Parallel Metaheuristic Approaches to Solve Combinatorial Optimization Problems

Thesis

Submitted in partial fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

Pramod Hanmantrao Yelmewad

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA
SURATHKAL, MANGALORE - 575 025

June, 2021

DECLARATION

by the Ph.D. Research Scholar

I hereby declare that the Research Thesis entitled **Parallel Metaheuristic Approaches to Solve Combinatorial Optimization Problems** which is being submitted to the **National Institute of Technology Karnataka, Surathkal** in partial fulfilment of the requirements for the award of the Degree of **Doctor of Philosophy** in Department of Computer Science and Engineering is a bonafide report of the research work carried out by me. The material contained in this Research Thesis has not been submitted to any University or Institution for the award of any degree.

Pramod Hanmantrao Yelmewad, 165046CS16F04
Department of Computer Science and Engineering

Place: NITK, Surathkal.
Date: June 2, 2021

CERTIFICATE

This is to certify that the Research Thesis entitled **Parallel Metaheuristic Approaches to Solve Combinatorial Optimization Problems** submitted by **Pramod Hanmantrao Yelmewad** (Register Number: 165046CS16F04) as the record of the research work carried out by him, is accepted as the Research Thesis submission in partial fulfillment of the requirements for the award of degree of **Doctor of Philosophy**.

7/June/2021

Dr. Basavaraj Talawar
Research Guide

(Signature with Date and Seal)

09/06/21.

Chairman - DRPC
(Signature with Date and Seal)

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deep and sincere gratitude to my research supervisor, Dr. Basavaraj Talawar, Assistant Professor in the Department of Computer Science and Engineering, for his valuable advice, encouragement, and suggestions throughout my Ph.D. journey. I want to mention a special thanks for allowing me to manage the college cluster and department servers that helped me learn many new things.

Besides my supervisor, I extend my sincere thanks to the research progress committee members, Dr. Manu Basavaraju, Associate Professor in the Department of Computer Science and Engineering, and Dr. Jidesh P, Assistant Professor in the Department of Mathematical and Computer Sciences, for their valuable comments and encouragement. I would also like to thanks other teaching and non-teaching staff of the department for their generous support.

Thank our SPARK lab members- Bheemappa, Anil, Khyamling, Prabhu, and Kallinatha for their suggestions and support during this journey. A special thanks go to the married researchers, Mr. & Mrs. Sachin Patil, Dr. Vishnu Yeralagadda, and Vishal Rathod for their delicious homemade food. I extend special thanks to my supporters, motivators, correctors, playing partners, and trekkers- Nikhil Mhala, Vishal Rathod, Pravin Ramteke, Bheemappa Halawar, Pradyoth Hegde, and Raghavan Santhanam.

Last but not least, I am incredibly grateful to my family members- father, mother, sisters, brothers, sisters-in-law, brothers-in-law, nieces, and nephews for their unconditional love, constant support, motivation, and encouragement in my life.

Pramod Hanmantrao Yelmewad

ABSTRACT

The time complexity of many optimization problems falls under either exponential time or factorial time. Finding an optimal solution for such optimization problems is of great significance for scientific and engineering applications. Metaheuristic approaches have vast influence while solving optimization problems that provide satisfactory solutions in a reasonable amount of time. A metaheuristic is a guiding strategy to an underlying heuristic approach to solve a specific optimization problem. The metaheuristic approach helps determine an acceptable solution by applying constraints on the exploration space of feasible solutions. However, metaheuristics consume large amounts of CPU time while solving larger instances.

Parallel computation reduces overall execution time by executing independent tasks simultaneously. Parallel computing enables faster convergence to solutions of large instances of optimization problems. Challenges facing designers implementing parallel strategies are - cost quality reduction in large problem instances, arriving at near-optimal solutions for a subset of input instances (but not all), large search-space exploration to reach a satisfactory solution. In this thesis work, efficient parallel metaheuristic models are developed that resolve some of the issues mentioned above. Further, satisfactory solutions are arrived at in a reasonable amount of time. The proposed parallel versions of metaheuristics have been applied to three combinatorial optimization problems: traveling salesman problem, minimum latency problem, and vehicle routing problem.

The Traveling Salesman Problem (TSP) is an NP-hard combinatorial optimization problem. Metaheuristic methods are used to produce the satisfactory solution by limiting the search-space exploration. Moreover, CPU implementations of metaheuristic methods are too time-consuming for large input instances. The GPU-based Parallel Iterative Hill Climbing (PIHC) approach is presented for solving large TSPLIB instances in a reasonable time. Multiple GPU-based thread mapping strategies have been presented to solve large-scale TSPLIB instances. The improved cost quality has been demonstrated using the symmetric TSPLIB instances having up to 85,900 cities. The PIHC

GPU implementation gives up to 193× speedup over its sequential counterpart and up to 979.96× speedup over a state-of-the-art GPU-based TSP solver. The PIHC gives a cost quality with error rate 0.72% in the best case and 8.06% in the worst case. Moreover, two GPU-based parallel strategies have been developed for ant colony algorithm to solve larger instances than existing approaches.

The Minimum Latency Problem (MLP) is an NP-Hard combinatorial optimization problem. Metaheuristics use perturbation and randomization to arrive at a satisfactory solution under time constraints. The proposed work uses Deterministic Local Search Heuristic (DLSH) to identify a satisfactory solution without setting up metaheuristic parameters. A move evaluation procedure is proposed for the swap approach which computes a move in a constant time. A GPU-based Parallel Deterministic Local Search Heuristic (PDLSH) is proposed to mitigate the execution time spent in the solution improvement phase. The PDLSH parallelizes the solution improvement phase and solves MLP for larger instances than the state-of-the-art. The DLSH and PDLSH implementations are tested on the TRP and TSPLIB standard instances. DLSH reaches new best solutions for five TSPLIB instances, namely $eil51$, $berlin52$, $pr107$, $rat195$, and $pr226$. The proposed PDLSH achieves a speedup of up to 179.75 for the instances of size 10-11849 nodes compared to its sequential counterpart.

The Vehicle Routing Problem (VRP) is an NP-hard, goods transportation scheduling problem with vehicle capacity and transportation cost constraints. This work presents GPU-based parallel strategies for the Local Search Heuristic (LSH) algorithm to solve the large-scale Capacitated Vehicle Routing Problem (CVRP) instances. This work employs a combination of five improvement heuristic approaches to improve the constructed feasible solution. It is noticed that a large amount of CPU time is spent in the feasible solution improvement phase. Two GPU-based parallel strategies, namely, route level and customer level parallel designs, have been developed to reduce the execution time of solution improvement phase. The proposed parallel version of the LSH has been tested on large-scale instances of up to 30000 customers. The customer level parallel design offers speedup up to 147.19× compared to the corresponding sequential version.

From this thesis work, the proposed parallel version of IHC solves larger TSPLIB instances up to 85900 cities with a speedup of up to 193 times. Also, it reduces error rates of local solutions, i.e., in the range of 0.72% - 8.06%. The limitation of existing GPU-based MLP solver has been overcome in the proposed GPU-based parallel strategy to solve instances above 1000 nodes. PDLSH significantly mitigates the overall execution time while solving MLP, which achieves a speedup of 179× over its sequential counterpart for instances up to 11849 nodes. In the case of CVRP, two parallel strategies, namely route-level and customer-level, are developed to reduce execution time spent in the improvement phase. The customer-level parallel design reduces the execution time significantly and generates the speedup up to 147× for the instances having up to 30000 nodes.

Keywords: Traveling Salesman Problem (TSP), Parallel Iterative Hill Climbing (PIHC) Algorithm, Minimum Latency Problem (MLP), Parallel Deterministic Local Search Heuristic (PDLSH), Capacitated Vehicle Routing Problem (CVRP).

CONTENTS

List of Figures **xii**

List of Tables **xvii**

List of Abbreviations **xix**

1 Introduction **1**

1.1 Combinatorial Optimization Problem 1

1.1.1 Traveling Salesman problem (TSP) 2

1.1.2 Minimum Latency Problem (MLP) 3

1.1.3 Vehicle Routing Problem (VRP) 3

1.2 Metaheuristic 4

1.2.1 Need of Metaheuristic 5

1.3 Parallel Computation 5

1.3.1 GPU Computing 6

1.4 Motivation 7

1.5 Problem Description 7

1.6 Objectives 8

1.7 Thesis Contributions 8

1.8 Thesis Organization 9

2 Literature Review **11**

2.1 Traveling Salesman Problem (TSP) 11

2.1.1 Single-core 11

2.1.2 Multi-core 12

2.1.3 GPU Computing 12

2.1.4 Limitations 17

2.1.5 Contribution 17

2.2 Minimum Latency Problem 18
2.2.1 Limitations 20
2.2.2 Contribution 21
2.3 Vehicle Routing Problem 21
2.3.1 Sequential Approach 21
2.3.2 Parallel Approach 23
2.3.3 Limitations 24
2.3.4 Contribution 24
2.4 Research Gaps 24
2.5 SUMMARY 25
3 Parallel Iterative Hill Climbing Algorithm to Solve TSP on GPU 27
3.1 Introduction 27
3.1.1 Exact and Heuristic Approaches 28
3.2 Parallel Iterative Hill Climbing Algorithm 29
3.2.1 Initial Solution Construction 30
3.2.1.1 Sequenced 30
3.2.1.2 Random 31
3.2.1.3 Nearest Neighborhood (NN) 31
3.2.1.4 Nearest Insertion (NI) 32
3.2.1.5 Greedy Algorithm 32
3.2.1.6 Minimum Spanning Tree (MST) 33
3.2.1.7 Christofides' Algorithm 34
3.2.1.8 Clarke-Wright Algorithm 34
3.2.2 2-opt Neighborhood Generation 35
3.2.2.1 Neighborhood Generation at CPU and its Cost Computation at GPU 36
3.2.2.2 Neighborhood Generation and its Cost Computation at GPU 36
3.2.3 Optimized Data for TSP 36
3.2.4 Distance Calculation Method 38
3.2.5 Thread Mapping Strategies 38

3.2.5.1 Threads per Row (TPR) 39
3.2.5.2 Threads per Row Equal Distribution (TPRED) 40
3.2.5.3 Threads per Row and Column (TPRC) 41
3.2.5.4 Threads per Neighborhood (TPN) 42
3.2.5.5 Usage of built-in functions 44
3.3 Result Analysis 45
3.3.1 Initial Solution Construction Result Analysis 46
3.3.1.1 Performance Analysis of Construction Heuristic Approaches 46
3.3.1.2 Time Analysis of NN with Sequenced and Random Approaches 52
3.3.1.3 Cost Analysis of NN with Sequenced and Random Approaches 56
3.3.2 Optimized Data for TSP 56
3.3.3 Performance Analysis of Thread Mapping Strategies 58
3.3.4 Performance Analysis with GPU based state-of-the-art TSP Solvers 60
3.3.4.1 Cost Analysis 61
3.3.4.2 Time Analysis 62
3.3.5 Performance Analysis with CPU based state-of-the-art TSP Solvers 65
3.3.6 Detailed Performance Analysis of the PIHC Approach 66
3.4 SUMMARY 68
4 Min-Max Ant System on GPU for Large TSP Instances **69**
4.1 Introduction 69
4.2 Background 70
4.2.1 Sequential Implementation 70
4.2.2 Parallel Implementation 70
4.3 Ant Colony Optimization 71
4.3.1 Tour Construction 72
4.3.2 Pheromone Update 72
4.4 Methodology 73
4.4.1 Task-level Parallel Approach 75

4.4.2 Data-level Parallel approach 75
4.4.3 2-opt 78
4.5 Result Analysis 78
4.5.1 Data versus Task-based Parallel Approach 79
4.5.2 Sequential versus Parallel 80
4.5.3 Solution Quality 81
4.5.4 Comparison with Previous Works 82
4.6 SUMMARY 82

5 Parallel Deterministic Local Search Heuristic for Minimum Latency Problem 85

5.1 Introduction 86
5.1.1 Exact Methods 87
5.1.2 Approximation and Metaheuristic Methods 87
5.1.3 MLP on the Graphics Processing Unit 88
5.2 Deterministic Local Search Heuristic (DLSH) 89
5.2.1 Swap Move 90
5.2.1.1 Move Evaluation 91
5.2.2 Two-opt Move 93
5.2.2.1 Move Evaluation 94
5.3 Parallel Deterministic Local Search Heuristic 95
5.3.1 Reduction using Built-in Function 97
5.3.2 Reduction using Vector 100
5.3.2.1 One-pass Reduction 102
5.3.2.2 Two-pass Reduction 102
5.4 Results Analysis 104
5.4.1 Deterministic Local Search Heuristic (DLSH) 105
5.4.1.1 Comparison with State-of-the-art MLP Solvers 109
5.4.2 Parallel Deterministic Local Search Heuristic (PDLSH) 112
5.4.2.1 DLSH vs. PDLSH 113
5.4.2.2 Reduction Methods 115
5.4.2.3 Vector Reduction Types 117

5.4.3 Solution Quality Precision 118

5.4.4 Discussions 123

5.5 SUMMARY 124

6 Parallel Version of Local Search Heuristic Algorithm to Solve Capacitated Vehicle Routing Problem 127

6.1 Introduction 127

6.1.1 Capacitated Vehicle Routing Problem 127

6.1.2 Heuristic Algorithm 129

6.1.3 Motivation 130

6.2 Local Search Heuristics 131

6.2.1 Solution Construction 133

6.2.2 Solution Improvement 134

6.2.2.1 2-opt Heuristic 135

6.2.2.2 or-opt Heuristic 136

6.2.2.3 3-opt Heuristic 136

6.2.2.4 Swap Heuristic 137

6.2.2.5 Relocate Heuristic 137

6.3 Parallel Local Search 137

6.3.1 Route Level Parallel Design 137

6.3.2 Customer Level Parallel Design 139

6.3.2.1 Intra-route Heuristics 140

6.3.2.2 Inter-route Heuristics 142

6.4 Results and Discussions 143

6.4.1 Result Analysis of Sequential Version over Different CVRP Benchmark Instance Sets 143

6.4.2 Route Level Parallel Design 148

6.4.3 Customer Level Parallel Design 151

6.4.4 Solution Quality 157

6.5 SUMMARY 158

7 Conclusions and Future Scope 161

Bibliography 165

Publications **179**

Bio-Data **181**

LIST OF FIGURES

3.1 The 2-opt neighborhood generation mechanism 36

3.2 Time analysis of the initial solution construction approaches. 50

3.3 Total PIHC execution time of sequenced, random, NN initial solution construction mechanism for TSPLIB instances 53

3.4 Throughput analysis of sequenced, random, NN initial solution construction mechanism for TSPLIB instances 54

3.5 Total execution time of the PIHC algorithm using Distance Matrix(DM), Triangular Distance Matrix(DM), X,Y coordinate(COORD) as data structures 57

3.6 Analysis of error rate in the final cost of the PIHC and TSP2.2 with 400,500 restarts. 61

3.7 Time analysis of the PIHC with different restarts of TSP2.2 TSP solver. 62

3.8 Execution Time analysis with the best CPU-based TSP Solvers 65

4.1 Parallel work-flow model for work distribution between CPU and GPU of ACO algorithm. 74

4.2 Data Parallel Approach 77

4.3 Speedup analysis of data-level and task-level approaches over sequential counterpart. 80

5.1 Neighborhood Generation Mechanism 91

5.2 An edge affected in swap move when $i = j - 1$ & $j = n - 1$ 92

5.3 Edges affected in swap move when $i = j - 1$ & $j < n - 1$ 92

5.4 Edges affected in swap move when $i \neq j - 1$ and $j < n$. 93

5.5 Subsequences of the solution after edge removal in two-opt move . . . 94

5.6 Execution time portion analysis in the initial solution construction and improvement. . . . 96
5.7 Pictorial representation of one-pass reduction. . . . 102
5.8 Pictorial representation of two-pass reduction. . . . 103

6.1 Phase-wise execution time analysis on Belgium Arnold and Sörensen (2019) data set. . . . 131
6.2 Solution improvement flowchart 135
6.3 Pictorial representation of 2-opt neighborhood generation technique. . . . 135
6.4 Pictorial representation of or-opt neighborhood generation technique. . . . 136
6.5 Pictorial representation of 3-opt neighborhood generation technique. . . . 136
6.6 Route level parallel design for improvement heuristics. . . . 138
6.7 Execution time analysis of LSH spent in the solution construction and improvement phases for Golden and Li data sets. . . . 145
6.8 Execution time analysis of LSH spent in the solution construction and improvement phases for Kytöjoki et al. (2007) and Belgium (Arnold and Sörensen 2019) data sets. . . . 148
6.9 Time comparison of route level parallel design with sequential version on Li et al. (2005) data set. . . . 149
6.10 Comparison of time spent in sequential and parallel versions of solution improvement phase using Golden and Li data sets. . . . 152
6.11 Comparison of time spent in sequential and parallel versions of solution improvement phase using Kytöjoki and Belgium data sets. . . . 154

LIST OF TABLES

2.1 Overview of multi-core based parallel TSP Solvers 12

2.2 Characteristics of GPU-based methods. Each row represents existing work, heuristic method used to solve TSP, neighborhood generation technique, instance size considered, error rate, speedup over sequential part, limitation of existing work. 15

2.3 Overview of existing metaheuristic approaches that solve the minimum latency problem. 19

2.4 Details of existing parallel implementations available for VRP variants. 24

3.1 GPU global memory requirement of the Distance Matrix (DM), Triangular Distance Matrix (TDM), X,Y coordinate (COORD) data structures for TSPLIB instances 37

3.2 Threads per Row (TPR) mapping strategy for a 5 cities graph. The columns represent cities from the graph. Thread 0 generates and evaluates neighbors using pairs (0,1), (0,2), (0,3), and (0,4). Thread 1 generates and evaluates neighbors using pairs (1,2), (1,3), and (1,4), and so on. 39

3.3 Threads per Row mapping with Equal work Distribution (TPRED) for a 5 cities graph. Here, n=5 and j=(i+1) mod n. The j columns indicate neighbors associated with each thread. Thread 0 evaluates pairs (0,1), and (0,2). Thread 1 evaluates pairs (1,2), and(1,3) and so on. 41

3.4 Threads per Row and Column (TPRC) mapping for 5 cities graph. Thread pairs, where row id $<$ column id, are used to generate and evaluate neighbors. 41

3.5 Threads per Neighbor (TPN) mapping for n cities where n $>$ 3. Each thread generates one neighbor. Each thread uses equations 3.6 and 3.7 to determine swapping cities. 43

3.6 Time analysis of constructing the initial solution using different construction approaches- Nearest Neighborhood (NN), Nearest Insertion (NI), greedy, Minimum Spanning Tree (MST), Christofides' algorithm, and Clarke-Wright algorithm. Time is given in seconds. 47

3.7 Analysis of the local optimal solution reaches with different construction heuristic approaches. 48

3.8 Analysis of execution time spent in the solution construction phase and improvement phase separately. Abbreviation used- C%: Percentage of time spent in the solution construction out of total execution time, I %: Percentage of time spent in the solution improvement out of total execution time. 51

3.9 Cost analysis of different initial solution construction techniques- sequenced, random, NN. Each technique has initial solution cost, locally reached final cost on the initial solution, total steps needed to reach the final solution and its error rate for TSPLIB instances. 55

3.10 Time analysis of thread mapping strategies- Threads per Row (TPR), Threads per Row equal Distribution (TPRED), Threads per Row and Column (TPRC), and Threads per Neighbor (TPN). Time is given in seconds which is average of ten trials. 59

3.11 Cost and time analysis of TSP2.2 solver for TSPLIB instances with different restart values. 63

3.12 Cost quality comparison of the PIHC algorithm with the state-of-the-art TSP solvers on various TSPLIB instances. Values in the parentheses are deviations from the optimal cost. 64

3.13 Detailed performance analysis of the proposed PIHC approach with time and cost analysis. Time analysis presents the time of host-device transfer, device-host transfer, actual GPU computation, Overall PIHC execution in seconds. Speedup on its corresponding sequential part. Cost analysis presents initial and final cost obtained using NN with error rate. 67

4.1 Time analysis of sequential, data-level, and task-level parallel approaches. Time is presented in milliseconds. The average execution time of the instances for 1000 iteration. 79
4.2 Comparison of solution quality with and without 2-opt. One iteration of execution time is given in seconds. 81

5.1 Thread per vertex pair mapping details for neighborhood evaluation methods. 97
5.2 Implementation details for DLSH and PDLSH with hardware specifications. The DLSH and PDLSH column shows the platform used for corresponding implementation. 104
5.3 Solution quality analysis for TRP instances of size 10, 20, and 50 using DLSH. Abbreviations used- I: Instance name, C: Calls; 106
5.4 Solution quality and execution time details for TRP instances of size 100, 150, and 200 using DLSH. Time is given in seconds. Abbreviations used- I: Instance name, C: Calls, LG: Local Gap; 107
5.5 Solution quality and execution time details for TRP instances of size 500 using DLSH. Time is given in seconds. 107
5.6 Solution quality and execution time details for TSPLIB instances using DLSH. Time is given in seconds. 108
5.7 DLSH solution quality comparison with Salehipour et al. (2011) for TSPLIB instances. 110
5.8 DLSH solution quality comparison with Abeledo et al. (2013) and Silva et al. (2012) for TSPLIB instances. 111

5.9 Solution quality comparison with Silva et al. (2012) for TRP instances of size 100, 200, and 500. 112
5.10 Speedup analysis of PDLSH over DLSH for TRP instances of size 100, 200 and 500. Time is given in seconds. 113
5.11 Speedup analysis of PDLSH over DLSH for TSPLIB instances. Time is given in seconds. 114
5.12 Execution times of built-in library API and vector reduction methods. Time is given in seconds. 116
5.13 Execution times of vector reduction types: one-pass, two-pass, and single thread. Time is given in seconds. 117
5.14 DLSH solution precision analysis using integer and floating-point calculation methods. TSPLIB instances are used for this evaluation. Difference is calculated using Eqn. 5.8. 120
5.15 DLSH solution precision analysis using integer and floating-point calculation methods on TRP instances of size 10, 20, and 50. Difference is calculated using Eqn. 5.8. 121
5.16 DLSH solution precision analysis using integer and floating-point calculation methods on TRP instances of size 100, 200, and 500. Difference is calculated using Eqn. 5.8. 122

6.1 Members of heuristic data structures used to hold each thread's best solution details. 139
6.2 Members of heuristic data structures used to hold each thread's best solution details. 141
6.3 The hardware and software details used for the sequential and parallel implementation of the local search heuristic. 144
6.4 The cost and time analysis of LSH on M set instances (Christofides et al., 1979). Time is given in seconds. Abbreviation used- N: Total customers, K: Minimum number of vehicles required to solve corresponding instance, SC: Solution Construction; BKS: Best Known Solution; SI: Solution Improvement. 144

6.5 The cost and time analysis over Golden et al. (1998) instance set. Time is given in seconds. 145
6.6 The cost and time analysis using twelve instances of Li et al. (2005). Time is given in seconds. 146
6.7 The cost and time analysis of LSH over Kytöjoki et al. (2007) instance set. Time is given in seconds. 146
6.8 The cost and time analysis of LSH over Belgium (Arnold and Sörensen 2019) instance set. Time is given in seconds. 147
6.9 The performance analysis of route level parallel design over Li et al. (2005) instances. Time is given in seconds. Abbreviation used- PSI: Parallel version of Solution Improvement. 149
6.10 The performance of route level parallel design over X (Uchoa et al. 2017) data set. Speedup is a ratio between sequential and parallel version of SI time. 150
6.11 The execution time analysis of customer level parallel design with sequential counterpart on M (Christofides et al. 1979) data set. 152
6.12 The execution time analysis of customer level parallel design with sequential counterpart on Golden et al. (1998) instance set. 153
6.13 The execution time analysis of customer level parallel design with sequential counterpart on Li et al. (2005) data set. 154
6.14 The execution time analysis of customer level parallel design with sequential counterpart on Kytöjoki et al. (2007) data set. 155
6.15 The execution time analysis of customer level parallel design with sequential counterpart on Belgium (Arnold and Sörensen 2019) data set. . 156
6.16 The performance of route level parallel design over Uchoa et al. (2017) data set. Abbreviations used- SI: Solution Improvement; RSI: Route level Solution Improvement; CSI: Customer level Solution Improvement; S1: Speedup ratio between SI and CSI; S2: Speedup ratio between RSI and CSI. 157

LIST OF ABBREVIATIONS

Abbreviations	Expansion
ACO	Ant Colony Optimization
AS	Ant System
BKS	Best Known Solution
CPU	Central Processing Unit
CSI	Customer-level Solution Improvement
CUDA	Compute Unified Device Architecture
CVRP	Capacitated Vehicle Routing Problem
CW	Clarke & Wright
DLSH	Deterministic Local Search Heuristic
DM	Distance Matrix
DtoH	Device to Host
GPU	Graphics Processing Unit
GRASP	Greedy Random Adaptive Search Procedure
HtoD	Host to Device
IHC	Iterative Hill Climbing
LKH	Lin Kernighan Heuristic
LSH	Local Search Heuristic
MLP	Minimum Latency Problem
MMAS	Min Max Ant System
MST	Minimum Spanning Tree
NI	Nearest Insertion
NN	Nearest Neighborhood
PDLSH	Parallel Deterministic Local Search Heuristic

Abbreviations	**Expansion**
PIHC	Parallel Iterative Hill Climbing
PSI	Parallel Solution Improvement
RSI	Route-level Solution Improvement
RVND	Random Variable Neighborhood Descent
SC	Solution Construction
SI	Solution Improvement
TDM	Triangular Distance Matrix
TPN	Threads Per Neighborhood
TPR	Threads Per Row
TPRC	Threads Per Row and Column
TPRED	Threads Per Row Equal Distribution
TRP	Traveling Repairman Problem
TSP	Traveling Salesman Problem
UHGS	Unified Hybrid Genetic Search
VND	Variable Neighborhood Descent
VNS	Variable Neighborhood Search
VRP	Vehicle Routing Problem

CHAPTER 1

INTRODUCTION

This thesis presents the GPU-based parallel models to the existing metaheuristic algorithms to solve different combinatorial optimization problems in lesser time. Many optimization problems deal with real-time applications. Providing faster solutions to such applications is of great importance. This thesis aims to reduce the execution time spent in the improvement phase of metaheuristic algorithms. This chapter briefly introduces different combinatorial optimization problems, metaheuristic algorithm and their limitations, and the need to apply parallelism. The problem statement of this thesis and its objectives are also presented.

1.1 COMBINATORIAL OPTIMIZATION PROBLEM

The combinatorial optimization problem is a study of operations research and theoretical computer science, where the objective is to find an optimal solution from the finite set of solutions. The combinatorial optimization problem plays a significant role in the science, engineering, and industrial domain. Some of the combinatorial optimization problems are the maximum clique problem (Bomze et al. 1999), minimum latency problem (Blum et al. 1994), graph coloring (Jensen and Toft 2011), quadratic assignment problem (Lawler 1963), traveling salesman problem (Lin 1965), and vehicle routing problem (Clarke and Wright 1964). There are several approaches available in the exact methods to solve such problems. Some of these are, namely, brute-force, branch and bound, branch and cut, and dynamic programming. The search-space involved in finding an optimal solution is too large. The exact method has to explore either exponential or factorial order search-space to determine an optimal solution. Finding an optimal solution for the large-scale problem is intractable, and hence in recent studies, metaheuristic algorithms are employed to find satisfactory solutions in a reasonable amount of time.

The thesis work presents faster, and improved cost quality solutions for larger input instances for three combinatorial optimization problems chosen based on their complexity. These three combinatorial optimization problems are Traveling Salesman Problem (TSP), Minimum Latency Problem (MLP), and Vehicle Routing Problem (VRP).

1.1.1 Traveling Salesman problem (TSP)

The Traveling Salesman Problem (TSP) (Gutin and Punnen 2002; Johnson and Mcgeoch 1997) is an NP-hard (Garey and Johnson 1990), $O(n!)$ combinatorial optimization problem. The time complexity of TSP is the factorial time when solved using the brute-force method and exponential time when solved using dynamic programming. For its large number of science and engineering applications, time-efficient TSP solutions are of great importance. Some practical applications are drilling of printed circuit boards, overhauling gas turbine engines, X-ray crystallography, and vehicle routing (Lenstra and Kan 1975; Matai et al. 2010).

The objective of TSP is to find the minimum cost route that passes through each city exactly once returning to the originating city. In short, in the map of n cities, the distance of a route π has to be minimized,

$$d(\pi) = \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} + d_{\pi(n),\pi(1)} \tag{1.1}$$

where $d_{i,j}$ is a distance between any two cities i and j, and π is the permutation from $(1, 2, ..., n)$ (Merz and Freisleben 2001).

According to graph theory, TSP is a simple weighted connected graph which has cities as nodes, paths between cities as edges and distance between cities as weights on edges and the goal of TSP is to find minimal Hamiltonian cycle from simple weighted connected graph $G = (V, A, d)$ where, V= set of cities to be visited, A= $\{(i, j)|(i, j) \in V \times V\}$ is set of paths between cities and $d : A \longrightarrow Z$ is a function which assigns distance $d_{i,j}$ to every path between city (i, j). A Hamiltonian cycle in a simple connected graph that visits each node exactly once and returns to the starting node (Angluin and Valiant 1979).

1.1.2 Minimum Latency Problem (MLP)

The Minimum Latency Problem is an NP-Hard (Blum et al. 1994; Sahni and Gonzalez 1976; Sitters 2002) combinatorial optimization problem. The objective of MLP is to find a Hamiltonian path that minimizes the overall waiting time of nodes. The formal definition of MLP is, consider a simple, directed, weighted graph that has n vertices, where $n - 1$ vertices act as service requesting nodes and one vertex act as service providing node. Each service requesting node v_i $(1 \leq i<n)$ has to wait until it is served. This waiting period is also known as latency. The latency is a sum of distance or time required to reach from the service providing node v_0 to the service requesting node v_i.

MLP has several applications in real life, such as data retrieval in the computer network, delivery services, disk head scheduling, and logistics services for emergency reliefs, etc. (Campbell et al. 2008; Ezzine et al. 2010; Méndez-Díaz et al. 2008).

1.1.3 Vehicle Routing Problem (VRP)

Vehicle Routing Problem (VRP) (Clarke and Wright 1964; Dantzig and Ramser 1959) is an NP-hard, combinatorial optimization problem, with applications in the field of goods and transportation. The objective of VRP is to schedule the number of vehicles for goods transportation such that its transporting cost is reduced. VRP has several variants based on its objective function, viz., Capacitated Vehicle Routing Problem (CVRP), Heterogeneous Fleet VRP (HFVRP), Multi-Depot VRP (MDVRP), Pickup and Delivery VRP (PDVRP) (Braekers et al. 2016; Eksioglu et al. 2009; Toth and Vigo 2001).

In this thesis, CVRP is considered. CVRP is defined as: Given a simple, connected, weighted, undirected graph $G(V, E)$, where, V is set of vertices, i.e., $V = \{v_0, v_1, .., v_n\}$ and E is set of edges i.e., $E = \{v_i, v_j\}$ where $(i < j < n)$, the objective is to find a set of routes $R = \{r_1, r_2, .., r_m\}$ for m vehicles such that, 1) each customer is visited exactly once, and 2) total traveling distance is minimum.

1.2 METAHEURISTIC

The metaheuristic is a high-level method used as a guiding strategy to an underlying heuristic approach to solve a specific optimization problem. It helps to find out an acceptable solution by putting limitations on the exploration of feasible solutions. Metaheuristic (Talbi, 2009) is the approximation method that explores a subset of feasible solutions putting a curb on the search-space exploration and derive satisfactory solutions in a reasonable amount of time.

Metaheuristic methods start with a feasible initial solution and iteratively improve on it to converge towards the global optimal solution until the termination criteria are met. The termination criteria may include the number of iterations, an optimal cost value, no improvement of solutions between iterations. Popular metaheuristic approaches are iterative local search, hill climbing, tabu search, simulated annealing, ant colony optimization, and genetic algorithm (Monmarché et al., 1999). The selection of a subset of feasible solutions depends on the initial solution.

These algorithms are further classified either into a single solution-based heuristic algorithm or the population of a solutions-based heuristic algorithm (Talbi, 2009). The algorithm is called a single solution-based heuristic algorithm that initiates a single initial solution and makes an iterative improvement on it until the termination condition occurs. Hill climbing algorithm, iterated local search, tabu search act as single solution-based improvement heuristic algorithm. A population of solutions-based heuristic determines multiple initial solutions, subsequently improves it simultaneously, and finally selects the best-improved solution out of it. Examples of these algorithms are ant colony, random-restart, and genetic algorithm.

Once the initial solution is determined, a subset of feasible solutions is generated repeatedly to improve the initiated solution. Some of the feasible solution generation methods are 2-opt, 3-opt, k-opt, relocate, and swap (Muyldermans et al., 2005).

1.2.1 Need of Metaheuristic

Optimization problems are solved using two broad types; viz., exact methods, and metaheuristic methods. In the worst-case, exact methods have to explore the entire search space to determine the optimal solution. Metaheuristic methods give near-optimal solutions by exploring limited search space in a reasonable amount of time (Talbi 2009). In the worst-case, for n city TSP exact method, optimal solution is determined after exploring $\frac{(n-1)!}{2}$ feasible solutions in case of brute-force approach, $n^2 \times 2^n$ feasible solutions in case of dynamic programming. While the exact method always assures an optimal solution, the CPU time is prohibitive for large n. Due to the factorial time complexity, exact methods cannot be time-efficient approaches to solve large-scale combinatorial optimization problems. Metaheuristic (Talbi 2009) is the approximation method that explores a subset of feasible solutions by constraining the search space and derive satisfactory solutions in a reasonable amount of time. Metaheuristic methods start with a random initial solution and iteratively improve on it to converge to a near-optimal solution.

1.3 PARALLEL COMPUTATION

The metaheuristic approach is the two-step approach. First, it determines the initial solution of an optimization problem. Second, it improves the initial solution iteratively by choosing the best-improved neighborhood solution until no further improvement is possible. Since the metaheuristic approach improves initiated solution repeatedly, one cannot accurately determine the number of iterations required to reach a locally optimal solution before its execution. Therefore, neighborhood generation continues until termination criteria are met. In metaheuristic approaches, it is observed that most of the execution time is being spent in the second phase, i.e., neighborhood generation. It is noticed from the experimental analysis that more than 90% execution time is spent in the neighborhood generation phase. This execution time spent in the neighborhood generation can be reduced, deploying neighborhood generation tasks over the parallel computing platforms. The neighborhood generation phase is suitable for parallel implementation since one solution's generation does not depend on other solution's generation. Growth in parallel computing has occurred in the following ways-

- **Shared Memory Architecture:** Advent of the multi-core processor allows us to distribute tasks among multiple cores with the help of OpenMP and pthreads programming paradigm.
- **Distributed Computing:** In addition to shared memory architecture, multiple CPU systems/nodes are connected to get more computational power. On such cluster, work distribution is being done using MPI programming paradigm on different devices.
- **GPU computing:** Recent trend in parallel computing is to use the Graphics Processing Unit (GPU) platform. GPU computing has emerged as a more powerful parallel computing platform that offers thousands of dedicated cores to handle several computational tasks simultaneously in SIMD fashion.

The GPU has been considered as the parallel implementation platform to design parallel strategies for metaheuristic algorithms to solve various combinatorial optimization problems.

1.3.1 GPU Computing

Graphics Processing Unit (GPU) computing has highly computational horsepower and remarkably high memory bandwidth compared to CPU-based architectures. In GPU, there are more transistors dedicated to data processing instead of data caching and flow control. It is highly suitable for the computationally complex task and highly parallel computations.

GPU exploits data, memory, and task-level parallelism in applications to improve their performance significantly (Che et al. 2008; Ryoo et al. 2008). NVIDIA's GPUs are programmed using the Compute Unified Device Architecture (CUDA) toolkit to deploy GPU-specific task over the GPU platform (NVIDIA). GPU computing is heterogeneous, involving the CPU and the GPU cooperating to execute serial and parallel portions of the applications. The CPU (host) deploys data required for parallel computation in the GPU's global memory through the PCIe bus. The GPU (device) performs computations on this data. The results are copied from the GPU's global memory back to the CPU's memory on completion. GPU performs computations on the data using many structured threads running on its several compute cores. High-performance GPU implementations handle host-device data transfer efficiently, have effective thread mapping, and have optimal synchronization amongst the execution threads. GPUs have been widely used to solve complex optimization problems in time-efficient manner.

1.4 MOTIVATION

- **Time bound**

  Due to its time complexity, the exact method becomes prohibitive while solving large-scale combinatorial optimization problems. This limitation can be eliminated using an approximation method known as a metaheuristic algorithm that gives a satisfactory solution in a reasonable amount of time.

- **Speedup**

  Metaheuristic algorithm improves the initial solution using the neighborhood generation methods. The generation of feasible neighborhood solutions and its cost computation can be implemented in parallel, which results in a significant speedup improvement over its sequential counterpart.

- **Cost quality**

  Most existing metaheuristic algorithms have set up the initial solution arbitrarily. If the initial solution is constructed using the construction heuristic approaches, the resultant solution's cost quality can be improved.

1.5 PROBLEM DESCRIPTION

The state-of-the-art metaheuristic algorithms that produce the best-known solutions for combinatorial optimization problems spend inadmissible CPU time to obtain the local solution for large-scale instances. In this thesis, the aim is to reduce the execution time and solve large-scale instances compared to existing works. The efficient work distribution parallel model helps to get a nonzero speedup over sequential counterpart. Existing GPU-based metaheuristics solve limited size instances. Large-scale instances can be solved using optimized GPU data, effective threads, and block parameter tuning.

1.6 OBJECTIVES

- Propose the single solution-based parallel model to solve the optimization problem.
- Propose the population of solutions-based parallel model to solve the optimization problem.
- Design an efficient parallel model that solves larger-size benchmarking instances.

1.7 THESIS CONTRIBUTIONS

In this research, several GPU-based parallel strategies have been designed to solve large-scale combinatorial optimization problems in a reasonable amount of time.

1. Four GPU-based parallel strategies for the single solution-based heuristic, namely, Iterative Hill Climbing Algorithm (IHC) to solve Traveling Salesman Problem are presented (Yelmewad and Talawar 2019). The proposed approach outperforms over the GPU-based state-of-the-art TSP solvers (Neil and Burtscher 2015; Rocki and Suda 2013) in terms of speedup and solution quality. The effect of setting up different types of the initial solution has been presented on the execution time and solution quality (Talawar and Yelmewad 2017; Yelmewad and Talawar 2018).

2. Two parallel strategies, namely, task-level and data-level, have been applied for the Ant Colony Optimization (ACO) to solve TSP. ACO is the population of solutions-based metaheuristic where the population of solutions is being constructed following ants' behavior (Yelmewad et al. 2019).

3. Deterministic Local Search Heuristic (DLSH) that assures the same solution for the same instances irrespective of multiple trials for solving the Minimum Latency Problem (MLP) is designed. Further, the move evaluation procedure that computes a move pair in O(1) order without using any preprocessed local data is presented. A GPU-based parallel model for DLSH, which solves larger instances

than existing parallel MLP solvers and achieves speedup up to 179.75 times is also presented (Yelmewad and Talawar 2020).

4. Finally, two GPU-based parallel strategies have been designed for the hybrid of five improvement heuristics to solve the Capacitated Vehicle Routing Problem (CVRP). In the existing study, it is observed that parallel strategies have been designed for only intra-route heuristics. Route level and customer level parallel designs for the inter-route heuristics in addition to the intra-route heuristics are designed (Yelmewad and Talawar 2020).

1.8 THESIS ORGANIZATION

The rest of thesis is organized as follows: Chapter 2 presents the survey of existing works done for solving several combinatorial optimization problems using metaheuristic algorithms. Chapter 3 presents the working of the Parallel Iterative Hill Climbing (PIHC) algorithm to solve TSP. Chapter 4 discusses the parallel version of the Min-Max Ant System (MMAS) to solve TSP. Chapter 5 presents the GPU-based Deterministic Local Search Heuristic (DLSH) to solve the Minimum Latency Problem in lesser time. Chapter 6 presents the parallel version of the local search algorithm to solve the Capacitated Vehicle Routing Problem (CVRP) on the GPU platform. Finally, the summary of all the proposed techniques and future research directions are given in Chapter 7.

CHAPTER 2

LITERATURE REVIEW

Nowadays, metaheuristic algorithms are widely used to solve several combinatorial optimization problems to provide satisfactory solutions in less time than exact methods. Recently published papers have been studied which solve either TSP, MLP, or CVRP using metaheuristic algorithms. Twofold objectives are targeted to study existing works. 1) What is the maximum size of the instances solved? 2) Is there any parallel strategy used to reduce the execution time? The brief observation of existing works is explained in the following subsections for these three optimization problems separately.

2.1 TRAVELING SALESMAN PROBLEM (TSP)

Several existing metaheuristic algorithms have been studied which solve TSP. The overview of this study is further classified based upon the hardware platform used for the experimental analysis.

2.1.1 Single-core

Dorigo and Gambardella (1997) present result analysis of different metaheuristic algorithms that include ant colony optimization, genetic algorithm, evolutionary algorithm, and simulated annealing approaches to solving TSP. Author evaluates the performance of these metaheuristic approaches using both symmetric and asymmetric TSPLIB instances. Based on experimental results, researchers conclude that the ant colony system produces the best quality solution. However, this approach spends inadmissible time to generate good results for solving large instances.

Baraglia et al. (2001) have applied a combination of genetic algorithm and local search heuristic as the metaheuristic to solve the TSP. Author used the k-opt mechanism to generate neighborhood solutions. This approach gives a better solution compared to the genetic algorithm alone. The execution time of the above approaches increases drastically as the input size increases.

2.1.2 Multi-core

Several parallel TSP implementations have been proposed to take advantage of multi-core processor architecture. Randall and Lewis (2002) present the task distribution strategy for ant colony optimization. Authors have analyzed their method's performance on TSPLIB instances ranging from 24-657 nodes and claims 3× speedup. However, the parallel code only allows for one ant per processor.

Work	Heuristic	Instance	Speedup	Limitations
Stützle (1998b)	Ant colony	198-1291	6.7	Consumes more CPU time.
Randall and Lewis (2002)	Ant colony	24-657	3	Allows one per processor.
Craus and Rudeanu (2004)	Ant colony	229	30	Cost quality is not considered.
Delisle et al. (2009)	Ant colony	763-13509	5.5	Consumes more CPU time.

Table 2.1: Overview of multi-core based parallel TSP Solvers

Stützle (1998b) presents parallelization strategies for ant colony optimization for multi-core processor architecture. In addition to this, the author has done a slight change in the algorithm to reduce the pheromone matrix size using the min-max approach. The efficiency of proposed algorithm has been shown using instances between 198 and 1291 nodes. Although work provides good cost quality, it cannot solve large instances due to heavy memory workload. Craus and Rudeanu (2004), Delisle et al. (2009) have presented a parallel strategy similar to the above work. Table 2.1 presents the overview of multi-core based parallel TSP Solvers. However, these present works consume more time while the instance size increases.

2.1.3 GPU Computing

Fujimoto and Tsutsui (2011) present a GPU-based parallel strategy of genetic algorithm to solve TSP. The 2-opt move and crossover techniques have been used to generate neighborhood solutions. Author allows 60 number of blocks to execute 60 initial solutions in parallel and find out the best-improved solution. Author has noted that 24.2× speedup in best-case on instances of 120 - 493 range. The main limitation of proposed approach is that it cannot solve instances which size is more than 1024. This is because work restricts the number of threads on each block is equal to instance size. Hence, up to 1024 threads are allowed on recent GPU architecture.

Zhao et al. (2011) present the hybrid of genetic algorithm and tabu search to solve TSP on GPU. Author implements a parallel immune algorithm to solve small-scale TSP instances and reports the speedup of 7.5 over the corresponding CPU implementation. Author evaluates the effectiveness of proposed approach with real-time application in the steel industry that arranges the cold rolling scheduling of a batch of steel coils. However, only small-size instances have been considered in this work (150 - 318 cities). The GPU part can be utilized more intensively.

O'Neil et al. (2011) present the iterative hill climbing approach along with a 2-opt move to solve TSPLIB 100 city problem. The work arrives at the optimal cost for four different variants of 100 cities, and for one instance i.e., kroE100 reaches a near-optimal cost with 0.07% error. However, this implementation is only limited to 100 city instance. Moreover, a large number of random restarts are required to get the optimal solution.

Delévacq et al. (2012) present parallel GPU implementation of iterative local search for TSP. Author has shown two parallel strategies to generate and compute the neighborhood solution. The 3-opt move has been used for neighborhood generation. The efficiency of proposed approach has been analyzed using TSPLIB instances ranging from 100 - 3038 nodes. Author claims 6.02× speedup over its sequential part. However, the proposed parallel strategy is not scalable, cannot solve instances beyond 3038 due to shared memory limitations.

Delévacq et al. (2013) present a parallel ant colony optimization approach to solve TSP. The ant colony approach's objective is to initially set the population of solutions and find the best out of it. Author uses three parallel strategies to identify efficient methods to solve TSP, which are thread level, block-level, and shared memory implementation. Moreover, a local search approach is applied to each ant solution to improve resultant cost. Author claims that the parallel ant colony approach gives a speedup of 23.60 over corresponding sequential implementation. However, the limitation of this implementation is more storage requirements to store the pheromone matrix, distance matrix, and candidate list. This memory requirement makes the proposed approach prohibitive while solving large input instances; therefore, reported experimental results up to d2103 instances.

Cecilia et al. (2013) present the data parallelism scheme for tour construction and pheromone update to solve TSP using ant colony optimization. The proposed parallel model needs a pheromone matrix and distance matrix on GPU before beginning the actual computation. The storage requirement makes the proposed approach inapplicable while solving large input instances. The proposed work has been evaluated using TSPLIB instances ranging from 198 - 2393 nodes and claims speedup of up to 20× over sequential counterpart.

Rocki and Suda (2012) evaluate the impact of 2-opt and 3-opt neighborhood generation methods to solve TSP efficiently to get the near-optimal solution. The work also constrains on minimizing memory utilization and efficient CPU-GPU data transfer. The proposed approach improves neighborhoods' local search time using the GPU 3 to 26 times compared to the parallel CPU code using 32 cores. Time analysis of different TSP instances ranging from 100 to 4461 nodes reveals the potential of 2-opt and 3-opt approaches. However, author does not report the solution quality.

Fosin et al. (2013) present the parallel local search implementation for 2-opt and 3-opt approaches for solving symmetric TSP instances. Author reports 27× speedup over corresponding CPU implementation. The proposed approach has used Nearest Neighborhood (NN) technique to determine the initial solution rather than choosing randomly. However, author does not report the solution quality of TSPLIB instances.

Luong et al. (2013) have used the tabu search mechanism as a metaheuristic approach with the 2-opt move neighborhood generation method to solve TSP on GPU.

Table 2.2: Characteristics of GPU-based methods. Each row represents existing work, heuristic method used to solve TSP, neighborhood generation technique, instance size considered, error rate, speedup over sequential part, limitation of existing work.

