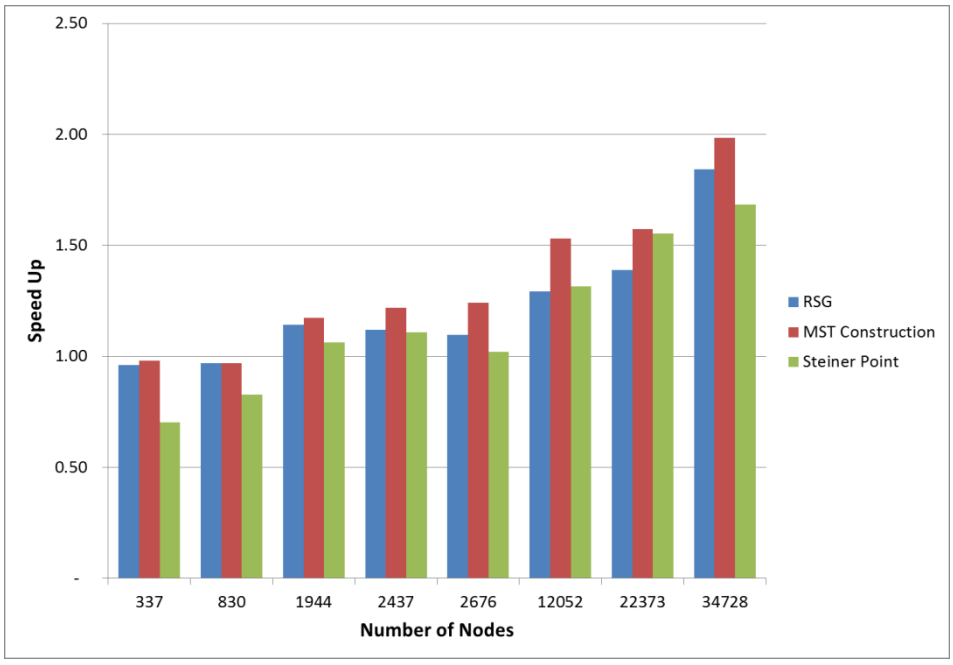

Figure 5-15: The Speed-Up of the RST Algorithm Blocks

Even for small data sizes such as 1944 and 2437 node cases speed up is possible for each algorithm block, proving that our algorithm is taking advantage of parallel execution opportunities successfully.

Highest speed up, 2x, is observed for MST construction for VLSI test instances. Although speed up depends on the size of the instance, the characteristics of the instance may also affect the performance significantly. For instance, for 2767 and 2437 node cases, the achieved speed up of larger node size case is slightly lower than the smaller node size case. Although most of the algorithm steps of the RST solution are implemented parallelly, it still contains some inevitable sequential operations. For instance, longest edge detection between two nodes requires sequential parent edge array search process or the nearest neighbor search operation requires serial active set operations. The increase in the runtime of these sequential operations degrades the GPU performance. Therefore, in some benchmark instances, due to the distribution of the node in the plane or the structure of the parent edge arrays, the overall speed of the parallel solution gets worse.

5.2.2.4 Algorithm Performance

The quality of our algorithm is evaluated as the length improvement in Rectilinear Steiner tree in comparison to the corresponding MST. The length improvement for VLSI test cases is shown in Figure 5-16.