Work	Heuristic	Neighbors Generation	Instances	Initial Solution	Cost quality	Speedup	Limitations
Fujimoto and Tsutsui (2011)	Genetic	2-opt, OX	120 - 493	random	0.5 %	24.2	Instance size >1024 can not be solved.
Zhao et al. (2011)	Genetic	OX	130 - 318	random	6.9%	7.54	Consumes more GPU time with error rate 6.9%.
O'Neil et al. (2011)	IHC	2-opt	100	random	0.07%	62	Limited to 100 city.
Delévacq et al. (2012)	IHC	3-opt	100 - 3038	random	NA	6.02	Limited by shared memory.
Delévacq et al. (2013)	ACO	3-opt	51 - 2103	random	Optimal	23.60	Limited by shared memory.
Cecilia et al. (2013)	ACO	2-opt	198 - 2392	random	Optimal	21.71	Requires large GPU memory.
Rocki and Suda (2012)	ILS	2-opt, 3-opt	100 - 4461	random	NA	26	Limited by shared memory.
Luong et al. (2013)	Tabu search	2-opt	101 - 5915	random	NA	19.9	Cost quality has not exposed.
Rocki and Suda (2013)	ILS	2-opt	52 - 85900	MF	11.66%	300	Error rate grows when size increases.
Neil and Burtscher (2015)	IHC	2-opt	32767	random	12.99%	8 on 20 cores	Large random restart is required.
Fosin et al. (2013)	ILS	2opt, 3opt	85900	40NN	NA	27	Cost quality has not exposed.
Zhou et al. (2016)	ILS	2-opt	4461	random	3.7%	279	Shared memory limitation.
Robinson et al. (2018)	IHC	2-opt	33810	random	NA	22.9	Large random restart is required.
Boqun et al. (2020)	GA	crossover, mutation	31 - 195	random	Optimal	200	Larger instances can be solved.

Author presents the texture memory approach to enhance the performance of general GPU implementation. The author provides efficient approaches for CPU-GPU data transfer optimization, thread control, neighborhood structure mapping to GPU threads, and memory management. The proposed algorithm obtains up to 19.9× speedup compared to the corresponding sequential part on instances ranging from 101 to 5915 nodes. Here, the author uses four different GPU devices to analyze the scalability of their GPU implementation. Moreover, author also evaluates GPU computing against the cluster, grid computing for combinatorial optimization problems. However, author does not report the solution quality for TSPLIB instances.

Rocki and Suda (2013) present the local search approach along with a 2-opt move. Author uses a problem division scheme to solve larger instances and reports 300× performance acceleration over sequential counterpart. The researcher presents different parallel strategies to enhance the performance of GPU computation. The proposed approach can solve large TSPLIB instances effectively. On the other side, the cost quality reported in the experimental result can be improved.

Neil and Burtscher (2015) illustrate the random hill climbing approach with the 2-opt move to solve TSP on GPU. The work reduces memory latency in the GPU implementation using the GPU memory hierarchy. The approach is 3× faster than existing methods and 8× faster than OpenMP implementation on 20 CPU cores. This approach solves large TSPLIB instances ranging from 105 - 33820 size. The resultant solution's cost quality can be improved by setting an initial solution using the Nearest Neighborhood approach (NN) rather than randomly setting it up.

Zhou et al. (2016) shows the parallel iterated local search algorithm with the 2-opt feasible solution generation method to solve TSP instances. The works' contribution is the efficient mapping between neighborhood solution and GPU threads, used the roofline model to analyze the performance of existing GPU methods. Moreover, author exhibits different distance calculation approaches such as the precalculated distance approach (LUT) and On-time distance calculation approach to find a fast TSP solving method. The author claims 279× speedup over corresponding sequential implementation and demonstrates their approach's performance with state-of-the-art methods. However, the implementation evaluation is done on moderate-size TSPLIB input instances ranging from 198-4461 nodes.

Robinson et al. (2018) have presented the GPU-based parallel model for the random-restart hill climbing algorithm. Authors have provided an improved version of the 2-opt move by allowing k multiple updates simultaneously to the current Hamiltonian path.

This improved version of 2-opt results in higher speedup (i.e., 4.5× - 22.9×) compared to the work (Neil and Burtscher 2015) for the instances in the range of 1400 - 33810. However, the solution quality can be further improved using the construction heuristic approach instead of using random initial solutions.

Boqun et al. (2020) present the GPU-based parallel model for Genetic Algorithm (GA). Authors have evaluated the performance of proposed parallel GA usingTSPLIB instances in the range of 31 - 195 nodes. In the best case, when population size is considered 20000, proposed approach receives a speedup of 200 times compared to its sequential version while solving an instance $chn31.tsp$.

2.1.4 Limitations

Table 2.2 presents the state-of-the-art in the GPU implementation of heuristic based TSP solvers. Columns represent heuristic methods, neighborhood generation approaches, TSPLIB input sizes, initial solution construction approaches, and limitation of each existing TSP solvers. The error rate is calculated using Eqn. 2.1, where $cost_{final}$ is the cost obtained using the construction approach on TSPLIB instance and $cost_{opt}$ is the optimal cost of the TSPLIB instance.

$$Error\ rate = \frac{cost_{final} - cost_{opt}}{cost_{opt}} \times 100 \tag{2.1}$$

From Table 2.2, it is observed that many TSP solvers (Cecilia et al. 2013; Delévacq et al. 2013; Fujimoto and Tsutsui 2011; O'Neil et al. 2011; Rocki and Suda 2012; Zhao et al. 2011; Zhou et al. 2016) could not solve TSPLIB (Reinelt 1991) instances of more than 5915 nodes. Although work (Neil and Burtscher 2015; Rocki and Suda 2013) solves the large size instances, the error rates are large up to 11.66%, 12.99% respectively.

2.1.5 Contribution

In Chapter 3, a GPU-based parallel iterative hill climbing algorithm has been designed to solve large-sized instances. The proposed GPU-based parallel approach has been evaluated using TSPLIB instances of up to 85900 nodes. The proposed approach produces good quality solutions with error rates in the range of 0.72%-8.06% in a reasonable amount of time. Moreover, Chapter 4 presents two GPU-based parallel strategies for the populations-based metaheuristic algorithm i.e., ant colony optimization to solve larger TSPLIB instances.

2.2 MINIMUM LATENCY PROBLEM

Extensive work has been carried out for solving MLP in the past decade. Several metaheuristic approaches have been proposed to produce better solution quality. Twofold objectives are targeted for studying existing work. First, identify the maximum size of instance considered for solving MLP. Second, discover whether any parallel strategy is applied to reduce the execution time of the metaheuristic algorithms using the parallel implementation.

Salehipour et al. (2011) and Silva et al. (2012) proposed a metaheuristic to solve MLP instances up to 1000 nodes. Ban and Duc (2014) propose the population-based algorithm that is a combination of the Genetic Algorithm (GA) and Ant Colony (AC) algorithm to solve MLP. In each subpopulation, AC is applied, followed by GA, to improve the respective subpopulation. This hybrid approach helps to maintain diversity in solution. The author shows the effectiveness of their approach using TRP and TSPLIB instances. However, the size considered in the evaluation is smaller, i.e., up to 493 nodes. The execution time increases as instance size increases.

A combination of Tabu Search (TS) and Variable Neighbourhood Search (VNS) has been proposed in (Ban and Nguyen, 2017) to solve MLP. TS is used to avoid generating the same solution. VNS is used to get out of trapping into local optima. The performance of the proposed algorithm has been evaluated using random and TRP instances up to 500 nodes. It produces a better solution quality on the TRP instances. However, the algorithm's running time can be reduced, and larger instances can be solved.

Avci and Avci (2017) solves an extension of MLP, i.e., Traveling Repairman Problem with Profits (TRPP). In TRPP, there is no rule to visit all nodes. The author uses GRASP to construct an initial solution and ILS for the solution improvement. The proposed metaheuristic produces the best solutions for 46 TRPP instances. The maximum size considered in this evaluation has 500 nodes. For a 500-size instance, the proposed algorithm takes time in minutes, which can be reduced using a parallel implementation.

Table 2.3: Overview of existing metaheuristic approaches that solve the minimum latency problem.

Author	Data Set	Methodology	Parallel	Advantages	Limitations
Salehipour et al. (2011)	TRP, TSPLIB	GRASP+VND, GRASP+VNS	No	- Design metaheuristic to solve MLP. - Develops 8 set of data sets for MLP.	- Instances are considered up to 1000 nodes.
Silva et al. (2012)	TRP, TSPLIB	GILS+RVND	No	- Constant time evaluation method is designed.	- Instances are considered up to 1000 nodes.
Ban and Duc (2014)	TRP, TSPLIB	GA+AC	No	- Reduces gap rate in final optima. - Hybrid of AC and GA constructs several initial solutions.	- Instances are solved up to 439 nodes. - Execution time requirement.
Ban and Nguyen (2017)	TRP, Random	TS+VNS	No	- TS avoids duplicate solutions. - VNS helps to avoid trapping in local optima.	- Running time can be improved. - Instances are solved up to 500 nodes.
Avci and Avci (2017)	TRPP	GRASP+ILS	No	- Provides an improved solution for 46 instances.	- Instances are considered up to 500 nodes.
Rios et al. (2018)	TRP, TSPLIB	GRASP+ILS	Yes	- Multi-improvement neighborhood exploration is used. - Produces solution quality better than previous works.	- Solve instances up to 1024 nodes.
Pereira Araujo et al. (2018)	TRP, TSPLIB	DVND	Yes	- Dataflow implementation is used for a local search. - Parallel strategy is designed.	- Parallel design solve instances up to 1024.
Araujo et al. (2020)	TRP, TSPLIB	DVND, RVND	Yes	- Improves the Sucuri dataflow library implementation.	- Parallel strategy is limited to 1024 nodes.
Santana et al. (2020)	TRP, TSPLIB	MDM-GILS-RVND	No	- Provides new best solution for 32 instances. - Used data mining technique to improve solution.	- Instances were considered up to 1379 nodes.
Lu et al. (2019)	TRPP	HESA	No	- Provides new best solutions for 39 TRPP instances.	- Instances are solved up to 500 nodes.

Rios et al. (2018) develop GPU-based metaheuristic using GRASP and ILS to solve MLP. The author proposes a new neighborhood exploration technique called multi-improvement. The designed parallel algorithm obtains up to 13.7× speedup over sequential counterpart for instances of size up to 1000 nodes. The existing GPU-based parallel strategy limits to solve instances up to 1024 nodes. The new GPU-based parallel strategy can be designed to solve instances larger than 1024 nodes.

Pereira Araujo et al. (2018) propose a dataflow implementation for Distributed VND (DVND) to solve MLP. A GPU-based parallel strategy is also proposed for DVND steps to run simultaneously. The effectiveness of the proposed strategy has been presented using TSPLIB and TRP instances. However, this parallel design does not solve instances of more than 1024 nodes. This can be overcome. Araujo et al. (2020) apply GPU computing in the dataflow framework for neighborhood search. The proposed approach improves the Sucuri library implementation using Multiple Output Gate (MOG) nodes. DVND is used to solve MLP over GPU platform. However, the proposed parallel strategy is limited to 1024 nodes. An instance larger than 1024 nodes can be solved.

Santana et al. (2020) have used data mining techniques with GRASP to solve MLP. The proposed algorithm finds new best solutions for 32 instances. The maximum size instance considered in the experiment has 1379 nodes. Lu et al. (2019) propose a population-based Hybrid Evolutionary Search Algorithm (HESA) to solve the MLP variant. Hybrid evolutionary search has used two crossover operators for crossover selection and solution recombination. The proposed approach produces the new best solution for 39 TRPP instances. However, the evaluation is done on the instance size up to 500 nodes.

2.2.1 Limitations

Table 2.3 presents an overview of existing work that uses metaheuristic algorithms to solve MLP. The first row represents the author name, data set used, metaheuristic used to solve, whether parallelism is applied, advantages, and limitations of each work. The literature study observed that the maximum instance size is considered up to 1379 nodes. The existing GPU-based parallel models cannot solve larger instances of more than

1024 nodes due to its parallel design strategy limitations. New parallel implementation can be designed to metaheuristic algorithms to reduce the time spent in the solution improvement phase and solve the very large-scale instances.

2.2.2 Contribution

In Chapter 5, Deterministic Local Search Heuristic (DLSH) is used to solve MLP. A GPU-based parallel strategy has been proposed to reduce execution time spent in the solution improvement. The DLSH and parallel DLSH have been evaluated using 187 instances from TRP and TSPLIB data sets, which size in the range of 10-11849 nodes. The proposed parallel approach receives a speedup of up to 179.75 times compared to DLSH, better than the existing state-of-the-art parallel implementations.

2.3 VEHICLE ROUTING PROBLEM

The existing works have been studied which use metaheuristics approaches while solving VRP. This study is classified according to its implementation nature, i.e., sequential and parallel approaches for metaheuristic algorithms.

2.3.1 Sequential Approach

The Vehicle Routing Problem (VRP) was formulated by Dantzig and Ramser (1959) in the context of scheduling gasoline delivery trucks from source depots to the service stations. Clarke and Wright (1964) provided the first heuristic algorithm to solve CVRP. The Clarke and Wright heuristic is basically used to solve a VRP variant where single depot and unfixed vehicles are used. A survey of VRP variants is presented in (Braekers et al. 2016; Eksioglu et al. 2009; Toth and Vigo 2001). Various heuristic and metaheuristic algorithms to solve the variants of VRP are presented in (Cordeau et al. 2002; Laporte et al. 2000; Pisinger and Ropke 2007).

Bullnheimer et al. (1999) have presented the Ant System (AS) algorithm for CVRP. They have presented AS's performance analysis on the Christofides' 14 benchmarking instances of size 50-150 customers (Christofides et al. 1979). AS constructs n feasible solutions and improves it repeatedly. These algorithms provide reasonable solutions but need to explore large search-space to converge to a good solution. The Particle Swarm

Optimization (PSO) has been used to solve CVRP in (Ai and Kachitvichyanukul 2009; Chen et al. 2006). Chen et al. (2006) uses a hybrid of local search and global search in the PSO to improve the solution and solves CVRP instances up to 135 customers. Ai and Kachitvichyanukul (2009) provides two solution representations and its decoding methods to solve CVRP instances up to 200 customers. Population-based heuristic algorithms like AS and PSO initialize multiple feasible solutions and apply improvements over them repeatedly.

Baker and Ayechew (2003) present the Genetic Algorithm (GA) to solve VRP. They have designed a hybrid of GA with neighborhood search methods that provide competitive results with simulated annealing and tabu search algorithms. This idea of hybridization is also used in (Prins 2004) for VRPs. Alba and Dorronsoro (2004) have proposed a GA that enhances the exploration and population diversity of the search-space.

Simulated Annealing (SA) is a heuristic algorithm with emulated thermodynamics concepts where heating and cooling are applied to the material to enhance crystals. SA is also used to solve VRP variants (Wei et al. 2018; Yu et al. 2017). Yu et al. (2017) proposes the Hybrid VRP (HVRP), i.e., an extension of Green VRP (GVRP), and applied SA to solve it. Wei et al. (2018) have considered two-dimensional loading constraints with CVRP.

Tabu search (TS) algorithm is designed to avoid visiting the same feasible solutions repeatedly. Initially, Taillard (1993) have proposed TS to solve CVRP. They have designed benchmark instances for CVRP to test the efficiency of their proposed algorithm. Later, Barbarosoglu and Ozgur (1999); Gendreau et al. (1994); Osman (1993); Rego and Roucairol (1996); Rochat and Taillard (1995); Toth and Vigo (2003) have provided a different variations to TS for improving solution quality to solve VRP variants. Gendreau et al. (2004) proposed TS for the VRP with two-dimensional load constraints.

Li et al. (2005) has built twelve benchmark instances of size 500 - 1200 customers. Kytöjoki et al. (2007) proposed a set of twenty very large-scale instances up to the size of 20000 customers. They have designed the Variable Neighborhood Search (VNS) heuristic to solve such a larger data set. Uchoa et al. (2017) have introduced a balanced and comprehensive set of 100 instances of size 100 - 1000 customers considering the characteristics that exist in real applications. Arnold and Sörensen (2019) have developed a set of larger instances which has customers up to 30000. These instances have been built considering the real-time scenario of parcel distribution in Belgium. Local search heuristic is used to solve these instances.

The limitation of sequential approach is that when large-scale instances are solved, existing metaheuristic algorithms spend huge CPU time to reach its local optima. Since metaheuristic uses neighborhood generation methods to improve the solution quality, these neighborhood methods can be implemented in parallel to reduce the execution time. The existing parallel models available for solving CVRP are overviewed in the next subsection.

2.3.2 Parallel Approach

Jin et al. (2014) have implemented TS in parallel to solve CVRP. Multiple TS threads work in parallel to solve CVRP. Some of the threads are responsible for solution intensification while the others are responsible for the solution diversification. These threads communicate with each other through a common solution pool. The efficiency of proposed parallel TS approach is shown using the Golden et al. (1998) and Li et al. (2005) data sets. Schulz (2013) has provided a GPU-based parallel strategy for the local search to solve Distance constrained VRP (DVRP). In particular, parallel strategies are designed for the 2-opt and 3-opt heuristics. These parallel designs' efficiency has been evaluated on the ten CVRP and DVRP instances of sizes 57 - 2401. Boschetti et al. (2017) have used the GPU platform to implement q-route and ng-route relaxations. The dynamic programming uses q-route and ng-route relaxations for computing their bounding components. A parallel version achieves the speedup up to 40 times over the sequential counterpart. Abdelatti and Sodhi (2020) have proposed the GPU-based parallel design for Genetic Algorithm (GA) to solve CVRP instances of size up to 76 customers.

Table 2.4: Details of existing parallel implementations available for VRP variants.

Articles	VRP variants	Method	Parallel Approach	Instance Size
Jin et al. (2014)	CVRP	Tabu Search	Multicore-based	240 - 1200
Schulz (2013)	CVRP, DVRP	Local Search	GPU-based	57 - 2401
Boschetti et al. (2017)	CVRP	Dynamic Programming	GPU-based	34 - 2000
Abdelatti and Sodhi (2020)	CVRP	Genetic Algorithm	GPU-based	16-76

2.3.3 Limitations

Table 2.4 shows the overview of the existing parallel models available for VRP variants. The first row represents the name of article that uses a parallel model, which variant of VRP is solved, whether heuristic is considered, what kind of parallel approach is used, and what is the range of instances, respectively. The performance of parallel approaches have been evaluated on smaller size instances i.e., up to 2401, in the existing works Boschetti et al. (2017); Jin et al. (2014); Schulz (2013). Real-time applications have to serve thousands of customers. Therefore solving such instances consumes much CPU times for heuristic algorithms (Typically in minutes to several hours). Kytöjoki et al. (2007) has been provided a larger size instances which can be considered for evaluating the performance of parallel strategies. In Schulz (2013), author provides the GPU-based parallel strategies only for intra-route improvement heuristics, i.e., 2-opt and 3-opt heuristics.

2.3.4 Contribution

Chapter 6 presents two GPU-based parallel strategies for intra-heuristics and inter-heuristics algorithm to solve large-scale CVRP. Implementing local search heuristics in parallel will yield better CVRP performance. Proposed work identifies hotspots in the local search heuristic implementation and parallelizes to achieve better performance for the same solution quality. Proposed work provides parallel strategies for three intra-route and two inter-route improvement heuristics approaches. The performance of proposed work has been evaluated on instances up to 30000 nodes.

2.4 RESEARCH GAPS

There are various places in heuristic approaches that need to be improved.

- **Instance Size:** The literature study shows that the efficiency of existing parallel strategies are evaluated using the smaller-size instances. In the case of MLP, the maximum size is considered up to 1379 nodes, whereas 2401 nodes are considered for CVRP. Real-time applications need to deals the large amount of customers. Therefore, designing scalable parallel strategies to handle large-scale instances has great importance.
- **Initial Solution Construction:** Construction of an initial solution is the first step in the optimization problem using metaheuristic methods. Initial route are either chosen sequentially or generating randomly. Setting up initial route using NN approach always assures a good quality solution at initialization stage itself rather than generating randomly or in sequence of city order. The upper bound on initial solution is within$\leq 0.5 \times (log_2 n + 1)$ (Rosenkrantz et al. 1974). In random and sequenced initial route, it is difficult to determine upper bound on initial solution and involves more number of iterations to get locally optimal solution than NN approach.
- **Memory Limitations in Population based Metaheuristic:** The existing population of solutions-based approaches requires large GPU memory to store distance matrix and pheromone matrix, therefore cannot solve large input instances. This can be tackled using on-the-fly distance calculation approach instead of using precalculated distance matrix.

2.5 SUMMARY

In this chapter, different metaheuristic algorithms have been presented which are used to solve combinatorial optimization problems. The existing literature study is categorized based on the combinatorial optimization problems. The limitation of existing works is presented in the respective section of optimization problems. Three combinatorial optimization problems have been considered as the case study for designing the parallel metaheuristic model to solve large-scale instances.

In the next chapters, efficient parallel models have been provided for metaheuristic algorithms while solving three optimization problems, namely Traveling Salesman

Problem (TSP), Minimum Latency Problem (MLP), and Capacitated Vehicle Routing Problem (CVRP). This thesis target to solve large-scale benchmarking instances than the existing works and reduce the execution time spent in the solution improvement phase.

CHAPTER 3

PARALLEL ITERATIVE HILL CLIMBING ALGORITHM TO SOLVE TSP ON GPU

In this chapter, four CUDA thread mapping strategies have been demonstrated to solve TSP on the GPU platform using the iterative hill climbing algorithm. Moreover, different initial solution construction approaches and impact of various data transfer techniques from the CPU to the GPU have been demonstrated. Performance evaluation of GPU implementation of PIHC is examined using TSPLIB instances up to 85900 cities and the performance is compared with state-of-the-art GPU-based TSP solvers (Source code link: `http://bit.ly/thesisSourceCodes`).

3.1 INTRODUCTION

The Traveling Salesman Problem (TSP) (Gutin and Punnen 2002; Johnson and Mcgeoch 1997) is an NP-hard (Garey and Johnson 1990), $O(n!)$ combinatorial optimization problem. In the worst case, the time complexity of TSP is factorial time when solved using the brute-force method and exponential time when solved using the dynamic programming (Cormen et al. 2009). For its large number of applications in science and engineering, time efficient TSP solutions are of great importance.

The objective of TSP is to find the minimal cost cycle that passes through each city exactly once and returns to the originating city. In short, in a map of n cities, distance of path π has to be minimized,

$$d(\pi) = \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} + d_{\pi(n),\pi(1)} \tag{3.1}$$

where $d_{i,j}$ is a distance between any two cities i and j, and π is a permutation of $(1, 2, ..., n)$ (Merz and Freisleben, 2001).

According to graph theory, the input to TSP is a simple, weighted, connected graph which has cities as nodes, paths between cities as edges and distance between cities as weights on the edges. The goal of TSP is to find a minimal weighted Hamiltonian cycle from the simple, weighted, connected graph $G = (V, A, d)$ where, V= set of cities to be visited, A= $\{(i, j)|(i, j) \in V \times V\}$ is the set of paths between cities and $d : A \longrightarrow R$ is a function which assigns distance $d_{i,j}$ to every path between cities i and j. A Hamiltonian cycle in a simple connected graph that visits each node exactly once and returns to the starting node (Cormen et al., 2009).

3.1.1 Exact and Heuristic Approaches

Optimization problems are solved using two broad approaches; viz., exact methods and heuristic methods. Generally, exact methods determine the optimal solution by exploring the entire search-space of the problem. Some examples of exact methods are brute-force algorithm, branch and bound, dynamic programming approaches. Heuristic methods give near-optimal solutions by exploring a limited search-space in a reasonable amount of time (Talbi, 2009). For n city TSP, an optimal solution is determined after exploring $\frac{(n-1)!}{2}$ different feasible solutions when the brute-force and the branch and bound are used (Cormen et al., 2009). When dynamic programming is used, an optimal solution is determined after exploration of $n^2 \times 2^n$ feasible solutions in the worst-case (Cormen et al., 2009). While the exact method always assures an optimal solution, the CPU time is prohibitive for large n. Due to the large search-space exploration, exact methods cannot be time-efficient approaches to solve the large TSP instances.

Heuristic algorithms explore a subset of feasible solutions by constraining the search-space and derive the satisfactory solutions in a reasonable amount of time. Heuristic methods start with the feasible solution and iteratively improve on it to converge to a near-optimal solution until the termination criteria is met (Talbi 2009). The termination criteria may include the number of iterations, an optimal cost value, no improvement of solutions between iterations. Popular heuristic approaches are iterative hill climbing, tabu search, simulated annealing, ant colony optimization, and genetic algorithm.

In this chapter, iterative hill climbing approach has been used as the base algorithm and GPU-based parallel strategies have been designed for 2-opt move. Note that a new heuristic algorithm is not presented to solve TSP. The CUDA-based thread mapping strategies have been applied to existing 2-opt move to reduce neighborhood generation time. Iterative hill climbing approach first sets the initial solution and later improves it iteratively till no further improvement is possible. Although heuristic methods to solve TSP are faster than the exact methods, the execution time increases as input size increases. For large input sizes, GPU implementation of TSP have been shown to complete in a reasonable amount of time (Zhao et al. 2011).

In this chapter, a GPU-based parallel iterative hill climbing algorithm has been designed to solve large size instances. The presented GPU-based parallel approach is used to evaluate TSPLIB instances of up to 85900 cities. The approach produces good quality solutions with error rates in the range of 0.72%-8.06% in a reasonable amount of time. Section 3.2 presents the PIHC heuristic algorithm in detail. Performance evaluation of the proposed approach has been demonstrated in Section 3.3.

3.2 PARALLEL ITERATIVE HILL CLIMBING ALGORITHM

This section presents the iterative hill climbing algorithm in detail, different ways to setup initial solutions, neighborhood generation techniques, different thread mapping strategies, and the optimized data to be considered on GPU platform.

Algorithm 3.1 presents a generic work-flow of the iterative hill climbing algorithm. Stepwise mechanism of the algorithm is follow. Determine the initial solution, also called the candidate solution, s. Once the initial solution is determined, its associating cost, $f(s)$, is calculated. Next, generate the neighborhood solutions, $N(s)$ of the initial solution using swapping techniques like 2-opt move. $\forall s^{'} \in N(s)$, $f(s^{'})$ is calculated using the distance calculation method, where $s^{'}$ is neighborhood solution and $f(s^{'})$ is corresponding cost of $s^{'}$. Cost of each $f(s^{'})$ is compared with the initial cost $f(s)$. The best improved $f(s^{'})$ on the $f(s)$ is considered as a local optimal cost. To move towards the global optimal solution, this algorithm must be called repeatedly. For each call, the locally found optimal solution $s^{'}$ acts as the initial solution s in the next iteration.

Algorithm 3.1: Generic Iterative Hill Climbing Algorithm

```
$s \leftarrow InitialSolution()$
$f(s) \leftarrow calculateDistance(s)$
while termination criteria not met do
    $N(s) \leftarrow GenerateNeighborhood(s)$
    for $\forall s^{'} \in N(s)$ do
        $f(s^{'}) \leftarrow calculateDistance(s^{'})$
        if $f(s^{'}) < f(s)$ then
            $f(s)=f(s^{'})$
        end
    end
end
```

This process continues until a termination criteria is met. Termination criteria could be, the number of iterations, further improvement not possible or optimal solution reached, etc. For each call of the hill climbing procedure, cost quality of the solution will potentially improve and move towards the global optimal. The important steps of this algorithm are the initial solution construction and generation of neighborhood solutions that are elaborated in following sections.

3.2.1 Initial Solution Construction

The initial solution construction is the first step in the IHC algorithm. Any feasible solution is chosen as the initial solution either in sequenced order of cities or arbitrarily or constructed using the construction heuristics. Various initial solution construction approaches are discussed below.

3.2.1.1 Sequenced

This is a fundamental way to construct the initial solution where the feasible solution is made up of cities in sequence Eg. 1, 2, 3, . . ., n, 1. In the worst-case, the sequenced initial solution gives maximal weighted Hamiltonian cycle since distance between cities is not considered while building the initial solution.

3.2.1.2 Random

Starting from the source node, the initial solution is constructed by choosing a random neighbor for each current node until a feasible solution is constructed. Most of the GPU-based TSP implementations use the randomly constructed initial solution. Both the sequenced and random methods do not consider the distances while identifying the neighbors.

3.2.1.3 Nearest Neighborhood (NN)

Nearest Neighborhood (NN) is a construction heuristic approach where a feasible solution is constructed instead of selecting a random feasible solution. The stepwise details of constructing the initial solution are presented in Algorithm 3.2. The mechanism of algorithm has been explained as follow. First, a random city is chosen as a source node. Next, a closest unvisited node of previously visited node is identified repeatedly until all cities are visited. The last hop returns to the source city to form a Hamiltonian cycle.

Algorithm 3.2: Generic Nearest Neighborhood Algorithm

```
Result: Returns constructed route as an initial solution
$route \leftarrow \emptyset$
select any random node $i$
$route \leftarrow$ add node $i$
while all nodes not visited do
    $j \leftarrow$ a neighbor of $i$ that has minimal cost
    $route \leftarrow$ append $j$
    $i \leftarrow j$
end
return $route$
```

Since symmetric TSPLIB instances have been considered in this work, for a n city problem, NN needs $O(n^2)$ time to construct the initial solution. The upper bound on the cost of the initial solution construction technique can be determined. Assume $cost_{opt}$ is the optimal cost and $cost_{route}$ is cost of the constructed initial solution. The upper bound on the cost of initial solution is $\frac{cost_{route}}{cost_{opt}} \leq 0.5 \times (log_2 n + 0.5)$ (Rosenkrantz et al. 1974). The NN mechanism identifies the initial solution that is typically closer to the optimal compared to the random and sequenced initial solutions. Compared to the sequenced and random approaches, NN needs fewer steps up to 17.20× and 19.16× respectively to reach a better local optimal solution. Experimental results shows that for TSPLIB instances of 200 - 18512 cities the error rate ranges are - 6.07% to 15.9% for sequenced mechanism, 7.49% - 14.26% for random, and 0.72% - 6.73% for NN.

3.2.1.4 Nearest Insertion (NI)

Nearest Insertion (NI) is a variant of the NN approach which constructs a Hamiltonian cycle by joining the nodes to a subtour. Algorithm 3.3 shows the stepwise details of the NI approach. An arbitrary node, i, is chosen as the starting node. Next, choose a minimal weighted unvisited neighbor j of the node i. Now, construct a subtour $i - j - i$. Next, find an unvisited closest node, k, to any node from the subtour. Once k is determined, find an edge, (i, j), from the subtour such that the difference is minimized, i.e., $diff = c_{i,k} + c_{j,k} - c_{i,j}$. Add node k into the subtour in between the edge (i, j). Repeat these steps until a subtour becomes a Hamiltonian cycle. The time complexity of constructing the initial solution is $O(n^2)$ (Rosenkrantz et al., 1974). The upper bound on the cost of initial solution is $\frac{cost_{tour}}{cost_{opt}} \leq 2$.

Algorithm 3.3: Generic Nearest Insertion Algorithm

```
Result: Returns constructed route as a initial solution
$i \leftarrow$ select any random node as source node
$j \leftarrow$ find closest node of $i$
$tour \leftarrow i, j, i$
while all nodes not visited do
    $k \leftarrow$ find a closest unvisited neighbor to the $tour$
    $edge_{i,j} \leftarrow min(c_{i,k} + c_{j,k} - c_{i,j})$
    $tour \leftarrow$ add $k$ to $tour$ between $i, j$
end
return $tour$
```

3.2.1.5 Greedy Algorithm

Algorithm 3.4 shows the greedy algorithm to construct a Hamiltonian cycle. The mechanism of greedy algorithm has been elaborated as follows. Pick an arbitrary node as a starting node. Lines (3-6): choose an unvisited minimal weighted neighbor, j, to the tour repeatedly until it forms a Hamiltonian cycle. Note that NN chooses a closest unvisited node of a recently added node of the tour, whereas, greedy approach selects a closest unvisited node to any nodes of the tour whose degree is less than two. The time complexity of greedy algorithm is $O(n^2 logn)$ (Johnson and Mcgeoch 1997). The upper bound on the cost of initial solution is $\frac{cost_{tour}}{cost_{opt}} \leq (0.5)(log_2 n + 1)$.

Algorithm 3.4: Generic Greedy Algorithm

```
$i \leftarrow$ an arbitrary source node
$tour \leftarrow$ add $i$
while all nodes not visited do
    $j \leftarrow$ find a closest unvisited neighbor to the $tour$
    $tour \leftarrow$ add $j$
end
return $tour$
```

3.2.1.6 Minimum Spanning Tree (MST)

MST is a construction heuristic algorithm and its steps are presented in Algorithm 3.5. First, construct the minimum spanning tree, MST, for a given input. Next, find a node, $start$, from MST which has degree one. Start traversing MST from a node $start$, doubling every edge. This will form the Eulerian cycle, eul_{tour}. Now convert the Eulerian cycle into the Hamiltonian cycle, ham_{tour}. To form a Hamiltonian cycle, start visiting the nodes from $start$, add unvisited nodes to the ham_{tour}. If a visited node is found, skip it and move to the next node. Repeat these until a Hamiltonian cycle is formed. The time complexity of MST algorithm is $O(n^2)$ (Johnson and Mcgeoch 1997) and the upper bound on the cost of the constructed Hamiltonian cycle is $\frac{cost_{tour}}{cost_{opt}} \leq 2$.

Algorithm 3.5: Generic Minimum Spanning Tree Algorithm

```
$MST \leftarrow$ construct the minimum spanning tree
$start \leftarrow$ one degree node of $MST$
$eul_{tour} \leftarrow$ doubles the edges of $MST$
$ham_{tour} \leftarrow$ add $start$
while all nodes not visited do
    $ham_{tour} \leftarrow$ Traverse $eul_{tour}$ from $start$ and add node $i \in eul_{tour}$ exactly
      once
end
return $ham_{tour}$
```

3.2.1.7 Christofides' Algorithm

Christofides' algorithm is a variant of the MST algorithm. The stepwise details of Christofides' algorithm are shown in Algorithm 3.6. First, construct a MST graph. Next, find out the odd degree nodes. Now calculate the minimum weight perfect matching for the odd degree nodes. Add these edges of the minimum weight perfect matching to the MST graph. The newly added edges may result in double the edges of MST, forming cycles in the MST. Now construct Eulerian cycle from the MST. Finally, convert an Eulerian cycle into a Hamiltonian cycle. The time complexity of Christofides' algorithm is $O(n^3)$ (Christofides, 1976). The cost upper bound of the constructed Hamiltonian cycle is $\frac{cost_{tour}}{cost_{opt}} \leq 1.5$.

Algorithm 3.6: Christofides' Algorithm

```
$MST$ ← construct the minimum spanning tree
$odd\_set$ ← find odd degree nodes of $MST$
$edges$ ← calculate the minimum weight perfect matching for $odd\_set$ nodes
$MST$ ← add $edges$
$eul_{tour}$ ← doubles the edges of $MST$
while all nodes not visited do
    $ham_{tour}$ ← Traverse $eul_{tour}$ and add unvisited node $i$
end
return $ham_{tour}$
```

3.2.1.8 Clarke-Wright Algorithm

Clarke-Wright algorithm is popularly used for vehicle routing problem (Clarke and Wright, 1964). The mechanism of Clarke-Wright is shown in Algorithm 3.7 and its stepwise details follow. First, select an arbitrary node, k, as a central node. Now, for each i, j pair, calculate the saving, $s_{i,j} \leftarrow c_{i,k} + c_{j,k} - c_{i,j}$, where, $i \neq j$, $i \neq k$, and $j \neq k$. Now sort the savings in descending order. Add k as a source to the tour. Traverse the savings from the highest saving to the lowest, add nodes i, j to the tour if they are not visited. Repeat visiting the savings until a Hamiltonian is constructed. The time complexity of Clarke-Wright algorithm is $O(n^2 logn)$ (Johnson and Mcgeoch, 1997). The cost upper bound of the constructed Hamiltonian cycle is $\frac{cost_{tour}}{cost_{opt}} \leq log_2 n$.

The performance analysis of these initial solution construction techniques have been

Algorithm 3.7: Generic Clarke-Wright Algorithm

```
$k \leftarrow$ select an arbitrary node as a center
$s_{i,j} \leftarrow c_{i,k} + c_{j,k} - c_{i,j}$ $\forall i, j$ where $i \neq j, i \neq k, j \neq k$
Arrange $s_{i,j}$ in descending order $\forall i, j$
$tour \leftarrow$ add $k$
while a Hamiltonian cycle is not formed do
    $tour \leftarrow$ add unvisited highest saving node pairs $i, j$
end
return $tour$
```

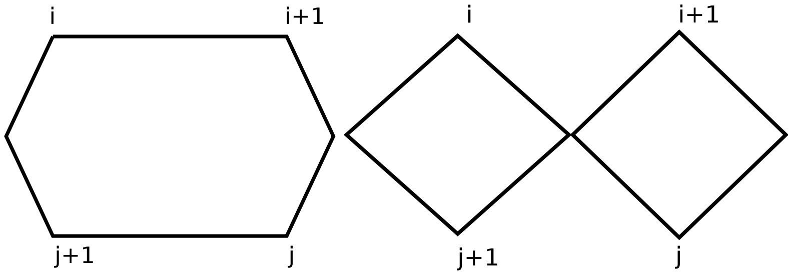
presented in more detail in Section 3.3.1.

3.2.2 2-opt Neighborhood Generation

Using the initial solution, candidate feasible solutions are generated from its permutations. In this work, 2-opt move is used to generate neighborhood solutions of the initiated solution. The n-city TSP problem has $(n-1)!$ different feasible solutions. 2-opt move reduces the search-space to be explored to an order of $O(n^2)$. In the 2-opt move, neighborhood solutions are generated by removing two edges from the initiated solution and reconnecting the newly created 2 sub-routes such that it forms a feasible solution. The stepwise mechanism of 2-opt move method follows.

- Consider a pair of cities, i and j.
- Remove the edge between city i and city $i+1$; remove the edge between city j and city $j+1$.
- Connect an edge between cities i and j; connect another edge between cities $i+1$ and $j+1$.

Figure 3.1 illustrates the 2-opt move mechanism between cities i and j. Figure 3.1 (a) is the original graph before applying the 2-opt move and Figure 3.1 (b) is the newly generated neighbor after applying the 2-opt move. The 2-opt neighborhood generation mechanism is performed on all possible pairs of the cities on the initial solution. Therefore, number of neighborhood solutions generated with 2-opt move is $\frac{n\times(n-1)}{2}$, where n is the total cities. The time complexity of 2-opt neighborhood generation mechanism is $O(n^2)$.



(a) Before 2-opt move (b) After 2-opt move

Figure 3.1: The 2-opt neighborhood generation mechanism

The 3-opt move method can also be used for neighborhood generation. It significantly slows down the searching process because in 3-opt, three edges are swapped at a time to generate a new feasible solution. The 3-opt mechanism explores $\frac{n\times(n-1)\times(n-2)}{6}$ feasible solutions. Neighborhood can be generated either at CPU or GPU.

3.2.2.1 Neighborhood Generation at CPU and its Cost Computation at GPU

In this approach, the neighborhood solutions are generated at the CPU, and the cost calculation of each neighborhood solution is computed at the GPU. For this case, when the cost calculation is done on the fly without using any pre-calculated distances, input coordinates are needed to organize in the neighborhood solution's order for each neighborhood solution. In this approach, $\frac{n\times(n-1)}{2}$ copies of coordinates need to be sent on the GPU. However, this approach is very basic and not feasible in practice which results in the communication overhead.

3.2.2.2 Neighborhood Generation and its Cost Computation at GPU

In this approach, the initial solution is sent to the GPU along with its cost. Each thread is involved in generating its corresponding neighborhood solution and its cost calculation. This approach has negligible data transfer time and result in fair work distribution for all threads. Further, this approach reduces computational time significantly. This approach is used in this work.

3.2.3 Optimized Data for TSP

Optimizing the CPU-GPU data transfer time is critical in all GPU implementations. TSPLIB instances are used as input to the TSP problem. These instances are defined in the form of X, Y coordinates. The distances between cities have to be calculated, stored, reused during the execution of the TSP solver. Following three variations of data structures are used to solve TSP on GPU.

Table 3.1: GPU global memory requirement of the Distance Matrix (DM), Triangular Distance Matrix (TDM), X,Y coordinate (COORD) data structures for TSPLIB instances

Instance	DM	TDM	COORD
kroA200	157.03 KB	78.52 KB	1.56 KB
pcb442	764.87 KB	382.43 KB	3.45 KB
vm1084	4.49 MB	2.24 MB	8.28 KB
u1432	7.83 MB	3.91 MB	11.18 KB
u2319	20.52 MB	10.26 MB	23.73 KB
pcb3038	35.22 MB	17.61 MB	23.73 KB
fnl4461	75.93 MB	37.97 MB	34.85 KB
rl5934	134.35 MB	67.17 MB	46.35 KB
rl11849	535.62 MB	267.81MB	92.57 KB
d15112	871.23 MB	435.61 MB	118.06 KB
d18512	1.28 GB	653.67 MB	144.62 KB
pla33810	4.26 GB	2.13 GB	264.14 KB
pla85900	27.48 GB	13.74 GB	671.09 KB
Average	2.66 GB	1.33 GB	111.04 KB

1. Distance Matrix (DM): The distances are stored in $N \times N$ 2D matrix. This matrix has to be sent over the GPU. This method occupies $n \times n \times 4$ bytes GPU memory and it needs the initial solution as well that occupies $n \times 4$ bytes GPU memory. Overall it allocates $(n^2 + n) \times 4$ bytes GPU memory. In GPU implementation, single precision floating point data is used to store city distances, thereby holding 4 bytes per distance.

2. Triangular Distance Matrix (TDM): As symmetric TSPLIB instances are used, there is no need to use the entire distance matrix. Instead, either upper triangular or lower triangular matrix can be used. TDM occupies $\frac{n \times (n-1)}{2} \times 4$ bytes GPU memory, overall it requires $(n + (\frac{n \times (n-1)}{2})) \times 4$ bytes along with initial solution.

3. X, Y coordinate (COORD): The location of each city is represented in x, y coordinate form. The initial solution is represented by arranging the coordinates in the

initial solution order. Therefore, the initial solution need not be sent to the GPU. In this approach, distances are computed on-the-fly, thereby eliminating the pre-calculated distance matrix. Therefore, COORD needs lesser memory compared to the DM and the TDM approaches, i.e., $(2 \times n) \times 4$ bytes.

Amongst the above data structures, COORD is the least bandwidth and least memory consuming representation. Table 3.1 reports the global memory consumption of various TSPLIB input instances using DM, TDM, COORD respectively. It is observed that COORD saves a significant amount of GPU memory when instance size increases. For $pla85900$ instance, COORD reserves 671.09 KB whereas both DM, TDM require 27.48 GB and 13.74 GB respectively. Such volume of data result in prohibitive data transfer time, memory access time in the GPU and requires large computation time. The COORD representation is used in this work to overcome the above disadvantages. Although COORD uses the least global memory for input instances, it needs to spend additional time calculating distances between two cities every time. The DM and TDM approaches calculate distance between any two cities once and allow using precalculated distances whenever needed, whereas the COORD has to calculate a distance between two cities on every request.

3.2.4 Distance Calculation Method

Generally, TSPLIB instances are used as standard benchmark input to identify the effectiveness of a specified heuristic approach to solve TSP. Most of the TSPLIB instances have 2D edge weight assignment type in form of x, y coordinates. Therefore, Euclidean distance formula has been used to calculate the distance of the solution. Consider the x, y coordinates of two cities, (x_1, y_1) and (x_2, y_2), then the Euclidean distance is:

$$d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{3.2}$$

3.2.5 Thread Mapping Strategies

In this work, four thread mapping strategies have been studied and evaluated for the 2-opt move. These strategies are explained below.

	0	1	2	3	4
Thread 0		x	x	x	x
Thread 1			x	x	x
Thread 2				x	x
Thread 3					x

Table 3.2: Threads per Row (TPR) mapping strategy for a 5 cities graph. The columns represent cities from the graph. Thread 0 generates and evaluates neighbors using pairs (0,1), (0,2), (0,3), and (0,4). Thread 1 generates and evaluates neighbors using pairs (1,2), (1,3), and (1,4), and so on.

3.2.5.1 Threads per Row (TPR)

In this strategy, each row is mapped to one CUDA thread. Table 3.2 shows the pictorial representation of threads per row mapping strategy where the indexes of marked blocks are used by respective threads to generate neighbors. Thread id is used as the first city i and the second city, j, is selected from $i+1$ to $n-1$ to generate neighbors of the initial solution. In this strategy, the first thread generates $n-1$ neighbors, second thread generates $n-2$, and so on. The entire 2-opt move search-space, $\frac{n\times(n-1)}{2}$ is covered using $n-1$ threads. This leads to an unequal thread work distribution. Moreover, at each thread, neighbors associated with threads are generated and computed in a serial manner.

$$blocks = \left\lceil \frac{n-1}{1024} \right\rceil \tag{3.3}$$

This strategy needs $n-1$ threads and the blocks are calculated using Eqn. 3.3, where 1024 is the maximum threads per CUDA block. Algorithm 3.8 presents the pseudo code of the 2-opt move using the TPR strategy. Lines (1-3): for each thread, $minchange$ is initialized to zero, cost of the initial solution is assigned, and global id is generated. Lines (4-13): each thread finds cost of every neighbor associated with it and chooses a minimal cost, $mincost$; generates a 1D index of the $mincost$ neighbor using i and j values and stores 1D value in a variable $locId$. The f function calculates the Euclidean distance between two nodes (Line 7). Lines (14-16): finally, a minimum cost of neighbor and its 1D index is chosen from minimum cost neighbors of all threads.

However, in the GPU implementation precautions have to be taken care. Race condition which occurs while selecting the $mincost$ and its associated 1D. The $mincost$

Algorithm 3.8: Pseudo code for GPU based 2-opt using TPR

```
$minchange \leftarrow 0$
$mincost \leftarrow initcost$
$i \leftarrow threadIdx.x + blockIdx.x * blockDim.x$
for $j=i+1; j < n; j{+}{+}$ do
    $cost = initcost$
    $change = 0$
    $change = f(i,j) + f(i+1,j+1) - f(i,i+1) - f(j,j+1)$
    $cost = cost + change$
    if $cost < mincost$ then
        $mincost = cost$
        $locId = i \times (n-1) + (j-1) - \frac{i \times (i+1)}{2}$
    end
end
if $mincost < initcost$ then
    $atomicMin(cost\_id, mincost << 32 | locId)$
end
```

must be communicated with all threads across all blocks. To handle race condition, $atomicMin$ function is used. To make $mincost$ available with all threads, it is stored in the global memory. Alternatively, if shared memory is used to store $mincost$. The minimum across all blocks is needed to be computed and stored in the global memory where each thread can access the latest value. Generally, this strategy is useful to deal with 2D matrix operations.

3.2.5.2 Threads per Row Equal Distribution (TPRED)

This is a variation of the threads per row mapping strategy that equally distributes neighborhood generation work among threads. At each thread, $\lfloor \frac{n}{2} \rfloor$ neighbors are generated. Since distances between pairs (i,j) and (j,i) are identical, $\lfloor \frac{n}{2} \rfloor$ neighbors are equally distributed among threads. Thread id i is used as first city and second city j is selected by increasing i by 1 until j covers all neighbors of the thread. j is calculated as, $j = (i+1)\ mod\ n$. If instance size, n, is even, threads having ids from $\frac{n}{2}$ to $n-1$, have repeated neighbors on the $\frac{n}{2} - 1$ column index. Table 3.3 shows an example of this thread mapping strategy for 5 city graph. This strategy needs n threads and blocks are

Thread id ↓	j →	
0	1	2
1	2	3
2	3	4
3	4	0
4	0	1

Table 3.3: Threads per Row mapping with Equal work Distribution (TPRED) for a 5 cities graph. Here, n=5 and j=(i+1) mod n. The j columns indicate neighbors associated with each thread. Thread 0 evaluates pairs (0,1), and (0,2). Thread 1 evaluates pairs (1,2), and(1,3) and so on.

calculated using Eqn. 3.4.

$$blocks = \left\lceil \frac{n}{1024} \right\rceil \tag{3.4}$$

The pseudo code representation of TPRED is similar to TPR. There is slight change needed for TPRED in Algorithm 3.8 at line 4. The for loop has to be iterated from k = 0 to $\left\lfloor \frac{n}{2} \right\rfloor - 1$. Inside the loop, the j is calculated as, $j = (i + k + 1)\%n$ for each thread i. This strategy is used to deal with the upper triangular and lower triangular matrix operations in order to distribute work equally to threads.

3.2.5.3 Threads per Row and Column (TPRC)

	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
Thread 0		x	x	x	x
Thread 1			x	x	x
Thread 2				x	x
Thread 3					x
Thread 4					

Table 3.4: Threads per Row and Column (TPRC) mapping for 5 cities graph. Thread pairs, where row id < column id, are used to generate and evaluate neighbors.

In this strategy, threads are mapped to both columns and rows which is presented in Table 3.4. The 2-opt move generates $\frac{n(n-1)}{2}$ elements which is equal to the number of elements in the triangular matrix excluding the diagonal elements. $2 \times^n C_2$ threads are involved in computation and the rest are in the idle state. Since maximum threads per block is 1024, 32×32 threads are used per block. The total blocks required is calculated using Eqn. 3.5.

$$blocks = \left\lceil \frac{n}{32} \right\rceil \tag{3.5}$$

Algorithm 3.9 presents the pseudo code representation of the TPRC strategy. Lines (1-4): for each thread, $minchange$ is initialized to zero, cost of the initial solution is assigned, and global id is generated. Lines (5-13): a thread pair is chosen which has the minimum cost and 1D index. At the CPU, these results are used to determine if further improvement is possible. Generally, this strategy is useful for square matrix operations where operation has to be performed on each element simultaneously.

Algorithm 3.9: Pseudo code for GPU based 2-opt using TPRC

```
1  $cost \leftarrow initcost$
2  $minchange \leftarrow 0$
3  $i \leftarrow threadIdx.x + blockIdx.x * blockDim.x$
4  $j \leftarrow threadIdx.y + blockIdx.y * blockDim.y$
5  if $i < j$ && $j < n$ then
6      $change = f(i,j) + f(i+1,j+1) - f(i,i+1) - f(j,j+1)$
7      $cost = cost + change$
8      if $change < minchange$ then
9          $globDist = cost$
10         $globId = i \times (n-1) + (j-1) - \frac{i\times(i+1)}{2}$
11         $minchange = change$
12     end
13 end
```

3.2.5.4 Threads per Neighborhood (TPN)

In TPR and TPRED, each thread strategy generates multiple neighbors. In TPRC, $n^2 - 2 \times {}^{n}C_2$ threads are idle. A more efficient thread mapping strategy maps the generation of a single neighborhood solution and its cost computation to one thread. For this Threads per Neighbor strategy, each thread needs 2 cities to apply the 2-opt move. This is done using Eqn. 3.6 and 3.7 (Luong et al., 2013).

$$i = n - 2 - \left\lfloor \frac{\sqrt{8 \times (N(s) - id - 1) + 1} - 1}{2} \right\rfloor \tag{3.6}$$

$$j = id - i \times (n-1) + \frac{i \times (i+1)}{2} + 1 \tag{3.7}$$

$$blocks = \left\lceil \frac{\frac{n(n-1)}{2}}{1024} \right\rceil \tag{3.8}$$

Thread id	Pair	Thread id	Pair
0	(0, 1)	n-1	(1, 2)
1	(0, 2)	n	(1, 3)
...	...	...	...
n-2	(0, n-1)	${}^{n}c_2 - 1$	(n-2, n-1)

Table 3.5: Threads per Neighbor (TPN) mapping for n cities where n > 3. Each thread generates one neighbor. Each thread uses equations 3.6 and 3.7 to determine swapping cities.

Therefore, $\frac{n(n-1)}{2}$ threads are required to generate all neighbors of the initial solution. Thread blocks required for computation are calculated using Eqn. 3.8. Global id of each thread is used to generate swapping edges. For example, Thread 0 will deal with pair (0,1) cities, Thread 1 will deal with pair (0, 2) cities and so on. Table 3.5 shows mapping strategy of Eqn. 3.6 and 3.7 to the thread. Algorithm 3.10 presents

Algorithm 3.10: Pseudo code for GPU based 2-opt using TPN

```
$cost \leftarrow initcost$
$soln \leftarrow \frac{n \times (n-1)}{2}$
$minchange \leftarrow 0$
$globalIdx \leftarrow threadIdx.x + blockIdx.x * blockDim.x$
if $globalIdx < soln$ then
    $i = n - 2 - \left\lfloor \frac{\sqrt{8 \times (soln - globalIdx - 1) + 1} - 1}{2} \right\rfloor$
    $j = globalIdx - i \times (n-1) + \frac{i \times (i+1)}{2} + 1$
    $change = f(i,j) + f(i+1, j+1) - f(i, i+1) - f(j, j+1)$
    $cost = cost + change$
    if $change < minchange$ then
        $globDist = cost$
        $globId = globalIdx$
        $minchange = change$
    end
end
```

the neighborhood generation and its cost computation using the 2-opt move. Stepwise details of the algorithm follow. Lines (1-2): Cost of initial solution, number of possible solutions with n is given to each thread. Line (4): Unique id for each thread is generated. Lines (6-7): The linear to 2D index conversion formula is used to generate i, j index to apply 2-opt move. Lines (8-9): Each thread checks the cost of removing pair of edges $(i, i+1)$, $(j, j+1)$ and adding pair of edges (i, j), $(i+1, j+1)$ to choose the best feasible solution. Lines (10-14): The best improved cost, $globDist$ and its $globalIdx$, are stored in GPU's global memory. The best improved solution that is obtained by Algorithm 3.10, is checked at the CPU side to decide the next 2-opt call. If the best improved solution is better than the initial solution, then the best improved solution will act as a new initial solution for next 2-opt call.

In general, Eqn. 3.6 and 3.7 are useful when operations on the 2-opt move, upper triangular, lower triangular, and symmetric matrix have to be computed in parallel and each element has to be distributed per thread. The number of index pairs required in each of these matrix types is ${}^nC_2$ excluding diagonal elements. These equations allow ${}^nC_2$ different threads to function over each matrix block in parallel. For the lower triangular matrix operations, Eqn. 3.6 & 3.7 are used with i and j swapped. When operations have to be done on symmetric matrix index pairs, either upper or lower triangular matrix is considered. Performance evaluation of these thread mapping strategies has been demonstrated in Section 3.3.3.

3.2.5.5 Usage of built-in functions

In TPN strategy, $floorf$ and $sqrtf$ are used to write the corresponding CUDA code for 3.6 & 3.7. Listing 1 show the CUDA code of corresponding equations.

Listing 3.1: CUDA code for equations 3.6 & 3.7

```
i = n-2-floorf((sqrtf(8*(sol-id-1)+1)-1)/2);
j = id-i*(n-1)+(i*(i+1)/2)+1;
```

Maximum instance size considered in work (Luong et al. 2013) is up to $rl5934$. When instance size goes above $rl5934$, these equations do not work correctly. This issue occurs while rounding a numerical value in Eqn. 3.6 for $sqrtf$ function. To illustrate, consider the $d15112$ instance. The number of neighborhood solutions possible with

$d15112$ instance is 114178716. For global id 0, the expected values of i, j indexes are 0 and 1 respectively. The calculation of i is shown below, Eqn. 3.6 becomes:

$$\begin{aligned} i &= 15112 - 2 - \left\lfloor \frac{\sqrt{8 \times (114178716 - 0 - 1) + 1} - 1}{2} \right\rfloor \\ &= 15112 - 2 - \left\lfloor \frac{30223}{2} \right\rfloor \quad \text{sqrtf rounds a value to 30223} \\ &= 15112 - 2 - 15111 \\ i &= -1 \end{aligned}$$

Since $sqrtf$ rounds the value of its output to 30223, a wrong i is generated. To overcome the lack of precision, a double precision floating point $\_\_dsqrt\_rn$ function is used. The $\_\_dsqrt\_rn$ function generates a more precise result than $sqrtf$. The $\_\_dsqrt\_rn$ function returns a value 30222.999868 and the final result of i is the expected zero. There are four rounding modes available with $sqrt$ function in CUDA. These are namely ru (rounds up, which act as ceiling function), rd (rounds down, which act as floor function), rz (rounds to zero, which truncate fractional digits), and rn (rounds to nearest even). The rn mode gives the expected index of 2-opt move.

3.3 RESULT ANALYSIS

The proposed GPU based Parallel Iterative Hill Climbing algorithm has been evaluated over a GPU Tesla K40m using CUDA programming interface with CUDA 8 version. The GPU has compute capability 3.5, 15 streaming multiprocessors and each multiprocessor has 192 cores running at 745 MHz, global memory is 12 GB, the shared memory available with each block is 48 KB with 65536 registers available at each block. The time analysis of sequential counterpart has been carried out on a 64 bit system which has 8 cores running at 3.6 GHz. The optimal solutions/best-known solutions of TSPLIB instances are known, therefore the symmetric TSPLIB (Reinelt 1991) instances have been considered to determine the effectiveness of PIHC TSP solver in terms of speedup and cost.

This section presents the time and cost analysis of initial solution construction methods, time analysis of different thread mapping strategies, the optimized data structures, time and cost comparison with the state-of-the-art GPU-based TSP implementations and time analysis with state-of-the-art CPU-based TSP implementations.

3.3.1 Initial Solution Construction Result Analysis

Initial solution construction is a crucial step in the PIHC method. TPN thread mapping strategy has been used for this analysis. Termination criteria of each approach is to improve until further improvement is not possible. The performance analysis of different initial setup approaches have been divided into two categories. First, the performance analysis of different construction heuristic approaches have been compared. Second, the sequenced and random approaches have been compared with faster construction heuristic approach.

3.3.1.1 Performance Analysis of Construction Heuristic Approaches

Construction heuristic approaches are used to construct a feasible solution of a given problem. In this work, multiple construction heuristics have been evaluated for use with PIHC for constructing the initial solution, namely, Nearest Neighborhood (NN), Nearest Insertion (NI), Greedy algorithm, Minimum Spanning Tree (MST), Christofides' algorithm, and Clarke-Wright algorithm. Cost of the initial solution for different TSPLIB instances and time required to construct it using different construction heuristic approaches have been presented in Table 3.6. Table 3.7 shows the local optimal solution found using different construction heuristic approaches, number of steps required to reach it, and the error rate for each TSPLIB instance for each construction approaches.

The NN approach constructs the initial solution faster than NI, greedy, MST, and Christofides' algorithm on an average. The error rate present in the cost of constructed initial solution are (best case - worst case): 15% - 41.46%, 13.40% - 33.35%, 15.10% - 41.46%, 20.68% - 53.44%, and 27.81% - 44.64% for NN, NI, greedy, MST, Christofides' algorithm respectively for instances ranging from 105 to 15112 cities. The error rate of initial solution is calculated using Eqn 3.9.

$$error = \frac{initial - optimal}{optimal} * 100 \tag{3.9}$$

Table 3.6: Time analysis of constructing the initial solution using different construction approaches- Nearest Neighborhood (NN), Nearest Insertion (NI), greedy, Minimum Spanning Tree (MST), Christofides' algorithm, and Clarke-Wright algorithm. Time is given in seconds.

Instance	NN		NI		Greedy		MST		Christofides' algorithm		Clarke-Wright	
	Initial	Time	Initial	Time	Initial	Time	Initial	Time	Initial	Time	Initial	Time
lin105	20341	0.000171	18851	0.001	20341	0.000360	20149	0.001	19588	0.01	40250	0.01
d198	18162	0.000294	17895	0.02	18163	0.000820	19043	0.01	20168	0.02	35403	0.17
kroA200	35715	0.000303	36049	0.02	40408	0.000906	39914	0.01	41389	0.02	190582	1.09
lin318	53959	0.000589	52082	0.12	52897	0.002021	59070	0.04	57069	0.07	273053	1.31
pcb442	61926	0.001143	60377	0.24	60824	0.002625	69676	0.14	67801	0.24	510445	19.15
d493	43244	0.001425	40702	0.47	43245	0.002632	44844	0.18	47152	0.34	309537	18.5
rat575	8431	0.002240	8111	0.61	8446	0.003480	9274	0.28	9130	0.5	46007	13.75
d657	61289	0.002357	59529	0.99	61290	0.004494	65515	0.47	65442	0.67	466672	77.84
rat783	10867	0.003186	10571	1.69	10702	0.006736	11983	0.71	12030	1.14	74316	69.92
vm1084	294693	0.006377	281212	5.38	307928	0.014880	313074	1.89	323240	3.41	3869840	1534.03
d1291	60078	0.008352	67744	8.45	60078	0.016660	77950	2.92	73389	5.09	1088341	935.46
rl1304	339621	0.01	323488	8.64	309775	0.018541	351229	3.3	365864	5.81	5047379	2762.28
u1432	188751	0.03	174811	11.39	193011	0.021006	222403	3.82	207321	6.83	2131644	2796.05
pcb3038	172791	0.11	169452	105.57	170990	0.1	197071	37.51	189454	66.23	-	-
fnl4461	223855	0.21	221679	427.6	225189	0.3	253688	161.25	251345	285.4	-	-
rl5934	679591	0.29	704694	906.41	682932	0.6	846993	347.09	785830	592.87	-	-
pla7397	28106018	0.31	28267400	1493.69	28371598	0.7	34314380	541.54	33115986	930.77	-	-
rl11849	1116175	1.16	1158293	8604.26	1129476	2.4	1365720	3919.83	1322008	5843.62	-	-
usa13509	24738055	1.47	25191616	8086.59	24924492	2.9	27955752	5075.35	27916358	7998.26	-	-
d15112	1942334	1.79	1918046	12195.86	1951689	3.8	2182334	8151.37	2172164	12006.22	-	-
Average		0.27		1592.90		0.54		912.39		1387.38		633.04

Table 3.7: Analysis of the local optimal solution reaches with different construction heuristic approaches.

Instance	NN			NI			Greedy			MST			Christofides' algorithm			Clarke-Wright		
	Final	Steps	Error	Final	Steps	Error	Final	Steps	Error	Final	Steps	Error	Final	Steps	Error	Final	Steps	Error
lin105	14691	24	2.17	14815	31	3.03	14691	24	2.17	15442	21	7.39	15659	19	8.90	14745	72	2.55
d198	16005	32	1.43	16852	33	6.79	16006	32	1.43	16196	37	2.64	16036	43	1.62	16525	123	4.72
kroA200	29579	36	0.72	34063	26	15.99	30068	41	2.38	31602	46	7.61	30521	50	3.93	31214	154	6.29
lin318	44278	60	5.35	46594	48	10.86	43391	59	3.24	44376	80	5.58	44551	74	6.00	44757	272	6.49
pcb442	53253	44	4.87	55481	53	9.26	53754	47	5.86	52431	73	3.26	53867	69	6.08	55710	370	9.71
d493	36101	87	3.14	37648	86	7.56	36102	87	3.14	37145	107	6.12	36638	126	4.67	38985	449	11.38
rat575	6933	87	2.36	7343	86	8.42	6995	87	3.28	6937	125	2.42	6961	121	2.78	7399	478	9.24
d657	51920	105	6.15	55616	100	13.71	51921	105	6.15	51388	136	5.06	51024	138	4.32	53914	586	10.23
rat783	9002	124	2.23	9683	106	9.96	9043	113	2.69	9029	168	2.53	9045	172	2.71	9584	691	8.83
vm1084	250998	133	4.89	260963	110	9.05	249591	159	4.30	249558	161	4.29	252275	179	5.42	260924	1057	9.04
d1291	52818	99	3.97	56791	166	11.79	52818	99	3.97	53908	143	6.12	53902	150	6.10	57595	1316	13.37
rl1304	273332	135	8.06	280575	165	10.92	273140	118	7.98	261479	155	3.37	270027	149	6.75	284194	1426	12.35
u1432	163278	172	6.74	166969	83	9.15	162921	174	6.51	160838	259	5.14	161553	222	5.61	169134	1289	10.57
pcb3038	145190	418	5.44	151517	417	10.04	145539	416	5.70	145582	604	5.73	146961	600	6.73	-	-	-
fnl4461	189881	619	4.01	204305	608	11.91	190353	613	4.27	191225	1000	4.74	191701	945	5.00	-	-	-
rl5934	581802	468	4.63	612927	608	10.23	586877	438	5.54	584859	583	5.18	591970	580	6.46	-	-	-
pla7397	24477726	565	5.23	25885374	680	11.28	24433189	565	5.04	24580733	910	5.67	24816385	947	6.69	-	-	-
rl11849	966496	1047	4.68	1019768	1313	10.45	971585	1057	5.23	969391	1668	4.99	969046	1603	4.96	-	-	-
usa13509	21144396	2016	5.81	22670503	2482	13.45	21118178	1971	5.68	21314623	2916	6.66	21274336	2916	6.46	-	-	-
d15112	1650340	2040	4.91	1772534	2200	12.68	1651190	2019	4.97	1664098	3418	5.79	1672071	3303	6.29	-	-	-
Average			4.34			10.33			4.48			5.01			5.37			8.83

The time requirement for constructing the initial solution, NN is up to 6813.33×, 6707.39×, 4553.84×, 2.10× faster compared to the NI, Christofides' algorithm, MST, and greedy algorithm respectively. This is because, the constraints involved in constructing the initial solution makes other approaches slower.

In the NI approach, generation of the initial solution starts with a subtour. There are two conditions for adding an unvisited node to a subtour. First, an unvisited node must be minimum to any one node of a subtour. Second, an edge i, j has to choose from a subtour such that the cost of adding unvisited node between i and j must be minimized. For first condition, an algorithm spends $O(nv)$ time for finding the unvisited closest nodes, where, v is total nodes present in the subtour. For the second condition, it spends $O(e)$ time to choose an edge where a closest unvisited node can be added, where, e, is total edges in the subtour. Though NI generates a better cost initial solution (error rates: 13.40% - 33.35%), these constrains make NI a slower approach.

Generation of the initial solution using the MST approach involves three major steps. First, a minimum spanning tree has to be generated. For constructing MST, Prim's algorithm is used. Second, an Eulerian cycle is generated which spends $O(n)$ time. Third, an Eulerian cycle is converted into a Hamiltonian cycle that spends $O(n)$ time additionally. Overall, constructing a initial solution using the MST spends the larger time than NN and greedy algorithm.

Christofides' algorithm is a variant of the MST approach. Christofides' algorithm uses the minimum weight perfect matching for odd degree nodes to improve the cost quality further. There are two additional overheads in Christofides' algorithm in addition to the MST approach. First, after MST is generated, an odd degree set of nodes to be determined. Second, edges are added to the MST using the minimum perfect weight matching. These constraints make Christofides' algorithm slower. Note that for finding the minimum perfect weight matching edges, the blossom V algorithm (Kolmogorov 2009) is used and its tool is available at link: `http://pub.ist.ac.at/~vnk/software.html`.

Clarke-Wright algorithm has been evaluated on the small size of TSPLIB instances

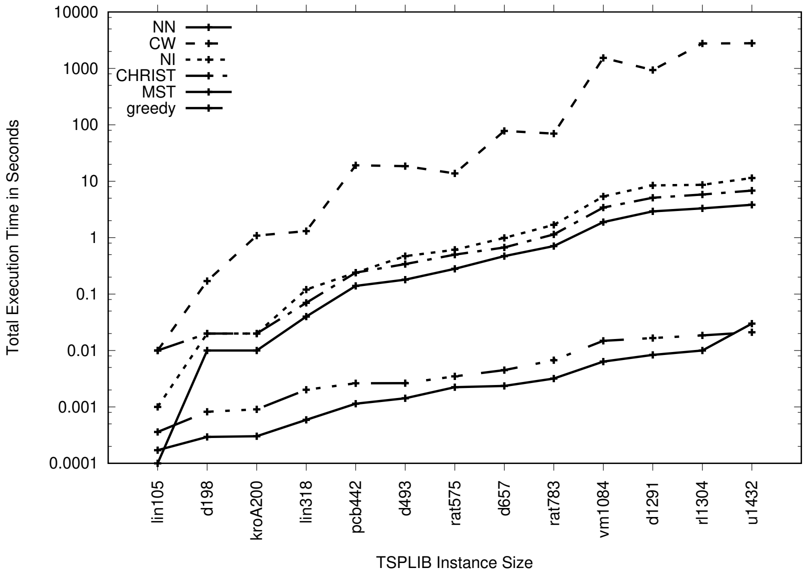


Figure 3.2: Time analysis of the initial solution construction approaches.

ranging from 105 - 1432. It is observed that Clarke-wright algorithm spends significantly larger time to construct the initial solution than other approaches. The error rates in the constructed solution are also larger, i.e., in the range of 124.35% - 2042.36%, which is worse than other heuristic approaches. The time requirement of constructing the initial solution of Clarke-wright algorithm is compared with NN, NI, greedy, MST, and Christofides' approaches and it is shown in Figure 3.2. The most time consuming part of Clarke-Wright algorithm is calculating the savings for ${}^{n}C_2$ node pairs and arranging these node pairs in the descending order of their savings.

The Greedy algorithm constructs the initial solution using the Prim's algorithm. However, in TSP, each city is visited exactly once, one node cannot have more than two neighbors in the feasible solution. Therefore, constructing the feasible solution using Prims' algorithm is similar to the NN approach except one difference. In NN, a closest unvisited node of the recently visited node is chosen, whereas, in greedy approach, a closest node is chosen for either a recently visited node or first node of the constructing tour. If closest unvisited node is neighbor to the first node of the tour, it is added before the first node otherwise it is added at the last. Therefore the performance of greedy algorithm is similar to the NN approach. Finding a closest unvisited neighbor from two places, add time overhead slightly in the computation. Therefore, NN is up to 2.10×

Table 3.8: Analysis of execution time spent in the solution construction phase and improvement phase separately. Abbreviation used- C%: Percentage of time spent in the solution construction out of total execution time, I %: Percentage of time spent in the solution improvement out of total execution time.

Instance	NN		NI		Greedy		MST		Christofides' Algorithm	
	C %	I %	C %	I %	C %	I %	C %	I %	C %	I %
lin105	0.05	99.95	0.31	99.69	0.02	99.98	0.26	99.74	2.50	97.50
d198	0.09	99.91	6.67	93.33	0.08	99.92	2.94	97.06	5.41	94.59
kroA200	0.09	99.91	6.67	93.33	0.11	99.89	2.86	97.14	5.41	94.59
lin318	0.17	99.83	27.27	72.73	0.20	99.80	10.81	89.19	15.91	84.09
pcb442	0.32	99.68	44.44	55.56	0.28	99.72	29.17	70.83	40.68	59.32
d493	0.39	99.61	53.41	46.59	0.27	99.73	33.33	66.67	47.89	52.11
rat575	0.60	99.40	65.59	34.41	0.34	99.66	45.16	54.84	57.47	42.53
d657	0.65	99.35	75.00	25.00	0.45	99.55	58.02	41.98	65.05	34.95
rat783	0.85	99.15	81.64	18.36	0.65	99.35	64.55	35.45	75.00	25.00
vm1084	1.74	98.26	94.89	5.11	1.40	98.60	82.89	17.11	89.97	10.03
d1291	2.21	97.79	95.80	4.20	1.58	98.42	88.22	11.78	93.05	6.95
rl1304	2.70	97.30	95.47	4.53	1.75	98.25	89.92	10.08	93.56	6.44
u1432	7.14	92.86	97.35	2.65	2.08	97.92	90.09	9.91	94.08	5.92
pcb3038	12.64	87.36	99.27	0.73	6.45	93.55	97.48	2.52	98.53	1.47
fnl4461	11.23	88.77	99.66	0.34	11.45	88.55	98.48	1.52	99.17	0.83
rl5934	12.13	87.87	99.75	0.25	18.63	81.37	99.27	0.73	99.57	0.43
pla7397	7.85	92.15	99.71	0.29	13.89	86.11	98.96	1.04	99.36	0.64
rl11849	6.80	93.20	99.86	0.14	12.59	87.41	99.36	0.64	99.59	0.41
usa13509	3.62	96.38	99.41	0.59	6.92	93.08	98.90	1.10	99.30	0.70
d15112	3.50	96.50	99.56	0.44	7.12	92.88	99.00	1.00	99.34	0.66
Average	3.74	96.26	72.09	27.91	4.31	95.69	64.48	35.52	69.04	30.96

faster over the greedy approach.

These constructed initial solutions using the different approaches have been further evaluated to identify the error rates in their local optimal solutions, which is presented in Table 3.7. Table shows the local optimal solution, total steps required to reach it, and the error rate present in it for each construction heuristic approach on TSPLIB instances. The error rates in the final cost for NN, NI, greedy, MST, and Christofides' approaches are (best case - worst case): 0.72% - 8.06%, 3.03% - 15.99%, 1.43% - 7.98%, 2.42% - 7.61%, and 1.62% - 8.90% respectively.

Table 3.8 shows the separate portion of execution time spent in both construction and improvement phases when different construction heuristic approaches are used to solve TSPLIB instances. NN spends 3.74% time in the solution construction and

96.26% time in the solution improvement phase. From Table 3.7, it is observed that NN produces better final solutions than remaining five approaches, therefore NN can be made a more time efficient approach by reducing the time spent in the solution improvement phase. It is observed that NN spends 96.26% on an average in the solution improvement phase. With the help of parallel models to improvement phase, the time portion of solution improvement can be reduced further.

From Figure 3.2 and Table 3.7, it is observed that NN and greedy approaches are faster and produces a good quality solutions. Although the MST and Christofides' algorithm construct the good solutions, they are slower than the NN and greedy approaches. NI constructs the better initial solution but result in a slower approach than the NN, greedy, MST and Christofides' approaches. Clarke-Wright algorithm are both the slower and worse in constructing the initial solution. Since NN is faster and produces a good cost, NN is used it in the rest of the result analysis as the initial solution construction approach.

3.3.1.2 Time Analysis of NN with Sequenced and Random Approaches

Figure 3.3 presents the total execution time of the NN, sequenced, random implementations. Each input instance is executed once for the three initial solution construction approaches. Results for the TSPLIB instances from 200 to 18512 cities are shown. For moderate size of TSP instances up to $u2319$, all three mechanisms spend similar time to find the near-optimal solution. As the input size increases above the $u2319$ instance, the total execution time of both the sequenced and random approaches increases significantly.

For the $d18512$ instance, the total execution time required using sequenced, random, NN are 873.67 s, 1258.91 s, and 76.60 s respectively. The NN as initial solution construction strategy makes PIHC GPU implementation 11.40× faster compared to the GPU implementation of sequenced, 16.43× faster compared to the random approach. This happens because NN gives a good quality solution at the initial phase. PIHC needs lesser number of steps to reach the local optimal solution compared to the sequenced and random approaches. For the $d18512$ instance, NN needs 8.5×, 11.19× fewer steps

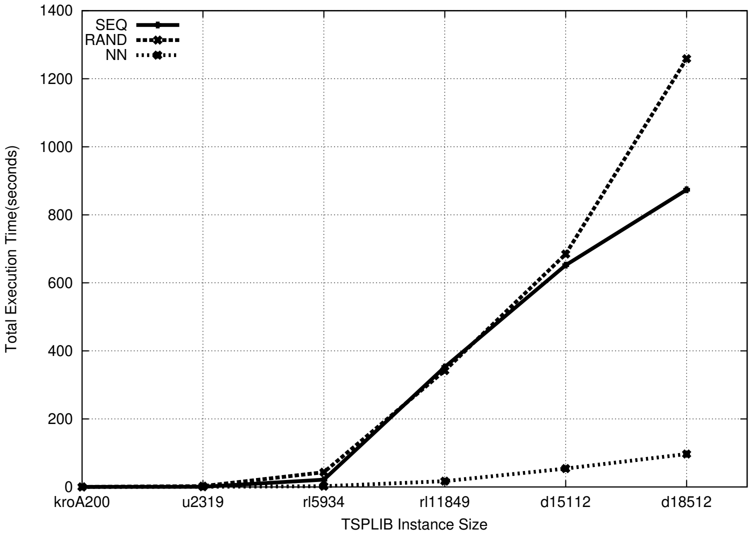


Figure 3.3: Total PIHC execution time of sequenced, random, NN initial solution construction mechanism for TSPLIB instances

compared to the sequenced and random approaches respectively. It is observed that the NN approach tends to yield a shorter initial tour, which is why it requires fewer 2-opt steps on average before reaching the local optimal, thus making PIHC tool faster than other tools, even if the tool uses the same 2-opt move implementation.

The time complexity of 2-opt move is O($k \times n^2$), where, k is the number of steps required to reach the local optimal solution. Since random and sequenced approach requires larger steps to reach their local optimal solutions that result in larger execution time requirement. For example, consider an instance $d1512$ for which sequenced, random, and NN need 21762, 22189, and 2040 steps (i.e., k) respectively. NN spends lesser time i.e., 42.79 seconds, sequenced approach which is the second best lesser steps required approach, spends 652.03 seconds, and the random approach spends 684.63 seconds which needs a higher steps than the NN and sequenced approaches respectively. Therefore, it makes sense to compute NN rather than using this time to run more 2-opt steps.

Figure 3.4 shows the throughput analysis of NN, sequenced, and random initial solution construction approaches. Throughput is the ratio between the total neighborhood solutions explored to reach a local optimal solution and the total execution time. NN has lesser throughput compared to other two approaches until the $u2319$ in-

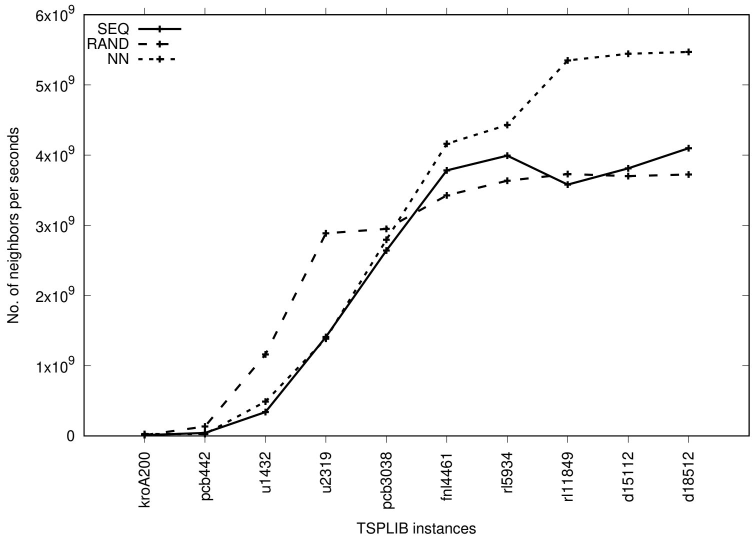


Figure 3.4: Throughput analysis of sequenced, random, NN initial solution construction mechanism for TSPLIB instances

stance. This is because, as far as total execution time is concerned, all three approaches spend similar time to reach their local optimal solution. When total steps required to reach their local optimal is concerned, the random and sequenced approaches require larger steps than NN. Eventually, the random and sequenced approaches result in higher throughput compared to NN for instances up to $u2319$ cities. For inputs $pcb3038$ and larger, the throughput of NN improves significantly than the random and sequenced approaches because NN needs fewer steps to reach its local optimal in lesser time. Random and sequenced approaches need larger steps to reach their local optimal solution and spend significantly more time than NN. This results in throughput improvement for NN while solving larger instances. For example, at the $d15112$ instance, sequenced, random, and NN require 21762, 22189, 2040 steps and 652.03, 684.63, 42.79 seconds respectively to reach their local optimal. Throughput of sequenced, random, and NN becomes 3810801983.95 (i.e., $\frac{21762}{652.03} *^{15112} C_2$), 3700555817.48 (i.e., $\frac{22189}{684.63} *^{15112} C_2$), and 5443434929.66 (i.e., $\frac{2040}{42.79} *^{15112} C_2$) respectively. Note that throughput calculation shows the number of neighborhood solutions calculated per second.

Table 3.9: Cost analysis of different initial solution construction techniques- sequenced, random, NN. Each technique has initial solution cost, locally reached final cost on the initial solution, total steps needed to reach the final solution and its error rate for TSPLIB instances.

Instance	Sequenced				Random				NN				Optimal
	Initial	Final	Steps	Error rate	Initial	Final	Steps	Error rate	Initial	Final	Steps	Error rate	
kroA200	373855	31151	234	6.07	323888	32599	218	11.0	35715	29579	36	0.72	29368
pcb442	221400	56624	192	11.51	778441	56522	511	11.31	61926	53253	44	4.87	50778
u1432	182991	169550	153	10.83	3947120	173259	1710	13.26	188751	163278	172	6.74	152970
u2319	281464	255304	299	8.98	6048275	251815	2717	7.49	278721	241566	232	3.12	234256
pcb3038	295345	148152	687	7.59	5423394	155367	4091	12.83	172791	145190	418	5.44	137694
fnl4461	5872300	202786	4866	11.07	8374159	204207	6076	11.85	223855	189881	619	4.0	182566
rl5934	9861054	607438	4852	9.24	41984032	635353	8947	14.26	679591	581802	468	4.63	556045
rl11849	86620224	1070098	17988	15.90	86899720	1049401	18186	13.65	1116175	966496	1047	4.68	923288
d15112	112310520	1752472	21762	11.40	134689056	1746739	22189	11.03	1942334	1650340	2040	4.91	1573084
d18512	29462514	724178	20899	12.23	59230728	720956	27363	11.73	785991	671000	2445	3.99	645238
Average				10.48				11.84				4.31	

3.3.1.3 Cost Analysis of NN with Sequenced and Random Approaches

The cost quality achieved using the NN, sequenced and random initial solution construction approaches are given in Table 3.9. This cost quality is evaluated with a single run of each input instance with each construction mechanism. The starting node of each construction technique is fixed, therefore it is run once. The error rate is calculated using Eqn. 3.10, where $cost_{final}$ is the cost obtained using the construction approach on TSPLIB instance and $cost_{opt}$ is the optimal cost of the TSPLIB instance.

$$Error\ rate = \frac{cost_{final} - cost_{opt}}{cost_{opt}} \times 100 \tag{3.10}$$

Choosing the NN as an initial solution construction mechanism saves search-space exploration time and result in a better cost quality solution. The worst-case cost of the initial solution using NN is within 1.08× optimal cost, whereas, for sequenced and random are 93.82× optimal cost, 94.12× optimal cost respectively. In the best-case, NN constructs the initial solution which cost is within 1.01× optimal cost (i.e., for kroA200 instance). In terms of cost quality, the error rate of NN ranges from 0.72% (best-case) and 6.73% (worst-case). The best quality solution found with sequenced, random approach have 6.02%, 7.49% error rate respectively whereas worst case cost quality have 15.90%, 14.26% error rate respectively.

The nearest neighborhood initial solution construction mechanism takes care to choose the minimal weighted neighboring city while constructing the initial solution, unlike sequenced and random mechanism. From the presented time and cost analysis, NN construction mechanism creates an initial solution which is closer to the optimal solution than the random and sequenced approaches for 200 - 18512 instances.

3.3.2 Optimized Data for TSP

In this section, the effect of three data structures, DM, TDM, COORD on the execution time of the PIHC algorithm have been evaluated. NN is used for the initial solution construction for this experiment. Figure 3.5 presents the total execution time of the PIHC algorithm using Distance Matrix (DM), Triangular Distance Matrix (TDM), X, Y coordinate (COORD) as data structures. TSPLIB input instances of size ranging from 3038 - 33810 cities is used for this analysis.

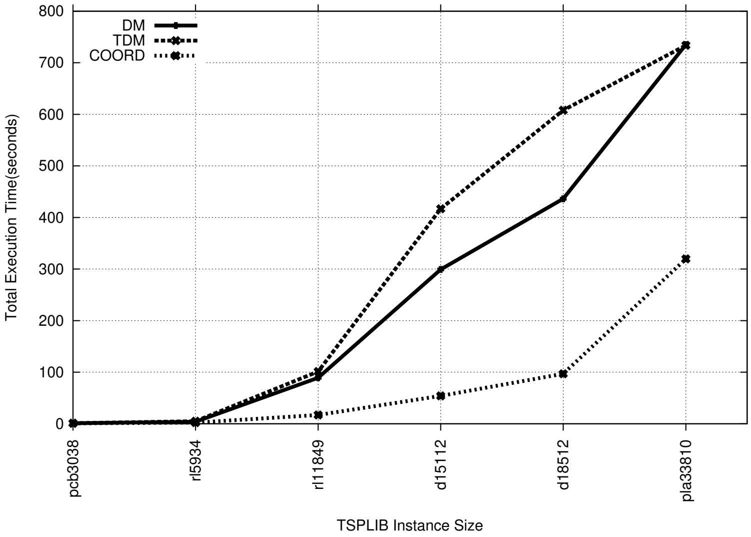


Figure 3.5: Total execution time of the PIHC algorithm using Distance Matrix(DM), Triangular Distance Matrix(DM), X,Y coordinate(COORD) as data structures

From Figure 3.5, for moderate input size instances up to rl5934, the data structures do not affect the final execution time significantly. When input size increases, the TDM implementation consumes more time than the DM and COORD. TDM reserves lesser memory than DM and needs more execution time. TDM stores the distances of the cities in the upper triangular matrix form which is converted into a 1D array to be sent to the GPU. In the GPU, the distance between any two cities i and j is accessed using an index of TDM matrix and the index is calculated using the following equation:

$$index = i \times (n-1) + j - 1 - \frac{i \times (i+1)}{2} \tag{3.11}$$

From Eqn. 3.11, the distance calculation involves 3 subtractions, 2 multiplications, 2 additions and 1 division operations. This results in an overall increase in the computational time to access one element from the TDM. Accessing elements from the matrix makes the TDM approach more time consuming than DM and COORD. In the case of DM, the entire distance matrix is sent over the GPU resulting in large data transfer delay.

For instances larger than 33810, the DM and TDM approaches were infeasible on the Tesla K40m GPU. With pla85900 instance, both DM and TDM approach cannot be solved on Tesla K40m since it needs 27.48 GB and 13.74 GB memory respectively. The K40m has a 12 GB global memory. From these experiments, it is concluded that

COORD approach outperforms DM and TDM approaches in the memory space consumption and execution time.

3.3.3 Performance Analysis of Thread Mapping Strategies

Section 3.2.5 introduces the TPR, TPRED, TPRC, and TPN thread mapping strategies applied in this work. Time analysis of the thread mapping strategies are reported in Table 3.10. Note that changing thread mapping strategy does not affect the local optimal solution of the input instances. Therefore, the time analysis of these thread mapping strategies has been presented in Table 3.10 for TSPLIB instances ranging from 198 to 85900 size. Average total execution time of each instance has been collected after ten trials for each mapping strategy.

Up to $rl11849$, TPRC performs better than other three mapping strategies. This is because, a pair of threads has been mapped to every neighbor generation and its cost computation. In TPR, the work of neighborhood generation and its cost calculation are distributed unequally. For n size input, first thread has to deal with n-1 neighbors computation, second thread has to deal with n-2 neighbors computation, and so on. In short, each thread works in parallel and at each thread, the associated neighbor computations are done in serial fashion. TPRED is a variant of TPR where the work of neighborhood generations are equally distributed among the threads. However, each thread has to deal with more than one neighbors computation and hence, spend more time compared to the TPRC strategy. Therefore, TPRED performs better than TPR. In TPN, one neighbor generation and its computation is mapped per thread. Since the computations of ${}^{n}C_2$ neighbors of the 2-opt move are performed by ${}^{n}C_2$ unique threads simultaneously, TPN perform better than the TPR, and TPRED approaches up to $rl11849$ instance. But compared to TPRC, TPN is slightly worse. This is because, TPRC uses two threads per neighborhood computation, whereas, TPN uses one thread per neighborhood computation. In addition to this, TPN has additional arithmetic computation overhead, i.e., each thread has to perform 18 arithmetic operations (Eqn. 6 and 7 involve 18 arithmetic operations) to get the ids of cities to be swapped. Hence, TPRC outperforms than TPN.

Table 3.10: Time analysis of thread mapping strategies- Threads per Row (TPR), Threads per Row equal Distribution (TPRED), Threads per Row and Column (TPRC), and Threads per Neighbor (TPN). Time is given in seconds which is average of ten trials.

Instance	TPR	TPRED	TPRC	TPN
d198	0.33	0.33	0.32	0.32
kroA200	0.33	0.32	0.32	0.32
lin318	0.35	0.34	0.32	0.32
pcb442	0.35	0.34	0.32	0.32
d493	0.38	0.37	0.32	0.33
rat575	0.41	0.38	0.33	0.32
d657	0.42	0.39	0.33	0.33
rat783	0.48	0.44	0.34	0.34
vm1084	0.55	0.50	0.35	0.34
d1291	0.51	0.48	0.33	0.33
rl1304	0.59	0.54	0.34	0.35
u1432	0.71	0.62	0.36	0.36
u2319	1.17	0.96	0.33	0.45
pcb3038	2.34	1.83	0.69	0.69
fnl4461	4.72	3.59	1.45	1.48
rl5934	4.75	3.64	1.83	1.86
pla7397	7.00	5.31	3.30	3.29
rl11849	20.20	15.10	13.36	13.74
usa13509	44.02	32.77	33.23	33.98
d15112	49.82	37.04	41.52	42.79
d18512	90.60	103.86	73.82	76.60
sw24978	177.20	205.28	200.83	210.21
pla33810	192.28	279.18	240.13	250.32
pla85900	2703.48	3393.34	3818.19	4323.17
Average	137.62	170.29	184.69	206.77

However, on the other side, it is observed that initiating the large number of threads result in slowing down the computation. This is because, when large numbers of threads involve in the computation, the more complex synchronization is required to get the accurate results. Another reason for slowing down the GPU computation is the thread control divergence. In CUDA, control divergence happens when some threads of a warp execute one set of instructions and other remaining threads execute another set of instructions which result in control divergence in the corresponding warp. When large number of threads are involved in computation, there exist a large number of warp, where, warp is a group of 32 threads. From Table 3.10, it is observed that though TPN and TPRC performs better than the TPR and TPRED until $rl11849$ instance, later it start slowing down the computation due to the synchronization and control divergence issues. Instances above $d15112$ cities, TPR mapping performs better than the other three. In TPRED, thread control divergence increases while choosing i, j pair. The indexes (i, j) are chosen such that resulting neighbor has the least cost. The (i, j) are used in Eqn. 3.11. If i is greater than j, values of i, j have to be shuffled in Eqn. 3.11 to extract the correct indexes at the host. In TPRC and TPN, $atomicMin$ function has to choose a minimal cost neighbor from $2 \times^n C_2$, $^nC_2$ threads respectively whereas in TPR and TPRED, a minimal cost neighbor is chosen from n threads. For $d15112$, in TPRC and TPN the $atomicMin$ function chooses a minimum cost neighbor from 228357432, 114178716 threads respectively. The function call overhead in TPR, TPRED is lesser as the minimum cost neighbor is chosen among 15112 threads.

3.3.4 Performance Analysis with GPU based state-of-the-art TSP Solvers

In this section, PIHC implementation has been compared with two well-known state-of-the-art GPU-based TSP solvers namely TSP2.2 (Neil and Burtscher, 2015) and LOGO (Rocki and Suda, 2013). TSPLIB (Reinelt, 1991) instances ranging from 198 to 85900 cities have been considered whose optimal costs are known in the TSPLIB testbed. The cost quality of LOGO TSP is reported in (Rocki and Suda, 2013). To obtain the cost quality of TSP2.2 (`http://cs.txstate.edu/~burtscher/research/TSP_GPU/index.html`) solver, each instance has been run ten times for each restart value using the flags *-O3 -arch=sm_35 -use_fast_math* and its cost and time are recorded. Seven different restart values have been considered to observe the change in the cost and execution time requirement. Restart value indicates the number of different initial solutions are considered while solving TSPLIB instances. Each restart gets its own local cost. Eventually, the minimum out of all restarts' local solutions is chosen and returned as the final cost. Table 3.11 shows the detailed performance analysis of TSP2.2 for different TSPLIB instances with different restarts.

3.3.4.1 Cost Analysis

Table 3.11 shows that the cost quality of TSP2.2 improves when restarts increase. The cost quality have been analyzed using multiple restarts: 10, 50, 100, 200, 300, 400, and 500. When larger restarts are considered, TSP2.2 slightly improves the final cost for small size TSPLIB instances but not better than PIHC. Figure 3.6 shows that the error rate comparison of TSP2.2 and proposed PIHC where error rate of the TSP2.2 has been calculated using the average final cost of 400 and 500 restarts. TSP2.2 produces better solution than PIHC for instances $lin318$ and $rl1304$. For these instances, error rates of the PIHC are 5.33% and 8.06% and the TSP2.2 with 500 restarts are 4.30% and 7.21% respectively. However for instances above $rl1304$, TSP2.2 produces large error rate. At $d15112$, error rates of the PIHC and TSP2.2 with 500 restarts are 4.91% and 10.52% respectively. For large size instances (i.e., above $pla7397$), despite large restarts such as 300, 400, and 500 which explores huge amount of search-space and spends more time, the final cost does not improve significantly compared to 10, 50, and 100 restarts. Moreover, while solving large instance, TSP2.2 produces large gap rate in the final cost compared to both LOGO and PIHC approaches.

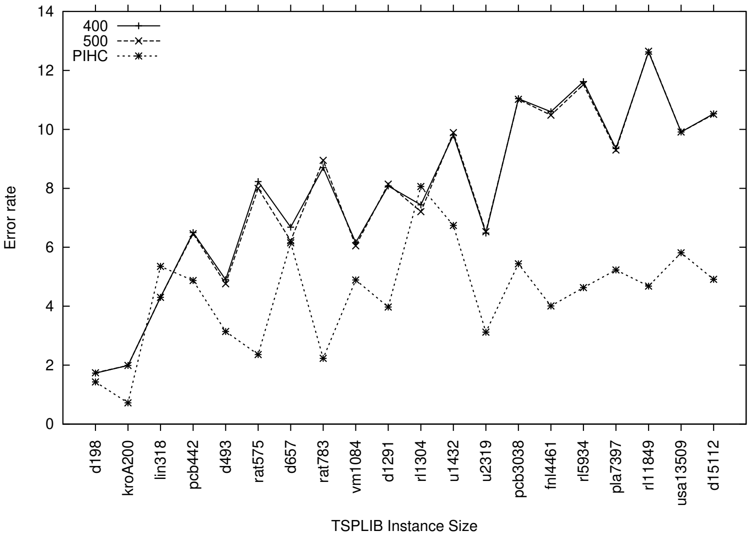


Figure 3.6: Analysis of error rate in the final cost of the PIHC and TSP2.2 with 400,500 restarts.

Table 3.12 presents the cost quality analysis of the LOGO and TSP2.2 with the PIHC algorithm. The PIHC algorithm gives the best quality solutions for TSPLIB input instance compared to the LOGO and TSP2.2. The error ranges (best case - worst case) for PIHC, LOGO, TSP2.2 are : 0.72% - 5.44%, 4.71% - 11.66%, 1.99% - 11.79% respectively. For TSP2.2, the error rate has been calculated using the average final cost of the best performing restarts. The TSP2.2 implementation for $pla85900$ instance did not finish.

The primary reason for the better quality results of the PIHC compared to the LOGO and TSP2.2 is the nearest neighborhood initial solution construction technique. NN provides an initial solution whose cost is within 0.5 $\times(log_2 n + 0.5)\times$ optimal cost for each input instance. Starting from such an initial solution, the search space for the hill climbing algorithm is reduced and a higher quality solution is obtained within a reasonable execution time. The TSP2.2 uses random tour to set the initial solution whereas LOGO uses Multiple Fragment (Bentley, 1990) heuristic.