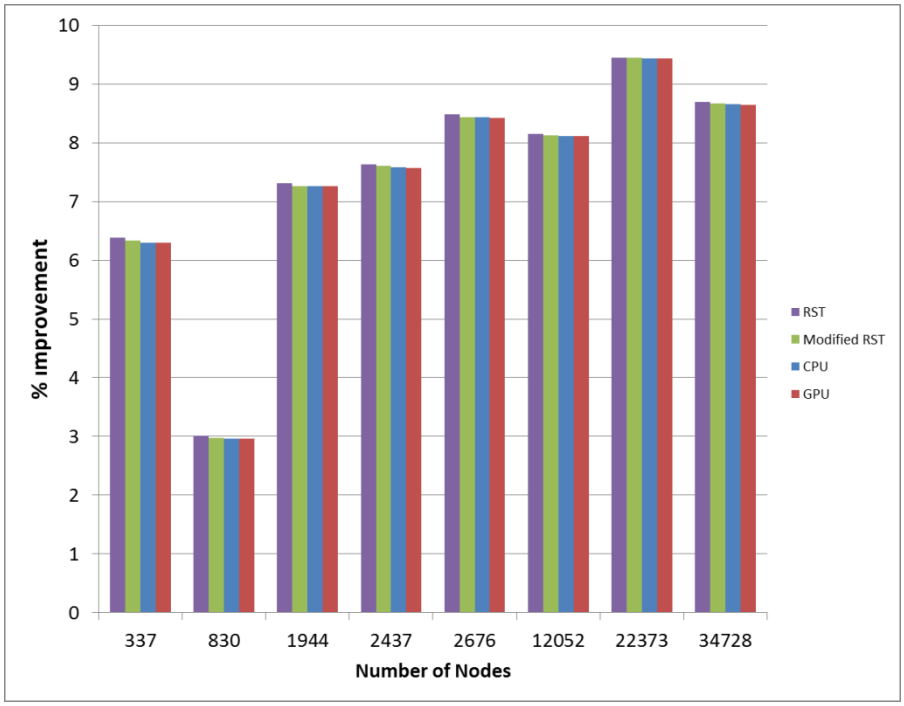

Figure 5-16: Steiner Tree Length Improvement for VLSI Test Instances

The same length improvements with RST and modified RST algorithms have been accomplished for our sequential implementation. The length improvement of our GPU implementation is slightly lower than our sequential implementation due to parallel programming optimizations. Nonetheless, the difference in the improvement is insignificant. As a conclusion, we demonstrated that with the implementation of our algorithm significant speed up can be also accomplished without deteriorating the algorithm performance on VLSI instances.

CHAPTER 6

CONCLUSION

The Steiner tree problem is a fundamental problem in network and VLSI design areas. It aims to generate the minimum total length tree that spans all the input terminals plus some possible additional nodes called Steiner points. If the distances are calculated in rectilinear plane, the problem is called Rectilinear Steiner Tree problem. For this case, the tree edges are restricted to horizontal and vertical lines, for which VLSI circuit routing is an important example of this type of the problem.

Although it is important to obtain an optimum solution for RSMT problem, it is NP-complete and there is no polynomial time exact solution for it. Known exact algorithms have long runtimes and they can only work on small set of terminals. As a result, finding an optimum solution for modern day industry size problems is impossible. There is more interest in heuristic solutions of the problem. In this study, we propose a heuristic solution approach that can be used in modern computer aided design (CAD) tools.

Among the modern RSMT heuristic algorithms, two of them are distinctive from others, namely BGA by Kahng [3] and RST by Zhou [2]. These two algorithms produce high quality results in reasonable time and achieve minimal runtime complexity. Meanwhile, they can be applied on instances with tens of thousands of terminals, which make them suitable for modern day applications. Modified RST algorithm [1] was proposed earlier by combining these two algorithms to generate a more efficient method. Modified RST algorithm is based on RST; however, RST's recursive steps are replaced with BGA's iterative steps.

We have chosen Modified RST as the basis of our parallel algorithm because it has better runtime in comparison to other algorithms without sacrificing the quality of the solution. Modified RST can work on instances with large number of terminals;

hence, we can achieve significant speed-up for such problem sizes with a proper parallel implementation due to GPUs massively data parallel structure. Moreover, most sub-steps of the modified RST are quite suitable for parallelization. Some modifications on the algorithm have been made in order to obtain better performance results. For instance, MST computation method is replaced with Boruvka's algorithm because of its parallel nature; thus, we accomplished 11.2x speed up for the MST computations.

The main contribution of this study is to propose a new parallel solution that can provide a RSMT in a time efficient way. Our algorithm achieves significant speed up in comparison to sequential methods in the literature without deteriorating the quality of the solution. With the advances in the chip industry, the number of nets in a design will grow and it can easily reach to tens of millions. Hence, there is a great demand in time efficient RSMT solutions. With the use of our parallel algorithm, problems with having a large number of input terminals can be solved in significantly shorter times in comparison to present methods. Furthermore, we provided an approach, which can achieve speed up for almost all data sizes; therefore, any RSMT problem instance can be solved faster with our algorithm.

Nowadays, every computer platform contains graphics hardware. The CAD tools or real world applications can take advantage of the speed up provided by our algorithm without a need to use an extra hardware resource. Hence, our implementation provides a cost efficient solution for the RSMT problem.

Our GPU implementation can be further improved in terms of total runtime. In RST flow, we replaced MST construction step and achieved important speed ups. However, in overall runtime of the algorithm, Steiner point operations block has more influence on the overall runtime than the other blocks. Hence, speed up obtained for RSMT is less than the speed ups obtained for RSG and MST constructions. As future work, Borah's edge substitution method can be replaced with another suitable approach and overall performance of the parallel algorithm can further be improved significantly.

Modern VLSI designs may contain obstacles in the layout such as IP blocks or pre routed nets. Adopting these obstacles to RSMT problem is also an important challenge. Routing large number of nets among thousands of obstacles may become

the new constraint of the CAD tools. Therefore, as future work, our parallel solution can be adapted to solve RSMT in the presence of obstacles.

REFERENCES

[1] S. Cinel and C. F. Bazlamaçcı, "A Distributed Heuristic Algorithm for the Rectilinear Steiner Minimal Tree Problem," *IEEE Trans. Comput. Des. Integr. CIRCUITS Syst.*, vol. 27, no. 11, pp. 2083–2087, 2008.

[2] H. Zhou, "Efficient Steiner Tree Construction Based on Spanning Graphs," *IEEE Trans. Comput. Des. Integr. CIRCUITS Syst.*, vol. 23, no. 5, pp. 704–710, 2004.

[3] A. B. Kahng, I. I. Mandoiu, and A. Z. Zelikovsky, "Highly scalable algorithms for rectilinear and octilinear Steiner trees," *Proc. ASP-DAC Asia South Pacific Des. Autom. Conf. 2003.*, pp. 827–833, 2003.

[4] J. B. J. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proc. Am. Math. Soc.*, vol. 7, no. 1, pp. 48–50, 1956.

[5] O. Boruvka, "O jistém problému minimálním (About a certain minimal problem)," *Práce mor. přírodověd. spol. v Brně III (in Czech Ger. Summ. 3*, pp. 37–58, 1926.

[6] M. Borah, R. M. Owens, and M. J. Irwin, "An Edge-Based Heuristic for Steiner Routing," *IEEE Trans. Comput. Des. Integr. CIRCUITS Syst.*, vol. 13, no. 12, pp. 1563–1568, 1994.

[7] G. Robins and A. Zelikovsky, "Minimum steiner tree construction," *Handb. Algorithms VLSI Phys. Autom.*, pp. 487–508, 2008.

[8] "Steiner tree problem," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Steiner_tree_problem. [Accessed: 29-Aug-2015].

[9] X. Wang, "Exact algorithms for the Steiner tree problem", Ph. D., University of Twente, 2008.

[10] S. Gueron and R. Tessler, "The Fermat-Steiner Problem," *Math. Assoc. Am.*, vol. 109, no. 5, pp. 443–451, 2002.

[11] M. Hanan, "On Steiner ' s Problem with Rectilinear Distance," *SIAM J. Appl. Math.*, vol. 14, no. 2, pp. 255–265, 1966.

[12] F. K. Hwang, “On Steiner Minimal Trees with Rectilinear Distance,” *SIAM J. Appl. Math.*, vol. 30, no. 1, pp. 104–114, 1976.

[13] M. R. Garey and D. Johnson, “The Rectilinear Steiner Tree Problem is NP-Complete,” *SIAM J. Appl. Math.*, vol. 32, no. 4, pp. 826–834, 1977.