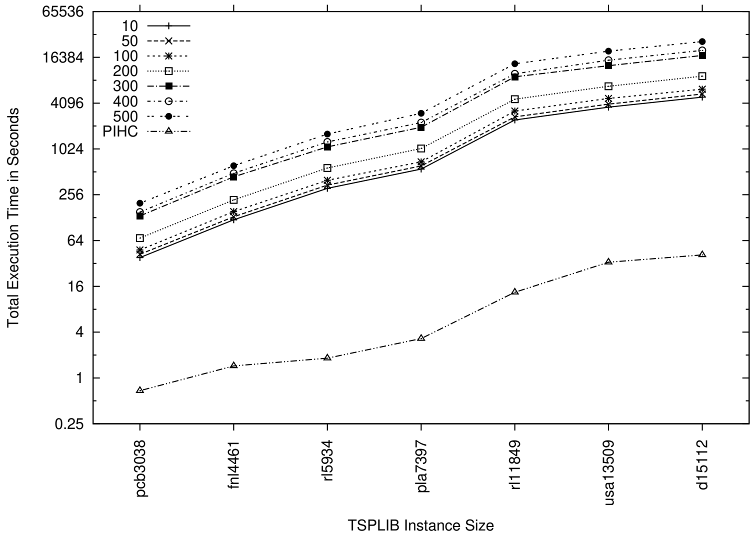


Figure 3.7: Time analysis of the PIHC with different restarts of TSP2.2 TSP solver.

3.3.4.2 Time Analysis

Since TSP2.2 source code is available, it was run on the same GPU as the PIHC and the execution times are recorded. The full execution time from the start to the end of the respective implementations (includes the CPU and GPU time) have been presented. Table 3.11 shows the execution time of each TSPLIB instances with seven different restart values. As restart value increases TSP2.2 produces slightly better solutions but consumes

Table 3.11: Cost and time analysis of TSP2.2 solver for TSPLIB instances with different restart values.

Instance	Restarts	Time (seconds)			Cost			Optimal	Error(%)
		min	avg	max	min	avg	max		
d198	10	0.02	0.02	0.02	16329	16329	16329	15780	3.48
	50	0.02	0.02	0.02	16111	16120.2	16157		2.16
	100	0.02	0.02	0.02	16087	16087	16087		1.95
	200	0.02	0.02	0.02	16037	16077.5	16087		1.89
	300	0.04	0.04	0.04	16087	16087	16087		1.95
	400	0.05	0.05	0.05	16037	16053.1	16057		1.73
	500	0.07	0.07	0.07	16037	16055	16057		1.74
u2319	10	14.86	14.88	14.89	250744	250744	250744	234256	7.04
	50	16.31	16.57	16.76	249506	250313.9	250977		6.85
	100	18.26	18.30	18.35	249349	249875.6	250389		6.67
	200	26.55	26.63	26.69	248787	249864.2	250260		6.66
	300	49.11	51.32	54.44	249374	249636.1	249921		6.57
	400	58.23	58.55	59.68	248826	249442.3	249978		6.48
	500	74.42	75.08	75.75	249098	249548.7	249850		6.53
fnl4461	10	119.88	119.95	120.03	203508	203508	203508	182566	11.47
	50	130.64	131.27	131.98	201370	202309.4	202703		10.81
	100	153.39	153.64	153.95	201858	202349	202761		10.84
	200	218.96	219.37	219.93	201683	201987.9	202300		10.64
	300	428.59	440.91	464.96	201238	201854.5	202372		10.57
	400	482.47	490.55	504.00	201559	201913.2	202170		10.60
	500	603.78	615.78	631.01	201084	201707.2	202223		10.48
rl5934	10	310.36	310.49	310.67	629665	629665	629665	556045	13.24
	50	340.72	342.47	344.80	621004	624782.5	627613		12.36
	100	395.59	397.09	398.46	620133	623253.2	625428		12.09
	200	574.50	575.81	577.57	621165	622530.7	624491		11.96
	300	1054.28	1090.93	1116.61	618505	621037.1	622983		11.69
	400	1247.36	1265.19	1296.95	618474	620639.5	622128		11.62
	500	1572.02	1608.14	1649.50	616026	620101	622030		11.52
rl11849	10	2459.28	2462.92	2467.60	1049048	1049048	1049048	923288	13.62
	50	2695.73	2703.89	2710.16	1035667	1041906.6	1045682		12.85
	100	3216.63	3229.43	3239.21	1038077	1041043.2	1044384		12.75
	200	4607.25	4616.82	4627.37	1034308	1039835.3	1043415		12.62
	300	8768.20	9090.31	9285.22	1037953	1039315.875	1040871		12.57
	400	9004.00	9948.72	10419.49	1036804	1039941	1041716		12.63
	500	12526.62	13467.54	14336.32	1037877	1040104.9	1042506		12.65
d15112	10	4911.54	4914.43	4919.89	1742811	1742811	1742811	1573084	10.79
	50	5378.59	5393.36	5408.23	1734645	1739606.9	1741944		10.59
	100	6226.06	6246.93	6276.38	1737920	1740109.7	1742137		10.62
	200	9254.35	9269.21	9282.23	1737168	1739583.3	1741142		10.58
	300	16803.20	17328.97	17718.21	1737103	1738153.75	1738854		10.49
	400	18074.74	20200.82	21126.87	1737261	1738755.9	1741520		10.53
	500	25285.88	26356.86	28859.90	1736388	1738466.2	1740073		10.51

Table 3.12: Cost quality comparison of the PIHC algorithm with the state-of-the-art TSP solvers on various TSPLIB instances. Values in the parentheses are deviations from the optimal cost.

Instance	LOGO	TSP2.2	PIHC	Optimal
kroA200	31685 (7.78%)	29953 (1.99%)	29579 (0.72%)	29368
rat783	9658 (9.67%)	9572.4 (8.70%)	9002 (2.23%)	8806
vm1084	267210 (11.66%)	253767.9 (6.05%)	250998 (4.89%)	239297
pcb3038	147690 (7.25%)	152865 (11.02%)	145190 (5.44%)	137694
fnl4461	194746 (6.67%)	201707.2 (10.48%)	189881 (4.01%)	182566
rl5934	582958 (4.84%)	620101 (11.52%)	581802 (4.63%)	556045
d15112	1652806 (5.06%)	1738466.2 (10.51%)	1650340 (4.91%)	1573084
d18512	675638 (4.71%)	716925 (11.11%)	671000 (3.99%)	645238
pla33810	69763154 (5.21%)	73840715 (11.79%)	69481379 (5.20%)	66048945
pla85900	149708033 (5.14%)	-	148437636 (4.25%)	142382641

more GPU time. Up to the $rat575$ instance, PIHC consumes more GPU time compared to the TSP2.2 TSP solver. For small size instances, PIHC suffers significant communication time overhead (i.e., host to device and device to host data transfer time). As instance sizes increase, communication overhead reduce significantly. Communication time requirement of TSPLIB instances are shown in Table 3.13. PIHC attains remarkably good speedup compared to TSP2.2 with seven different restarts which is shown in Figure 3.7. Compared to 10, 50, 100, 200, 300, 400, and 500 restarts, PIHC gives the speedup in the range of 0.06× - 179.21×, 0.06× - 196.75×, 0.06× - 234.99×, 0.08× - 335.94×, 0.14× - 661.45×, 0.16× - 723.91×, and 0.20× - 979.96× respectively for the instances of sizes 198 - 33810. The reason behind higher speedup is that PIHC initiates single initial solution which is constructed using the NN approach whereas, TSP2.2 initiates multiple initial solutions randomly. From Table 3.9, it is observed that NN approach builds a shorter initial solution than the random, therefore it require fewer steps to reach a local optimal solution, thus it makes PIHC faster, even if TSP2.2 use the 2-opt move implementation. Note that the average execution time of ten trials have been considered for each instances up to $d15112$ cities and for each restart value. Instances above $d15112$ sizes have been executed once with 100 restarts and their execution time are recorded.

3.3.5 Performance Analysis with CPU based state-of-the-art TSP Solvers

Concorde (David Applegate) and LKH (Helsgaun 2000) are the best CPU-based TSP solvers. Concorde provides the exact solution TSP whereas LKH is a heuristic algorithm which is known to provide the optimal solution of TSP instances. Concorde uses the branch and bound and linear programming techniques to solve TSP instances. Since concorde and LKH provides optimal solutions, PIHC cannot be compared with these approaches in terms of solution quality. PIHC provides the local optimal solution which has error rate in the range of 0.72%-8.06% for the instances of size 198 - 85900 cities.

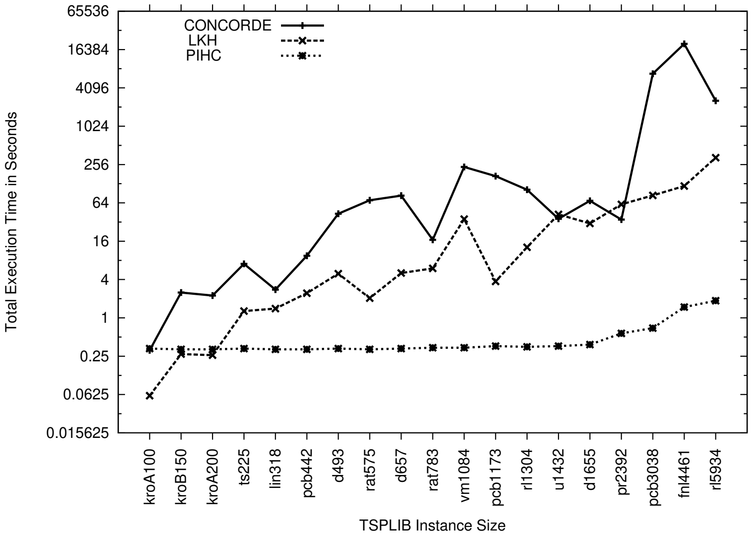


Figure 3.8: Execution Time analysis with the best CPU-based TSP Solvers

Figure 3.8 shows the total execution time requirement for different TSPLIB instances ranging from 100 - 5934 cities. LKH have been run ten times and the average execution time of each instance has been considered. Execution time of concorde for instances ranging from 100 - 2392 have been collected from the link: `http://www.math.uwaterloo.ca/tsp/concorde/benchmarks/bench.html`. For instances above 2393 to 5934, execution time is recorded by a single run for each instance. Up to 200 cities, PIHC suffers communication overhead. When size increase above 200, PIHC provides a solution in the lesser time. For each instance above 200 size, PIHC spends significantly lesser time than Concorde. For instance $fnl4461$, Concorde needs 20135.05 seconds to reach an optimal solution whereas PIHC needs 1.48 seconds to reach a local optimal which has 4.01% error rate. For $rl5934$ instance, LKH needs

327.89 seconds to reach an optimal solution whereas PIHC produces a solution in 1.83 seconds and which has a error rate 4.63%.

3.3.6 Detailed Performance Analysis of the PIHC Approach

Table 3.13 reports a detailed time analysis of Parallel Iterative Hill Climbing algorithm to solve TSP. The nvprof (NVIDIA) profiler is used to obtain the isolated times from the overall execution time. For this analysis, the best performing thread mapping strategies have been considered. Execution times slightly vary from the times presented in Table 3.10 due to the additional profiling overhead. The values reported for the PIHC are: host to device and device to host data transfer time, GPU kernel execution time, and total execution time in seconds (columns 2 - 5) respectively. The sequential implementation execution time in seconds is reported in column 6. The speedup of the PIHC compared to its sequential implementation is reported in column 7. The cost quality results are presented in columns 8 - 11. The cost of initial solution using NN, the cost of the PIHC local optimal solution and the optimal are shown.

PIHC is a single solution-based iterative hill climbing algorithm that repeatedly improves the initial solution until it reaches the local optimal solution. If the local optimal solution is suboptimal, the improvement process halts at the suboptimal value. This is because, the initial solution is constructed using the nearest neighborhood with fixed source node that result in the same initial solution irrespective of multiple trials.

In addition to a single initial solution based PIHC, n initial solution can be constructed using NN and later improved repeatedly at each initial solution. When no further improvement is possible, the best improved solution is selected among n local optimal solutions. This n initial solution based PIHC produce better solution compared to a single initial solution based PIHC but the same best solution is produced irrespective of multiple trials and spends $n\times$ more execution time approximately. A solution to get rid of trapping into the local optimal solution is to use the random restart.

Table 3.13: Detailed performance analysis of the proposed PIHC approach with time and cost analysis. Time analysis presents the time of host-device transfer, device-host transfer, actual GPU computation, Overall PIHC execution in seconds. Speedup on its corresponding sequential part. Cost analysis presents initial and final cost obtained using NN with error rate.

Instance	PIHC time				Sequential	Speedup	PIHC Cost			
	HtoD	DtoH	GPU	Total			Initial	Reached	Optimal	Error rate
kroA200	103.27us	72.321us	275.17us	0.43	0.026	0.06	35715	29579	29368	0.72
lin318	192.43us	121.06us	946.30us	0.42	0.1	0.24	53959	44278	42029	5.35
pcb442	142.38us	88.805us	1.1326ms	0.42	0.15	0.36	61926	53253	50778	4.87
d493	285.52us	186.21us	2.5782ms	0.42	0.28	0.67	43244	36101	35002	3.14
rat575	295.88us	174.60us	3.3895ms	0.43	0.72	1.67	8431	6933	6773	2.36
d657	366.03us	210.19us	5.2448ms	0.42	0.8	1.90	61289	51920	48912	6.15
rat783	452.88us	247.72us	8.5573ms	0.43	1.38	3.21	10867	9002	8806	2.23
vm1084	539.32us	265.77us	16.995ms	0.44	2.62	5.95	294687	250998	239297	4.89
rl1304	587.26us	269.78us	24.684ms	0.44	4.13	9.39	339621	273332	252948	8.06
u1432	774.27us	341.23us	37.762ms	0.45	6.28	13.96	188751	163278	152970	6.74
u2319	1.3254ms	464.28us	132.04ms	0.57	21.53	37.77	278721	241566	234256	3.12
pcb3038	2.8042ms	839.77us	406.85ms	0.85	70.69	83.16	172791	145190	137694	5.44
fnl4461	5.4027ms	5.4027ms	1.46017s	1.8	226.4	125.78	223855	189881	182566	4.01
rl5934	4.9694ms	944.61ms	1.95215s	2.27	293.73	129.40	679591	581802	556045	4.63
rl11849	19.391ms	2.1301ms	13.9813s	14.47	2537.07	175.34	1116175	966496	923288	4.68
d15112	46.780ms	4.1789ms	44.1279s	44.76	8638.85	193.00	1942334	1650340	1573084	4.91
d18512	72.308ms	5.0780ms	78.6453s	79.36	15184.18	191.33	785991	671000	645238	3.99
pla33810	118.53ms	5.1259ms	191.534s	192.43	27157.77	141.13	77321491	69481379	66048945	5.20
pla85900	649.92ms	11.226ms	2714.000s	2714.67	499976.66	184.18	163374026	148437636	142382641	4.25
Min				0.42	0.026	0.06				0.72
Max				2714.67	499976.66	193				8.06
Avg				160.81	29,164.39	68.34				4.46

3.4 SUMMARY

In this chapter, a GPU-based parallel iterative hill climbing (PIHC) algorithm have been presented to solve the symmetric TSPLIB instances up to 85900 cities in a reasonable amount of time with a good quality cost. Six construction heuristic approaches and four CUDA thread mapping strategies have been demonstrated for 2-opt move to solve TSP on the GPU.

It is observed that NN and greedy are the faster approaches to construct a good cost initial solution. Although NI, MST, and Christofides' approaches construct a better cost solution but result in slower approaches. The Clarke-Wright algorithm is slower and worse in constructing an initial solution compared to other approaches.

Four CUDA thread mapping strategies, Threads per Row (TPR), Threads per Row Equal Distribution (TPRED), Threads per Row and Column (TPRC), and Threads per Neighbor (TPN) have been designed and experimented. For moderate size instances up to 18512 cities, TPRC and TPN are the best mapping strategies which operates on each neighbor simultaneously. When instance size goes above 18512, TPR performs better than other three approaches. Both TPN and TPRC suffers large function call overhead. Though TPRED spends lesser time than TPR for instances up to 15112, it suffers thread control divergence afterwards.

Proposed approach is evaluated and compared with LOGO and TSP2.2 GPU based state-of-the-art TSP solvers and it is observed that PIHC approach accounts the good quality results with error rate 0.72% in the best case and 8.06% in the worst case. The proposed PIHC implementation produce the speedup up to 979.66× over the GPU based TSP2.2 implementation with 500 restarts. Overall, PIHC receives 68.34× on average and 193× in the best case over its corresponding sequential implementation.

CHAPTER 4

MIN-MAX ANT SYSTEM ON GPU FOR LARGE TSP INSTANCES

In this chapter, two GPU-based parallel strategies have been demonstrated, namely task-level and data-level strategies for Min-Max Ant System (MMAS), to solve TSPLIB instances over the GPU platform. Contributions of this chapter includes:

- Solve larger size instances up to 33810 cities in a reasonable amount of time.
- Receive a speedup up to $60\times$ over sequential counterpart.
- Produce the satisfactory solutions with error rates in the range of 0.52% - 4.97%.

4.1 INTRODUCTION

Finding an optimal solution of the TSP problem becomes intractable while solving larger instances due to its time-bound, which is exponential time or factorial time (Cormen et al. 2009). Therefore, approximation algorithms and heuristic algorithms have been designed to find good solutions in an acceptable amount of time (Gutin and Punnen 2002; Johnson and Mcgeoch 1997). This work focus on the heuristic-based algorithm, Ant Colony Optimization (ACO), to solve the TSP problem (Dorigo and Gambardella 1997).

In the ACO algorithm, n ants construct feasible solutions. Later, it improves it, and finally, the best-improved solution is chosen (Further details are presented in Section 4.3). The sequential implementation of this approach consumes much CPU time for larger size instances. The most time-consuming part of ACO is constructing a feasible solution and improving it for each ant. Each ant constructs its feasible solution independently with others, and hence, this part can be implemented in parallel. Papers have shown (Cecilia et al. 2013; Delévacq et al. 2013) that parallel implementation of ACO reduces the total execution time effectively.

This work aims to reduce the execution time of the sequential implementation using the parallel processing power of the Graphic Processing Unit.

4.2 BACKGROUND

In this section, different existing sequential and parallel implementation works of the ACO algorithm have been presented briefly.

4.2.1 Sequential Implementation

Dorigo and Gambardella (1997) firstly proposed the Ant Colony Optimization (ACO) algorithm. Authors have used ACO to solve TSP problem for instances of size 1577 cities. Their algorithm outperforms over the existing nature-inspired algorithm, such as simulated annealing and evolutionary algorithm. Moreover, applying local search after ACO finishes results in good solutions. Stützle and Hoos (2000) has presented a variant of the ACO algorithm, named, Min-Max Ant System (MMAS). Author has provided the solution to the limitation of Dorigo and Gambardella (1997)'s work, which occurs while solving larger instances. MMAS has used a greedier search approach to avoid redundant exploration of search-space.

4.2.2 Parallel Implementation

The advent of multi-core CPU allows programmers to distribute computational work over multiple cores independently. Stützle (1998b) proposed the first multi-core parallel strategy, which implements multiple ant colonies on various processors. It avoids the communication overhead. This does not improve the iteration time, but it requires fewer iterations. It is thus reducing the time to converge the solution. Zhou et al. (2018) present a SIMD based CPU implementation that takes advantage of vector extensions of CPUs. They claimed up to $57.8\times$ speedup in AS and up to $8.7\times$ speedups in MMAS with 3-opt local search for instances up to 4461 cities.

Nowadays Graphics Processing Unit (GPU) is available for general-purpose computations (Che et al. 2008). Earlier, GPU was dedicated only to the graphics acceleration purpose. Li et al. (2009) proposed a fine grain model for GPU acceleration. The parallel strategy used in this work is that one ant is mapped per thread block. Thus, this parallel strategy utilizes the GPU resources better and getting significant speedups over its counterpart.

Skinderowicz (2016) proposed a parallel approach for ACO on GPUs. They have implemented a warp level element selection technique which reduces the number of block synchronization. They provides a better implementation of the roulette wheel selection algorithm. They have claimed up to $24.29\times$ speedup for up to 2392 cities. Cecilia et al. (2013) proposed an I-Roulette method to implement the roulette wheel selection algorithm for the fine-grain approach. Moreover, author provides an improved pheromone depositing phase using the scatter to gather based design. Their work receives speedup up to $20\times$ for the instances up to 2392 cities. Audrey Delévacq et al. present the GPU implementation for the MMAS. Four GPU parallel strategies have been presented for the MMAS algorithm. $23.60\times$ speedup is received for the TSPLIB instances up to 2103 cities.

The above-described papers presented the efficiency of their parallel strategies for ACO over smaller TSPLIB instances, i.e., up to 2392 cities. Here, a GPU based parallel design has been presented for the ACO algorithm to solve larger TSPLIB instances up to 33810 cities. This chapter presents a parallel variant of the Ant Colony Optimization (PACO) algorithm called Min-Max Ant System (Stützle and Hoos 2000) to solve TSP over the GPU platform.

4.3 ANT COLONY OPTIMIZATION

Ant colony optimization is inspired by the real ant colony (Dorigo and Gambardella 1997) and the way they search their food. When an ant finds a path to food, it deposits a substance called pheromone on the path. Other ants can sense this pheromone trail and follow it. The pheromone evaporates with time. Other ants also deposit pheromones on the path they choose. Thus the best path gets high pheromone reinforcement. This concept is emulated in the ACO approach. It is a meta-heuristic approach that forms a solution using previous experience or knowledge. Each ant holds information, like its tour(list of cities), list of visited cities, and tour length. Each ant knows the distance between all the cities and the pheromone value associated with each path. For the purpose of this demonstration, a variation of the ant system called MIN-MAX ANT System (MMAS) is used. Each phase of ACO is described below:

4.3.1 Tour Construction

Each ant is placed on a randomly chosen city. Based on the pheromone information and the distance between the cities, the ant calculates the probability of visiting the next city with Eqn. 4.1.

$$p_{ij}^{k} = \begin{cases} \frac{[\tau_{ij}]^{\alpha} . [\eta_{ij}]^{\beta}}{\sum_{l \in N_i^k} [\tau_{il}(t)]^{\alpha} [\eta_{il}]^{\beta}} & j \in N_i^k \\ 0 & otherwise \end{cases} \tag{4.1}$$

Where i is the current city, p_{ij}^{k} is the probability of choosing the next city, j. τ_{ij} is the pheromone value between cities i and j. η_{ij} is the heuristic information, which is $1/d$, where d is the distance between cities i and j. N is the feasible set of neighbors (i.e., cities ant k has not visited yet). α and β controls the effect of pheromone trail and heuristic information on the probability.

4.3.2 Pheromone Update

After ants have constructed the solution, they update the pheromone value using Eqn. 4.2.

$$\tau(t+1) = \rho\tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t) \tag{4.2}$$

where ρ is called pheromone trail persistence which emulates trail evaporation. $\Delta\tau_{ij}^{k}(t)$ is defined as in Eqn. 4.3.

$$\begin{cases} 1/L^{k}(t) & if\ arc(i,j)\ is\ used\ by\ ant\ k\ in\ iteration\ t \\ 0 & otherwise \end{cases} \tag{4.3}$$

The MMAS algorithm uses Eqn. 4.4 for the pheromone update instead of Eqn. 4.2.

$$\tau(t+1) = \rho\tau_{ij}(t) + \Delta\tau_{ij}^{best} \tag{4.4}$$

where $\Delta\tau_{ij}^{best}$ is $1/f(s^{best})$ which is the inverse of either global best solution or iteration best solution. In MMAS, only the best ant updates the pheromone matrix. Also, to avoid stagnation, the pheromone values are restricted between τ_{min} and τ_{max} which are defined in equations Eqns. 4.5 and 4.6 respectively.

$$\tau_{max} = \frac{1}{1-\rho}\frac{1}{f(s^{gb})} \tag{4.5}$$

$$\tau_{min} = \frac{\tau_{max}(1-p_{dec})}{(avg-1)p_(dec)} \tag{4.6}$$

where s^{gb} is the global best solution. $p_{dec} = \sqrt[n]{p_{best}}$ in this work the value of p_{best} is set to 0.005 and $avg = n/2$ as described in Stützle and Hoos (2000) and so $\tau_{min} = \tau_{max}/2n$ where n is the number of cities. The pheromone matrix is initialized with τ_{max} as in Stützle and Hoos (2000). The MMAS algorithm can be further improved using the local search algorithm (Stützle, 1998a).

4.4 METHODOLOGY

The general flow of a parallel approach is given in Figure 4.1. The stepwise details have been elaborated as follows. First, the initial solution is constructed at the CPU side. The nearest neighborhood (NN) algorithm is used to construct the initial solution. The NN starts from a random city, also known as a source city. Later, it selects the next unvisited closest city of the current city. This procedure is repeated until all cities are visited. Then the cost of this tour is used to initialize variables, t_max and t_min using Eqn. 4.5. Next, the pheromone matrix is allocated, and it is initialized to t_max. The total matrix is then calculated using Eqn. 4.1. Next, the total matrix is copied to GPU. In this implementation, the unified memory model is used, which handles copying tasks. Then the tour construction phase begins on GPU using one of the two parallel approaches described below. After the tour construction phase is completed, the best ant is chosen on the CPU side. Then the pheromone values are evaporated. This is done on the GPU side to reduce execution time. Later, the pheromone is deposited on the tour of the best ant. This is done on CPU as it takes $O(n)$ iterations. The total matrix is calculated again since this is a matrix operation which is taken place on the GPU. This procedure is repeated until a specified number of iterations.

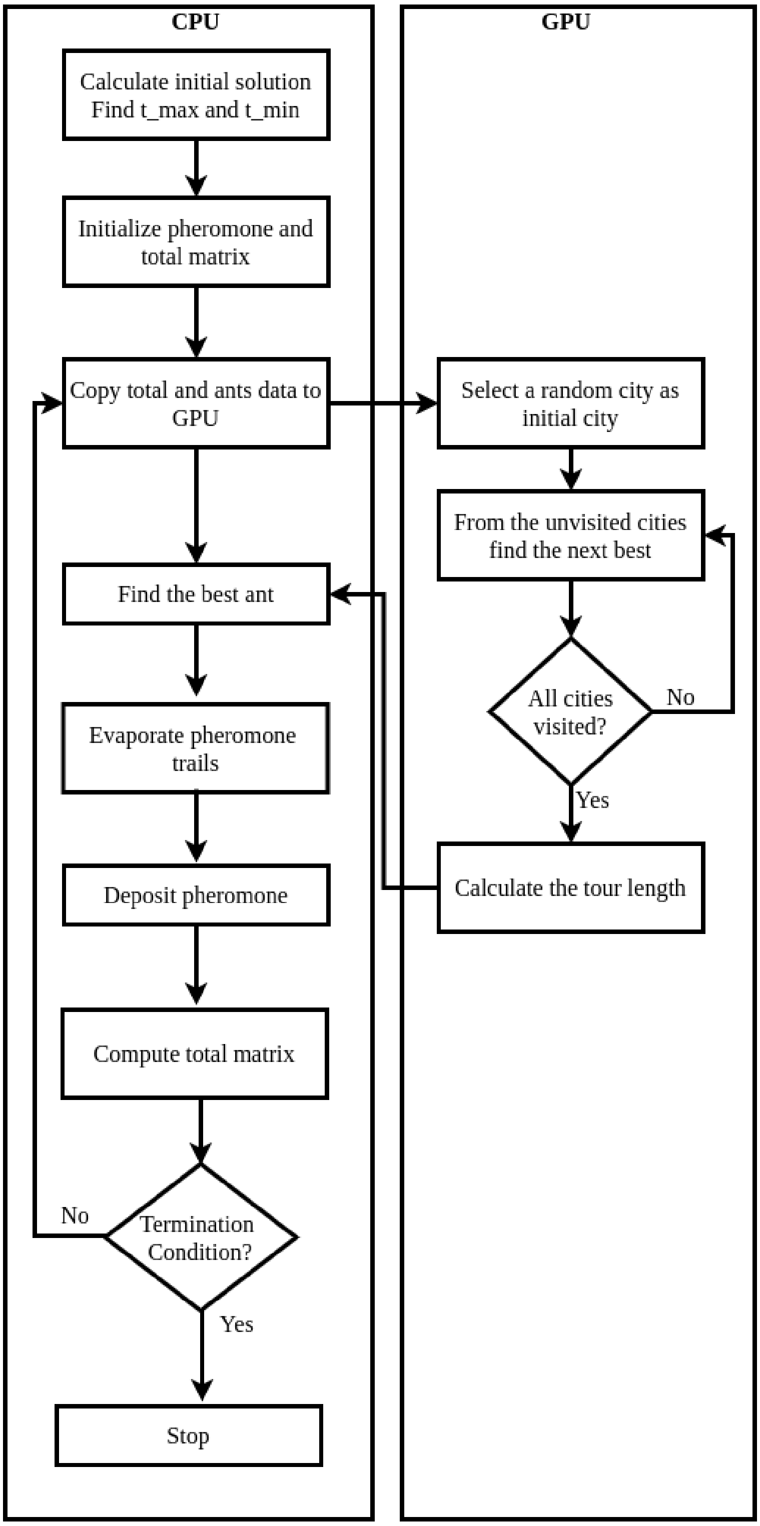

CPU
GPU
Calculate initial solution
Find t_max and t_min
Initialize pheromone and total matrix
Copy total and ants data to GPU
Find the best ant
Evaporate pheromone trails
Deposit pheromone
Compute total matrix
Termination Condition?
No
Yes
Stop
Select a random city as initial city
From the unvisited cities find the next best
All cities visited?
No
Yes
Calculate the tour length

Figure 4.1: Parallel work-flow model for work distribution between CPU and GPU of ACO algorithm.

Two parallel strategies have been designed for the ACO algorithm which are explained below:

4.4.1 Task-level Parallel Approach

In this parallel approach, each tour is constructed by a separate thread. So the number of threads to be launched is equal to the number of ants. The step-wise detail of this approach is presented in Algorithm 4.1. Each thread has to go through the entire list of cities to find the city with maximum value. Here, the maximum value is the probability rate of city j from a current city i. Overall, this procedure takes $O(n)$ time. After finding the city with maximum total info, it marks the corresponding city as the next city to be visited. This limits the number of threads that can be launched per block. This whole process runs for n number of times until all cities are visited. Note that the number of ants is equal to total cities available per instance. The number of threads and blocks needed is given in Eqns. 4.7 and 4.8 respectively.

$$threads = ants \tag{4.7}$$

$$blocks = \frac{ants - 1}{threadsPerBlock} + 1 \tag{4.8}$$

Since the number of ants (threads) is equal to total threads, the degree of parallelization is shallow. This leads to very low GPU optimization.

4.4.2 Data-level Parallel approach

This method tries to mitigate problems that arise in the first method by increasing the degree of parallelization. This method constructs one tour per block instead—threads within the block, work to parallelize the tour construction phase. The flow of this approach is presented in Algorithm 4.2. In this approach, one city is handled by one thread. First, the visited vector is reset. Then, one thread of the blocks initializes the tour by assigning the initial city. This city is selected randomly. Then each thread reads the total value corresponding to one city to find the next city. Next, a block-wide reduce operation is performed to find the maximum value of total from the unvisited cities and its thread id(which represents the city index). The number of blocks and threads

Algorithm 4.1: Task Parallel Approach

```
Result: Feasible solution is created for each ant
Each thread act as an $ant$
$max\_total \leftarrow 0$
$next\_city \leftarrow 0$
$step \leftarrow 1$
while $step < n$ do
    $i \leftarrow 0$
    while $i < n$ do
        if $visited[i] == 0$ then
            if $max\_total < total[i]$ then
                $max\_total = total[i]$
                $next\_city = i$
            end
        end
    end
    $tour[step] = next\_city$
    $visited[next\_city] = 1$
    $step++$
end
```

required for this approach is calculated using Eqns. 4.9 and 4.10 respectively.

$$blocks = ants \tag{4.9}$$

$$threads = \begin{cases} 1024, & \text{if } cities \geq 1024 \\ cities, & \text{otherwise} \end{cases} \tag{4.10}$$

Checking whether a city is visited is a decision problem that leads to thread divergence. To mitigate this problem, the visited cities are assigned a value zero, and unvisited cities are assigned the value 1. Then, the corresponding value from the visited vector is multiplied with the total info. If the city is already visited, the total value corresponding to that city becomes zero (which is the minimum possible total value), resulting in zero probability of the city being selected. But if the city is unvisited, then the total remains the same. Each thread performs this multiplication.

Since the data to be operated is stored in threads' local memory, its access speed is much faster. This operation takes $O(log_2n)$ time. This will significantly reduce the execution time. After this operation, the city with a maximum total value is assigned as the next city. The above step is repeated until all cities are visited(n times) to complete the tour construction phase. If the number of cities is larger than the maximum number of threads in a block, each thread has to handle multiple cities. This further increases GPU occupancy.

Thus this approach is better than the previous approach as it utilizes the GPU properly and solves the thread divergence problem. It also reduces the time as it parallelizes the tour construction phase. It is thus reducing the time complexity.

Algorithm 4.2: Data Parallel Approach

```
Result: Feasible solution is created for each ant
while $step < n$ do
    $p = selProb[threadIdx.x]$
    $v = ants[blockIdx.x].visited[threadIdx.x]$
    $result = p * v$
    $nextBest = reduce(result)$
    $tour[step] = nextBest$
    $visited[nextBest] = 0$
    $step++$
end
```

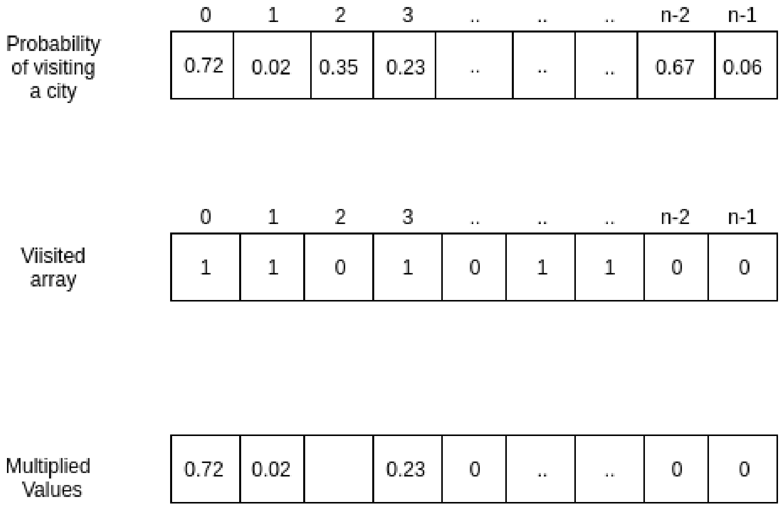


Figure 4.2: Data Parallel Approach

Then a block-wide reduce operation is performed to find the maximum value among the threads. The city corresponding to the thread with maximum probability is selected as the next city. In this approach, only one city's worth of data is required by a thread to be stored in its local memory, which facilitates fast data access. The reduce operation can be done in parallel. To compute tours having more cities than the block dimension, one thread needs to process multiple cities. This also increases utilization and leads to better overall efficiency. In this implementation, the maximum block dimension is 1024, and each thread can handle at most 35 cities. Thus the implementation can handle TSP instances up to 35840 cities. This approach is shown in Figure 4.2.

4.4.3 2-opt

The solution quality obtained by the MMAS alone can be greatly improved by applying local search on the final cost of MMAS. A parallel approach is implemented for the 2-opt. The 2-opt is applied after tour construction. The threads are needed to be mapped to the pair of cities. Thread per row equal distribution strategy is used to achieve this. The number of pairs to be examined is given by Equation 4.11, Equation 4.12 gives the number of threads required, and Equation 4.13 tells the number of CUDA blocks required.

$$solutions = \frac{n * (n-1)}{2} \tag{4.11}$$

$$threads = \begin{cases} 1024, & \text{if } cities \geq 1024 \\ cities, & \text{otherwise} \end{cases} \tag{4.12}$$

$$blocks = (cities - 1)/threads + 1 \tag{4.13}$$

Since the number of solutions are greater than number of threads. Each thread need to process more than one pair in loop. The number of pair each threads need to process is given in Equation 4.14.

$$item\_per\_thread = \frac{solutions}{threads} \tag{4.14}$$

After calculating the above values, threads are mapped to two cities having city index as $threadIdx.x$ and the value $j = (threadIdx.x + j + 1)$ where j varies form 0 to $item\_per\_thread$. This parallel strategy reduces the time to calculate the 2-opt route.

4.5 RESULT ANALYSIS

The performance of GPU-based parallel ACO algorithm has been evaluated using TSPLIB instances up to 33810 cities. The GPU device, which is used to execute a parallel version, has the following specifications. A GPU device has a NVIDIA Tesla P100 GPU card with a compute capability version 6.0, 16 GB of the global memory, 56 Streaming Multiprocessors (SMs), where each SM has 64 cores operating at 1.33 GHz. Average execution time is collected, running each instance using 1000 iterations.

4.5.1 Data versus Task-based Parallel Approach

Two parallel strategies named, data level and task level parallelism have been evaluated on the TSPLIB instances of ranges $100-33810$ cities. The runtime requirement of both approaches is presented in Table 4.1. Data-level and task-level strategies are up to $60\times$, $22\times$ faster over the sequential approach for the instances up to 7397 cities. As instance size increases, the data-level approach outperforms over task-level parallel approach.

Instance	Average time (ms) for serial version	Average Time (ms) For data parallel	Average Time (ms) For task based
KroA100	6	3.96	4.03
d198	38	14.45	19.849
lin318	152	34.11	55.22
pcb442	438	63.15	112.281
d493	621	78.76	148.406
rat575	926	110.5	215.019
d657	1426	140.284	281.693
rat783	3131	204.259	574.782
vm1084	7145	447.183	1329.97
d1291	9461	652.24	2068.33
rl1304	11475	664.44	1987.15
u1432	13733	799.44	2219.84
u2319	62053	2223.59	7436.67
pcb3038	119140	3925.48	123383.7
fnl4461	536427	8851.46	29886.7
rl5934	1034504	16391.5	51477.9
pla7397	1626091	55672.3	75451.5
usa13509	-	44589	347244
d15112	-	60524.3	784392
pla33810	-	240172	3763810
Average		21778.1203	259604.952

Table 4.1: Time analysis of sequential, data-level, and task-level parallel approaches. Time is presented in milliseconds. The average execution time of the instances for 1000 iteration.

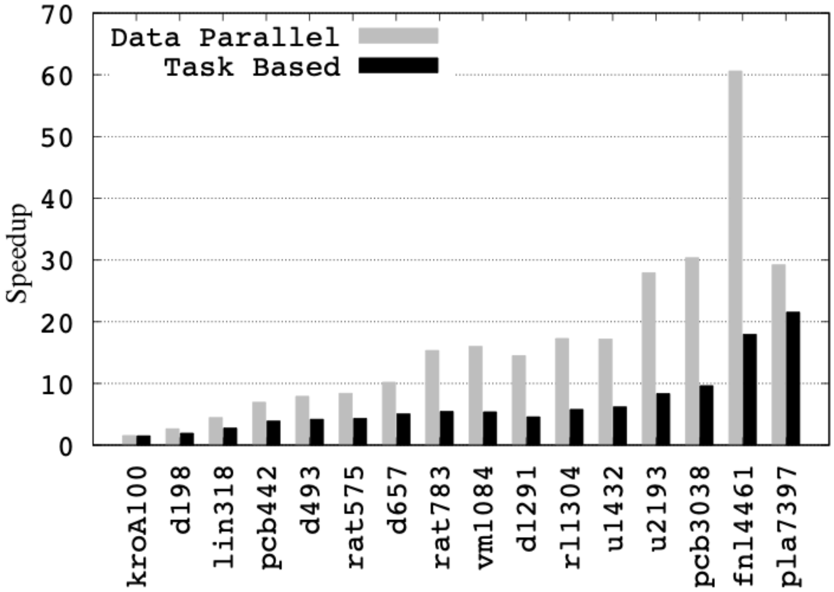


Figure 4.3: Speedup analysis of data-level and task-level approaches over sequential counterpart.

Figure 4.3 shows the speedup analysis of both parallel approaches over sequential counterpart. Data level approach is $3.37\times$ faster in the best-case and $1.02\times$ faster in the worst-case. In a task-level approach, each thread has to do $O(n^2)$ work. Thus, this method is quite slow. Each ant requires storing the whole thread data and the visited vector to handle the whole tour. From Table 4.1, it is noticed that the task-level approach is slower. This is due to poor work division as each thread constructs one whole tour. Hence, this leads to less GPU utilization.

4.5.2 Sequential versus Parallel

The sequential implementation of the ACO is compared with its parallel counterpart to identify the speedup. The detailed time analysis of sequential and parallel approach is presented in Table 4.1. Figure 4.3 shows the speedups obtained in parallel approaches versus serial approach. It can be seen that the speedup is lesser (about $10\times$) for smaller sized instances. This is because of the memory transfer overhead, and GPU is not fully utilized. As instances grows, the speedup increases up to $60\times$ over sequential counterpart. Speedup comparison is shown for the instances up to 7397 cities. The reason is that for larger instances running the serial approach was time-consuming.

4.5.3 Solution Quality

In terms of solution quality, the performance of ACO has been analyzed over TSPLIB instances ranging from 100 - 33810. The cost quality received for each instance has been presented in Table 4.2. Table 4.2 contains instance name, its optimal cost, the local optimal solution where ACO stops, error rate concerning optimal cost, local cost when 2-opt is considered, and the error rate after applying 2-opt.

Instance	Optimal Cost	Local Cost (without 2-opt)	Error Rate	Local Cost (With 2-opt)	Error Rate	Time (seconds) (With 2-opt)
KroA100	21282	23188	8.95	21394	0.52	0.11
d198	15780	17395	10.23	15873	0.58	0.29
lin318	42029	47969	14.13	43503	3.5	0.77
pcb442	50778	56144	10.53	51483	1.38	1.54
d493	35002	38836	10.95	35579	1.64	2.02
rat575	6773	7707	13.79	6859	1.27	2.67
d657	48912	55826	14.13	51168	4.61	3.33
rat783	8806	10207	15.90	8962	1.77	5.06
vm1084	239297	278158	16.23	248436	3.81	9.61
d1291	50801	55248	8.75	51450	1.27	10.30
rl1304	252948	280100	10.73	258307	2.11	12.38
u1432	152970	180817	18.20	157205	2.65	13.15
u2319	234256	268694	14.70	238264	1.71	36.35
pcb3038	137694	164955	19.79	143766	4.4	58.12
fnl4461	182566	217632	19.20	189327	3.7	150.72
rl5934	556045	665676	19.7	573930	3.2	225.18
pla7397	23260728	27231392	17.07	24259925	4.2	337.03
usa13509	19982859	24958865	24.90	20976170	4.97	1445.10
d15112	1573084	1887879	20.01	1635492	3.96	1953.99
pla33810	66048945	77008662	16.69	68774275	4.12	17426.63
Average			15.23		2.77	1084.72

Table 4.2: Comparison of solution quality with and without 2-opt. One iteration of execution time is given in seconds.

When 2-opt is not considered, the final cost of ACO has a 8.75% error rate in the best-case and 20.01% in the worst-case. To reduce these error-rates further, 2-opt is applied. The 2-opt local search improves the ACO's cost quality significantly. With 2-opt, it has a 0.52% error in the best-case and 4.97% error in the worst-case. From Table 4.2, it is observed that applying the local search algorithm in addition to the ACO algorithm improves local solution significantly. Note that the time presented in Table 4.1 is collected without considering the 2-opt local search.

4.5.4 Comparison with Previous Works

The GPU-based thread mapping strategies presented in (Delévacq et al. 2013; Skinderowicz 2016) are similar with implementation level differences. Delévacq et al. (2013) have presented ant-level and colony-level thread mapping strategies for the tour construction phase. Once a feasible solution is constructed per ant, the 3-opt move is applied to improve the solution quality further. Skinderowicz (2016) have presented three GPU-based parallel models to Ant Colony System (ACS). Authors have implemented the warp-level element selection technique, which reduces the number of block synchronization. In both (Delévacq et al. 2013; Skinderowicz 2016) implementations, three 2D matrices need to be transferred on the GPU for the tour construction. These three matrices are distance matrix, pheromone matrix, and n routes of n ants. The work presented in this chapter uses only two 2D matrices, which are: a probability matrix and n routes of n ants. The probability matrix stores the probability rates between any two cities i and j, and the rate is calculated using Equation 4.1. This probability matrix helps us to avoid transferring distance and pheromone matrices over the GPU. This memory-efficient approach helps to solve large-scale instances over the GPU platform.

4.6 SUMMARY

This chapter presents a GPU-based parallel approach for Ant Colony Optimization (ACO) algorithm to solve the TSP problem. This work aims to solve larger size TSPLIB instances over the GPU platform. ACO algorithm constructs n ants and later improves it to converge towards the near-optimal solution. The most time-consuming part of the ACO algorithm is to construct n ants and repeatedly improve it until no further improvement is possible. Constructing n ants are independent of each other. Therefore, this part of the ACO algorithm can be implemented in parallel to reduce total runtime.

Two GPU-based parallel strategies have been implemented, namely data-level and task-level parallel approaches for the ACO algorithm. Task-level parallel approach maps one ant per thread whereas, in a data-level approach, one ant is mapped per thread block. The task-level approach is up to $22\times$ faster over the sequential approach. In a data-level approach, multiple threads construct each ant's solution. Therefore, data level approach outperforms over the task-level approach, i.e., 1.02 to 3.37 times faster than the task-level approach for instances up to 33810 cities. When data-level approach is compared with a sequential counterpart, up to $60\times$ speedup is observed for the instances in the range of $100 - 33810$ cities. When solution quality is concerned, ACO with 2-opt produces good quality solutions with error rates in the range of $0.52\% - 4.97\%$ for instances up to 33810 cities.

CHAPTER 5

PARALLEL DETERMINISTIC LOCAL SEARCH HEURISTIC FOR MINIMUM LATENCY PROBLEM

This chapter focuses on a deterministic local search heuristic algorithm that provides the same local solution for a given MLP instance irrespective of multiple runs. Existing metaheuristic algorithms for MLP use randomness in setting up the initial solution, and perturbation to escape local minima. A heuristic usually does not produce the same local cost for an MLP instance on successive runs. Adding randomness and perturbation does help to explore a wide range of feasible solutions and results in a better solution. However, the aim of this study is to determine the closeness of local optimal solution of the deterministic heuristic algorithm with a solution obtained using the greedy randomized metaheuristic algorithms. A parallel deterministic local search heuristic algorithm for the GPU to reduce the neighborhood computation time is also presented. The contribution of this chapter includes:

- Provide the Deterministic Local Search Heuristic (DLSH) that assures the same solution for the same instances irrespective of multiple trials.

- Propose the move evaluation procedure to the swap neighborhood that computes a move pair in constant time without using any preprocessed local data.

- Demonstrates the performance analysis of heuristic algorithm over the wide range of instances, i.e., 10-11849 nodes.

- Presents a GPU-based parallel model for DLSH, which solves larger instances than existing parallel MLP solvers.
- Proposed parallel model achieves the speedup up to 179.75 times for the instances up to size 7397. (Proposed source code is available at `http://bit.ly/thesisSourceCodes`).