[14] Z. a. Melzak, “On the problem of Steiner,” *Bull. Can. mathématiques*, vol. 4, no. 0, pp. 143–148, 1961.

[15] J. Van Laarhoven, “Exact and heuristic algorithms for the Euclidean Steiner tree problem,” Ph. D., University of Iowa, 2010.

[16] D. M. Warme, P. Winter, and M. Zachariasen, “Exact algorithms for plane Steiner tree problems: A computational study,” *Tech. Rep. DIKU-TR-98/11*, pp. 81–116, 1998.

[17] J. S. Salowe and D. M. Warme, “An exact rectilinear Steiner tree algorithm,” *Proc. 1993 IEEE Int. Conf. Comput. Des. ICCD’93*, pp. 1–4, 1993.

[18] J. Lee, N. Bose, and F. Hwang, “Use of Steiner’s problem in suboptimal routing in rectilinear metric,” *IEEE Trans. Circuits Syst.*, vol. 23, no. 7, pp. 470–476, 1976.

[19] F. Hwang, “An O(n log n) algorithm for suboptimal rectilinear Steiner trees,” *IEEE Trans. Circuits Syst.*, vol. 26, no. 1, pp. 1978–1980, 1979.

[20] J. E. Beasley, “A heuristic for Euclidean and rectilinear Steiner problems,” *Eur. J. Oper. Res.*, vol. 58, no. 2, pp. 284–292, 1992.

[21] R. C. Prim, “Shortest Connection Networks And Some Generalizations,” *Bell System Technical Journal*, vol. 36, no. 6. pp. 1389–1401, 1957.

[22] A. B. Kahng and G. Robins, “A new class of iterative Steiner tree heuristics with good performance,” *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 11, no. 7, pp. 893–902, 1992.

[23] A. B. Kahng and G. Robins, *On Optimal Interconnections for VLSI*. 1995: Kluwer Academic Publishers

[24] U. Fößmeier, M. Kaufmann, and a. Zelikovsky, “Faster Approximation Algorithms for the Rectilinear Steiner Tree Problem,” *Discrete Comput. Geom.*, vol. 18, no. 1, pp. 93–109, 1997.

[25] L. J. Guibas and J. Stolfi, “On Computing All North-East Nearest Neighbors in the L1 Metric,” *Inf. Process. Lett.*, vol. 17, no. 4, pp. 219–223, 1983.

[26] H. Zhou, N. Shenoy, and W. Nicholls, “Efficient minimum spanning tree construction without Delaunay triangulation,” *Inf. Process. Lett.*, vol. 81, no. 5, pp. 271–276, 2002.

[27] D. Harel and R. E. Tarjan, “Fast algorithms for finding nearest common ancestors,” *SIAM J. Comput.*, vol. 13, no. 2, pp. 338–355, 1984.

[28] T. Barrera, J. Griffith, S. a. McKee, G. Robins, and T. Zhang, “Toward a Steiner engine: enhanced serial and parallel implementations of the iterated 1-Steiner MRST algorithm,” *Des. Autom. High Perform. VLSI Syst. Proc.*, pp. 90–94, 1993.

[29] C. Fobel and G. Gréwal, “A parallel steiner tree heuristic for macro cell routing,” *26th IEEE Int. Conf. Comput. Des. 2008, ICCD*, pp. 27–33, 2008.

[30] W. K. Chow, L. Li, E. F. Y. Young, and C. W. Sham, “Obstacle-avoiding rectilinear Steiner tree construction in sequential and parallel approach,” *Integr. VLSI J.*, vol. 47, no. 1, pp. 105–114, 2014.

[31] J. L. Ganley and J. P. Cohoon, “Routing a multi-terminal critical net: Steiner tree construction in the presence of obstacles,” *Proc. IEEE Int. Symp. Circuits Syst. - ISCAS ’94*, vol. 1, pp. 113–116, 1994.