5.1 INTRODUCTION

The Minimum Latency Problem is an NP-Hard (Blum et al. 1994; Sahni and Gonzalez 1976; Sitters 2002) combinatorial optimization problem. The objective of MLP is to find a Hamiltonian path that minimizes the overall waiting time of nodes. The formal definition of MLP is, consider a simple, directed, weighted graph that has n vertices, where, $n-1$ vertices act as service requesting nodes and one vertex act as service providing node. Each service requesting node v_i $(1 \leq i<n)$ has to wait until it is served. This waiting period is also known as latency. The latency is a sum of distance or time required to reach from the service providing node v_0 to the service requesting node v_i. Objective of MLP is to determine a Hamiltonian path $l(p)$ that has minimum latency (Eqn. 5.1). Note that vertex v_0 is considered as a service providing node, also called depot node.

$$l(p) = \sum_{i=1}^{n} l(v_i) \text{ where } l(v_i) = \sum_{j=1}^{i} D(v_{j-1}, v_j) \tag{5.1}$$

MLP is also known with other names, namely, Cumulative Traveling Salesman Problem (Bianco et al. 1993), Delivery Man Problem (Fischetti et al. 1993), School Bus Driver Problem (Chaudhuri et al. 2003), and Traveling Repairman problem (Tsitsiklis 1992). MLP has several applications in real life, such as data retrieval in the computer network, delivery services, disk head scheduling, and logistics services for emergency reliefs etc. (Campbell et al. 2008; Ezzine et al. 2010; Méndez-Díaz et al. 2008). MLP and TSP, though related, differ in their main objectives. MLP focuses on reducing the average waiting time of all the nodes (customers), whereas TSP focuses on reducing the total traveling distance. The optimal solution of TSP may not be optimal for MLP. MLP is customer-centric, and TSP is distributor centric. Moreover, in TSP, generating feasible solutions from initial solution and calculating its cost is straightforward, a change in affected subsequence has to be considered (Yelmewad and Talawar 2018; Yelmewad and Talawar 2019). In MLP, the latency of each node has to be recalculated after a small modification in the base solution. Therefore, MLP involves more computations per route cost calculation than TSP.

5.1.1 Exact Methods

There are various exact algorithms available to solve MLP that assures an optimal solution (Blum et al. 1994; García et al. 2002; Ha Bang et al. 2013; Lucena 1990; Wu et al. 2004). Lucena (1990) presents the branch and bound algorithm using the lower bound scheme. The lower bound scheme divides lower bound into multiple components and subsequently optimize each component. Blum et al. (1994) proposes two exact methods, namely, dynamic programming, and unweighted trees and depth-first search algorithms. Wu (2000) proposes the dynamic programming approach to solve MLP. Wu et al. (2004) proposes another exact method, namely, branch and bound algorithm for solving MLP. The branch and bound is a more efficient approach than the dynamic programming which consumes less CPU time. For an MLP of 25 vertices, the fastest approach of the branch and bound algorithm consumes 100 seconds, outperforming over the CPLEX (Salehipour et al. 2011). Solving MLP for medium to large instances for exact solutions take an inordinately large time. Therefore, approximation and heuristic algorithms are built to provide near-optimal solutions in lesser time.

5.1.2 Approximation and Metaheuristic Methods

Several approximation algorithms have been proposed to solve MLP (Archer and Blasiak 2010; Arora and Karakostas 2003; Blum et al. 1994; Fakcharoenphol et al. 2007; Goemans and Kleinberg 1998). Approximation algorithms solve NP-Hard optimization problems aiming to provide near-optimal solutions with minimum gap rate. An approximation algorithm assures an upper or lower bound on the constructed feasible solution. The closeness of the approximation algorithm with an optimal solution is determined using the approximation factor. Blum et al. (1994) propose the first approximation algorithm for MLP. This algorithm had an approximation factor of 144 over the optimal solution. Archer and Blasiak (2010) provide an approximate approach which obtains the best approximation factor, i.e., 3.03 in case of edge-weighted tree, the best approximation factor. When the general metric space is considered, Chaudhuri et al. (2003) provide a solution which is within 3.59 times.

A metaheuristic is another approximation method where the cost bound on the obtained solution is not assured (Talbi 2009). Metaheuristic algorithms begin with an arbitrary or constructed initial solution. The initial solution is continuously improved using local search mechanisms. A few metaheuristic algorithms have been proposed to solve MLP (Dewilde et al. 2013; Ribeiro and Laporte 2012; Salehipour et al. 2011; Silva et al. 2012). Nguevue et al. (2010) present a memetic algorithm for the cumulative capacitated vehicle routing problem. They have presented the move evaluation method which provides the cost of the newly generated solution in constant time, $O(1)$, for the particular neighborhood structure where the orientation of the base solution is not changed. Salehipour et al. (2011) propose two combinations of metaheuristic algorithms, first, Greedy Adaptive Search Procedure (GRASP) with Variable Neighborhood Descent (VND), and second, GRASP with Variable Neighborhood Search (VNS). They build eight sets of MLP instances of size 10, 20, 50, 100, 150, 200, 500, and 1000. Each instance size set has another twenty randomly generated instances. Silva et al. (2012) proposes a metaheuristic, which is a combination of GRASP, ILS, and RVND. They have presented a generic move evaluation method to calculate the cost of the solution after applying the move in $O(1)$ time. This time bound is achieved using a preprocessed data structure for the base solution.

Dewilde et al. (2013) propose the tabu search approach for the MLP with profit. All these metaheuristic approaches use iterative local search to improve the solution quality. A large fraction of execution time is spent in exploring different neighborhood structures. This neighborhood exploration time can be reduced by distributing tasks on multi-core processors or many-core co-processors.

5.1.3 MLP on the Graphics Processing Unit

The proposed sequential version of DLSH consumes 39 days to reach a local solution for a TSPLIB instance $rl11849$. In the metaheuristic method, more than 90% execution time on an average is being spent in the solution improvement phase. The parallel implementation reduces the execution time significantly when the instructions involved in the computation have less dependency. Most of the steps present in the neighborhood generation techniques are independent. These neighborhood generation steps can be implemented in parallel. For this work, the parallel computation has been done using the GPU platform. Afif et al. (2020) and Kim et al. (2020) have shown the effectiveness of the GPU platform for the general-purpose computation.

Programmable massively parallel processors, such as Graphics Processing Unit (GPU) (Alawneh et al. 2020; Carvalho et al. 2020; Geng et al. 2020) can be employed in various stages of the MLP execution to arrive at near-optimal solutions in a reasonable amount of time. The existing GPU-based parallel strategy limits to solve instances up to 1024 nodes. In this chapter, a GPU-based parallel strategy have been proposed to address the large-scale instances in lesser time.

5.2 DETERMINISTIC LOCAL SEARCH HEURISTIC (DLSH)

Algorithm 5.1 presents the stepwise details of Deterministic Local Search Heuristic (DLSH) algorithm. First (line 1), the initial solution is constructed using the nearest neighborhood approach. The first node in a feasible solution is fixed to node 0. Later NN starts adding the closest unvisited neighbors of a recently visited node until a feasible solution is constructed. Once the solution is constructed, its cost is calculated (line 2). Equation 5.2 is used for the cost calculation (Silva et al. 2012), where, n is a sum of total clients and one depot node, $D(P_i, P_j)$ returns an Euclidean distance between nodes P_i and P_j, and P is a Hamiltonian path.

$$f(s) = \sum_{j=1}^{n-1} (n - j) \times D(P_{j-1}, P_j) \tag{5.2}$$

Local search is applied to the initial solution using two neighborhood mechanisms, namely, swap and 2-opt move (lines 3-8). If any improved solution is found in any of neighborhood approaches, the neighborhood approaches are applied repeatedly on the improved solution until no further improvement is possible. The order considered for the neighborhood evaluation is, swap move followed by the 2-opt move. This order

Algorithm 5.1: Generic Deterministic Local Search Heuristic

```
Result: Returns the local optimal solution
$s \leftarrow$ initial solution
$f(s) \leftarrow$ cost calculation
while $s$ is improved do
    $s' \leftarrow swap(s)$
    $s' \leftarrow 2 - opt(s')$
    if $s' < s$ then
        $s \leftarrow s'$
    else
        return $s$ as local optimal
    end
end
```

is fixed on the complexity of neighborhood move computation. Neither perturbation, multiple initial solutions, nor randomization have been applied for the DLSH approach. There is no diversification in DLHS to get out of the local optimal solution. Instead, DLHS converges towards the local optimal. Therefore, a term *heuristic* is used instead of *metaheuristic* for this local search algorithm (Talbi 2009). Since there is no randomization exist in the DLHS, it always generates the same solution for an input instance.

Two neighborhood generation approaches have been considered, namely swap move and two-opt move. This is because when any move is applied on the base solution, either it changes the orientation of the base solution or just respective vertex pair of the move is exchanged without changing the orientation. These neighborhood evaluation approaches are explained in the following subsections in detail.

5.2.1 Swap Move

In the swap move, a pair of vertices (i, j) is swapped without changing the orientation of the base solution. Figure 5.1 shows the pictorial representation of the neighborhood generation mechanism. Figure 5.1 (A) shows the original path, and 5.1 (B) shows the path after swapping is applied on the vertex pair (i, j). Total feasible solutions possible with swap approach are $\frac{n(n-1)}{2}$, where, n is total nodes in the Hamiltonian path, excluding the depot node. When a move is applied, the latency of i^{th} onward nodes are changed. Therefore for each move, the total latency of the solution is to be calculated. This cost calculation process consumes $O(n)$ time for each move.

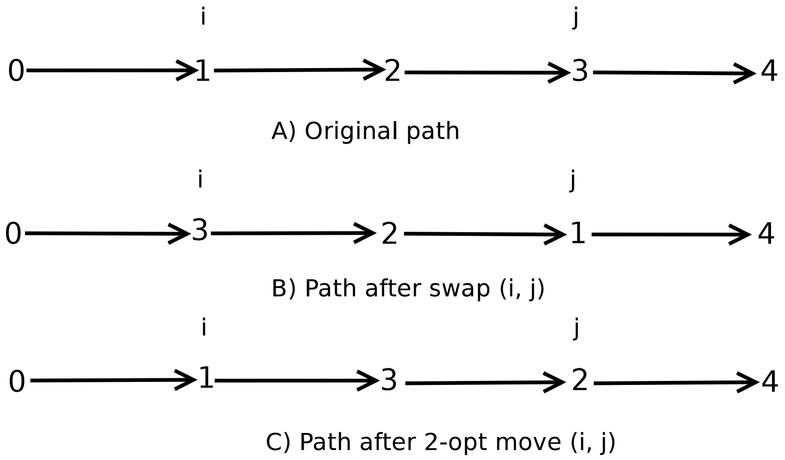


Figure 5.1: Neighborhood Generation Mechanism

5.2.1.1 Move Evaluation

Silva et al. (2012) has proposed the move evaluation procedure, which consumes a constant time, i.e., $O(1)$. They have achieved the move evaluation in constant time using the local data structure. Local data structures are preprocessed before applying move operations. This local data structure contains three matrices, namely duration (T), cost (C), and the delay cost (W) for the base solution. The computation of these matrices is additional overhead for this move evaluation procedure.

In this work, another move evaluation procedure is proposed for the swap move eliminating additional preprocessing of local data. A study has been carried out to investigate the number of edges affected during the swap move evaluation. A study reveals that at most four edges are affected in the initial solution while evaluating a vertex (i, j) pair. Consider an instance of size n, having i and j vertices, where $1 \leq i < j < n$. Three equations have been modeled to evaluate a pair in constant time based upon the number of edges affected.

1) One edge is affected when vertex $i = n - 2$ and vertex $j = n - 1$:

When a swap move is applied on the vertex pair (i, j), where vertices i and j are adjacent and vertex j lies at $n-1$ position in the base solution, only one edge is affected in the base solution. Therefore, one edge, i.e. $(i - 1, i)$, is removed from the base solution and one edge, i.e. $(i-1, j)$, is added back to form a new solution. Figure 5.2 shows a pictorial mechanism of a swap move when applied on the vertex (i, j) pair. In Figure 5.2 (a), the dotted line shows an edge that is removed, and a new edge is added back to the base solution, as shown in Figure 5.2 (b), to form a new feasible solution. For this type of vertex pair, the change is calculated using Eqn. 5.3.

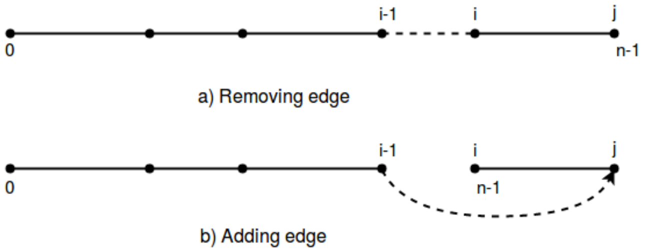


Figure 5.2: An edge affected in swap move when $i = j - 1$ & $j = n - 1$

$$change = (n-i) \times D(P_{i-1}, P_j) - (n-i) \times D(P_{i-1}, P_i) \tag{5.3}$$

2) Two edges are affected when vertex $i = j - 1$ and vertex $j < n - 1$:

When two vertices i and j are adjacent and vertex j is placed at a position $< n - 1$, two edges are affected in a base solution. Figure 5.3 shows a pictorial mechanism of a swap move when vertex $i = j - 1$ and vertex $j < n - 1$. When a swap is applied on such vertex pair, two edges are are removed, namely $(i-1, i)$ and $(j, j+1)$, and two edges, namely $(i-1, j)$ and $(i, j+1)$, are added back to form a feasible solution. Eqn. 5.4 is used to calculate impact of such swapping.

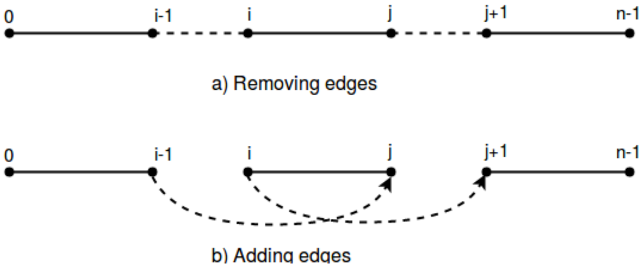


Figure 5.3: Edges affected in swap move when $i = j - 1$ & $j < n - 1$

$$\begin{aligned} change = {} & ((n-i) \times D(P_{i-1}, P_j) + (n-i-2) \times D(P_i, P_{j+1})) \\ & -((n-i) \times D(P_{i-1}, P_i) + (n-i-2) \times D(P_j, P_{j+1})) \end{aligned} \tag{5.4}$$

3) Four edges are affected when $i \neq j - 1$ and $j < n$:

When vertices i and j are not adjacent and j is placed at a position $< n$, four edges are affected in the base solution. Figure 5.4 presents the affected edges on the solution. These affected edges are $(i-1, i)$, $(i, i+1)$, $(j-1, j)$, and $(j, j+1)$. Four new edges are added in following form: $(i-1, j)$, $(j, i+1)$, $(j-1, i)$, and $(i, j+1)$. The impact of applying a swap move on such vertex pair is calculated using Eqn. 5.5.

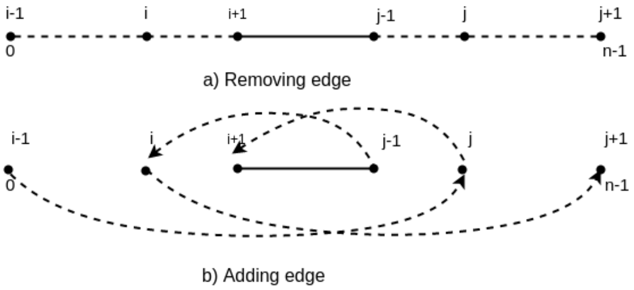


Figure 5.4: Edges affected in swap move when $i \neq j-1$ and $j < n$.

$$\begin{aligned} change = ((n-i) \times D(P_{i-1}, P_j) + (n-i-1) \times D(P_j, P_{i+1}) \\ +(n-j) \times D(P_{j-1}, P_i) + (n-j-1) \times D(P_i, P_{j+1})) \\ -((n-i) \times D(P_{i-1}, P_i) + (n-i-1) \times D(P_i, P_{i+1}) \\ +(n-j) \times D(P_{j-1}, P_j) + (n-j-1) \times D(P_j, P_{j+1})) \end{aligned} \tag{5.5}$$

One of three equations is used while calculating a swap move based on the vertex position. The $change$ is a difference of adding edges and removing edges. If the change is negative, the corresponding swapping pair (i, j) generates a better solution than the base solution. This change is calculated on-the-fly using X, Y coordinates of the corresponding instance. Since each move effect is computed in $O(1)$ time for $\frac{n(n-1)}{2}$ neighbors, the time complexity of swap move becomes $O(n^2)$.

5.2.2 Two-opt Move

In the two-opt move, two edges of the base solution are swapped. Total solutions possible with this move is $\frac{n(n-1)}{2}$, where n is the total clients size excluding depot node. Figure 5.1 (C) shows the pictorial representation of the two-opt move. The order used to remove two edges are $(i, i+1)$ and $(j, j+1)$ respectively, and added back in order (i, j) and $(i+1, j+1)$. The orientation of the base solution changes after applying this neighborhood move. The subsequence from $i+1$ to j is reverted and added back to the solution. Here, the variable number of edges are affected for each move, and hence Eqns. 5.3-5.5 cannot be applied for the two-opt move.

5.2.2.1 Move Evaluation

A constant time move evaluation procedure presented in Silva et al. (2012) can be used for the two-opt move. The procedure requires preprocessing of a local data structure of size $4n^2$. In a GPU based MLP implementation, using the local data structure, the communication time between the CPU and the GPU overshadows the computation time (Rios et al. 2018). Therefore, the total latency for each move is calculated in linear time, instead of using preprocessed data. The time complexity of two-opt move is $O(n^3)$ since each move needs $O(n)$ time, and there are $O(n^2)$ vertex pairs to be evaluated.

When a move is applied, the base solution cannot be changed. The original base solution has to be retained until all the moves are evaluated. A separate array of size n will be needed to store intermediate solution generated after each move. The overall space required to store these intermediate solutions is $n \times \frac{n(n-1)}{2}$. This is avoided in our approach by calculating the latency of each solution using Eqn. 5.2 on the fly.

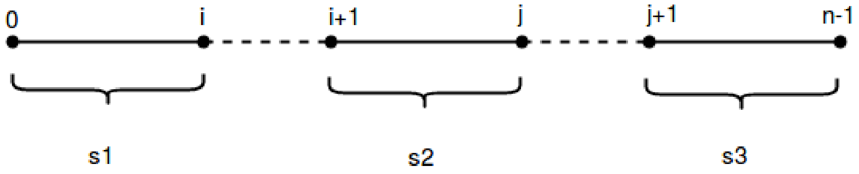


Figure 5.5: Subsequences of the solution after edge removal in two-opt move

The total latency of the generated solution is calculated as follows. Assume a Hamiltonian path of size n. When two-opt is applied on the vertex pair (i, j), the path is separated into three subsequences. Figure 5.5 presents all three possible subsequences of the path when $i \neq j - 1$. When two-opt move is applied on the vertex pair (i, j), where $i = j - 1$, it will be equivalent to the swap move. Therefore, in two-opt, the value of j is considered from $i + 2$ index. First subsequence $s1$ will be from index 0 to i, $s2$ is from $i + 1$ to j, and $s3$ is from the index $j + 1$ to $n - 1$. A new solution is formed by joining $s1$, a reverted sequence of $s2$, and $s3$. The latencies of subsequences are calculated using Algorithm 5.2 to get the total latency of move (i, j). Lines (2-4)

Algorithm 5.2: Move Evaluation Procedure for Two-opt Move

```
    Result: Returns $f(s')$ for the move pair $(i, j)$
 1  d1, d2, d3← 0
 2  for x = 0, y = 1; y ≤ i; x++, y++ do
 3      d1 += (n - y) * D(P[x], P[y]);
 4  end
 5  for x = i, y = j, z = i; y < i; x = y, y − −, z++ do
 6      d2 += (n - z - 1) * D(P[x], P[y]);
 7  end
 8  for y = j + 1, x = i + 1; y < n; x = y, y++ do
 9      d3 += (n - 1) * D(P[x], P[y]);
10  end
11  f(s') = d1 + d2 + d3
```

presents the latency calculation for the subsequence $s1$. $f(s1)$ is calculated by summing the product of $(n - y)$ and $D(P_x, P_y)$ until $x{<}i$, where, $x = 0$ and $y = 1$ initially, later incremented by 1 until $x{<}i$. Function $D(P_x, P_y)$ returns an Euclidean distance between nodes x^{th} and y^{th} of P. For $f(s2)$ calculation, x and y are initialized with value of i, j respectively (lines 5-7). Because j is a first node in reverted $s2$ and has to connect with a last node of $s1$, i.e., i, to join subsequences $s1$ and $s2$ together. Now sum of product of $(n - z - 1)$ and (P_x, P_y) is computed until $z{<}i$. After first iteration, x is assigned with a current value of y, and y is decremented. For the third subsequence $f(s3)$ calculation, the last node of $s2$, i.e., $i + 1$ is initialized to x and y start with first node of $s3$, i.e., $j + 1$ (line 8-10). Now, $f(s3)$ is calculated until $y{<}n$ (line 9). Next, value of y is assigned to x before y increments. Finally, all three $f(s1)$, $f(s2)$, and $f(s3)$ are summed up to get total latency of a solution on-the-fly without using a separate array (line 11).

5.3 PARALLEL DETERMINISTIC LOCAL SEARCH HEURISTIC

The most time-consuming part of the DLSH is the solution improvement phase. The experimental analysis has been performed to segregate time spent in the initial solution construction and solution improvement using TSPLIB instances of size 1000-2392 nodes. Figure 5.6 shows the time portion spent in the solution construction and its improvement. Out of total execution time, more than 99% of the time is spent in the solution improvement phase. DLSH spends less than a few seconds time to build a feasible solution, whereas minutes to hours is needed to get a local optimal. The initial solution is improved using neighborhood generation methods, such as swap and two-opt, which have $O(n^2)$ and $O(n^3)$ time complexity, respectively. These methods are called repeatedly until the local optimal solution is obtained. The overall time complexity of these methods becomes $i \times (n^2 + n^3)$, where i is total calls of the neighborhood methods. Therefore, this work aimed to reduce the time spent in the neighborhood generation methods using a parallel implementation.

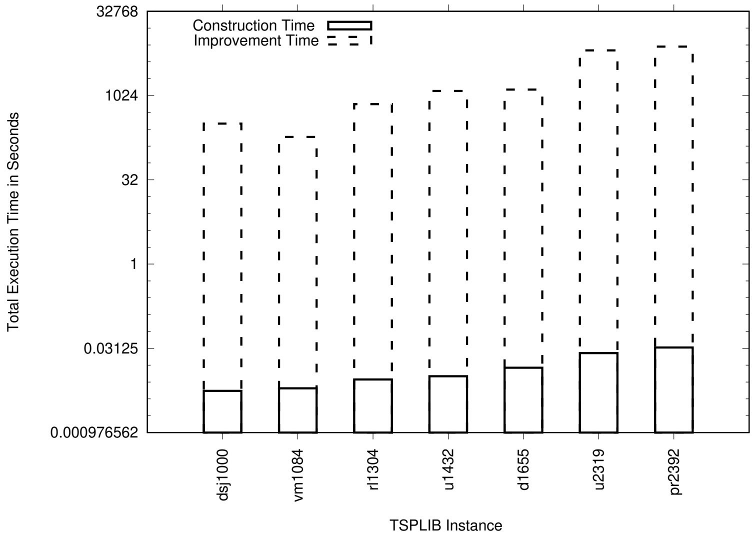


Figure 5.6: Execution time portion analysis in the initial solution construction and improvement.

Parallel DLSH (PDLSH) leverages the GPU's massively parallel processors to execute parallel tasks for the DLSH algorithm. The move evaluations with their independent tasks are a good candidate to execute in parallel. Rios et al. (2018) have implemented MLP over GPU with maximum speedup up to 13.7 times. For the current work, each move evaluation is assigned to a single thread as in (Rios et al., 2018). However, the proposed work overcomes a limitation of the strategy used in the previous work. While evaluating the vertex pair (i, j), the previous work assigns CUDA block id as i and the thread ids inside the block as j. This limits the number of j vertices that can be evaluated with i to the maximum threads allowed in a block (1024, typically). This limitation can be overcome by assigning more evaluations per thread or eliminating node j dependency on the maximum threads per block limit.

Parallel DLSH employs one move evaluation per thread strategy overcoming the maximum threads per block limit. $n(n-1)/2$ threads are used across multiple blocks for $n(n-1)/2$ moves possible with n size instance. Vertex pair (i, j) is generated using the global id of thread using Eqns. 5.6, 5.7 (Luong et al. 2013).

$$i = n - 2 - \left\lfloor (\sqrt{8 * (N(s) - id - 1) + 1} - 1)/2 \right\rfloor \tag{5.6}$$

$$j = id - i * (n - 1) + (i * (i + 1)/2) + 1 \tag{5.7}$$

Thread id	0	1	...	$n-2$	$n-1$	n	...	${}^{n}C_2 - 1$
(i, j) pair	$(0, 1)$	$(0, 2)$	...	$(0, n-1)$	$(1, 2)$	$(1, 3)$	...	$(n-2, n-1)$

Table 5.1: Thread per vertex pair mapping details for neighborhood evaluation methods.

Table 5.1 presents the thread mapping strategy for the corresponding move evaluation pair. The first row indicates global thread ids, and the second row indicates the corresponding vertex pair to be evaluated by the thread. In this strategy, first, $n-2$ threads do not compute move evaluation since the first node 0 has to remain at its original place in the solution.

There are two versions of PDLSH have been designed based on the reduction type. These two versions are presented in Algorithms 5.3 and 5.6. The common steps of both algorithms are illustrated below. The initial solution is created using Nearest Neighborhood (NN) approach, $f(s)$ is computed using Eqn. 5.2, and $N(s)$ indicates the total moves possible with swap and two-opt methods (lines 1-4). Next, GPU memory is allocated, and data is transferred to the GPU, which is required for the GPU computation (line 5). Total threads and blocks required for neighborhood computation are determined (lines 6-12). The remaining steps of algorithms are explained based on their reduction type in the following subsections.

5.3.1 Reduction using Built-in Function

In Algorithm 5.3, the swap and two-opt kernels are called repeatedly until a current solution is trapped in the local optima (lines 13-42). A 64-bit variable $ltcy\_id$ is used to hold the thread id and total latency of the corresponding threads together. At the CPU side, $f(s)$ is bound into the most significant 32 bit (line 14). A kernel swap is called

Algorithm 5.3: Parallel Deterministic Local Search Heuristic using Built-in Function

```
Result: Returns the local optimal solution
$s \leftarrow$ initial solution
$s^{''} \leftarrow s$
$f(s) \leftarrow$ cost calculation
$N(s) \leftarrow n(n-1)/2$
$GPU \leftarrow$ allocate and copy $ltcy\_id$, $s$ $x, y$ coords
if $N(s) < 256$ then
    $thrds \leftarrow N(s)$
    $blks \leftarrow 1$
else
    $thrds \leftarrow 256$
    $blks \leftarrow (N(s)-1)/256+1$
end
while $s$ is improved do
    $ltcy\_id \leftarrow (((long)f(s)+1) << 32)-1$
    $swap <<< blks, thrds >>> (s, x, y, ltcy\_id)$
    $f(s') \leftarrow dst\_tid >> 32$
    while $f(s') < f(s)$ do
        $f(s) \leftarrow f(s')$
        $id = ltcy\_id \& ((1ull << 32)-1)$
        $x = n-2-\left\lfloor (\sqrt{8*(N(s)-id-1)+1}-1)/2 \right\rfloor$
        $y = id - x*(n-1)+(x*(x+1)/2)+1$
        $s \leftarrow move(x, y)$
        $ltcy\_id \leftarrow (((long)f(s)+1) << 32)-1$
        $swap <<< blks, thrds >>> (s, x, y, ltcy\_id)$
        $f(s') \leftarrow dst\_tid >> 32$
    end
    $two-opt <<< blks, thrds >>> (s, x, y, ltcy\_id)$
    $f(s') \leftarrow dst\_tid >> 32$
    while $f(s') < f(s)$ do
        $s \leftarrow s'$
        $id = ltcy\_id \& ((1ull << 32)-1)$
        $x = n-2-\left\lfloor (\sqrt{8*(N(s)-id-1)+1}-1)/2 \right\rfloor$
        $y = id - x*(n-1)+(x*(x+1)/2)+1$
        $s \leftarrow move(x, y)$
        $ltcy\_id \leftarrow (((long)f(s)+1) << 32)-1$
        $two-opt <<< blks, thrds >>> (s, x, y, ltcy\_id)$
        $f(s') \leftarrow dst\_tid >> 32$
    end
    if $f(s) < f(s^{''})$ then
        $s^{''} \leftarrow s$
    end
end
```

Algorithm 5.4: Swap Kernel Pseudocode for Built-in Function

```
   Result: Returns $ltcy\_id$ to CPU
1  $id \leftarrow threadIdx.x + blockIdx.x * blockDim.x$
2  if $id < N(s)$ then
3      $i = n - 2 - \lfloor 8 * (N(s) - id - 1) + 1) - 1)/2 \rfloor$
4      $j = id - i * (n - 1) + (i * (i + 1)/2) + 1$
5      if $i \neq 0$ then
6          $change \leftarrow change(i, j)$ using Eqns. 5.3-5.5
7          if $change<0$ then
8              $cost+ = change$
9              $atomicMin(ltcy\_id, cost << 32|id)$
10         end
11     end
12 end
```

(line 15). Algorithm 5.4 presents the kernel pseudocode for the swap neighborhood approach. A global identification id is assigned to each thread (line 1). Threads with $id<N(s)$ are involved in move evaluation (lines 2-12). The built-in $atomicMin$ is used to find the best-improved solution $cost$ across threads having negative $change$ value (lines 7-10). The $atomicMin$ binds the best improved solution $cost$ and its corresponding thread id in the $ltcy\_id$ variable (line 9).

Returning to Algorithm 5.3, latency $f(s')$ is extracted from $ltcy\_id$ (line 16). If $f(s')<f(s)$, the swap kernel is called (lines 17-26). Thread id of the best found solution is extracted from the lower 32 bits (line 19). The best improved vertex pair (x, y) is extracted from id using 1D to 2D conversion equations (lines 20-21). The move between (x, y) pair is applied on the base solution s (line 22). The updated $f(s)$ is added back in $ltcy\_id$ variable and kernel $swap$ is called recursively (lines 23-25). The two-opt kernel (lines 27-38) and the $swap$ kernel (lines 14-26) differs in their working. The two-opt is illustrated in pseudocode, in Algorithm 5.5. ${}^{n}C_2$ threads are involved in move evaluation and latency of each move is calculated using Algorithm 5.2 (lines 1-11). A thread which gets the minimum latency is chosen, and its id and $cost$ are combined in the $ltcy\_id$ variable using $atomicMin$ (lines 7-10). Finally, Algorithm 5.3 returns the local optimal when further improvement is not possible. CUDA atomic function supports operation on 64-bit numbers for devices having compute capability

Algorithm 5.5: Two-opt Kernel Pseudocode for Built-in Function

```
   Result: Returns $ltcy\_id$ to CPU
1  $id \leftarrow threadIdx.x + blockIdx.x * blockDim.x$
2  if $id < N(s)$ then
3      $i = n - 2 - \lfloor 8 * (N(s) - id - 1) + 1) - 1)/2 \rfloor$
4      $j = id - i * (n - 1) + (i * (i + 1)/2) + 1$
5      if $i \neq 0$ && $i \neq j - 1$ then
6          $f(s') \leftarrow$ calculate latency for $move(i, j)$ using Algo. 5.2
7          if $f(s')<f(s)$ then
8              $atomicMin(ltcy\_id, f(s') << 32|id)$
9          end
10     end
11 end
```

3.5 and above. The $atomicMin$ function is used such that when a 32-bit minimum value is chosen from threads at the same time, a 32-bit thread id of corresponding thread which has minimum value is combined into a 64-bit global memory variable. Therefore, it will be easier to identify which vertex pair (i, j) through the thread id, and has the best-improved solution $f(s)$ without any race condition. However, the strategy of combining two 32-bit numbers into one 64-bit will fail when either the thread id or the $f(s)$ does not fit into 32-bits.

5.3.2 Reduction using Vector

Algorithm 5.6 is an alternative to the Algorithm 5.3, where the best-improved solution and its associated vertex pair (i, j) are chosen manually. To do this, a separate triple min_data vector is required to hold three elements, namely $cost$, i, and j for each thread, involved in the move evaluation. Once the neighborhood kernel finishes its execution, a separate function is used to find the best-improved solution s', i, and j from the triple and written into the first position min_data vector. The swap kernel for this reduction approach is similar to the Algorithm 5.4 except that each thread will write their $cost$, i, and j values to the min_data instead of line 9 in Algorithm 5.4. Similarly, threads will update min_data instead of line 8 in Algorithm 5.5 in case of reduction using vector. Reduction on the vector can be done in two ways, which are explained below.

Algorithm 5.6: Parallel Deterministic Local Search Heuristic using Vector

```
Result: Returns the local optimal solution
$s \leftarrow$ initial solution
$s'' \leftarrow s$
$f(s) \leftarrow$ cost calculation
$N(s) \leftarrow n(n-1)/2$
$GPU \leftarrow$ allocate and copy $min\_data$, $s$ $x, y$ coords
if $N(s) < 256$ then
    $thrds \leftarrow N(s)$
    $blks \leftarrow 1$
else
    $thrds \leftarrow 256$
    $blks \leftarrow (N(s)-1)/256+1$
end
kernel to initialize $min\_data$
while $s$ is improved do
    $swap <<< blks, thrds >>> (s, x, y, min\_data)$
    $find\_min <<< blks, thrds >>> (min\_data)$
    $f(s') \leftarrow min\_data[0].ltcy$
    while $f(s') < f(s)$ do
        $f(s) \leftarrow f(s')$
        $x = min\_data[0].i$
        $y = min\_data[0].j$
        $s \leftarrow move(x, y)$
        $swap <<< blks, thrds >>> (s, x, y, min\_data)$
        $find\_min <<< blks, thrds >>> (min\_data)$
        $f(s') \leftarrow min\_data[0].ltcy$
    end
    $two-opt <<< blks, thrds >>> (s, x, y, min\_data)$
    $find\_min <<< blks, thrds >>> (min\_data)$
    $f(s') \leftarrow min\_data[0].ltcy$
    while $f(s') < f(s)$ do
        $f(s) \leftarrow f(s')$
        $x = min\_data[0].i$
        $y = min\_data[0].j$
        $s \leftarrow move(x, y)$
        $two-opt <<< blks, thrds >>> (s, x, y, min\_data)$
        $find\_min <<< blks, thrds >>> (min\_data)$
        $f(s') \leftarrow min\_data[0].ltcy$
    end
    if $f(s) < f(s'')$ then
        $s'' \leftarrow s$
    end
end
```

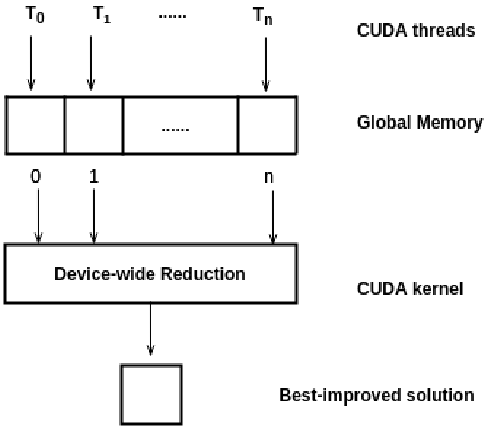


Figure 5.7: Pictorial representation of one-pass reduction.

5.3.2.1 One-pass Reduction

An additional data structure is needed to allocate memory on the GPU device for manual reduction. For one-pass reduction, a triple vector of $^{n}C_{2}$ size is used to hold the result of each thread in each kernel call. This vector is allocated in global memory. The workflow of one-pass reduction is presented in Figure 5.7.

5.3.2.2 Two-pass Reduction

The two-pass reduction is designed to reduce the global memory allocations required in the one-pass method. The pictorial workflow of two-pass reduction is presented in Figure 5.8. All threads within a block write their move evaluation results in the shared memory. When threads finishes move evaluation, the best-improved solution within each block is determined. Subsequently, the details of best-improved solution of each block, i.e., a vertex pair (i, j) and its $cost$, is written to the global memory. Next, device-wide reduction is applied to find the best-improved solution from the global memory. For that a separate kernel function is required. A kernel function that is built to find a minimum is called $log_2 n$ times, where n will be the total blocks used for the neighborhood kernel computation. In this technique, a triple vector of size $block\_width$ is created in the shared memory to hold $cost$, i, and j values of each thread within a block. The triple values of each block are then written to the triple vector of size $total\_blocks$ in the global memory.

The difference between both the one-pass and two-pass reductions is in the memory

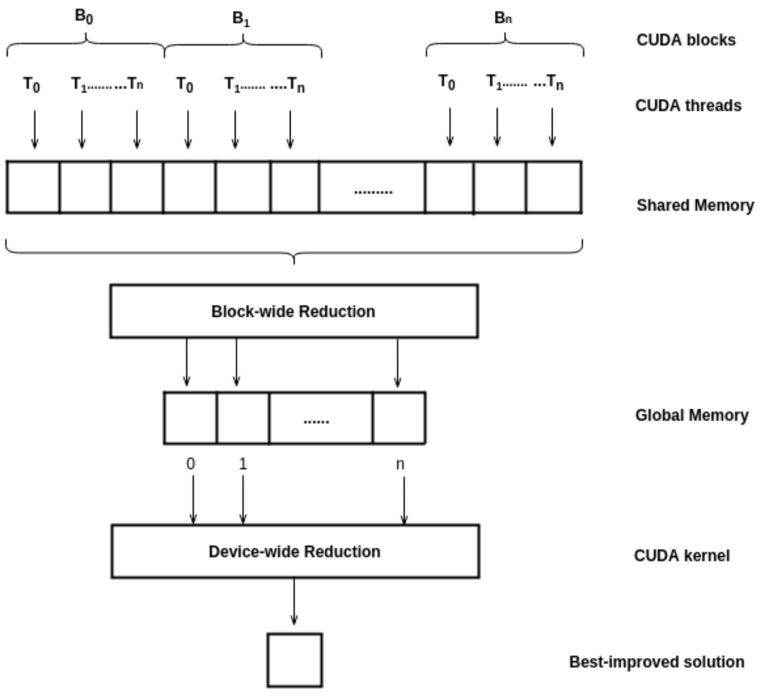


Figure 5.8: Pictorial representation of two-pass reduction.

space allocation. The two-pass method allocates $3 \times n$ memory space in the global memory, where n is total blocks, and $3 \times block\_width$ in the shared memory, where $block\_width$ is total threads in a single block. The one-pass reduction approach allocates $3 \times^n C_2$ memory space in the global memory.

The atomic operation can be used within and across the block. When an atomic operation is applied within a block, it consumes lesser time than the atomic operation is applied across the block. This is because when an atomic operation is applied within a block, the atomic function creates the serialization at the block level. Only threads within the same CUDA block compete for accessing the same shared memory location. Moreover, the shared memory has the least latency. When an atomic operation is applied across the block, the atomic function creates serialization at the device level. Several threads across multiple blocks compete for accessing the same global memory location. The global memory is visible to all threads across all blocks, and it is high latency memory in the GPU memory hierarchy.

5.4 RESULTS ANALYSIS

The performance analysis of DLSH and PDLSH have been tested using the wide variety of TRP (Salehipour et al. 2011) and TSPLIB (Reinelt 1991) instances. Seven sets of TRP instances of size 10, 20, 50, 100, 150, 200, and 500 nodes are used. Each TRP set has 20 randomly generated instances. For time analysis, each instance has been run ten times, and the average execution time is recorded. Moreover, TSPLIB instances have been used to observe the functioning of PDLSH while solving larger instances, i.e., up to 11849 nodes. Note that results are recorded using instances up to 11849 nodes. If any larger instance which has more than 11849 nodes, it can be solved using PDLSH. The hardware specification details used for DLSH and PDLSH analysis is shown in Table 5.2. The sequential implementation of DLSH was run on the Intel Core i7, which has eight cores running at 3.6 GHz, and a RAM of size 16 GB. The sequential DLSH is coded in C language, and parallel DLSH has been coded in the CUDA framework (version: CUDA 10.1). The specifications of GPU which is used for executing the parallel code are followed. A Tesla K40m GPU card is used, which has compute capability version 3.5, the global memory of size 12 GB, 15 Streaming Multiprocessors (SMs), and each SM has 192 cores running at 745 MHz. The performance analysis of DLSH and PDLSH is explained in the following subsections.

Table 5.2: Implementation details for DLSH and PDLSH with hardware specifications. The DLSH and PDLSH column shows the platform used for corresponding implementation.

Description	DLSH	PDLSH
Language	C	CUDA
CPU / GPU	Core i7-4790	Tesla K40m
Architecture	Haswell	Kepler
Streaming Multiprocessor	NA	15
Cores	8	2880
Frequency	3.6 GHz	0.75 GHz
Global Memory	16 GB	12 GB
Shared Memory	NA	48 KB

5.4.1 Deterministic Local Search Heuristic (DLSH)

The solution quality obtained with DLSH is presented in Tables 5.3-5.6. Table 5.3 presents the solution obtained on the TRP instances of size 10, 20 and 50 nodes. The first column represents 20 instances for each size. Columns 2-6 represent the initial solution (i.e., obtained using NN), local optimal solution, number of times a pair of neighborhood methods (i.e., swap and two-opt) is called, optimal solution, and the gap rate from an optimal solution, respectively, for TRP size 10. Columns 7-11 and 12-16 represents the same information as columns 2-6 for 20 and 50 size instances. The optimal solutions for TRP sets of size 10, 20, and 50 are known (Silva et al. 2012). The gap percentage is used to compute the farness of local solution from the optimal solution. The gap rate is calculated as $(Local - Opt)/Opt * 100$.

From Table 5.3, it is noticed that DLSH builds an optimal solution for 11 instances for 10-size set. Out of 11 optimal solutions, eight solutions are reached in the solution construction phase itself. For the remaining three optimal solutions, the neighborhood generation pair is called at least twice in order to reach global optimal solutions. For a 10-size data set, nine instances trap in the local optima. The gap rate observed in these nine instances is in the range of $0.20 - 8.38\%$. DLSH obtains optimal solutions for four instances, namely R12, R16, R17, and R20, respectively, for a 20-size data set. DLSH provides near-optimal solutions for the other 16 instances, which has a gap rate of up to 11.67%. DLSH could not converge to an optimal solution in a 50-size data set. The local solutions found in the 50-size data set has a 2.02% gap rate in the best-case and 13.48% in the worst-case.

From these results, it can be inferred that an optimal solution may not be reached as instance size increases. This is because, DLSH explores a limited search-space and stops exploring the feasible solutions when traps into a local optima. The number of feasible solutions explored in the DLSH search-space are: $i \times (j \times^n C_2 + k \times^n C_2)$, where n is instance size, i represents number of times a neighborhood pair (i.e., swap and two-opt) is called, j and k indicates the number of times swap and two-opt is called respectively for each i call. For example, consider an instance R1 from a 50-size data set. DLSH produces a local solution (i.e., 13842) after exploring 39200 (i.e.,

$2\times(6\times{}^{50}C_2+10\times{}^{50}C_2) = 39200$) feasible solutions with a gap rate of 13.48%. Exact methods have to explore either $O(2^n)$ or $O(n!)$ search-space to determine an optimal solution for the same instance. Therefore, metaheuristic algorithms are used to find a near-optimal solution in lesser time. As far as execution time is concerned, DLSH spends a negligible time (i.e., few milliseconds) to get local solutions for these 10, 20, and 50 size data sets.

Table 5.3: Solution quality analysis for TRP instances of size 10, 20, and 50 using DLSH. Abbreviations used- I: Instance name, C: Calls;

I	10					20					50				
	Init	Local	C	Opt	Gap %	Init	Local	C	Opt	Gap %	Init	Local	C	Opt	Gap %
R1	1328	1328	1	1303	1.92	3356	3181	2	3175	0.19	15530	13842	2	12198	13.48
R2	1520	1520	1	1517	0.20	3305	3290	2	3248	1.29	13693	13046	2	11621	12.26
R3	1233	1233	1	1233	0.00	4244	3621	2	3570	1.43	15058	13238	2	12139	9.05
R4	1395	1395	1	1386	0.65	3416	3331	2	2983	11.67	14656	13605	2	13071	4.09
R5	1060	1060	1	978	8.38	3521	3322	2	3248	2.28	12955	12713	2	12126	4.84
R6	1477	1477	1	1477	0.00	3479	3461	2	3328	4.00	13638	13325	2	12684	5.05
R7	1221	1169	2	1163	0.52	2910	2910	1	2809	3.60	14123	11725	2	11176	4.91
R8	1234	1234	1	1234	0.00	3741	3627	2	3461	4.80	13281	13171	2	12910	2.02
R9	1402	1402	1	1402	0.00	3654	3626	2	3475	4.35	14141	13739	2	13149	4.49
R10	1410	1394	2	1388	0.43	3582	3582	1	3359	6.64	14291	13164	2	12892	2.11
R11	1405	1405	1	1405	0.00	3253	3136	2	2916	7.54	13849	13574	2	12103	12.15
R12	1164	1150	2	1150	0.00	4227	3314	2	3314	0.00	11256	11058	2	10633	4.00
R13	1561	1561	1	1531	1.96	3698	3560	2	3412	4.34	13386	12747	2	12115	5.22
R14	1442	1219	2	1219	0.00	3935	3450	2	3297	4.64	14160	13586	2	13117	3.58
R15	1129	1129	1	1087	3.86	3369	2871	2	2862	0.31	13394	12522	2	11986	4.47
R16	1315	1315	1	1264	4.03	3909	3433	2	3433	0.00	13596	13285	2	12138	9.45
R17	1058	1058	1	1058	0.00	3175	2913	2	2913	0.00	14172	13126	2	12176	7.80
R18	1083	1083	1	1083	0.00	3288	3262	2	3124	4.42	14435	13953	2	13357	4.46
R19	1482	1394	2	1394	0.00	4441	3659	3	3299	10.91	12398	12316	2	11430	7.75
R20	951	951	1	951	0.00	3140	2796	2	2796	0.00	12745	12612	2	11935	5.67
Min					0					0					2.02
Avg					1.10					3.62					6.34
Max					8.38					11.67					13.48

Table 5.4 presents the solution quality and execution time analysis for 20 instances of each 100, 150, and 200 size data set. There is no known optimal solutions are available for instance size ≥ 100. Therefore, an improvement percentage (i.e., local gap $\%$) is calculated from the initial solution to the local optima as $(Local - Init)/Init * 100$. In the best-case, the improvement rates observed in DLSH are -14.57%, -12.64%, and -12.06% for 100, 150, and 200 size data sets, respectively. There is one constraint applied in the DLSH. The constraint is to maintain the deterministic solution irrespective of multiple runs for the same instance. This makes DLSH improve the solution until there is no further improvement is possible and hence DLSH stops when the local

Table 5.4: Solution quality and execution time details for TRP instances of size 100, 150, and 200 using DLSH. Time is given in seconds. Abbreviations used- I: Instance name, C: Calls, LG: Local Gap;

I	100					150					200				
	Init	Local	C	LG %	Time	Init	Local	C	LG %	Time	Init	Local	C	LG %	Time
R1	34726	34255	2	-1.36	0.025	61256	57800	2	-5.64	0.175	109130	98480	2	-9.76	0.680
R2	38189	34977	2	-8.41	0.069	73855	66598	3	-9.83	0.294	111697	98230	3	-12.06	1.176
R3	37642	33404	2	-11.26	0.055	67767	64205	2	-5.26	0.212	113545	103959	3	-8.44	0.544
R4	40316	35631	2	-11.62	0.087	68260	59631	2	-12.64	0.211	104810	101155	2	-3.49	0.362
R5	36017	35050	2	-2.68	0.016	68538	61524	2	-10.23	0.191	102359	94078	3	-8.09	0.626
R6	36298	35805	3	-1.36	0.019	66783	64108	2	-4.01	0.120	113768	102947	2	-9.51	0.559
R7	39236	37570	2	-4.25	0.030	68003	62507	3	-8.08	0.263	103359	96995	2	-6.16	0.470
R8	38749	33103	2	-14.57	0.053	68802	64799	2	-5.82	0.193	100896	94898	2	-5.94	0.510
R9	41262	36354	2	-11.89	0.055	69749	64354	2	-7.73	0.151	106238	96319	2	-9.34	0.969
R10	37860	32545	2	-14.04	0.048	69934	64508	2	-7.76	0.162	105386	97113	2	-7.85	0.728
R11	38962	37077	2	-4.84	0.038	71069	66766	2	-6.05	0.118	101519	97026	2	-4.43	0.646
R12	35032	33670	2	-3.89	0.039	62843	59766	2	-4.90	0.189	106774	99578	2	-6.74	0.822
R13	38406	35416	2	-7.79	0.022	72133	66538	2	-7.76	0.200	99401	92648	2	-6.79	0.527
R14	34517	32601	2	-5.55	0.022	70496	65155	2	-7.58	0.100	110028	100741	2	-8.44	0.580
R15	37769	35559	2	-5.85	0.032	67276	63180	2	-6.09	0.166	105445	97719	2	-7.33	0.623
R16	39860	37642	2	-5.56	0.029	65796	65181	2	-0.93	0.067	103499	95245	2	-7.97	0.710
R17	40749	39763	2	-2.42	0.029	66970	62379	2	-6.86	0.279	105663	93552	2	-11.46	0.589
R18	38839	35862	2	-7.66	0.053	71622	68349	2	-4.57	0.116	104599	98375	2	-5.95	0.395
R19	39752	37178	2	-6.48	0.020	72316	67204	2	-7.07	0.077	107054	103036	2	-3.75	0.339
R20	39346	35128	2	-10.72	0.049	73719	68334	2	-7.30	0.336	95935	92868	2	-3.20	0.337

Table 5.5: Solution quality and execution time details for TRP instances of size 500 using DLSH. Time is given in seconds.

I	500				
	Init	Local	C	LG %	Time
R1	2256050	2045627	3	-9.33	26.14
R2	2238193	2062200	3	-7.86	15.24
R3	2227313	2035358	3	-8.62	18.16
R4	2119535	2002354	2	-5.53	14.75
R5	2253349	2013264	2	-10.65	35.88
R6	2109395	1970414	2	-6.59	9.08
R7	2185996	2044015	2	-6.50	18.88
R8	2149991	2005965	3	-6.70	18.58
R9	2074414	1867533	2	-9.97	16.94
R10	2045557	1935231	2	-5.39	11.38
R11	2226155	1993875	3	-10.43	13.96
R12	2091972	1943465	2	-7.10	13.58
R13	2246410	2038807	2	-9.24	13.98
R14	2104882	1980946	2	-5.89	19.21
R15	2067337	1930097	3	-6.64	14.65
R16	2072183	1999019	2	-3.53	9.97
R17	2072851	1969347	2	-4.99	14.53
R18	2136634	2059754	2	-3.60	18.61
R19	2081332	1960612	3	-5.80	16.85
R20	2191334	2000517	2	-8.71	12.44

Table 5.6: Solution quality and execution time details for TSPLIB instances using DLSH. Time is given in seconds.

Instance	Init	Local	C	LG %	Time
berlin52	144761	139316	2	-3.76	0.004
st70	21844	20278	2	-7.17	0.020
rat99	61575	60074	2	-2.44	0.056
kroA100	1176490	996163	2	-15.33	0.067
kroB100	1159631	1011943	2	-12.74	0.086
kroD100	1081377	986717	2	-8.75	0.062
kroE100	1074901	1025899	2	-4.56	0.037
lin105	692956	645135	2	-6.90	0.070
pr107	2078587	1983521	2	-4.57	0.031
ch130	376931	356674	2	-5.37	0.117
ch150	483528	465248	3	-3.78	0.137
kroA150	2269515	1891421	3	-16.66	0.224
kroB150	1954630	1870029	2	-4.33	0.164
rat195	221834	211049	2	-4.86	0.242
d198	1312478	1209289	2	-7.86	0.616
ts225	14139523	13649963	2	-3.46	0.647
pr226	11124323	7181227	3	-35.45	2.188
a280	395607	367765	2	-7.04	1.511
pr299	7546036	7204856	2	-4.52	1.139
lin318	7022116	6117163	3	-12.89	4.146
pr439	21760303	19084000	2	-12.30	7.900
pcb442	10765404	10657722	2	-1.00	4.441
d493	9675285	7700153	3	-20.41	27.984
att532	25085998	19160615	3	-23.62	30.771
ali535	372174	309613	2	-16.81	31.733
rat575	2052289	1947078	2	-5.13	26.340
d657	16547216	15379075	3	-7.06	55.597
rat783	3711989	3470429	2	-6.51	62.613
dsj1000	9819816218	8646553613	3	-11.95	347.642
vm1084	129225071	105275207	3	-18.53	200.898
rl1304	178715703	162008652	2	-9.35	742.259
u1432	118600489	111871593	3	-5.67	1305.046
d1655	53432497	49464859	2	-7.43	1377.627
u2319	287165931	276381186	3	-3.76	6882.538
pr2392	501222898	464856059	3	-7.26	8352.751
pcb3038	228422815	209522527	3	-8.27	20904.295
fnl4461	429244345	406179078	3	-5.37	89308.930
rl5934	1721991412	1633486986	3	-5.14	155700.828
pla7397	105277496147	70428563340	4	-33.10	729210.250
rl11849	5937577967	5448313366	3	-8.24	3356448
Average				-9.73	109276.00

optima is found. DLSH spends a less than second time for executing an instance of 100, 150, and 200 size sets.

Table 5.5 shows a 500-size data set that consumes 9-36 CPU seconds to reach local optima for a single instance. The improvement rate observed for this data set is in the range of -3.53 to -10.65%. Although instance size is the same, i.e., 500 nodes, every instance spends unequal execution time. This happens due to 500 nodes are placed on different positions in each instance. Therefore, the number of feasible solutions explored and time consumed in each instance is purely independent and cannot be related. From Table 5.5, it is noticed that the execution time increases as instance size increases. These execution times required for exploring the feasible solutions can be reduced.

DLSH has been evaluated on the wide range of TSPLIB instances to identify the execution time required for very-large instances, i.e., up to 11849 nodes. The solution quality and execution time required for these TSPLIB instances are presented in Table 5.6. DLSH has a -35.44% improvement rate in the best-case for $pr226$ instance. DLSH reaches a -1.00% improvement in the worst-case for a $pcb442$ instance. From Table 5.6, it is noticed that the size and improvement rate cannot be correlated. The second-best improvement rate is found for a $pla7397$ instance, which has a -33.10% improvement rate. When the execution time is concerned, the size and execution time requirement can be correlated. DLSH produces a local optima in a few milliseconds for smaller instances up to 200 nodes. For medium-size instances, i.e., > 200 and < 1000 nodes, linear growth is observed in the execution time. When instance size goes beyond 1000 nodes, DLSH significantly consumes a lot of CPU time to obtain a local optima. For example, DLSH spends 729210.25 seconds to solve a $pla7397$ instance, i.e., eight days approximately. For a $rl11849$ instance, DLSH spends approximately 39 days to get local optima. The time required for such large instances is inadmissible, which needs a consistent power supply. Therefore, a GPU-based parallel strategy is proposed to reduce the execution time for solving such large-scale instances.

5.4.1.1 Comparison with State-of-the-art MLP Solvers

DLSH has been compared with the state-of-the-art metaheuristic MLP solvers to evaluate the solution quality. Table 5.7 present the solution quality comparison of DLSH with Salehipour et al. (2011) using TSPLIB instances. The first row indicates an instance name, Salehipour local solution, DLSH local solution, and gap rate observed in

DLSH solution from Salehipour local solution, respectively. Salehipour et al. (2011) presents the local solutions for TSPLIB instances of size up to 532 nodes. Salehipour metaheuristic produces better local solutions than DLSH. This is because Salehipour metaheuristic uses five neighborhood generation methods, namely swap, swap-adjacent, two-opt, or-opt, and remove-insert, to explore feasible solutions for finding better local optima. In the other side, DLSH uses two neighborhood generation methods. There is a high possibility of getting a better solution when a large search-space is explored. Although DLSH explores limited search-space, it finds a new best solutions for two instances, namely $rat195$ and $pr226$. These two solutions indicate that a better solution may be obtained with limited search-space exploration as well. For other TSPLIB instances, DLSH has a gap rate in the range of 0.002-10.12% compared to Salehipour metaheuristic.

Table 5.7: DLSH solution quality comparison with Salehipour et al. (2011) for TSPLIB instances.

Instance	Salehipour	DLSH	Gap %
st70	19553	20278	3.708
rat99	56994	60074	5.404
kroD100	976830	986717	1.012
lin105	585823	645135	10.125
pr107	1983475	1983521	0.002
rat195	213371	211049	-1.088
pr226	7226554	7181227	-0.627
lin318	5876537	6117163	4.095
pr439	18567170	19084000	2.784
att532	18448435	19160615	3.860

DLSH is compared with another two state-of-the-art MLP solvers, namely Abeledo et al. (2013) and Silva et al. (2012), where TSPLIB instances are used for performance evaluation. TSPLIB instances with edge type EUC_2D are considered for comparison. Table 5.8 presents the solution quality received using DLSH, Abeledo et al. (2013), and Silva et al. (2012) for TSPLIB instances of size 51-107 nodes. DLSH obtains new best solutions for five instances, $eil51$, $berlin52$, $st70$, $eil76$, and $pr107$. For the remaining ten instances, DLSH has a gap rate in the range of 0.88-6.83%.

Salehipour et al. (2011) have developed MLP benchmarking data sets, which are

Table 5.8: DLSH solution quality comparison with Abeledo et al. (2013) and Silva et al. (2012) for TSPLIB instances.

Instance	DLSH	Abeledo		Silva	
		Sol	Gap %	Sol	Gap %
eil51	9908	10178	-2.65	10178	-2.65
berlin52	139316	143721	-3.06	143721	-3.06
st70	20278	20557	-1.36	20557	-1.36
eil76	17914	17976	-0.34	17976	-0.34
pr76	3485539	3455242	0.88	3455242	0.88
rat99	60074	58288	3.06	57986	3.60
kroA100	996163	983128	1.33	983128	1.33
kroB100	1011943	986008	2.63	986008	2.63
kroC100	1002165	961324	4.25	961324	4.25
kroD100	986717	976965	1.00	976965	1.00
kroE100	1025899	971266	5.62	971266	5.62
rd100	359559	340047	5.74	340047	5.74
eil101	27816	27519	1.08	27513	1.10
lin105	645135	603910	6.83	603910	6.83
pr107	1983521	2026626	-2.13	2026626	-2.13

also known as TRP instances. The solution quality of DLSH is further compared with Salehipour et al. (2011) using TRP instances. Table 5.9 presents DLSH solution, GILS-RVND (Silva et al. 2012) solution, and the gap rate (i.e., $(DLSH - GILS\_RVND)/GILS\_RVND * 100$) for 20 different instances of 100, 200, and 500 size data sets. It is observed that GILS-RVND outperforms over the DLSH. DLSH has 0.52%, 4.57%, and 7.71% gap rates in the best-case, whereas 16.27%, 12.40%, and 13.52% in the worst-case for 100, 200, and 500 instance sets, respectively compared to the GILS-RVND approach. GILS-RVND obtains a better solution quality since a large search-space is explored in the GILS-RVND approach.

There are five neighborhood generation methods, namely swap, two-opt, or-opt1, or-opt2, and or-opt3, are used in the GILS-RVND. The initial solution is constructed ten times.These five neighborhood methods are applied to improve the solution after each initial solution is constructed. If any neighborhood method finds an improvement, these five methods are repeatedly applied. If improvement is not observed in these five methods, the perturbation technique is applied at least $min(100, n)$ times

Table 5.9: Solution quality comparison with Silva et al. (2012) for TRP instances of size 100, 200, and 500.

Instance	100			200			500		
	DLSH	GILS-RVND	Gap %	DLSH	GILS-RVND	Gap %	DLSH	GILS-RVND	Gap %
R1	34255	32779	4.50	98480	88787	10.92	2045627	1841386	11.09
R2	34977	33435	4.61	98230	91977	6.80	2062200	1816568	13.52
R3	33404	32390	3.13	103959	92568	12.31	2035358	1833044	11.04
R4	35631	34733	2.59	101155	93174	8.57	2002354	1809266	10.67
R5	35050	32598	7.52	94078	88737	6.02	2013264	1823975	10.38
R6	35805	34159	4.82	102947	91589	12.40	1970414	1786620	10.29
R7	37570	33375	12.57	96995	92754	4.57	2044015	1847999	10.61
R8	33103	31780	4.16	94898	89048	6.57	2005965	1820846	10.17
R9	36354	34167	6.40	96319	86326	11.58	1867533	1733819	7.71
R10	32545	31605	2.97	97113	91552	6.07	1935231	1762741	9.79
R11	37077	34188	8.45	97026	92655	4.72	1993875	1797881	10.90
R12	33670	32146	4.74	99578	91457	8.88	1943465	1774452	9.52
R13	35416	32604	8.62	92648	86155	7.54	2038807	1873699	8.81
R14	32601	32433	0.52	100741	91882	9.64	1980946	1799171	10.10
R15	35559	32574	9.16	97719	88912	9.91	1930097	1791145	7.76
R16	37642	33566	12.14	95245	89311	6.64	1999019	1810188	10.43
R17	39763	34198	16.27	93552	89089	5.01	1969347	1825748	7.87
R18	35862	31929	12.32	98375	93619	5.08	2059754	1826263	12.79
R19	37178	33463	11.10	103036	93369	10.35	1960612	1779248	10.19
R20	35128	33632	4.45	92868	86292	7.62	2000517	1820813	9.87
Min			0.52			4.57			7.71
Max			16.27			12.40			13.52
Average			7.05			8.06			10.18

to get out of trapping into local optima. In mathematical term, GILS-RVND explores $10 * min(100, n) * i * ({}^{n}C_2 +^{n} C_2 + (n-1) + (n-2) + (n-3))$ feasible solutions to obtain its local optima. The terms ${}^{n}C_2$, ${}^{n}C_2$, $n-1$, $n-2$ and $n-3$ indicates total feasible solutions possible with swap, two-opt, or-opt1, or-opt2, and or-opt3, respectively. Unlike the GILS-RVND, DLSH starts with a single initial solution, and later two neighborhood approaches are repeatedly called until no further improvement is possible. DLSH explores $i * (j *^{n} C_2 + k *^{n} C_2)$ feasible solutions to produce local solution. Therefore, there will be a high possibility of getting a better solution in GILS-RVND.

5.4.2 Parallel Deterministic Local Search Heuristic (PDLSH)

The execution time of PDLSH has been collected for TRP and TSPLIB instances of size up to 11849 nodes. The execution time comprises the total time required for reading X, Y coordinates of an instance, its initial solution setup, and the improvement phase. This section presents the time analysis of both DLSH and PDLSH versions. Moreover, the impact of applying different optimization techniques to PDLSH is also presented.

The solution quality analysis of PDLSH is not presented since PDLSH obtains the same local optima as DLSH.

Table 5.10: Speedup analysis of PDLSH over DLSH for TRP instances of size 100, 200 and 500. Time is given in seconds.

Instance	100			200			500		
	DLSH	PDLSH	Speedup	DLSH	PDLSH	Speedup	DLSH	PDLSH	Speedup
R1	0.025	0.05	0.50	0.680	0.07	9.71	26.14	0.39	67.02
R2	0.069	0.05	1.38	1.176	0.08	14.70	15.24	0.25	60.96
R3	0.055	0.05	1.09	0.544	0.07	7.77	18.16	0.3	60.55
R4	0.087	0.06	1.45	0.362	0.07	5.17	14.75	0.25	59.02
R5	0.016	0.05	0.33	0.626	0.08	7.82	35.88	0.48	74.75
R6	0.019	0.05	0.39	0.559	0.07	7.99	9.08	0.18	50.44
R7	0.030	0.09	0.34	0.470	0.07	6.72	18.88	0.3	62.94
R8	0.053	0.07	0.75	0.510	0.06	8.50	18.58	0.3	61.94
R9	0.055	0.06	0.91	0.969	0.09	10.77	16.94	0.27	62.73
R10	0.048	0.06	0.80	0.728	0.06	12.14	11.38	0.2	56.92
R11	0.038	0.06	0.63	0.646	0.07	9.23	13.96	0.25	55.85
R12	0.039	0.06	0.65	0.822	0.08	10.28	13.58	0.24	56.59
R13	0.022	0.06	0.36	0.527	0.08	6.59	13.98	0.25	55.90
R14	0.022	0.04	0.55	0.580	0.08	7.25	19.21	0.31	61.98
R15	0.032	0.05	0.64	0.623	0.08	7.78	14.65	0.26	56.33
R16	0.029	0.05	0.57	0.710	0.08	8.87	9.97	0.2	49.84
R17	0.029	0.06	0.48	0.589	0.08	7.36	14.53	0.24	60.56
R18	0.053	0.05	1.06	0.395	0.07	5.65	18.61	0.3	62.04
R19	0.020	0.04	0.51	0.339	0.06	5.65	16.85	0.27	62.39
R20	0.049	0.06	0.82	0.337	0.07	4.81	12.44	0.21	59.25
Average			0.71			8.24			59.90

5.4.2.1 DLSH vs. PDLSH

From Section 5.4.1, it is observed that DLSH spends a few milliseconds to solve instances of less than 100 nodes. Therefore, TRP instances ≥ 100 are considered for comparative analysis. Table 5.10 presents a comparison between the execution time of DLSH and PDLSH for TRP instances. In Table 5.10, columns 2-4 represents DLSH execution time, PDLSH execution time, and speedup factor for 100-size data set, whereas columns 5-7 and 8-10 presents the same information for 200 and 500-size data sets, respectively. The speedup is calculated dividing the sequential execution time by the parallel execution time of the corresponding instance. PDLSH spends more time in execution than DLSH for a 100-size data set. This happens due to the communication time overhead. For smaller instances, communication time overshadows the actual

Table 5.11: Speedup analysis of PDLSH over DLSH for TSPLIB instances. Time is given in seconds.

Instance	DLSH	PDLSH	Speedup
berlin52	0.004	0.046	0.08
st70	0.020	0.046	0.43
rat99	0.056	0.049	1.14
kroA100	0.067	0.052	1.28
kroB100	0.086	0.051	1.69
kroD100	0.062	0.051	1.21
kroE100	0.037	0.05	0.74
lin105	0.070	0.052	1.35
pr107	0.031	0.053	0.59
ch130	0.117	0.053	2.21
ch150	0.137	0.053	2.58
kroA150	0.224	0.057	3.93
kroB150	0.164	0.056	2.92
rat195	0.242	0.055	4.41
d198	0.616	0.062	9.94
ts225	0.647	0.061	10.61
pr226	2.188	0.094	23.28
a280	1.511	0.073	20.70
pr299	1.139	0.071	16.05
lin318	4.146	0.114	36.37
pr439	7.900	0.136	58.08
pcb442	4.441	0.101	43.97
d493	27.984	0.345	81.11
att532	30.771	0.382	80.55
ali535	31.733	0.412	77.02
rat575	26.340	0.308	85.52
d657	55.597	0.569	97.71
rat783	62.613	0.558	112.21
dsj1000	347.642	1.934	179.75
vm1084	200.898	2.00	100.45
rl1304	742.259	6.83	108.74
u1432	1305.046	9.46	138.00
d1655	1377.627	10.85	126.92
u2319	6882.538	46.08	149.35
pr2392	8352.751	53.94	154.85
pcb3038	20904.295	142.03	147.18
fnl4461	89308.930	826.95	108.00
rl5934	155700.828	1282.93	121.36
pla7397	729210.25	5605.88	130.08
rl11849	3356448	36272.12	92.53
min			0.08
max			179.75
average			57.50

computation time. The communication time means a time required for transferring data from CPU to GPU and vice-versa. This indicates that the smaller size instances are not suitable to run over the GPU platform. For a 200-size data set, PDLSH start showing its influence. PDLSH is five times faster compared to DLSH for R4 instance in the worst-case. In the best-case, PDLSH is 14.70 times faster for an R2 instance. PDLSH receives speedup in the range of 49.84-74.75× for a 500-size data set. From Table 5.10, it is noticed that PDLSH outperforms when the instance size increase.

Table 5.11 presents the execution time comparison between PDLSH and DLSH for TSPLIB instances of size 52-11849 nodes. Up to $pr107$, PDLSH suffers communication time overhead. PDLSH has obtained speedups of up to 179.75× over corresponding sequential DLSH. This highest speedup factor is recorded while running TSPLIB instance $dsj1000$ where DLSH consumes 347.644 seconds, and PDLSH gives the same result in 1.934 seconds. On average, PDLSH obtains 57.50 speedups for 39 instances of size 52-11849 nodes. DLSH spends CPU times in days for solving instances namely $fnl4461$, $rl5934$, $pla7397$, and $rl11849$ instances. The most execution times are required for $pla7397$ and $rl11849$ instances, which are 8 and 39 days approximately. These same instances are solved using PDLSH in hours. PDLSH consumes 1.55 and 10.07 hours to solve $pla7397$ and $rl11849$ instances, respectively.

PDLSH could achieve a significant time reduction due to the proposed parallel strategy. The proposed parallel strategy allows a large number of threads to participate in neighborhood generation techniques. PDLSH maps one thread per neighborhood evaluation. The swap and two-opt moves evaluate ${}^{n}C_2$ neighborhood moves per call. For example, consider an instance $rl11849$. PDLSH allows 70193476 threads across 274194 CUDA blocks to evaluates neighborhood moves simultaneously for each swap and two-opt call. Since a large amount of threads compute neighborhood moves simultaneously, PDLSH obtains a local optimum in 10.07 hours compared to 39 days of DLSH time.

5.4.2.2 Reduction Methods

Table 5.12 presents the time analysis of reduction methods. The large-scale TSPLIB instances of size 1084-5934 nodes have been considered for this analysis. For instances up to $pcb3038$ nodes, the built-in reduction method needs a slightly lesser time than others. This is because the vector method first needs to write the triple values of threads into the shared memory or global memory. Later, a separate minimum finding kernel is called $log_2 n$ times. In each kernel call, these data from the GPU memory are read multiple times until a minimum triple value is determined. These multiple kernel calls from the CPU result in a slight increase in the execution time for the vector method. In the built-in method, each thread reads the 64-bit value written in the global memory, compares it with its triple values, and writes to the same address.

Table 5.12: Execution times of built-in library API and vector reduction methods. Time is given in seconds.

Instance	vm1084	rl1304	u1432	d1655	u2319	pr2392	pcb3038	fnl4461	rl5934
Built-in	1.91	6.65	8.47	10.34	42.27	48.03	132.92	833.64	1289.46
Vector	2.00	6.83	9.46	10.85	46.08	53.94	142.03	826.95	1282.93

However, once the instance size goes beyond $pcb3038$ size, the vector reduction method performs better. This is because $atomicMin$ function is applied on all threads across all the blocks to find the minimum latency and the corresponding thread id. When threads write to the same memory address at the same time, $atomicMin$ function runs serially. The $atomicMin$ function consumes $O(n)$ time to find minimum triple values, where n is total threads used in the computation. The vector reduction method needs $O(log_2 n)$ time to find the minimum triple values, where n will be either total threads or total blocks, dependent on the reduction types. For one pass reduction, n will be total threads, and for two-pass reduction, n is total blocks required for solving a particular instance.

The disadvantage of the built-in reduction method is that when the value of either thread id or latency of corresponding vertex pair (i, j) does not fit in the unsigned 32-bit number, this reduction approach will fail. This is because the $atomicMin$ function supports reduction on either 32-bit or 64-bit numbers. Here, two unsigned 32-bit numbers are bound into a single 64-bit number and write it into the global memory while choosing a minimum using $atomicMin$ function to avoid the race condition. The range of unsigned 32-bit number is 0 to 4294967295. For example, the initial solution of $pla7397$ is 70428563340, which cannot fit in the unsigned 32-bit number and hence the binding of thread id and latency in $atomicMin$ function will fail. Therefore, results for $pla7397$ and $rl11849$ instances are not provided in Table 5.12.

5.4.2.3 Vector Reduction Types

The claim from (Rios et al. 2018) is tested that the serial version of choosing the minimum triple values is faster than a reduction kernel on the vector data. Two strategies have been used for a reduction on the GPU: one-pass and two-pass reduction. Table 5.13 presents the execution time required for TSPLIB instances for each of these reductions. Up to the $rl1304$ instance, the single-thread approach consumes less time than the GPU reduction kernels. When larger instances are considered, i.e., $fnl4461 - rl11849$, two pass reduction performs slightly better than the single-thread reduction. For these larger instances, two-pass has time improvements in milliseconds to a few seconds over the single-threaded approach. The one-pass method performs better than the other two methods for medium-size TSP instances (i.e., $u1432 - pcb3028$). The difference of execution time between the single-thread and two-pass approaches is in range of -0.057 to 8.15 seconds. The single-thread and one-pass have a gap between -305.71 to 65.77 seconds. This is because, although one-pass and two-pass perform the reduction in parallel, the reduction kernel is called $log_2 n$ times, CPU to GPU and vice-versa data

Table 5.13: Execution times of vector reduction types: one-pass, two-pass, and single thread. Time is given in seconds.

Instance	GPU		CPU
	Two-pass	One-pass	Single-thread
vm1084	2.00	2.02	1.95
rl1304	6.83	6.83	6.78
u1432	9.46	8.72	9.40
d1655	10.85	10.49	10.83
u2319	46.08	42.43	46.04
pr2392	53.94	48.40	53.89
pcb3038	142.03	132.94	141.99
fnl4461	826.95	834.84	826.98
rl5934	1282.93	1287.47	1283.24
pla7397	5605.88	5542.17	5607.94
rl11849	35457.14	35771.00	35465.29
Average	3949.46	3971.57	3950.39

transfer is also happened $log_2 n$ times. The actual computation time is dominated by the communication time in the one-pass and two-pass reductions that result in poor improvement in the execution time in both one-pass and two-pass reduction compared to the single-thread approach.

5.4.3 Solution Quality Precision

MLP solutions have been given in integers in the past work. The latency between two nodes in MLP is calculated using the 2D Euclidean distance formula. Euclidean distance formula uses the square-root to get the distance between two points. If an integer variable is considered to hold the distance between two points, precision is lost as square-root returns a floating-point number. The performance analysis of calculating latency using integer and floating-point methods has been evaluated on TRP and TSPLIB instances. The results are presented in Tables 5.14-5.16.

Table 5.14 presents the precision calculation results on TSPLIB instances. The difference rate indicates the closeness of the floating-point solution with its corresponding integer solution. The difference rate is calculated using Eqn. 5.8.

$$\frac{\text{floating local} - \text{integer local}}{\text{integer local}} * 100 \tag{5.8}$$

If the difference rate is greater than 0, it shows the integer calculation loses accuracy while calculating the latency. If rate < 0, indicates floating-point calculation obtains the better solution than the integer method. If rate ≈ 0, it shows that both calculation methods provide similar solutions. The floating-point calculation provides better solutions for seven instances out of 39 TSPLIB instances. It is observed that the integer calculation loses accuracy for the remaining 32 instances. Table 5.15 show the precision calculation analysis for TRP instances of 10, 20, 50 size data sets. There is only a single occurrence of a better solution using the floating-point calculation in each 10 and 50 data sets. The floating-point calculation obtains an improved solution for an input R4 from 10-size and R1 from 50-size data sets compared to the integer method. In Table 5.16, floating-point calculation provides an improved solution for one and two instances in 100 and 500-size data sets, respectively. In a 100-size data set, floating-point calculation improves the local solution by -1.48 rate for the instance R16. In a 500-size data set, the local solutions of two inputs, namely R2 and R14, are improved using the floating-point method, which has improved rates of -0.88 and -0.61, respectively.

From Tables 5.14-5.16, it is conferred that integer calculation loses accuracy while calculating the latencies of solutions for most instances. For this analysis, 159 instances have been considered. Out of 159, the floating-point calculations yield better solutions for nine instances. There are two reasons for building different solutions for the same instance. 1) The next closest unvisited node is chosen more precisely in the floating-point calculation while constructing the initial solution using the NN method. 2) The fractional part is skipped while summing up the latency of a solution in the integer calculation method. Therefore DLSH constructs a different initial solution that leads to different local optima in each calculation method. The floating-point method is always a better choice, which avoids losing fractional differences.

Table 5.14: DLSH solution precision analysis using integer and floating-point calculation methods. TSPLIB instances are used for this evaluation. Difference is calculated using Eqn. 5.8.

Instance	Integer			Floating-point			Difference
	Init	Local	C	Init	Local	C	
berlin52	144761	139316	2	145263.41	139788.27	2	0.34
st70	21844	20278	2	24802.08	22004.93	2	8.52
rat99	61575	60074	2	62253.55	60128.07	2	0.09
kroA100	1176490	996163	2	1169959.25	995705.25	2	-0.05
kroB100	1159631	1011943	2	1159714.13	1010408	2	-0.15
kroD100	1081377	986717	2	1083653.38	988907.25	2	0.22
kroE100	1074901	1025899	2	1080603.88	1027854.19	2	0.19
lin105	692956	645135	2	693880.38	646193.38	2	0.16
pr107	2078587	1983521	2	2079422.25	1984768.38	2	0.06
ch130	376931	356674	2	381157.41	360942.19	2	1.20
ch150	483528	465248	3	493104	485919.62	2	4.44
kroA150	2269515	1891421	3	2263668.5	1937367.25	3	2.43
kroB150	1954630	1870029	2	1959116.38	1874999.62	2	0.27
rat195	221834	211049	2	225705.23	218795.45	2	3.67
d198	1312478	1209289	2	1295058.75	1274517.5	2	5.39
ts225	14139523	13649963	2	14139545	13650161	2	0.00
pr226	11124323	7181227	3	11126393	7165387	3	-0.22
a280	395607	367765	2	398456.22	370254.12	2	0.68
pr299	7546036	7204856	2	7555756	7213945.5	2	0.13
lin318	7022116	6117163	3	7032892.5	6128251	3	0.18
pr439	21760303	19084000	2	21775316	19097888	2	0.07
pcb442	10765404	10657722	2	10774320	10666490	2	0.08
d493	9675285	7700153	3	9122479	7338664.5	3	-4.69
att532	25085998	19160615	3	25143350	20016770	2	4.47
ali535	372174	309613	2	448284.34	380797.44	3	22.99
rat575	2052289	1947078	2	2160181.75	1951499.62	2	0.23
d657	16547216	15379075	3	16361273	15349201	3	-0.19
rat783	3711989	3470429	2	3832228.75	3490663.75	3	0.58
dsj1000	9819816218	8646553613	3	9820071936	8646803456	3	0.00
vm1084	129225071	105275207	3	129752360	105125344	3	-0.14
rl1304	178715703	162008652	2	178821328	162103600	2	0.06
u1432	118600489	111871593	3	118634352	111973496	3	0.09
d1655	53432497	49464859	2	54952276	50416184	3	1.92
u2319	287165931	276381186	3	287211456	276434144	3	0.02
pr2392	501222898	464856059	3	501816320	465431136	3	0.12
pcb3038	228422815	209522527	3	227252048	214473408	3	2.36
fnl4461	429244345	406179078	3	444218496	408697984	3	0.62
rl5934	1721991412	1633486986	3	1718665216	1627402752	3	-0.37
pla7397	105277496147	70428563340	4	105280004096	70480502784	5	0.07
Average							1.43

Table 5.15: DLSH solution precision analysis using integer and floating-point calculation methods on TRP instances of size 10, 20, and 50. Difference is calculated using Eqn. 5.8

I	10							20							50						
	Integer			Floating-point			Diff	Integer			Floating-point			Diff	Integer			Floating-point			Diff
	Init	Local	C	Init	Local	C		Init	Local	C	Init	Local	C		Init	Local	C	Init	Local	C	
R1	1328	1328	1	1348.17	1348.16	1	1.52	3356	3181	2	3414.55	3256.5	2	2.37	15530	13842	2	13907.95	13762.61	2	-0.57
R2	1520	1520	1	1537.37	1537.37	1	1.14	3305	3290	2	3374.04	3359.86	2	2.12	13693	13046	2	13931.64	13638.93	2	4.54
R3	1233	1233	1	1253.43	1253.43	1	1.66	4244	3621	2	4330.45	3702.18	2	2.24	15058	13238	2	16614.95	13420.05	2	1.38
R4	1395	1395	1	1451.82	1394.93	2	-0.01	3416	3331	2	3485.91	3401.06	2	2.10	14656	13605	2	15062.25	14117.11	2	3.76
R5	1060	1060	1	1086.99	1086.99	1	2.55	3521	3322	2	3605.06	3404.19	2	2.47	12955	12713	2	13731.25	13197.02	2	3.81
R6	1477	1477	1	1496.29	1496.28	1	1.31	3479	3461	2	3561.14	3542.96	2	2.37	13638	13325	2	14284.55	13870.31	2	4.09
R7	1221	1169	2	1238.93	1189.79	2	1.78	2910	2910	1	2989.32	2989.32	1	2.73	14123	11725	2	14326.90	12054.14	2	2.81
R8	1234	1234	1	1264.85	1264.84	1	2.50	3741	3627	2	3829.32	3716.58	2	2.47	13281	13171	2	13832.08	13688.26	2	3.93
R9	1402	1402	1	1415.34	1415.33	1	0.95	3654	3626	2	3733.31	3718.95	2	2.56	14141	13739	2	14727.18	14177.12	2	3.19
R10	1410	1394	2	1429.53	1412.22	2	1.31	3582	3582	1	3620.96	3620.96	2	1.09	14291	13164	2	14706.84	13622.65	2	3.48
R11	1405	1405	1	1420.75	1420.75	1	1.12	3253	3136	2	3325.48	3211.92	2	2.42	13849	13574	2	13998.74	13746.94	2	1.27
R12	1164	1150	2	1193.69	1177.9	2	2.43	4227	3314	2	4309.54	3388.42	2	2.25	11256	11058	2	12543.62	11456.58	2	3.60
R13	1561	1561	1	1584.44	1584.44	1	1.50	3698	3560	2	3784.11	3641.8	2	2.30	13386	12747	2	13716.61	13191.8	2	3.49
R14	1442	1219	2	1459.61	1233.75	2	1.21	3935	3450	2	3897.04	3612.05	2	4.70	14160	13586	2	14577.11	14033.23	2	3.29
R15	1129	1129	1	1138.73	1138.73	1	0.86	3369	2871	2	3419.00	2923.67	2	1.83	13394	12522	2	13591.94	13328.87	2	6.44
R16	1315	1315	1	1336.92	1336.92	1	1.67	3909	3433	2	3972.44	3521.75	2	2.59	13596	13285	2	14449.06	14193.56	2	6.84
R17	1058	1058	1	1077.40	1077.39	1	1.83	3175	2913	2	3272.02	3009.96	2	3.33	14172	13126	2	15067.12	13375.93	2	1.90
R18	1083	1083	1	1098.77	1098.77	1	1.46	3288	3262	2	3360.62	3327.9	2	2.02	14435	13953	2	14804.79	14326.87	2	2.68
R19	1482	1394	2	1500.82	1425.22	2	2.24	4441	3659	3	4245.17	3991.04	2	9.07	12398	12316	2	12912.48	12638.41	2	2.62
R20	951	951	1	973.95	973.95	1	2.41	3140	2796	2	3227.96	2874.17	2	2.80	12745	12612	2	13209.50	13092.03	2	3.81
Average							1.57							2.79							3.32

Table 5.16: DLSH solution precision analysis using integer and floating-point calculation methods on TRP instances of size 100, 200, and 500. Difference is calculated using Eqn. 5.8

I	100							200							500						
	Integer			Floating-point			Diff	Integer			Floating-point			Diff	Integer			Floating-point			Diff
	Init	Local	C	Init	Local	C		Init	Local	C	Init	Local	C		Init	Local	C	Init	Local	C	
R1	34726	34255	2	36285.10	35913.46	2	4.84	109130	98480	2	111467.17	102271.06	3	3.85	2256050	2045627	3	2219064.25	2063379.88	3	0.87
R2	38189	34977	2	39883.77	36629.17	2	4.72	111697	98230	3	114286.59	104908.69	2	6.80	2238193	2062200	3	2340919.5	2043976.5	3	-0.88
R3	37642	33404	2	37448.48	34537.05	2	3.39	113545	103959	3	114071.03	106008.3	2	1.97	2227313	2035358	3	2278522.25	2074556.38	2	1.93
R4	40316	35631	2	41765.24	38776.12	2	8.83	104810	101155	2	115209.85	107404.8	2	6.18	2119535	2002354	2	2093390.75	2019776	2	0.87
R5	36017	35050	2	38185.20	36925.47	2	5.35	102359	94078	3	107447.23	101585.12	2	7.98	2253349	2013264	2	2279551.5	2061675.62	3	2.40
R6	36298	35805	3	40522.93	37950.39	2	5.99	113768	102947	2	117902.80	105829.46	2	2.80	2109395	1970414	2	2169371	2018286.75	2	2.43
R7	39236	37570	2	40551.31	37690.13	2	0.32	103359	96995	2	112823.40	104376.41	2	7.61	2185996	2044015	2	2280056.75	2096777.88	2	2.58
R8	38749	33103	2	40681.43	35966.15	3	8.65	100896	94898	2	100672.94	99044.02	2	4.37	2149991	2005965	3	2207849.5	2055648.38	2	2.48
R9	41262	36354	2	38457.38	37000.3	2	1.78	106238	96319	2	109855.73	102245.2	2	6.15	2074414	1867533	2	2095168.625	1918312.38	2	2.72
R10	37860	32545	2	38094.05	34289.59	2	5.36	105386	97113	2	106613.22	102136.23	2	5.17	2045557	1935231	2	2110112.5	1993131.25	2	2.99
R11	38962	37077	2	40777.37	39443.48	2	6.38	101519	97026	2	114924.05	108894.77	2	12.23	2226155	1993875	3	2233189	2102794	3	5.46
R12	35032	33670	2	36941.71	35526	2	5.51	106774	99578	2	112266.42	103997.99	3	4.44	2091972	1943465	2	2153525.75	1999638.12	2	2.89
R13	38406	35416	2	36591.67	35548.02	2	0.37	99401	92648	2	106968.40	103552.89	2	11.77	2246410	2038807	2	2293263.5	2074448.62	3	1.75
R14	34517	32601	2	38326.72	35506.98	2	8.91	110028	100741	2	118634.70	113931.3	2	13.09	2104882	1980946	2	2055053.375	1968847.62	2	-0.61
R15	37769	35559	2	41506.70	38066.36	2	7.05	105445	97719	2	111445.51	107893.65	2	10.41	2067337	1930097	3	2099942.75	1993830.12	2	3.30
R16	39860	37642	2	41869.86	37085.38	3	-1.48	103499	95245	2	105728.13	100204.14	2	5.21	2072183	1999019	2	2122850.25	2046780.62	2	2.39
R17	40749	39763	2	42662.68	41569.66	2	4.54	105663	93552	2	110846.08	103207.38	2	10.32	2072851	1969347	2	2124344.5	2007515.5	2	1.94
R18	38839	35862	2	39966.67	37482.21	2	4.52	104599	98375	2	114323.77	109448.31	2	11.26	2136634	2059754	2	2156032	2063403	2	0.18
R19	39752	37178	2	40567.59	38259.46	2	2.91	107054	103036	2	109312.99	105349.45	2	2.25	2081332	1960612	3	2142302.5	2000185.75	2	2.02
R20	39346	35128	2	41293.56	36839.17	3	4.87	95935	92868	2	116876.07	99637.58	3	7.29	2191334	2000517	2	2202715.75	2080423	3	3.99
Average							4.64							7.06							2.09

5.4.4 Discussions

In existing metaheuristic algorithms, the initial solution is constructed using Iterated Local Search (ILS) or Guided Local Search (GLS). ILS and GLS use the randomization technique while setting up the initial solution. It means that for the same instance, the different initial solution is constructed at each run. In this work, the Deterministic Local Search Heuristic (DLSH) constructs the same initial for the same instance irrespective of multiple runs. In DLSH, the initial solution is constructed using the Nearest Neighbourhood (NN) approach.

The swap method needs the preprocessing of data structure to evaluate a neighborhood move in constant time. In this work, a constant time evaluation method is proposed for the swap method, which does not require data preprocessing. The latency between nodes is calculated on-the-fly using X, Y coordinates of a corresponding instance. This saves allocating additional memory required for data preprocessing.

The GPU-based parallel strategy is proposed for DLSH called as PDLSH. The existing GPU-based parallel strategy limits solving instances beyond 1024 nodes. This limitation is overcome in the proposed strategy, which solves larger instances > 1024 nodes. The effectiveness of PDLSH has been examined on large-scale instances up to 11849 nodes. PDLSH provides the same local solution in 10.07 hours, where DLSH spends 39 days to obtain it. The PDLSH receives a speedup of up to 179.75 times for instances up to 11849 nodes compared to corresponding DLSH implementation.

For PDLSH, two reduction methods, namely built-in and vector, are applied to determine the best-improved solution for providing synchronized results. The built-in method is useful while solving instances up to 4461 nodes, whereas the vector method is found useful for solving instances > 3038 nodes. The built-in method has limitations, which fails when either thread id or latency does not fit in a 32-bit number.

The precise latency calculation method is presented. The impact of calculating latency using integer and floating-point methods are demonstrated over TRP and TSPLIB instances. The floating-point method helps to calculate latency more accurately, whereas the integer method loses the fractional part while summing up Euclidean distance between two nodes.

The disadvantage of DLSH is its solution quality. DLSH does not reach the best-known-solution compared to the state-of-the-art MLP solvers. It has gap rates in the range of 0.52-16.27%, 4.57-12.40%, and 7.71-13.52% for TRP instances of 100, 200, and 500 size data sets, compared to the GILS-RVND metaheuristic. GILS-RVND uses five neighborhood generation methods, whereas DLSH uses a two generation method. Moreover, GILS-RVND applies perturbation $min(n, 100)$ times to avoid trapping in local optimal. In the future, perturbation technique and additional neighborhood generation methods can be applied to the DLSH for obtaining a better solution quality.

5.5 SUMMARY

The Deterministic Local Search Heuristic (DLSH) that always reaches the same local solution on multiple trials, as presented. Metaheuristics often use perturbation and randomization. Perturbation is used to escape from the local optima. Randomization helps to explore diversified solutions. DLSH does not use metaheuristic parameters such as perturbation and randomization. DLSH constructs the initial solution using the Nearest Neighborhood (NN) approach.

The swap and two-opt methods evaluate several moves to determine the best-improved move. The state-of-the-art move evaluation procedure evaluates a move in $O(1)$ time using the preprocessed local data. Current work proposes the move evaluation procedure for the swap method, which computes a move in $O(1)$ time that does not need any preprocessed data. A linear time move evaluation procedure is used for the two-opt move calculating the move effect on-the-fly without storing intermediate solutions.

The most time-consuming part of the DLSH is its solution improvement phase. The swap and two-opt methods consume $O(n^2)$ and $O(n^3)$ time, respectively, for a single call. These neighborhood methods are called many times to improve the solution. PDLSH strategy maps one thread per move evaluation. The parallel strategy of PDLSH does not depend on the maximum threads per block limit and hence solves larger instances than existing works. When all threads evaluate their moves, the next important job (i.e., reduction) is to choose the best-improved vertex pair and its associated latency. Two reduction methods have been presented, namely built-in and vector. The built-in method consumes less execution time for the instances less than 4461 sizes, whereas the vector method performs better for the larger instances with sizes above 3038. However, the built-in method fails when either of the thread id or the latency of the associated thread does not fit in the unsigned 32-bit number. Because $atomicMin$ binds the thread id and its latency into an unsigned 64-bit number to choose the best-improved solution. Vector reduction method overcomes the issue associated with the built-in method. PDLSH achieves a speedup up to 179.75 for TRP and TSPLIB instances of sizes 10 - 7397.

CHAPTER 6

PARALLEL VERSION OF LOCAL SEARCH HEURISTIC ALGORITHM TO SOLVE CAPACITATED VEHICLE ROUTING PROBLEM

The purpose of this chapter is to identify independent, time-consuming portions of the heuristic algorithms in order to reduce the total runtime of the heuristic algorithms using data level, and thread level parallelism. Note that this chapter does not propose any new heuristic algorithm to find a better solution quality. Instead, we focus on finding GPU-based parallel approaches for the heuristic algorithms to solve larger CVRP instance in a reasonable amount of time. Contributions of this chapter includes:

- GPU-based parallel implementation for the intra-route and inter-route improvement heuristics.
- Solve the large-scale benchmarking instances up to 30000 customers.
- The proposed customer level parallel implementation achieves a speedup of up to $147.19\times$ over its sequential counterpart.

6.1 INTRODUCTION

6.1.1 Capacitated Vehicle Routing Problem

The Vehicle Routing Problem (VRP) (Clarke and Wright 1964; Dantzig and Ramser 1959) is an NP-hard, combinatorial optimization problem, with applications in the field of goods and transportation. The objective of VRP is to schedule the number of vehicles for goods transportation such that its transporting cost is reduced. VRP has several variants based on its objective function, viz., Capacitated Vehicle Routing Problem (CVRP), Heterogeneous Fleet VRP (HFVRP), Multi-Depot VRP (MDVRP), Pickup and Delivery VRP (PDVRP) (Braekers et al. 2016; Eksioglu et al. 2009; Toth and Vigo 2001).

CVRP is considered in this work. CVRP is defined as: Given a simple, connected, weighted, undirected graph $G(V, E)$, where, V is set of vertices, i.e., $V = \{v_0, v_1, .., v_n\}$ and E is set of edges i.e., $E = \{v_i, v_j\}$ where $(i < j < n)$, the objective is to find a set of routes $R = \{r_1, r_2, .., r_m\}$ for m vehicles such that, 1) each customer is visited exactly once, and 2) total traveling distance is minimum.

The constrains considered for CVRP are given in Equations 6.1 to 6.5. Vertex v_0, the depot is the starting point for all vehicles. Each vehicle completes its journey and returns to v_0. All vehicles have uniform capacity Q. Each customer v_i $(i = 1, 2, .., n-1)$ has a corresponding demand d_i to be met by a vehicle. A vehicle, m_i, services one or more customers on its route, r_i, such that its own capacity does not exceed Q.

$$Length(R) = min \sum_{i=1}^{m} r_i \tag{6.1}$$

subject to,

$$R = r_1 \cup r_2 \cup \cdots \cup r_m = \{v_0, v_1, v_2, \dots v_{n-1}\} \tag{6.2}$$

$$r_1 \cap r_2 \cap \cdots \cap r_m = \{v_0\} \tag{6.3}$$

$$capacity(r_i) \leq Q \text{ where } 1 \leq i \leq m \tag{6.4}$$

$$\sum_{i=1}^{n-1} d_i \leq Q * m \tag{6.5}$$

Exact methods such as brute-force approach, branch and bound, and dynamic programming have been used to find an optimal solution of VRP problems (Laporte and Nobert 1987). Although exact methods assure an optimal solution, it becomes intractable while solving larger size instances. Exact methods explore the entire search-space, an $O(n!)$ or $O(2^n)$ operation, to determine optimal solutions (Cormen et al.

2009). Real world applications have several thousands of consumers to serve making exact methods infeasible.