[32] “Graphics Processing Unit.” [Online]. Available: https://en.wikipedia.org/wiki/Graphics_processing_unit. [Accessed: 31-Aug-2015].

[33] “Minimum spanning tree.” [Online]. Available: https://en.wikipedia.org/wiki/Minimum_spanning_tree. [Accessed: 11-Aug-2015].

[34] G. Robins and J. S. Salowe, “Low-Degree Minimum Spanning Trees*,” *Discrete Comput. Geom.*, vol. 14, no. 1, pp. 151–165, 1995.

[35] Z. Feng, Y. Hu, T. Jing, X. Hong, X. Hu, and G. Yan, “An O (n log n) algorithm for obstacle-avoiding routing tree construction in the λ-geometry plane,” *Proc. 2006 Int. Symp. Phys. Des. - ISPD ’06*, p. 48, 2006.

[36] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, “Scan primitives for GPU computing,” *Graph. Hardw.*, pp. 97–106, 2007.

[37] A. S. Arefin, C. Riveros, R. Berretta, and P. Moscato, “k NN-Boruvka-GPU : A Fast and Scalable MST Construction from k NN Graphs on GPU,” *ICCSA*, pp. 71–86, 2012.

[38] V. Vineet, P. Harish and S. Patidar, “Efficient Obstacle-Avoiding Rectilinear Steiner Tree Construction”, Proceedings of the Conference on High Performance Graphics, pp. 167-171, 2009.

[39] D. a. Bader and G. Cong, “Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs,” *J. Parallel Distrib. Comput.*, vol. 66, no. 11, pp. 1366–1378, 2006.

[40] P. Harish, V. Vineet, and P. J. Narayanan, “Large Graph Algorithms for Massively Multithreaded Architectures,” *IIIT/TR*, no. 74, pp. 1–20, 2009.

[41] A. B. Kahng and I. Mandoiu, “FastSteiner : Highly Scalable Rectilinear and Octilinear Minimum Steiner Tree Algorithms,” 2015. [Online]. Available: http://vlsicad.ucsd.edu/GSRC/bookshelf/Slots/RSMT/FastSteiner/. [Accessed: 27-Aug-2015].

[42] C.-W. Lin, S.-Y. Chen, C.-F. Li, Y.-W. Chang, and C.-L. Yang, ``Efficient obstacle-avoiding rectilinear Steiner tree construction,'' in Proceedings of ACM International Symposium on Physical Design (ISPD-2007), pp. 127--134, Austin, TX, March 2007.

[43] L. Li and E. F. Y. Young, “Obstacle-avoiding rectilinear steiner tree construction,” *IEEE/ACM Int. Conf. Comput. Des. Dig. Tech. Pap. ICCAD*, pp. 523–528, 2008.

[44] D. M. Warme, P. Winter, and M. Zachariasen, “GeoSteiner 3.1 Package. [Online].” [Online]. Available: ftp.diku.dk/diku/users/martinz/ geosteiner-3.1.tar.gz. [Accessed: 16-Oct-2015].

[45] Ajwani, G.; Chu, C.; Wai-Kei Mak, "FOARS: FLUTE Based Obstacle-Avoiding Rectilinear Steiner Tree Construction," in Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on , vol.30, no.2, pp.194-204, Feb. 2011

[46] “GeoSteiner” [Online]. Available at: http://www.geosteiner.net/. [Accessed: 21-Oct-2015].

[47] Wikipedia, "Moore's law", 2015. [Online]. Available at: https://en.wikipedia.org/wiki/Moore%27s_law. [Accessed: 20- Dec- 2015].

[48] “The death of CPU scaling: From one core to many,” ExtremeTech. [Online]. Available at: http://www.extremetech.com/computing/116561-the-death-of-cpu-scaling-from-one-core-to-many-and-why-were-still-stuck/2. [20- Dec- 2015].

[49] “Accelerated Computing Toolkit,” NVIDIA Developer, 2015. [Online]. Available at: https://developer.nvidia.com/accelerated-computing-toolkit. [Accessed: 16-Oct-2015].