6.1.2 Heuristic Algorithm

The CVRP is an intractable problem due to the execution time requirements of the exact methods. Heuristic algorithms search a subset of the search space for a satisfactory solution in a reasonable amount of time (Campos and Mota 2000; Laporte et al. 2000; Pisinger and Ropke 2007). There are two important phases in the heuristic algorithm, solution setup and solution improvement. In the solution setup phase, a feasible solution is determined. A feasible solution is either fixed arbitrarily or constructed using construction heuristic methods. In the solution improvement phase, the initiated solution is continuously improved until there is no further progress. Because CVRP has m routes in its feasible solution, a solution can improve within a route or between two routes during the solution improvement phase. As a result, two types of improvement heuristics are used in the solution improvement phase: intra-route and inter-route improvement heuristics.

Campos and Mota (2000) design two heuristic approaches for solving CVRP. The proposed heuristic uses Tabu Search (TS) to improve the feasible solution and shows the effectiveness of proposed heuristic algorithm using TSPLIB (Reinelt 1991), Christofides et al. (1979), and Fisher (1994) instances of size up to 135 nodes. Laporte et al. (2000) present the survey of heuristic algorithms used for VRP and provide the computational analysis of various heuristic algorithms that solve different VRP variants. Pisinger and Ropke (2007) present a unified heuristic algorithm called Adaptive Large Neighborhood Search (ALNS), which solve five variants of VRP, namely CVRP, MDVRP, site-dependent VRP, VRP with time windows, and open VRP. Altabeeb et al. (2021) present a cooperative hybrid firefly algorithm (CHFA) with multiple solutions to solve CVRP. The proposed CHFA helps to get out of trapping into local solution and overcome the limitation of the single swarm firefly algorithm. Authors have demonstrated the efficacy of the proposed heuristic on small-scale instances, i.e., up to 200 customers, in previous works. The majority of the time spent by these heuristic algorithms is spent in the improvement phase. Because neighbourhood solutions are generated repeatedly during the improvement phase until the termination criteria are met. Furthermore, the proposed heuristic efficiency is tested on smaller size instances (i.e., typically up to 200 nodes). Because real-time applications must deal with thousands of nodes to determine a better feasible solution, the heuristic algorithm should now solve large-scale instances.

Máximo and Nascimento (2021) present a hybrid of iterated local search and path relinking to solve CVRP. The proposed hybrid version of heuristic is tested on CVRP instances of size up to 1000 nodes. Kytöjoki et al. (2007) have designed the large-scale instances for CVRP that has sized up to 20000 nodes. Moreover, the author developed a heuristic algorithm called Variable Neighborhood Search (VNS) to solve these large-scale instances. Recently, Arnold and Sörensen (2019) created very large-scale instances considering a real-world problem that contains up to 30000 nodes and provides a heuristic approach for solving it. Both in (Arnold and Sörensen, 2019; Kytöjoki et al., 2007), have used several improvement heuristic approaches to improve the constructed solutions. The time spent in the solution improvement phase can be reduced by applying GPU-based parallel computation (Carvalho et al., 2020).

6.1.3 Motivation

The most time-consuming part of heuristic algorithm is its solution improvement phase. The solution improvement phase is a critical step in the heuristic algorithm that attempts to move the constructed solution closer to the global optimal solution. Therefore, a study has been carried out to identify what portion of execution time is being spent in the solution improvement phase. Figure 6.1 shows the execution time analysis of both heuristic phases over the Belgium (Arnold and Sörensen, 2019) instances. From Figure 6.1, it can be determined that more than 99% of the time out of total execution time is being spent in the solution improvement phase.

Our motivation for this work is to reduce the amount of time spent in the solution improvement phase when dealing with large-scale instances. This large portion of execution time can be reduced using parallel computation because the steps involved in the solution improvement phase are independent. In the solution improvement phase, the improvement heuristic generates several neighborhood solutions repeatedly. The

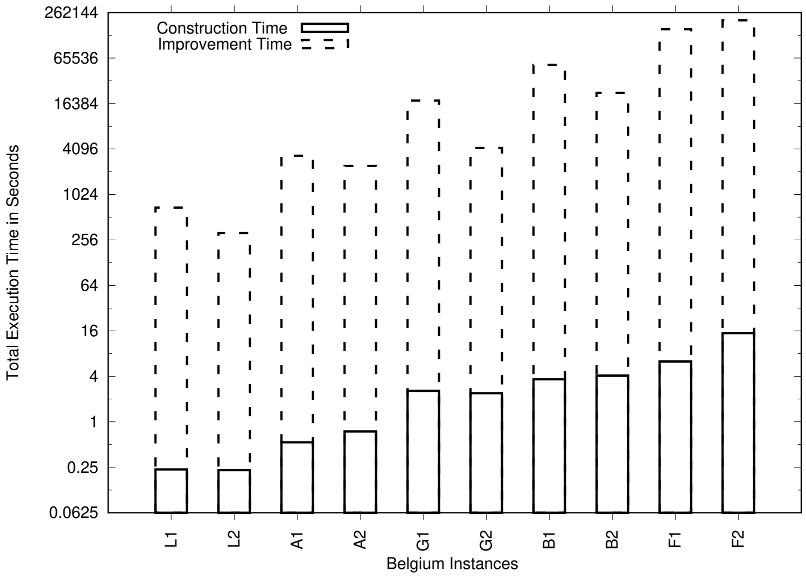


Figure 6.1: Phase-wise execution time analysis on Belgium Arnold and Sörensen (2019) data set.

generation of one neighborhood solution does not depend on the others. These independent neighborhood generations are key components required for using the parallel platform. The GPU-based parallel computation has already shown the effectiveness in reducing the execution time (Yelmewad and Talawar 2019). The parallel computation for heuristic algorithms are applied in (Jin et al. 2014; Schulz 2013) and solve instances up to 2401 nodes. The GPU-accelerated heuristic (Abdelatti and Sodhi 2020) solves CVRP instances of up to 76 nodes. Moreover, we aim to develop a parallel heuristic algorithm that solves large-scale instances than the existing parallel version of heuristic algorithms.

Section 6.2 explains the mechanism of heuristic algorithms to solve CVRP. Section 6.3 presents the detailed illustration of GPU-based parallel approaches for heuristic algorithms. Section 6.4 presents performance evaluation of parallel approaches.

6.2 LOCAL SEARCH HEURISTICS

The Local Search Heuristic (LSH) is a heuristic approach that initiates a feasible solution and explores its neighboring solutions. The initialization of a feasible solution is done in one of two ways: a) randomly. b) using the construction heuristic approaches (Yelmewad and Talawar 2019). The neighborhood solutions are generated to improve the initial solution further. The neighborhood solutions are generated using 2-opt, 3-opt, or-opt, relocate and swapping techniques.

Algorithm 6.1: Generic Local Search Heuristic Algorithm

```
   Result: Local optimal solution
1  $s \leftarrow$ create a feasible solution.
2  $f(s) \leftarrow$ find cost of $s$;
3  while $s$ is improved do
4      Apply swap;
5      Apply relocate;
6      if $s$ is improved then
7          Apply 2-opt;
8          Apply or-opt;
9          Apply 3-opt;
10         $s \leftarrow$ initialize the best improved solution.
11     else
12         return current $s$ as local optimal solution.
13     end
14 end
```

Algorithm 6.1 presents the local search heuristic considered for solving CVRP in this work. Five heuristic approaches, namely, 2-opt, or-opt, 3-opt, relocate, and swap are used. These heuristics are further classified, intra-route and inter-route heuristics. The methods 2-opt, 3-opt, and or-opt are the intra-route improvement heuristics, and relocate and swap methods are the inter-route improvement heuristics. The feasible solution is created using Nearest Neighborhood (NN) approach, and its cost is calculated (lines 1-2). Later, the inter-route improvement heuristics are applied to the constructed solution (lines 4-5). The swap heuristic is applied first, followed by the relocate. If any improved solution is found in any of the inter-route heuristics, the intra-route heuristics are used repeatedly until no further improvement is possible (lines 6-11). The loop (lines 3-14) is repeated until there is no further improvement possible.

The solution construction phase and its improvement methods are elaborated in the following subsections.

6.2.1 Solution Construction

Algorithm 6.2 presents the stepwise details of the initial solution construction phase. The list, $farList$, contains the list of customers, which is arranged in the descending order of their distances with the depot (line 1). For m routes, the first customer is chosen from the $farList$ list (line 4). One route is constructed at a time. A strategy is used in which the farthest unvisited customer is assigned as the first node for each route (i.e., a list $rSource$ holds the first node of each route; line 5). The list $rSource$ and $custOrder$ records the feasible solution, where $rSource$ keeps track of the first node of each route and $custOrder$ maintains the sequence of remaining customers on each route (Kytöjoki et al., 2007). The lists $visit$ and cap maintains the list of customers visited

Algorithm 6.2: Feasible Solution Construction Procedure

```
   Result: Creates a feasible solution
 1 $farList \leftarrow$ create a descending order customers' list.
 2 $count \leftarrow 0$;
 3 for $r = 0; r < m; r{+}{+}$ do
 4     $cust \leftarrow$ choose first unvisited customer from $farList$;
 5     $rSource[r] \leftarrow cust$;
 6     $visit[cust] \leftarrow 1$;
 7     $cap[r] += demands[cust]$;
 8     $curCust \leftarrow cust$;
 9     $count++$;
10     while $cap[r] \leq Q$ do
11         $cust \leftarrow$ finds the closest unvisited customer of $curCust$;
12         if $cap[r] + demands[cust] \leq Q$ then
13             $custOrder[curCust] \leftarrow cust$;
14             $visit[cust] \leftarrow 1$;
15             $cap[r] += demands[cust]$;
16             $curCust \leftarrow cust$;
17             $count++$;
18         else
19             break;
20         end
21     end
22 end
23 if $count < n$ then
24     $m++$;
25     $goto\ back$;
26 end
```

and currently filled capacity of each vehicle, respectively (lines 6-7). The variables $curCust$ and $count$ keeps track of the current node of the present route, and the total customers visited until r routes, respectively. Once the first customer is visited, the next node of the route r is chosen using the NN method (lines 10-21). The criteria which are used to choose the next node for route r are given below.

- Next node should be the closest customer of the current customer on route r.
- The nearest customer should not be visited previously by any of m routes.
- The demands of the next node should fit into the capacity of the route r (i.e., $cap[r] + demands[cust] \leq Q$).

If these criteria are satisfied, an unvisited customer is added to the current route r. If an unvisited customer does not meet any of these conditions, the job of assigning customers to the current route is terminated (line 19). The remaining unvisited nodes will be assigned to the next routes. If a feasible solution is not constructed by the loop (lines 3-22), the vehicle count, m, is incremented (lines 23-26). The initial m value tells the minimum number of vehicles required to satisfies the demands of n customers.

6.2.2 Solution Improvement

The inter-route and intra-route heuristics are applied once the feasible solution is constructed. Figure 6.2 presents the order in which five improvement heuristics are applied. First, the constructed solution is passed to the swap heuristic. The swap heuristic generates neighboring solutions and tries to find the best-improved solution. Either the improved or the constructed solution is passed to the relocate heuristic. The relocate heuristic uses its neighbor generation technique for solution exploration. If an improvement is observed in any of inter-route heuristics, three intra-route heuristics are applied subsequently in the following sequence: 2-opt, or-opt, and 3-opt. If an improved solution is found, both intra-route and inter-route heuristics are considered for the next initial solution. The same neighbor generation procedure is repeated until there is no further progress possible. The generation of neighborhood solution techniques for each heuristic is explained below.

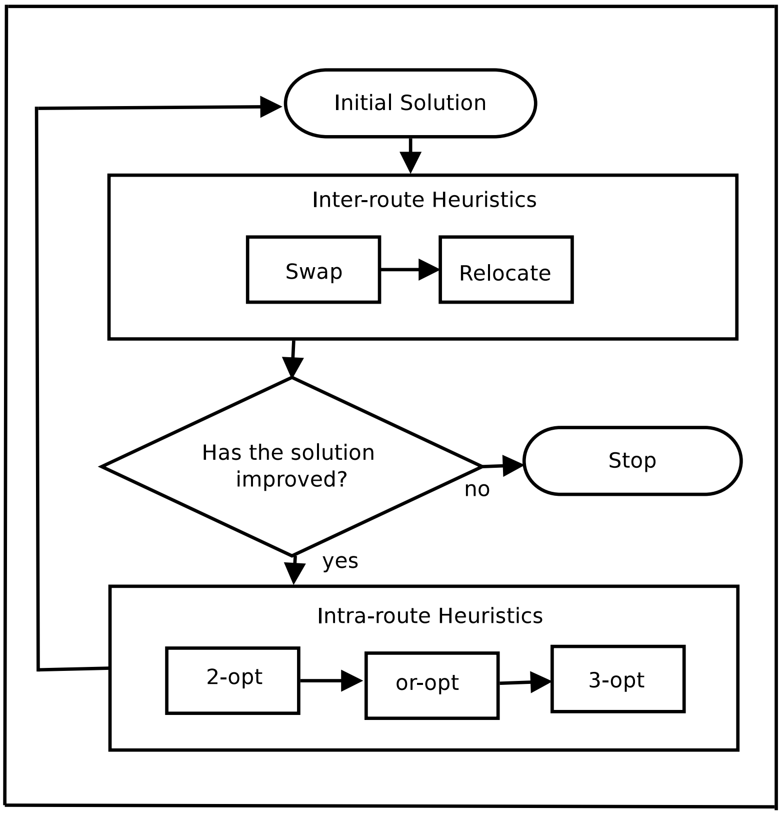
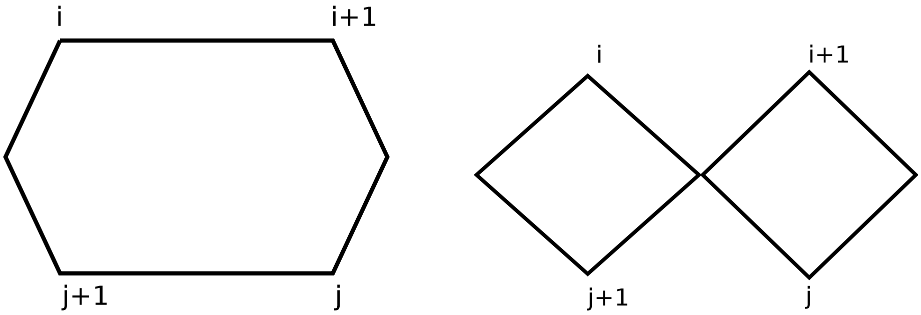


Figure 6.2: Solution improvement flowchart

6.2.2.1 2-opt Heuristic

In the 2-opt method, two edges of the same route are exchanged to generate a new route. This exchanging of edges is only considered if it has resulted in an improved route. Figure 6.3 shows the neighborhood generation in the 2-opt heuristic.

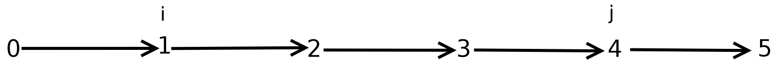


(a) Original route (b) After 2-opt on the nodes i, j

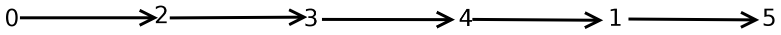
Figure 6.3: Pictorial representation of 2-opt neighborhood generation technique.

6.2.2.2 or-opt Heuristic

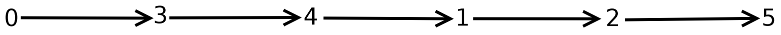
In or-opt heuristic, l consecutive customers are relocated in the same route r. A chain of one, two, and three customers have been considered for the relocate heuristic. Figure 6.4 shows the mechanism of or-opt heuristic in a pictorial form when the value of l is 1, 2, and 3 consecutive customers. In Figure 6.4 (a), a variable i indicates the starting position of l consecutive customers and j indicates a neighbor where l is relocated.



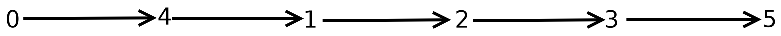
(a) Original route



(b) or-opt with 1 move



(c) or-opt with 2 move

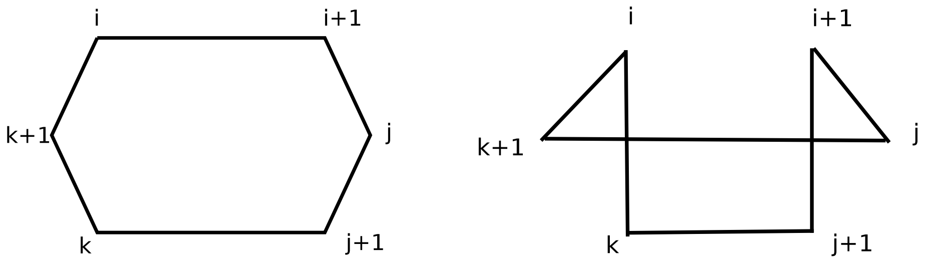


(d) or-opt with 3 move

Figure 6.4: Pictorial representation of or-opt neighborhood generation technique.

6.2.2.3 3-opt Heuristic

The 3-opt is a variant of the 2-opt in which three edges are exchanged. Figure 6.5 shows the neighborhood generation in the 3-opt heuristic. The order of removing three edges for the nodes i, j, and k are $(i, i+1)$, $(j, j+1)$, and $k, k+1$, respectively. The order used for adding back three edges are (i, k), $(i+1, j+1)$, and $(j, k+1)$, respectively. For a route r, total possible new routes generated using the 3-opt heuristic is $\frac{n(n-1)(n-2)}{6}$.



(a) Original route (b) After 3-opt on the nodes i, j, k

Figure 6.5: Pictorial representation of 3-opt neighborhood generation technique.

6.2.2.4 Swap Heuristic

In the swap heuristic, a customer from one route is swapped with a customer from another route. This heuristic allows exchanging of two customers from different routes to find its suitable place. If customer a is in route r, a can be swapped with a customer b from the routes $r + 1$ to $m - 1$. The total customers in b's route should be greater than one. This swapping is only be done if the demands of both customers can be met by the capacity of routes.

6.2.2.5 Relocate Heuristic

The relocate heuristic moves a customer from one route to another. This moving of customers will be done when two constraints are satisfied. 1) Improve the initial solution. 2) Demand of the moving customer does not exceed the capacity of the route where it is moved.

6.3 PARALLEL LOCAL SEARCH

LSH requires a significant amount of time for the execution of larger input instances. It is observed that the solution improvement phase consumes hours of time on an average for improving the feasible solution for Kytöjoki data set (Kytöjoki et al. 2007). In this work, five heuristics are applied for solution improvement. Improvement heuristics are employed after the feasible solution is constructed. In this work, parallel strategies are applied after the feasible solution is constructed.

This work parallelizes the improvement phase in two ways: at the route level, and the customer level. The route level parallel approach improves m routes in parallel using local improvement heuristics. A customer level approach assigns a customer per thread. Each customer finds its best place on m routes in parallel using the improvement heuristics. The mechanism of these strategies are elaborated in the following subsections.

6.3.1 Route Level Parallel Design

In this parallel strategy, one route is mapped to a thread. The total number of threads created is equal to m, the number of vehicles of each CVRP instance. Total blocks required is calculated using Eqn. 6.6 where, $threadsPerBlock$ is the number of threads to be used per block (1024 threads are the maximum threads allowed per block on Tesla P100 GPU card).

$$blocks = \frac{m-1}{threadsPerBlock} + 1 \tag{6.6}$$

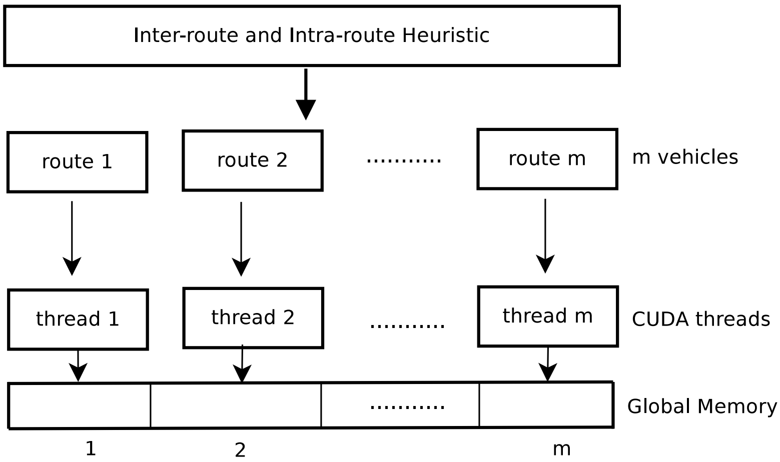


Figure 6.6: Route level parallel design for improvement heuristics.

Figure 6.6 shows the generic workflow of the route level parallel design. Note that this parallel strategy is applied inside each improvement heuristic. There is no change in the order of applying improvement heuristics. In the intra-route heuristics, each thread generates potential neighboring routes from the current route and selects the best-improved route. No race conditions occur in intra-route heuristics as each thread works independently on different routes. During inter-route heuristics, each thread finds the best swapping or relocation in routes maintained by other threads. This work avoids race conditions using the following strategy.

- For each inter-route heuristic, a separate data structure is maintained to hold each thread's best solution.
- An additional function is necessary to find the best solution from all threads' individual best solutions.

Table 6.1 shows a separate data structure of $\mathcal{O}(m)$ size is created in the global memory to hold each thread's best solution for the inter-route heuristic. For the swap heuristic,

Table 6.1: Members of heuristic data structures used to hold each thread's best solution details.

Heuristic	Data Structure	Members
swap	swap_data	2 customers from different routes, 2 route ids, 2 change effects for 2 routes, 1 total change effect
relocate	relocate_data	1 customer, 1 neighbor, 2 route ids, 2 change effects for 2 routes, 1 total change effect

seven members are needed in the data structure, $swap\_data$. These seven members are two routes' id where swapping is to be done. Next two members are customer ids which is to be swapped, two scalar variable that holds the changing effect of both routes, and a seventh member will hold the actual change value in the final generating solution. A seven-member data structure, $relocate\_data$, is also used for the relocate heuristic. The role of the first four members is changed in the relocate heuristic. Now the first member will represent a route from which the customer is removed. The second member represents the route where the customer is to be relocated. Next, two members, i.e., customers id, one will act as a neighboring node where the second customer will be relocated, and second will be a customer id, which is to be relocated. Once all threads write its best solution to the data structure, a separate function is used to find the best-improved solution. This minimum finding function can be implemented at either the CPU or the GPU side. If it is done at the CPU side, it takes $\mathcal{O}(m)$ time. Instead, it can be done in log_2m steps over the GPU.

However, this parallel strategy does not produce a noticeable improvement in the execution time. The performance analysis of this strategy is presented in Section 6.4.2.

6.3.2 Customer Level Parallel Design

The customer level parallel design allows every thread to act as a customer. In this strategy, the total threads used for neighborhood generation is equal to the instance size, n. The number of blocks created is calculated using Eqn 6.7.

$$blocks = \frac{n-1}{threadsPerBlock} + 1 \tag{6.7}$$

The job of threads changes according to the improvement heuristics. The mechanism of customer level parallel design for intra-route and inter-route heuristics are explained in the following subsections.

6.3.2.1 Intra-route Heuristics

Each thread deals with a set of customers that are allocated on the same route. Algorithm 6.3 presents the pseudocode for the intra-route heuristics. There are three separate kernels used for the 2-opt, or-opt, and 3-opt heuristics. Algorithm 6.3 present an abstract pseudocode view of a parallel version for all three heuristics together. Each thread gets its associating global id (line 1). If thread id is in between 1 and m, associated threads participates in neighborhood generation computation. Id $i = 0$ represents a depot id; it cannot be moved or relocated. Thus threads whose id is greater than zero are allowed for the solution improvement (lines 2-15). The first step of each thread (i.e., a customer) is to identify on which route it has been allocated (line 3). The previous and next customers of i are determined i.e., $i - 1$ and $i + 1$, respectively (lines 4-5). Next, thread i visits all the customers of the same route to determine the best-improved solution (lines 6-14).

The change calculation equation determines whether a respective pair (i, j) or tuple (i, j, k) can improve the solution (lines 7-9). If change is negative, the pair can generate an improved solution over the existing solution. Each intra-route heuristic has a separate change calculation equation. For 2-opt, or-opt, and 3-opt, equations on lines 7, 8, and 9 are used, respectively. If an improvement is found, thread writes solution details in the respective data structure at the i^{th} position in the global memory (lines 10-13). There are three data structures created for each intra-route heuristic, Table 6.2. Upon improvement, the thread collects the best improved customer pair (i, j) for 2-opt, chunk size l and customer pair (i, j) for or-opt, and a tuple (i, j, k) for the 3-opt heuristics respectively. When all threads finish execution, a separate minimum finding kernel is used to select the best solution among n solutions available in the respective data structure.

Algorithm 6.3: Pseudocode for generating and finding the best solution per thread for intra-route heuristics.

```
Result: Generates and find the best solution per thread
$i \leftarrow threadIdx.x + blockIdx.x * blockDim.x$;
if $i > 0$ && $i < n$ then
    $r \leftarrow$ identify where $i$ is allocated.
    $i + 1 \leftarrow$ identify a next neighbor of $i$;
    $i - 1 \leftarrow$ identify a previous neighbor of $i$;
    while all members of $r$ is not visited do
        $change \leftarrow d(i,j) + d(i+1,j+1) - d(i,i+1) - d(j,j+1)$;
        // this change calculation is used for 2-opt
        $change \leftarrow d(i,j) + d(i+l,j+1) + d(i-1,i+l+1) - d(i-1,i) - d(i+l,i+l+1) - d(j,j+1)$;
        // this change calculation is used for or-opt
        $change \leftarrow d(i,k)+d(i+1,j+1)+d(j,k+1)-d(i,i+1)-d(j,j+1)-d(k,k+1)$;
        // this change calculation is used for 3-opt
        if $change < minChange$ then
            $minData \leftarrow$ writes current solution details.
            $minChange \leftarrow change$;
        end
    end
end
```

Table 6.2: Members of heuristic data structures used to hold each thread's best solution details.

Heuristic	Data Structure	Members
2-opt	twoOptData	2 customers, route id, change effects
3-opt	threeOptData	3 customers, route id, change effects
or-opt	orOptData	x, y, chunk size, route id, change effects

Algorithm 6.4: Pseudocode for generating and finding the best solution per thread for inter-route heuristics.

```
   Result: Generates and find the best solution per thread
1  $i \leftarrow threadIdx.x + blockIdx.x * blockDim.x$;
2  if $i > 0$ && $i < n$ then
3      $r1 \leftarrow$ identify where $i$ is allocated.
4      $i + 1 \leftarrow$ identify a next neighbor of $i$;
5      $i - 1 \leftarrow$ identify a previous neighbor of $i$;
6      for r2 = r1+1; r2 < m; r2++ do
7          while All members of r2 is not visited do
8              $change1 \leftarrow d(i-1, j) + d(j, i+1) - d(i-1, i) - d(i, i+1)$;
9              $change2 \leftarrow d(j-1, i) + d(i, j+1) - d(j-1, j) - d(j, j+1)$;
10             $change \leftarrow change1 + change2$;
               // this change calculation is used for swap
11             $change1 \leftarrow d(i-1, i+1) - d(i-1, i) - d(i, i+1)$;
12             $change2 \leftarrow d(j-1, i) + d(i, j+1) - d(j-1, j) - d(j, j+1)$;
13             $change \leftarrow change1 + change2$;
               // this change calculation is used for
                  relocate
14             if $change < minchange$ then
15                 $minData \leftarrow$ writes current solution details.
16                 $minchange \leftarrow change$;
17             end
18         end
19     end
20 end
```

6.3.2.2 Inter-route Heuristics

The parallel version of inter-route heuristic is a variant of the intra-route heuristic, where each thread explores customers of other routes (Algorithm 6.4). The first five steps of an algorithm are the same as the intra-route heuristic (lines 1-5). Once each thread receives its neighboring customers' detail, it starts exploring all customers on the routes $r + 1$ to m independently in pursuit of an improved solution (lines 6-19). A separate change calculation equation is used for the swap and relocate heuristics. The steps 8-10 are used to get a change affect on routes $r1$ and $r2$ if customers i and j are exchanged. A variable $change1$ tells the impact of removing a customer i from route $r1$ and adding a customer j to a route $r1$. A variable $change2$ indicates an effect on the route $r2$ if j is removed and i is added back to the route $r2$.

The steps 11-13 are used to get an impact of applying the relocate on customer i from route $r1$ to $r2$. A variable $change1$ indicates a resultant cost of a route $r1$ after removing a customer i. Variable $change2$ has the same meaning in the swap heuristic. If a $change$ is negative, thread writes the current solution details in the data structure $swap\_data$ or $relocate\_data$, which is located in the global memory. After the kernel finishes its execution, a minimum finding kernel is applied to the data structure to determine the best solution. The performance analysis of the route level and customer level parallel designs have been presented in the following section.

6.4 RESULTS AND DISCUSSIONS

The performance analysis of the parallel version for local search heuristic has been presented in this section. A wide variety of benchmark instances, i.e., up to 30000 customers, have been used to evaluate the performance of the proposed GPU-based parallel designs. Table 6.3 shows the computational platform details which are used for collecting results of both sequential and parallel implementations. Sequential implementation of the LSH was tested on the Intel Core i7, which has eight cores operating at a 3.6 GHz frequency and a 16 GB RAM. The sequential version is coded in C language, and its corresponding parallel version has been coded using the CUDA paradigm (version: CUDA 10.1). The GPU device, which is used to execute a parallel version, has the following specifications. A GPU device has a NVIDIA Tesla P100 GPU card with a compute capability version 6.0, 16 GB of the global memory, 56 Streaming Multiprocessors (SMs), where each SM has 64 cores operating at 1.33 GHz. The cost and time of constructing a feasible solution and improving the solution have been collected. The empirical performance analysis of the sequential version and corresponding parallel version of LSH have been explained in the following subsections.

6.4.1 Result Analysis of Sequential Version over Different CVRP Benchmark Instance Sets

The experimental analysis has been carried out on the following CVRP data sets: M set of Christofides instances (Christofides et al. 1979), Golden et al. (1998) instances, Li et al. (2005) instances, Kytöjoki et al. (2007) instances, Uchoa et al. (2017) instances,

Table 6.3: The hardware and software details used for the sequential and parallel implementation of the local search heuristic.

Description	Sequential	Parallel
Language	C	CUDA
CPU / GPU	Core i7-4790	Tesla P100
Architecture	Haswell	Pascal
Streaming Multiprocessor	NA	56
Cores	8	3584
Frequency	3.6 GHz	1.33 GHz
RAM / Global Memory	16 GB	16 GB
Shared Memory	NA	48 KB

and Belgium (Arnold and Sörensen, 2019) instance set. Tables 6.4, 6.5, 6.6, 6.7, and 6.8 presents results for five benchmark instance suites. For these tables, the first row represents instance name, cost of the constructed solution, the time required to construct a solution, a local optimal solution where solution improvement phase terminates, a best-known-solution for the corresponding instance, and total time spent in the solution improvement phase, respectively. There are five, twenty, twelve, twenty, and ten instances available in M, Golden, Li, Kytöjoki, and Belgium sets, respectively.

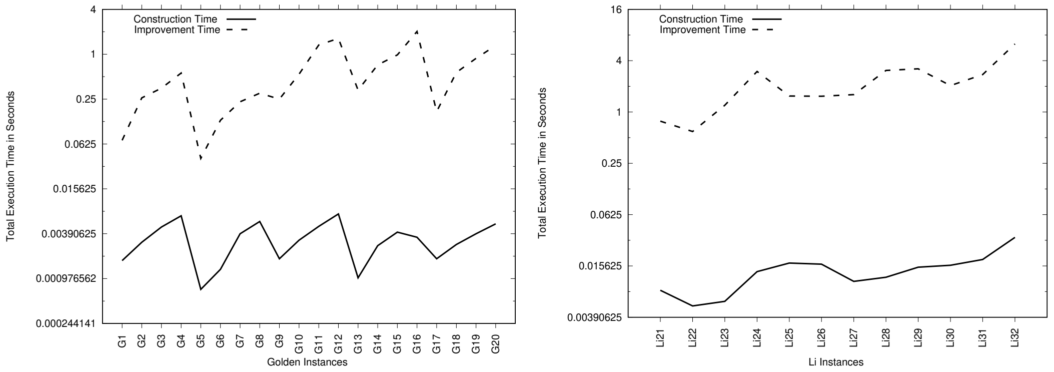
Table 6.4: The cost and time analysis of LSH on M set instances (Christofides et al. 1979). Time is given in seconds. Abbreviation used- N: Total customers, K: Minimum number of vehicles required to solve corresponding instance, SC: Solution Construction; BKS: Best Known Solution; SI: Solution Improvement.

Instance	N	K	Initial cost	SC Time	Final cost	BKS	SI Time
M-n101-k10.vrp	101	10	1404.35	0.00036	1118.83	820	0.0110
M-n121-k7.vrp	121	7	1888.68	0.00046	1325.8	1034	0.0138
M-n151-k12.vrp	151	12	1740.52	0.00075	1401.91	1053	0.0313
M-n200-k16.vrp	200	16	2167.54	0.00134	1638.36	1274	0.1201
M-n200-k17.vrp	200	17	2167.54	0.00133	1638.36	1275	0.1200

Table 6.4 presents the solution and time analysis for the M data set instances. These instances are small in size, i.e., up to 200 customers. Solving such instances using an LSH algorithm spends time in milliseconds. When solution quality is concerned, LSH is 1.36 times away from the BKS in the worst-case and 1.28× away in the best-case scenario.

Table 6.5: The cost and time analysis over Golden et al. (1998) instance set. Time is given in seconds.

Instance	N	K	Initial cost	SC Time	Final cost	BKS	SI Time
Golden_1.vrp	241	10	7687.3	0.0017	6670.62	5627.54	0.07
Golden_2.vrp	321	10	11994.33	0.0030	10014.21	8447.92	0.26
Golden_3.vrp	401	10	16247.29	0.0048	13645.36	11036.22	0.35
Golden_4.vrp	481	12	22293.6	0.0068	18584.15	13624.52	0.56
Golden_5.vrp	201	5	11804.83	0.0007	8939.28	6460.98	0.04
Golden_6.vrp	281	8	14305.56	0.0013	11679.71	8412.8	0.13
Golden_7.vrp	361	9	16223.67	0.0039	13504.84	10181.75	0.23
Golden_8.vrp	441	11	16540.95	0.0057	14294.06	11663.55	0.30
Golden_9.vrp	256	14	1075.35	0.0018	783.34	585.43	0.25
Golden_10.vrp	324	16	1305.28	0.0032	987.98	741.56	0.54
Golden_11.vrp	400	18	1666.67	0.0049	1245.2	918.45	1.33
Golden_12.vrp	484	19	2067.48	0.0072	1544.93	1107.19	1.63
Golden_13.vrp	253	27	1412.09	0.0010	1087.03	859.11	0.33
Golden_14.vrp	321	30	1867.05	0.0027	1382.57	1081.31	0.72
Golden_15.vrp	397	34	2357.36	0.0041	1742.36	1345.23	0.98
Golden_16.vrp	481	38	2833.35	0.0035	2043.87	1622.69	2.02
Golden_17.vrp	241	22	1155.04	0.0018	901.82	707.79	0.17
Golden_18.vrp	301	28	1664.74	0.0028	1176.9	997.52	0.57
Golden_19.vrp	361	33	2394.8	0.0039	1681.35	1366.86	0.88
Golden_20.vrp	421	41	3350.68	0.0053	2228.23	1820.09	1.31
Average				0.0035			0.63



(a) Golden data set

(b) Li data set

Figure 6.7: Execution time analysis of LSH spent in the solution construction and improvement phases for Golden and Li data sets.

When instance size grows, LSH also starts spending more time in the solution improvement phase. From Tables 6.5 and 6.6, it is noticed that LSH starts gradually

Table 6.6: The cost and time analysis using twelve instances of Li et al. (2005). Time is given in seconds.

Instance	N	K	Initial cost	SC Time	Final cost	BKS	SI Time
Li_21.vrp	561	10	27533.96	0.0081	22877.34	16212.74	0.78
Li_22.vrp	601	14	19435.62	0.0053	16860.79	14597.18	0.59
Li_23.vrp	641	10	33537.45	0.0060	26287.64	18801.12	1.20
Li_24.vrp	721	10	39948.51	0.0134	31312.44	21389.33	2.99
Li_25.vrp	761	17	23069.62	0.0169	19206.98	16902.16	1.54
Li_26.vrp	801	10	45574.75	0.0164	36016.48	23971.74	1.53
Li_27.vrp	841	19	23331.72	0.0103	19943.63	17488.74	1.60
Li_28.vrp	881	10	52823.72	0.0115	40817.43	26565.92	3.07
Li_29.vrp	961	10	58545.52	0.0151	45258.54	29154.34	3.21
Li_30.vrp	1041	10	63386.64	0.0159	49803.48	31742.51	2.04
Li_31.vrp	1121	10	70615.77	0.0186	54682.97	34330.84	2.75
Li_32.vrp	1201	10	75904.4	0.0338	58459.62	36919.24	6.28
Average				0.0143			2.30

Table 6.7: The cost and time analysis of LSH over Kytöjoki et al. (2007) instance set. Time is given in seconds.

Instance	N	K	Initial cost	SC Time	Final cost	BKS	SI Time
33.vrp	2401	10	169711.59	0.166	122634.02	75754.55	37.72
34.vrp	3601	10	262921.38	0.296	187110.06	114579.08	123.49
35.vrp	6001	10	460197.28	0.902	315759	192228.14	757.08
36.vrp	7201	10	542640.12	0.925	360941.94	231052.66	1504.20
37.vrp	8401	10	635326.56	1.761	413496.81	269877.19	1840.28
38.vrp	9601	10	723210.88	2.406	444206.12	308702.91	3008.22
39.vrp	10801	10	817841.38	1.967	502171	347537	3591.77
40.vrp	12001	10	903524.06	2.358	567184.19	386350.78	10702.50
41.vrp	13201	10	1008933	2.540	617296.25	425175.31	8115.99
42.vrp	14401	10	1106824.62	5.670	668950.69	463999.84	13234.90
43.vrp	16801	10	1260851	6.123	733591.75	541648.88	75085.16
44.vrp	20001	10	1517404.5	6.345	892366.31	645180.94	39346.01
E.vrp	9517	17	6905641.5	2.395	6077346	4757566.36	1221.84
M.vrp	10218	17	4572542.5	2.974	3670555.75	3170932.21	1800.14
R12.vrp	12001	812	1020104.75	5.416	792441.38	680832.79	55710.43
R3.vrp	3001	204	289439.69	0.144	218700.17	186219.59	736.40
R6.vrp	6001	406	535301.69	1.003	414486.25	352701.73	6673.78
R9.vrp	9001	609	775353.75	2.223	603867.44	517443.4	22376.54
S.vrp	8455	15	6429944	2.043	4320665.5	3333695.83	1471.28
W.vrp	7799	16	8235416.5	0.872	5955757	4559986.36	1474.01
Average				2.426			12440.59

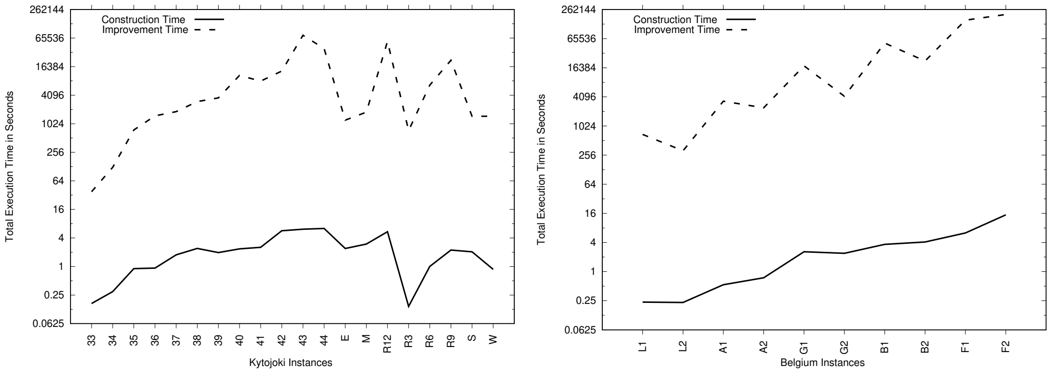
Table 6.8: The cost and time analysis of LSH over Belgium (Arnold and Sörensen 2019) instance set. Time is given in seconds.

Instance	N	K	Initial cost	SC Time	Final cost	BKS	SI Time
L1.vrp	3001	203	380510.19	0.234	246467.48	194381	685.19
L2.vrp	4001	46	204400.17	0.230	144827.23	113484	315.55
A1.vrp	6001	343	868495.31	0.535	598249.88	481338	3329.28
A2.vrp	7001	120	578501.94	0.744	399403.72	296055	2440.42
G1.vrp	10001	485	934140.19	2.568	614145.19	473568	17940.77
G2.vrp	11001	110	434350.31	2.388	331532.44	264512	4217.82
B1.vrp	15001	512	1055067.62	3.654	645495.25	507103	53010.61
B2.vrp	16001	182	673065.81	4.091	458719.41	355779	22647.48
F1.vrp	20001	684	13826885	6.289	9362481	7295447	158141.80
F2.vrp	30001	256	10103465	14.909	6192097	4504416	206708.16
Average				3.564			46943.71

spending more time in the solution improvement phase because the size of instances in Golden and Li sets are up to 481 and 1201 nodes, respectively. Figure 6.7 (a) shows that LSH spends at least 97.62% time and at most 99.82% time in the solution improvement phase, whereas Figure 6.7 (b) shows LSH spends 98.91% -99.62% portion of time in the solution improvement phase while solving Li et al. instances. These customer sizes are small in amount. When size goes beyond 1200 nodes, LSH significantly spends ample amount of CPU time in the solution improvement phase.

The real-time application requires thousands of customers to be satisfied. Kytöjoki et al. (2007) instance set contains instances of size between 3001-20001 nodes. Table 6.7 presents the instance wise solution quality and execution time analysis which is spent in the solution construction and solution improvement phases for Kytöjoki et al. (2007) instance set. In Table 6.7, the time required for the solution improvement phase is in hours. For instance, $43.vrp$, the solution improvement phase spends 20.85 hours to get its local optimal solution. On average, it spends 1.20 hours CPU time for 20 instances. Figure 6.8 (a) shows LSH consumes 99.56%-99.99% of time in the solution improvement phase for Kytöjoki et al. (2007) instance set.

Arnold and Sörensen (2019) have proposed the large-scale data set that contains up to 30000 customers, considering the real-time scenario. As customer size grows, the execution time also increases significantly. Table 6.8 shows that the solution improvement



(a) Kytöjoki data set

(b) Belgium data set

Figure 6.8: Execution time analysis of LSH spent in the solution construction and improvement phases for Kytöjoki et al. (2007) and Belgium (Arnold and Sörensen, 2019) data sets.

phase spends 13.03 hours on an average for ten instances of size up to 30000 customers. An instance of 30000 customers (i.e., F2.vrp) needs 2.39 days to trap into the local optimal solution. From figure 6.8 (b), it is noticed that LSH spends 99.92%-99.99% of time in the solution improvement phase. This observation tells that computing the solution for larger CVRP problems is a time-consuming process; hence finding the parallel version is important. Two GPU-based parallel strategies have been designed to mitigate execution time, which is being spent in the solution improvement phase. The following subsections explain the performance analysis of proposed GPU-based parallel strategies, namely route level and customer level designs.

6.4.2 Route Level Parallel Design

The route level parallel design is the natural way to apply parallelization in the solution improvement phase. In route level parallel design, each thread works separately on its associated route simultaneously in the one improvement heuristic at a time. Table 6.9 shows the cost and time analysis of this parallel strategy on Li et al. (2005) data set. It is noticed that this strategy does not reduce execution time. Instead, it worsens the speedup. Compared to the sequential approach, this parallel strategy is $1.81\times$ slower in the best-case, $5.77\times$ in the worst-case, and $3.39\times$ on an average, which is shown in Figure 6.9.

The reasons for this worse performance are followed. In this strategy, the number

Table 6.9: The performance analysis of route level parallel design over Li et al. (2005) instances. Time is given in seconds. Abbreviation used- PSI: Parallel version of Solution Improvement.

Instance	N	K	Initial cost	SC Time	Final cost	BKS	SI Time	PSI Time
Li_21.vrp	561	10	27533.96	0.0081	22877.34	16212.74	0.78	2.708867
Li_22.vrp	601	14	19435.62	0.0053	16860.79	14597.18	0.59	1.793499
Li_23.vrp	641	10	33537.45	0.0060	26287.64	18801.12	1.20	4.10188
Li_24.vrp	721	10	39948.51	0.0134	31312.44	21389.33	2.99	7.810976
Li_25.vrp	761	17	23069.62	0.0169	19206.98	16902.16	1.54	2.787952
Li_26.vrp	801	10	45574.75	0.0164	36016.48	23971.74	1.53	5.109024
Li_27.vrp	841	19	23331.72	0.0103	19943.63	17488.74	1.60	4.919494
Li_28.vrp	881	10	52823.72	0.0115	40817.43	26565.92	3.07	8.568896
Li_29.vrp	961	10	58545.52	0.0151	45258.54	29154.34	3.21	12.251009
Li_30.vrp	1041	10	63386.64	0.0159	49803.48	31742.51	2.04	11.79972
Li_31.vrp	1121	10	70615.77	0.0186	54682.97	34330.84	2.75	12.434641
Li_32.vrp	1201	10	75904.4	0.0338	58459.62	36919.24	6.28	19.343243
Average				0.014275			2.298333	7.802433

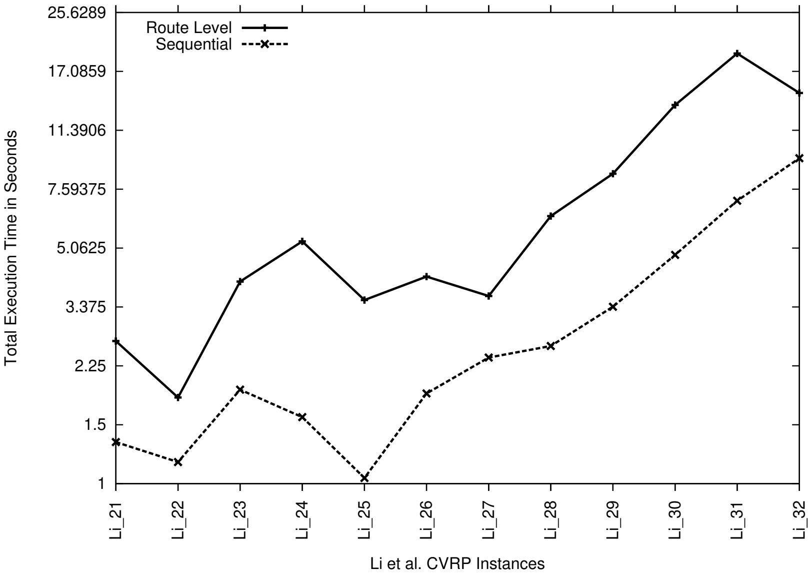


Figure 6.9: Time comparison of route level parallel design with sequential version on Li et al. (2005) data set.

of threads used is equal to the number of vehicles allowed per instance. The maximum and minimum number of vehicles permitted to use in the Li data set are 19 and 10, respectively. It means the maximum number of threads used in this parallel strategy are 19. In CUDA, a bunch of 32 threads, known as warp, is mapped to the Streaming Multiprocessor (SM). If a warp does not suffer control divergence, all 32 threads perform their computations simultaneously on the same instruction. In this parallel

Table 6.10: The performance of route level parallel design over X (Uchoa et al. 2017) data set. Speedup is a ratio between sequential and parallel version of SI time.

Instance	N	K	Initial Cost	SC Time	Final Cost	PSI Time	SI Time	Speedup
X-n219-k73.vrp	219	13	154908.61	0.0011	118750.68	0.16	0.38	2.34
X-n266-k58.vrp	266	58	124439.59	0.0016	82355.02	0.71	0.84	1.18
X-n317-k53.vrp	317	53	116344.2	0.0023	81323.53	1.23	2.56	2.07
X-n336-k84.vrp	336	84	227384.73	0.0026	157520.09	1.36	1.51	1.11
X-n376-k94.vrp	376	94	203926.78	0.0029	151054.22	0.75	2.88	3.82
X-n384-k52.vrp	384	52	105075.69	0.0031	74123.53	1.86	2.47	1.33
X-n420-k130.vrp	420	130	157091.25	0.0040	122377.92	1.60	2.22	1.39
X-n429-k61.vrp	429	61	100465.99	0.0044	74340.88	1.98	2.92	1.48
X-n469-k138.vrp	469	138	331598.12	0.0047	243563.02	1.60	4.42	2.77
X-n480-k70.vrp	480	70	131992.22	0.0051	99666.37	3.03	5.17	1.71
X-n548-k50.vrp	548	50	130800.34	0.0055	100258.44	4.49	6.32	1.41
X-n586-k159.vrp	586	159	285243.81	0.0071	204905.28	2.64	6.00	2.27
X-n599-k92.vrp	599	92	171988.58	0.0077	120882.88	4.11	7.67	1.87
X-n655-k131.vrp	655	131	134284.16	0.0099	108906.77	3.07	13.51	4.40
X-n733-k159.vrp	733	159	205729.66	0.0116	152020.62	5.65	6.86	1.22
X-n749-k98.vrp	749	98	124846.94	0.0113	90490.3	9.18	11.34	1.24
X-n819-k171.vrp	819	171	211374.03	0.0123	173226.84	5.01	15.38	3.07
X-n837-k142.vrp	837	142	286703.12	0.0157	210848.55	9.97	23.84	2.39
X-n856-k95.vrp	856	95	125451.69	0.0151	97690.31	8.52	20.97	2.46
X-n916-k207.vrp	916	207	489238.47	0.0165	362133.62	9.23	32.83	3.56
X-n957-k87.vrp	957	87	138820.48	0.0186	99555.78	13.60	34.56	2.54
							min	1.11
							max	4.40
							avg	2.17

strategy, only one warp is involved in the computation since 19 threads are required at the most. Another reason is that the neighborhood generation and its cost computation are performed sequentially on each route by an associated thread.

An investigation has been carried out to identify the number of vehicles resulting in positive speedup. It is observed that the route level parallel design offers a slight benefit when the number of vehicles used are more than 50 and the total number of customers are more than 200. Note that there are hardly some benchmarking instances available where the vehicle count constraint per instance is more than 50. Table 6.10 shows the results of route level parallel design, where it produces speedup gain over the Uchoa et al. (2017) data set. Two conditions have been considered to selects the instances for this experiment. 1) An instance should have more than 200 customers. 2) The minimum number of vehicles are allowed to use should be more than 50. The speedup achieved with this parallel strategy is $1.11\times$ faster in the worst-case, $4.40\times$ in the best-case, and $2.17\times$ on an average as compared to the sequential version for the same data set.

From these result analyses, it is noticed that this parallel strategy under-utilizes GPU compute capability since a lesser number of threads are involved in the actual computation. Moreover, the neighborhood generation and its cost computation are performed in a serial manner at each thread. Therefore, it is necessary to find a parallel strategy such that a large number of threads perform computation. A customer level parallel strategy has been designed, which improves the work distribution. The performance analysis of customer level parallel design is presented in the following subsection.

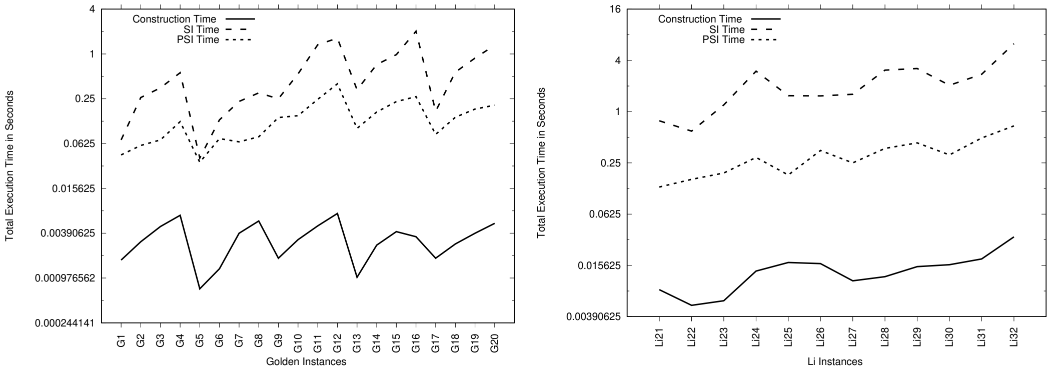
6.4.3 Customer Level Parallel Design

The customer level parallel strategy is designed to overcome issues associated with route level parallel design. In this strategy, each customer acts as a thread. This strategy allows a larger number of threads to involve in the computation. Table 6.11 shows the execution time spent in the sequential and parallel versions for the M data set (Christofides et al. 1979). It is noticed that the parallel version spent more time compared to its sequential counterpart for smaller instances, i.e., up to 120 customers. There are two main reasons for this slower performance. First, for smaller instances, the computational time is dominated by the communication time. The communication time is the time required for transferring data, which is required for performing the computation over the GPU, from the CPU to the GPU, and vice-versa. Second, lesser threads are involved in the computation for smaller instances, which under-utilizes the GPU's compute capability.

As instance size grows, customer level parallel design reveals its noticeable performance. Tables 6.12 and 6.13 shows the instance wise time comparison between the sequential and parallel version of the solution improvement phase for Golden and Li instances. For these medium-scale instances, i.e., up to 1200 customers, this proposed parallel strategy reduces execution time significantly. The customer level parallel design achieves 1.12×, 3.89×, and 7.56× speedup in the worst-case, average case, and best-case, respectively, for the Golden data set. Compared with the Li data set, this

Table 6.11: The execution time analysis of customer level parallel design with sequential counterpart on M (Christofides et al. 1979) data set.

Instance	N	K	SI Time	PSI Time	Speedup
M-n101-k10.vrp	101	10	0.011	0.029	0.39
M-n121-k7.vrp	121	7	0.014	0.025	0.55
M-n151-k12.vrp	151	12	0.031	0.031	1.00
M-n200-k16.vrp	200	16	0.120	0.079	1.53
M-n200-k17.vrp	200	17	0.120	0.074	1.62
				min	0.39
				max	1.62
				avg	1.02



(a) Golden data set

(b) Li data set

Figure 6.10: Comparison of time spent in sequential and parallel versions of solution improvement phase using Golden and Li data sets.

Table 6.12: The execution time analysis of customer level parallel design with sequential counterpart on Golden et al. (1998) instance set.

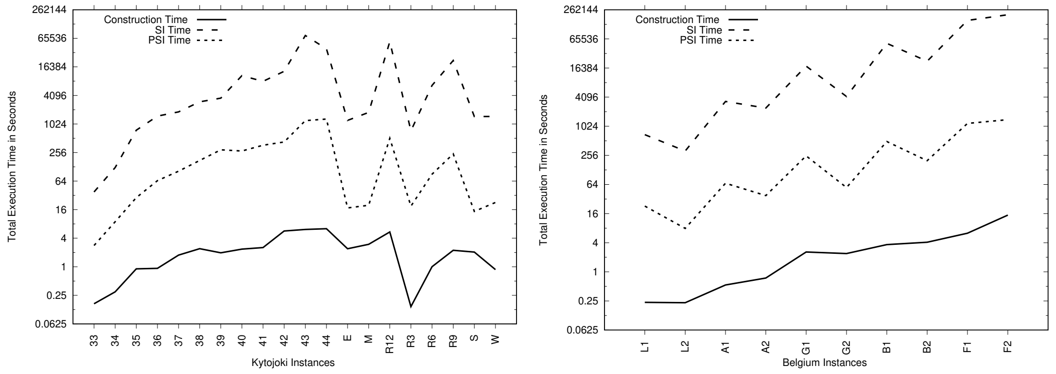
Instance	N	K	SI Time	PSI Time	Speedup
Golden_1.vrp	241	10	0.073	0.044	1.65
Golden_2.vrp	321	10	0.264	0.059	4.47
Golden_3.vrp	401	10	0.350	0.070	4.98
Golden_4.vrp	481	12	0.562	0.123	4.57
Golden_5.vrp	201	5	0.039	0.035	1.12
Golden_6.vrp	281	8	0.130	0.073	1.78
Golden_7.vrp	361	9	0.225	0.066	3.44
Golden_8.vrp	441	11	0.302	0.077	3.93
Golden_9.vrp	256	14	0.252	0.140	1.81
Golden_10.vrp	324	16	0.540	0.148	3.64
Golden_11.vrp	400	18	1.326	0.246	5.40
Golden_12.vrp	484	19	1.633	0.396	4.12
Golden_13.vrp	253	27	0.331	0.100	3.32
Golden_14.vrp	321	30	0.724	0.166	4.37
Golden_15.vrp	397	34	0.980	0.229	4.27
Golden_16.vrp	481	38	2.016	0.267	7.56
Golden_17.vrp	241	22	0.173	0.083	2.07
Golden_18.vrp	301	28	0.571	0.141	4.06
Golden_19.vrp	361	33	0.879	0.182	4.84
Golden_20.vrp	421	41	1.306	0.205	6.38
				min	1.12
				max	7.56
				avg	3.89

parallel design achieves 3.80×, 6.92×, and 10.37× in the worst-case, average case, and best-case, respectively. Figure 6.10 shows the time portion analysis of total execution time spent in solution construction, sequential solution improvement, and parallel solution improvement for all instances of Golden and Li data sets. Figure 6.10 (a) shows that SI spends at least 97.62% time and at most 99.82% time in the solution improvement phase, which is reduced down to 93.10%-99.00% using PSI, whereas Figure 6.10 (b) shows SI spends 98.91%-99.62% portion of time, which is now reduced to 91.41%-96.98% using PSI.

When this parallel design's performance is evaluated on the large-scale data sets, namely Kytöjoki et al. (2007) and Belgium (Arnold and Sörensen, 2019), it is noticed

Table 6.13: The execution time analysis of customer level parallel design with sequential counterpart on Li et al. (2005) data set.

Instance	N	K	SI Time	PSI Time	Speedup
Li_21.vrp	561	10	0.78	0.13	6.12
Li_22.vrp	601	14	0.59	0.16	3.80
Li_23.vrp	641	10	1.20	0.19	6.24
Li_24.vrp	721	10	2.99	0.29	10.37
Li_25.vrp	761	17	1.54	0.18	8.60
Li_26.vrp	801	10	1.53	0.35	4.37
Li_27.vrp	841	19	1.60	0.25	6.44
Li_28.vrp	881	10	3.07	0.37	8.34
Li_29.vrp	961	10	3.21	0.43	7.54
Li_30.vrp	1041	10	2.04	0.31	6.48
Li_31.vrp	1121	10	2.75	0.49	5.62
Li_32.vrp	1201	10	6.28	0.68	9.16
				min	3.80
				max	10.37
				avg	6.92



(a) Kytöjoki data set

(b) Belgium data set

Figure 6.11: Comparison of time spent in sequential and parallel versions of solution improvement phase using Kytöjoki and Belgium data sets.

that the customer level parallel design obtains higher speedup. Tables 6.14 and 6.15 shows the significant reduction in the execution time of solution improvement phase using parallel version. The parallel version achieves 12.31×, 47.54×, and 109.00× speedup in the worst-case, average case, and best-case, respectively compared to its sequential counterpart for Kytöjoki data set. Compared with Belgium instances, this parallel design obtains 29.91×, 83.52×, and 147.20× speedup in the worst-case, aver-

Table 6.14: The execution time analysis of customer level parallel design with sequential counterpart on Kytöjoki et al. (2007) data set.

Instance	N	K	SI Time	PSI Time	Speedup
33.vrp	2401	10	37.72	2.81	13.43
34.vrp	3601	10	123.49	8.88	13.90
35.vrp	6001	10	757.08	28.68	26.40
36.vrp	7201	10	1504.20	65.44	22.98
37.vrp	8401	10	1840.28	103.62	17.76
38.vrp	9601	10	3008.22	174.07	17.28
39.vrp	10801	10	3591.77	291.78	12.31
40.vrp	12001	10	10702.50	275.95	38.78
41.vrp	13201	10	8115.99	364.93	22.24
42.vrp	14401	10	13234.90	426.95	31.00
43.vrp	16801	10	75085.16	1207.23	62.20
44.vrp	20001	10	39346.01	1302.29	30.21
E.vrp	9517	17	1221.84	17.39	70.25
M.vrp	10218	17	1800.14	19.80	90.91
R12.vrp	12001	812	55710.43	511.09	109.00
R3.vrp	3001	204	736.40	18.97	38.81
R6.vrp	6001	406	6673.78	89.24	74.79
R9.vrp	9001	609	22376.54	240.00	93.23
S.vrp	8455	15	1471.28	14.67	100.31
W.vrp	7799	16	1474.01	22.67	65.02
				min	12.31
				max	109.00
				avg	47.54

age case, and best-case, respectively. The maximum speedup is recorded for Belgium instance, $F2.vrp$, which is $147.20\times$ faster. Figure 6.11 shows a comparative time analysis of time spent in sequential and parallel versions of the solution improvement phase using Kytöjoki and Belgium data sets. Figure 6.11 (a) shows that SI spends 99.56%-99.99% time in the solution improvement phase, whereas its parallel counterpart spends 86.94%-99.51%. Figure 6.11 (b) shows SI spends 99.92%-99.99% portion of time, whereas PSI spends 95.86%-99.46%. For example, LSH spends 2.39 days of execution time to reach its local solution for a $F2.vrp$ instance. In contrast, PLSH solves the same instance in 23.65 minutes with customer-level parallel design.

The primary reason for this speedup is that this parallel strategy allows many threads

Table 6.15: The execution time analysis of customer level parallel design with sequential counterpart on Belgium (Arnold and Sörensen, 2019) data set.

Instance	N	K	SI Time	PSI Time	Speedup
L1.vrp	3001	203	685.19	22.91	29.91
L2.vrp	4001	46	315.55	7.91	39.90
A1.vrp	6001	343	3329.28	67.83	49.08
A2.vrp	7001	120	2440.42	37.71	64.72
G1.vrp	10001	485	17940.77	249.66	71.86
G2.vrp	11001	110	4217.82	55.30	76.28
B1.vrp	15001	512	53010.61	493.77	107.36
B2.vrp	16001	182	22647.48	198.61	114.03
F1.vrp	20001	684	158141.80	1172.46	134.88
F2.vrp	30001	256	206708.16	1404.30	147.20
				min	29.91
				max	147.20
				avg	83.52

to participate in the actual computation. For example, consider an instance $F2.vrp$. Since each customer act as a thread, the number of threads involved in the computation are 30000. For each CUDA block, we have setup to use at most 128 threads. Therefore, a total of 30000 threads across 235 blocks are involved in computation, resulting in a higher speedup. Table 6.15 shows that the customer-level parallel design obtains a speedup gain in the linear order as the customer size increases. This strategy is also compared with the route level design. Table 6.16 shows the execution time comparison of customer level parallel design with sequential counterpart and the route level parallel approach over the X data set. Compared to the sequential version, this parallel design is $1.99\times$ faster in the worst-case, $9.83\times$ on an average, and $22.97\times$ in the best-case. The speedup achieved in the best-case, average, and the worst-case is $9.03\times$, $4.69\times$, and $1.99\times$, respectively, compared to the route level parallel design. From these result analysis, it can be determined that the more threads involve in computation independently, the better speedup it will produce.

Table 6.16: The performance of route level parallel design over Uchoa et al. (2017) data set. Abbreviations used- SI: Solution Improvement; RSI: Route level Solution Improvement; CSI: Customer level Solution Improvement; S1: Speedup ratio between SI and CSI; S2: Speedup ratio between RSI and CSI.

Instance	N	K	SI Time	RSI Time	CSI Time	S1	S2
X-n219-k73.vrp	219	13	0.38	0.16	0.08	4.66	1.99
X-n266-k58.vrp	266	58	0.84	0.71	0.23	3.70	3.14
X-n317-k53.vrp	317	53	2.56	1.23	0.29	8.73	4.21
X-n336-k84.vrp	336	84	1.51	1.36	0.36	4.23	3.82
X-n376-k94.vrp	376	94	2.88	0.75	0.26	11.05	2.89
X-n384-k52.vrp	384	52	2.47	1.86	0.34	7.27	5.47
X-n420-k130.vrp	420	130	2.22	1.60	0.51	4.35	3.13
X-n429-k61.vrp	429	61	2.92	1.98	0.36	8.09	5.48
X-n469-k138.vrp	469	138	4.42	1.60	0.53	8.30	3.00
X-n480-k70.vrp	480	70	5.17	3.03	0.52	9.86	5.77
X-n548-k50.vrp	548	50	6.32	4.49	0.63	10.05	7.14
X-n586-k159.vrp	586	159	6.00	2.64	0.86	6.99	3.08
X-n599-k92.vrp	599	92	7.67	4.11	0.83	9.20	4.93
X-n655-k131.vrp	655	131	13.51	3.07	0.72	18.67	4.25
X-n733-k159.vrp	733	159	6.86	5.65	1.22	5.64	4.64
X-n749-k98.vrp	749	98	11.34	9.18	1.35	8.38	6.78
X-n819-k171.vrp	819	171	15.38	5.01	1.41	10.91	3.55
X-n837-k142.vrp	837	142	23.84	9.97	1.89	12.61	5.27
X-n856-k95.vrp	856	95	20.97	8.52	1.15	18.16	7.37
X-n916-k207.vrp	916	207	32.83	9.23	2.62	12.53	3.52
X-n957-k87.vrp	957	87	34.56	13.60	1.50	22.97	9.03
					min	1.99	1.99
					max	22.97	9.03
					avg	9.83	4.69

6.4.4 Solution Quality

As far as solution quality is concerned, the local search heuristic used in this chapter does not offer better solutions compared to the existing state-of-the-art metaheuristics algorithms (Kytöjoki et al. 2007; Li et al. 2005; Uchoa et al. 2017). LHS stops exploring neighborhood solutions when it traps into a local optima. The perturbation and randomization are not incorporated, as in metaheuristics algorithms. The focus of this work is to find data and thread level parallelism in improvement heuristics. The lower bound observed in the final solutions are $1.28\times$, $1.18\times$, $1.14\times$, $1.16\times$, and $1.24\times$ far compared to the best-known solutions of M, Golden, Li, Kytöjoki, and Belgium data sets, respectively. The upper bound on final solutions are within $1.31\times$, $1.40\times$, $1.59\times$, $1.64\times$, and $1.38\times$ far compared to the best-known solutions of M, Golden, Li, Kytöjoki, and Belgium data sets, respectively. The LSH can be replaced with the well-known metaheuristics such as variable neighborhood search heuristic (VNSH) (Kytöjoki et al. 2007), iterated local search based metaheuristic algorithm (ILS-SP) (Subramanian et al. 2013), and unified hybrid genetic search (UHGS) (Vidal et al. 2012, 2014), in order to reduce the gap rates present in the final solutions. Finding the independent portion and designing the parallel strategies for such metaheuristics is a future work.

6.5 SUMMARY

In this chapter, two GPU-based parallel strategies have been designed for the local search heuristic to solve the Capacitated Vehicle Routing Problem (CVRP) to reduce the execution time.

The improvement heuristic is applied after the feasible solution is created in order to improve the solution quality. In the solution improvement phase, five improvement heuristic approaches have been applied in which three are intra-route heuristics, and two are inter-route heuristics. The three intra-route heuristics are 2-opt, or-opt, and 3-opt. Two inter-route heuristics are swap and relocate. Experiment shows that more than 90% of time is spent in solution improvement phase. The execution time requirement is in hours and days for Belgium instances which sizes in range of 3000-30000 customers. The execution time of solution improvement phase can be reduced using the data and thread level parallelisms since there exists less dependency in its computation.

Two GPU-based parallel strategies have been designed for the improvement heuristic phase. The first is the route level, and another is the customer level strategy. In the route level parallel strategy, one thread per route is mapped to improve routes simultaneously in each heuristic. The route level strategy is found in underutilizing GPU computability. This happens due to the number of vehicles allowed to use per CVRP instance is short in size. Therefore another parallel strategy is designed, namely customer level parallel design, in which each customer can act as a thread. The performance of this strategy is performed on a wide variety of data sets ranging from 101 to 30000 customers. The average speedup achieved with the customer level parallel design is $3.89\times$, $6.92\times$, $47.52\times$, and $83.52\times$ for Golden, Li, Kytöjoki, and Belgium data sets, respectively over its sequential counterpart. The maximum speedup is achieved up to 147.19 times faster with respect to sequential version. This maximum speedup recorded is for Belgium $F2.vrp$ instance.

CHAPTER 7

CONCLUSIONS AND FUTURE SCOPE

This thesis presents the GPU-based parallel models to the existing metaheuristic algorithms to solve different combinatorial optimization problems in lesser time. Many optimization problems deal with real-time applications. Providing faster solutions to such applications is of great importance. This thesis aims to reduce the execution time spends in the improvement phase of metaheuristic algorithms. This thesis presents cost quality and speedup improved implementations of Combinatorial Optimization NP hard problems, viz., Travelling Salesman Problem (TSP), Minimum Latency Problem (MLP), and Capacitated Vehicle Routing Problem (CVRP), on the GPU.

The first part of the thesis work presents four GPU-based parallel strategies for the single solution-based metaheuristic, namely, Iterative Hill Climbing (IHC) algorithm to solve large-size Traveling Salesman Problem (TSP) instances. The Parallel IHC (PIHC) approach is evaluated and compared with LOGO and TSP2.2 GPU based state-of-the-art TSP solvers and it is observed that PIHC approach obtains results with error rate in the range of 0.72% (best case) - 8.06% (worst case). The PIHC produces a speedup of up to 979.66 over the GPU based TSP2.2 implementation. Overall, PIHC receives 68.34× on average and 193× in the best case over its corresponding sequential implementation. PIHC is the single solution-based heuristic designed to solve TSP instances. Subsequent chapter in the thesis work is the design of parallel strategies for the population of solutions-based heuristic.

The next contribution is the GPU-based parallel approach for Ant Colony Optimization (ACO) algorithm to solve large-scale TSP. The compute and time intensive section of the ACO algorithm is the construction of ants and the iterative improvement till convergence. Two GPU-based parallel strategies, namely, data-level and task-level parallel approaches for the ACO algorithm have been implemented. Task-level parallel approach maps one ant per thread whereas in data-level approach one ant is mapped per thread block. Task-level approach is up to 22 faster over the sequential approach. In data-level approach, multiple threads construct each ant's solution. Therefore, data level approach outperforms the task-level approach by 1.02 to 3.37 times for instances up to 33810 cities. When data-level approach is compared with sequential counterpart, up to 60 speedup is observed for the instances in the range of 100 - 33810 cities. When solution quality is concerned, ACO with 2-opt produces good quality solutions which has error rates in the range 0.52% - 4.97% for instances up to 33810 cities.

In the next piece of work, a Deterministic Local Search Heuristic (DLSH) that always reaches the same local solution on multiple trials is presented. The state-of-the-art evaluates a move in O(1) time using the preprocessed local data. The thesis presents the move evaluation procedure for the swap method, which computes a move in O(1) time without the need for any preprocessed data. DLSH obtains optimal solutions for 15 out of 40 instances. For TRP instances of size 50, gap rates are in the range of 2.02-13.48%. DLSH is compared with the state-of-the-art MLP solvers. DLSH has gap rates in the range of 0.52-16.27%, 4.57-12.40%, and 7.71-13.52% for the TRP instance sets of sizes 100, 200, and 500 compared to the solutions of GILS-RVND. However, DLSH reaches the new best solutions for the five TSPLIB instances, namely eil51, berlin52, pr107, rat195, and pr226. The GPU-based parallel DLSH (PDLSH) is proposed to compute neighborhood methods on the GPU. PDLSH achieves a speedup up to 179.75 for TRP and TSPLIB instances of sizes 10-7397.

In the last contribution of the thesis, two GPU-based parallel strategies for the improvement heuristic phase are presented. The first is the route level, and the second is the customer level strategy. In the route level parallel strategy, one thread per route is mapped to improve routes simultaneously in each heuristic. The route level strategy is found in underutilizing GPU computability. This happens due to the number of vehicles allowed to use per CVRP instance is short in size. Therefore another parallel strategy is designed, namely customer level parallel design, in which each customer can act as a thread. The performance of this strategy is performed on a wide variety of data sets ranging from 101 to 30000 customers. The average speedup achieved with the customer level parallel design is $3.89\times$, $6.92\times$, $47.52\times$, and $83.52\times$ for Golden, Li, Kytöjoki, and Belgium data sets, respectively over its sequential counterpart. The maximum speedup is achieved up to 147.19 times faster with respect to sequential version. This maximum speedup recorded is for Belgium $F2.vrp$ instance.

From this thesis, it can be inferred that the execution time portion requires in the improvement phase of the metaheuristic algorithms have been significantly reduced with the help of an efficient GPU-based parallel model. Sequential metaheuristic spends an inadmissible time solving the combinatorial optimization problem, where instance size is more than 10000 nodes. When existing metaheuristic algorithms are studied considering TSP, MLP, and CVRP, it is found that more than 90% of execution time is being spent in the solution improvement phase. Therefore, reducing the time portion spent in the solution improvement phase is of great importance. The GPU-based parallel models presented in the thesis have shown the efficacy to reduce the solution improvement phase time portion. Setting up initial solution using the nearest neighborhood is better than random approach that results in improved solution quality. Therefore, GPU-based parallel models can be used for solving various combinatorial optimization problems efficiently.

FUTURE SCOPE

The metaheuristic features, such as, randomization and perturbation, have not been considered in metaheuristics used in this thesis. The solution quality is further improved considering these mentioned features. In chapter 5, as an extension, a GPU-based parallel model can be designed for the metaheuristics, such as Variable Neighborhood Descent (VND), Variable Neighborhood Search (VNS), Generalized Random Adaptive Search Procedure (GRASP), and Iterative Local Search (ILS) metaheuristics. In chapter 6, parallelization can be applied to the best-performing metaheuristics approaches, which produce the best-known solutions for the benchmarking instances. The local search heuristic algorithm used in this chapter does not offer better solution quality compared to the state-of-the-art metaheuristic algorithms.

BIBLIOGRAPHY

Abdelatti, M. F. and Sodhi, M. S. (2020). “An improved gpu-accelerated heuristic technique applied to the capacitated vehicle routing problem.” In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO ’20, Association for Computing Machinery, New York, NY, USA, 663–671.

Abeledo, H., Fukasawa, R., Pessoa, A. and Uchoa, E. (2013). “The time dependent traveling salesman problem: polyhedra and algorithm.” *Mathematical Programming Computation*, 5(1), 27–55.

Afif, M., Said, Y. and Atri, M. (2020). “Computer vision algorithms acceleration using graphic processors NVIDIA CUDA.” *Cluster Computing*.

Ai, T. J. and Kachitvichyanukul, V. (2009). “Particle swarm optimization and two solution representations for solving the capacitated vehicle routing problem.” *Computers & Industrial Engineering*, 56(1), 380 – 387.

Alawneh, L., Shehab, M. A., Al-Ayyoub, M., Jararweh, Y. and Al-Sharif, Z. A. (2020). “A scalable multiple pairwise protein sequence alignment acceleration using hybrid CPU-GPU approach.” *Cluster Computing*.

Alba, E. and Dorronsoro, B. (2004). “Solving the vehicle routing problem by using cellular genetic algorithms.” In Gottlieb, J. and Raidl, G. R., editors, *Evolutionary Computation in Combinatorial Optimization*, Springer Berlin Heidelberg, Berlin, Heidelberg, 11–20.

Altabeeb, A. M., Mohsen, A. M., Abualigah, L. and Ghallab, A. (2021). “Solving capacitated vehicle routing problem using cooperative firefly algorithm.” *Applied Soft Computing*, 108, 107403.

Angluin, D. and Valiant, L. G. (1979). "Fast probabilistic algorithms for hamiltonian circuits and matchings." *Journal of Computer and System Sciences*, 18(2), 155–193.

Araujo, R. P., Coelho, I. M. and Marzulo, L. A. J. (2020). "A multi-improvement local search using dataflow and gpu to solve the minimum latency problem." *Parallel Computing*, 102661.

Archer, A. and Blasiak, A. (2010). "Improved approximation algorithms for the minimum latency problem via prize-collecting strolls." In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 429–447.

Arnold, F. and Sörensen, K. (2019). "Knowledge-guided local search for the vehicle routing problem." *Computers & Operations Research*, 105, 32 – 46.

Arora, S. and Karakostas, G. (2003). "Approximation schemes for minimum latency problems." *SIAM Journal on Computing*, 32(5), 1317–1337.

Avci, M. and Avci, M. G. (2017). "A grasp with iterated local search for the traveling repairman problem with profits." *Computers & Industrial Engineering*, 113, 323 – 332.

Baker, B. M. and Ayechew, M. (2003). "A genetic algorithm for the vehicle routing problem." *Computers & Operations Research*, 30(5), 787 – 800.

Ban, H. B. and Duc, N. N. (2014). "A parallel algorithm combines genetic algorithm and ant colony algorithm for the minimum latency problem." In *Proceedings of the Fifth Symposium on Information and Communication Technology*, SoICT '14, Association for Computing Machinery, New York, NY, USA.

Ban, H.-B. and Nguyen, D.-N. (2017). "A meta-heuristic algorithm combining between tabu and variable neighborhood search for the minimum latency problem." *Fundamenta Informaticae*, 156, 21–41.

Baraglia, R., Hidalgo, J. I. and Perego, R. (2001). “A hybrid heuristic for the traveling salesman problem.” *IEEE Transactions on Evolutionary Computation*, 5(6), 613–622.

Barbarosoglu, G. and Ozgur, D. (1999). “A tabu search algorithm for the vehicle routing problem.” *Computers & Operations Research*, 26(3), 255 – 270.

Bentley, J. L. (1990). “Experiments on traveling salesman heuristics.” *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, 91–99.

Bianco, L., Mingozzi, A. and Ricciardelli, S. (1993). “The traveling salesman problem with cumulative costs.” *Networks*, 23(2), 81–91.

Blum, A., Chalasani, P., Coppersmith, D., Pulleyblank, B., Raghavan, P. and Sudan, M. (1994). “The minimum latency problem.” In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC ’94, ACM, New York, NY, USA, 163–171.

Bomze, I. M., Budinich, M., Pardalos, P. M. and Pelillo, M. (1999). *The Maximum Clique Problem*, 1–74. Springer US, Boston, MA.

Boqun, W., Hailong, Z., Jun, N., Jie, W., Xinchen, Y., Toktonur, E., Meng, Z., Jia, L. and Wanqiong, W. (2020). “Multipopulation genetic algorithm based on gpu for solving tsp problem.” *Mathematical Problems in Engineering*, 2020.

Boschetti, M. A., Maniezzo, V. and Strappaveccia, F. (2017). “Route relaxations on gpu for vehicle routing problems.” *European Journal of Operational Research*, 258(2), 456 – 466.

Braekers, K., Ramaekers, K. and Nieuwenhuyse, I. V. (2016). “The vehicle routing problem: State of the art classification and review.” *Computers & Industrial Engineering*, 99, 300 – 313.

Bullnheimer, B., Hartl, R. and Strauss, C. (1999). “An improved ant system algorithm for thevehicle routing problem.” *Annals of Operations Research*, 89(0), 319–328.

Campbell, A. M., Vandenbussche, D. and Hermann, W. (2008). "Routing for relief efforts." *Transportation Science*, 42(2), 127–145.

Campos, V. and Mota, E. (2000). "Heuristic procedures for the capacitated vehicle routing problem." *Computational Optimization and Applications*, 16(3), 265–277.

Carvalho, P., Cruz, R., Drummond, L. M. A., Bentes, C., Clua, E., Cataldo, E. and Marzulo, L. A. J. (2020). "Kernel concurrency opportunities based on GPU benchmarks characterization." *Cluster Computing*, 23, 177–188.

Cecilia, J. M., García, J. M., Nisbet, A., Amos, M. and Ujaldón, M. (2013). "Enhancing data parallelism for ant colony optimization on gpus." *Journal of Parallel and Distributed Computing*, 73(1), 42 – 51.

Chaudhuri, K., Godfrey, B., Rao, S. and Talwar, K. (2003). "Paths, trees, and minimum latency tours." In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, 36–45.

Che, S., Che, S., Boyer, M., Boyer, M., Meng, J., Meng, J., Sheaffer, J. W., Sheaffer, J. W., Skadron, K. and Skadron, K. (2008). "A performance study of general-purpose applications on graphics processors using CUDA." *Journal of Parallel and Distributed Computing*, 68(10), 1370–1380.

Chen, A.-l., Yang, G.-k. and Wu, Z.-m. (2006). "Hybrid discrete particle swarm optimization algorithm for capacitated vehicle routing problem." *Journal of Zhejiang University-SCIENCE A*, 7(4), 607–614.

Christofides, N. (1976). "Worst-case analysis of a new heuristic for the travelling salesman problem." Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University.

Christofides, N., Mingozzi, A., Tooth, P. and Sandi, C. (1979). *Combinatorial optimization.* Wiley Publication.

Clarke, G. and Wright, J. W. (1964). "Scheduling of vehicles from a central depot to a number of delivery points." *Operations Research*, 12(4), 568–581.

Cordeau, J.-F., Gendreau, M., Laporte, G., Potvin, J.-Y. and Semet, F. (2002). “A guide to vehicle routing heuristics.” *Journal of the Operational Research Society*, 53(5), 512–522.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2009). *Introduction to Algorithms*, The MIT Press.

Craus, M. and Rudeanu, L. (2004). “Parallel framework for ant-like algorithms.” In *Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, 36–41.

Dantzig, G. B. and Ramser, J. H. (1959). “The truck dispatching problem.” *Manage. Sci.*, 6(1), 80–91.

David Applegate, Ribert Bixby, V. C. W. C. “Concorde tsp solver.” `http://www.math.uwaterloo.ca/tsp/concorde.html`). [online; accessed 01-mar-2018].

Delévacq, A., Delisle, P., Gravel, M. and Krajecki, M. (2013). “Parallel ant colony optimization on graphics processing units.” *Journal of Parallel and Distributed Computing*, 73(1), 52 – 61. Metaheuristics on GPUs.

Delévacq, A., Delisle, P. and Krajecki, M. (2012). *Parallel GPU Implementation of Iterated Local Search for the Travelling Salesman Problem*, 372–377. Springer Berlin Heidelberg, Berlin, Heidelberg.

Delisle, P., Krajecki, M. and Gravel, M. (2009). “Multi-colony parallel ant colony optimization on smp and multi-core computers.” In *2009 World Congress on Nature Biologically Inspired Computing (NaBIC)*, 318–323.

Dewilde, T., Cattrysse, D., Coene, S., Spieksma, F. and Vansteenwegen, P. (2013). “Heuristics for the traveling repairman problem with profits.” *Computers & Operations Research*, 40(7), 1700 – 1707.

Dorigo, M. and Gambardella, L. (1997). "Ant colony system: a cooperative learning approach to the traveling salesman problem." *IEEE Transactions on Evolutionary Computation*, 1(1), 53–66.

Eksioglu, B., Vural, A. V. and Reisman, A. (2009). "The vehicle routing problem: A taxonomic review." *Computers & Industrial Engineering*, 57(4), 1472 – 1483.

Ezzine, I. O., Chabchoub, H. and Semet, F. (2010). "New formulations for the traveling repairman problem." In *Proceedings of the 8th ENIM IFAC International Conference of Modeling and Simulation, Lavoisier*, 1889–1894.

Fakcharoenphol, J., Harrelson, C. and Rao, S. (2007). "The k-traveling repairmen problem." *ACM Trans. Algorithms*, 3(4).

Fischetti, M., Laporte, G. and Martello, S. (1993). "The delivery man problem and cumulative matroids." *Operations Research*, 41(6), 1055–1064.

Fisher, M. L. (1994). "Optimal solution of vehicle routing problems using minimum k-trees." *Operations Research*, 42(4), 626–642.

Fosin, J., Davidović, D. and Carić, T. (2013). "A GPU implementation of local search operators for symmetric travelling salesman problem ." *Promet Traffic Transp*, 25(3), 225–234.

Fujimoto, N. and Tsutsui, S. (2011). "A highly-parallel tsp solver for a gpu computing platform." In *Proceedings of the 7th International Conference on Numerical Methods and Applications*, NMA'10, Springer-Verlag, Berlin, Heidelberg, 264–271.

García, A., Jodrá, P. and Tejel, J. (2002). "A note on the traveling repairman problem." *Networks*, 40(1), 27–31.

Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA.

Gendreau, M., Hertz, A. and Laporte, G. (1994). "A tabu search heuristic for the vehicle routing problem." *Management Science*, 40(10), 1276–1290.

Gendreau, M., Iori, M., Laporte, G. and Martello, S. (2004). "A tabu search heuristic for the vehicle routing problem with two-dimensional loading constraints." *Networks*, 51, 4–18.

Geng, X., Zhang, H., Zhao, Z. and Ma, H. (2020). "Interference-aware parallelization for deep learning workload in GPU cluster." *Cluster Computing*.

Goemans, M. and Kleinberg, J. (1998). "An improved approximation ratio for the minimum latency problem." *Mathematical Programming*, 82(1), 111–124.

Golden, B. L., Wasil, E. A., Kelly, J. P. and Chao, I.-M. (1998). *The Impact of Metaheuristics on Solving the Vehicle Routing Problem: Algorithms, Problem Sets, and Computational Results*, 33–56. Springer US, Boston, MA.

Gutin, G. and Punnen, A. P., editors (2002). *The traveling salesman problem and its variations*, Combinatorial optimization, Kluwer Academic, Dordrecht, London.

Ha Bang, B., Nguyen, K., CUONG NGO, M. and Nguyen Duc, N. (2013). "An efficient exact algorithm for the minimum latency problem." *Progress in Informatics*, 167–174.

Helsgaun, K. (2000). "An effective implementation of the lin-kernighan traveling salesman heuristic." *European Journal of Operational Research*, 126(1), 106 – 130.

Jensen, T. R. and Toft, B. (2011). *Graph coloring problems*, volume 39, John Wiley & Sons.

Jin, J., Crainic, T. G. and Løkketangen, A. (2014). "A cooperative parallel metaheuristic for the capacitated vehicle routing problem." *Computers & Operations Research*, 44, 33 – 41.

Johnson, D. S. and Mcgeoch, L. A. (1997). *The Traveling Salesman Problem: A Case Study in Local Optimization*.

Kim, S., Kim, D., Son, Y. and Eom, H. (2020). "Towards predicting GPGPU performance for concurrent workloads in Multi-GPGPU environment." *Cluster Computing*.

Kolmogorov, V. (2009). “Blossom v: a new implementation of a minimum cost perfect matching algorithm.” *Mathematical Programming Computation*, 1(1), 43–67.

Kytöjoki, J., Nuortio, T., Bräysy, O. and Gendreau, M. (2007). “An efficient variable neighborhood search heuristic for very large scale vehicle routing problems.” *Comput. Oper. Res.*, 34(9), 2743–2757.

Laporte, G., Gendreau, M., Potvin, J.-Y. and Semet, F. (2000). “Classical and modern heuristics for the vehicle routing problem.” *International Transactions in Operational Research*, 7(4), 285 – 300.

Laporte, G. and Nobert, Y. (1987). “Exact algorithms for the vehicle routing problem.” In *Surveys in Combinatorial Optimization*, volume 132 of *North-Holland Mathematics Studies*, North-Holland, 147 – 184.

Lawler, E. L. (1963). “The quadratic assignment problem.” *Management science*, 9(4), 586–599.

Lenstra, J. K. and Kan, A. H. G. R. (1975). “Some simple applications of the travelling salesman problem.” *Operational Research Quarterly (1970-1977)*, 26(4), 717–733.

Li, F., Golden, B. and Wasil, E. (2005). “Very large-scale vehicle routing: new test problems, algorithms, and results.” *Computers & Operations Research*, 32(5), 1165 – 1179.

Li, J., Hu, X., Pang, Z. and Qian, K. (2009). “A parallel ant colony optimization algorithm based on fine-grained model with gpu-accelerated.” *International Journal of Innovative Computing, Information and Control*, 24, 3707 – 3716.

Lin, S. (1965). “Computer solutions of the traveling salesman problem.” *Bell System Technical Journal*, 44(10), 2245–2269.

Lu, Y., Hao, J.-K. and Wu, Q. (2019). “Hybrid evolutionary search for the traveling repairman problem with profits.” *Information Sciences*, 502, 91 – 108.

Lucena, A. (1990). “Time-dependent traveling salesman problem–the deliveryman case.” *Networks*, 20(6), 753–763.

Luong, T. V., Melab, N. and Talbi, E.-G. (2013). "GPU Computing for Parallel Local Search Metaheuristics." *IEEE Transactions on Computers*, 62(1), 173–185.

Matai, R., Mittal, M. and Singh, S. (2010). *Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches*, INTECH Open Access Publisher.

Máximo, V. R. and Nascimento, M. C. (2021). "A hybrid adaptive iterated local search with diversification control to the capacitated vehicle routing problem." *European Journal of Operational Research*.

Merz, P. and Freisleben, B. (2001). "Memetic algorithms for the travelling salesman problem." *Complex Systems*, 297–345.

Monmarché, N., Ramat, E., Dromel, G., Slimane, M. and Venturini, G. (1999). "On the similarities between as, bsc and pbil: toward the birth of a new meta-heuristic." In *Complex Systems*, Citeseer.

Muyldermans, L., Beullens, P., Cattrysse, D. and Oudheusden, D. V. (2005). "Exploring variants of 2-opt and 3-opt for the general routing problem." *Operations Research*, 53(6), 982–995.

Méndez-Díaz, I., Zabala, P. and Lucena, A. (2008). "A new formulation for the traveling deliveryman problem." *Discrete Applied Mathematics*, 156(17), 3223 – 3237. Cologne/Twente Workshop on Graphs and Combinatorial Optimization.

Neil, M. A. O. and Burtscher, M. (2015). "Rethinking the Parallelization of Random-Restart Hill Climbing A Case Study in Optimizing a 2-Opt TSP Solver for GPU Execution." *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 99–108.

Ngueveu, S. U., Prins, C. and Calvo, R. W. (2010). "An effective memetic algorithm for the cumulative capacitated vehicle routing problem." *Computers & Operations Research*, 37(11), 1877 – 1885. Metaheuristics for Logistics and Vehicle Routing.

NVIDIA. "Cuda programming guide." `http://docs.nvidia.com/cuda/index.html`). [Online; accessed 01-jul-2020].

O'Neil, M., Tamir, D. and Burtscher, M. (2011). "A parallel gpu version of the traveling salesman problem." *The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, 348–353.

Osman, I. H. (1993). "Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem." *Annals of Operations Research*, 41(4), 421–451.

Pereira Araujo, R., Machado Coelho, I. and Marzulo, L. A. J. (2018). "A dvnd local search implemented on a dataflow architecture for the minimum latency problem." In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 1250–1259.

Pisinger, D. and Ropke, S. (2007). "A general heuristic for vehicle routing problems." *Comput. Oper. Res.*, 34(8), 2403–2435.

Prins, C. (2004). "A simple and effective evolutionary algorithm for the vehicle routing problem." *Computers & Operations Research*, 31(12), 1985 – 2002.

Randall, M. and Lewis, A. (2002). "A parallel implementation of ant colony optimization." *Journal of Parallel and Distributed Computing*, 62(9), 1421 – 1432.

Rego, C. and Roucairol, C. (1996). *A Parallel Tabu Search Algorithm Using Ejection Chains for the Vehicle Routing Problem*, 661–675. Springer US, Boston, MA.

Reinelt, G. (1991). "Tsplib-a traveling salesman problem library." volume 3, 376–384.

Ribeiro, G. M. and Laporte, G. (2012). "An adaptive large neighborhood search heuristic for the cumulative capacitated vehicle routing problem." *Computers & Operations Research*, 39(3), 728 – 735.

Rios, E., Ochi, L. S., Boeres, C., Coelho, V. N., Coelho, I. M. and Farias, R. (2018). "Exploring parallel multi-gpu local search strategies in a metaheuristic framework." *Journal of Parallel and Distributed Computing*, 111, 39 – 55.

Robinson, J. A., Vrbsky, S. V., Hong, X. and Eddy, B. P. (2018). "Analysis of a high-performance tsp solver on the gpu." *ACM J. Exp. Algorithmics*, 23.

Rochat, Y. and Taillard, É. D. (1995). "Probabilistic diversification and intensification in local search for vehicle routing." *Journal of Heuristics*, 1(1), 147–167.

Rocki, K. and Suda, R. (2012). "Accelerating 2-opt and 3-opt Local Search Using GPU in the Travelling Salesman Problem." *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*.

Rocki, K. and Suda, R. (2013). "High performance GPU accelerated local optimization in TSP." *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013*, 1788–1796.

Rosenkrantz, D. J., Stearns, R. E. and Lewis, P. M. (1974). "Approximate algorithms for the traveling salesperson problem." 15th Annual Symposium on Switching and Automata Theory (swat 1974), 33–42.

Ryoo, S., Rodrigues, C. I., Stone, S. S., Stratton, J. A., Ueng, S. Z., Baghsorkhi, S. S. and Hwu, W. m. W. (2008). "Program optimization carving for GPU computing." *Journal of Parallel and Distributed Computing*, 68(10), 1389–1401.

Sahni, S. and Gonzalez, T. (1976). "P-complete approximation problems." *J. ACM*, 23(3), 555–565.

Salehipour, A., Sörensen, K., Goos, P. and Bräysy, O. (2011). "Efficient grasp+vnd and grasp+vns metaheuristics for the traveling repairman problem." *4OR*, 9(2), 189–209.

Santana, Í., Plastino, A. and Rosseti, I. (2020). "Improving a state-of-the-art heuristic for the minimum latency problem with data mining." *International Transactions in Operational Research*.

Schulz, C. (2013). "Efficient local search on the gpu—investigations on the vehicle routing problem." *Journal of Parallel and Distributed Computing*, 73(1), 14 – 31. Metaheuristics on GPUs.

Silva, M. M., Subramanian, A., Vidal, T. and Ochi, L. S. (2012). "A simple and effective metaheuristic for the minimum latency problem." *European Journal of Operational Research*, 221(3), 513 – 520.

Sitters, R. (2002). "The minimum latency problem is np-hard for weighted trees." In Cook, W. J. and Schulz, A. S., editors, *Integer Programming and Combinatorial Optimization*, Springer Berlin Heidelberg, Berlin, Heidelberg, 230–239.

Skinderowicz, R. (2016). "The gpu-based parallel ant colony system." *Journal of Parallel and Distributed Computing*, 98, 48 – 60.

Stützle, T. (1998a). *Local search algorithms for combinatorial problems — analysis, improvements, and new applications.*

Stützle, T. (1998b). *Parallelization strategies for Ant Colony Optimization*, 722–731. Springer, Berlin, Heidelberg.

Stützle, T. and Hoos, H. H. (2000). "Max–min ant system." *Future Generation Computer Systems*, 16(8), 889 – 914.

Subramanian, A., Uchoa, E. and Ochi, L. S. (2013). "A hybrid algorithm for a class of vehicle routing problems." *Computers & Operations Research*, 40(10), 2519 – 2531.

Taillard, (1993). "Parallel iterative search methods for vehicle routing problems." *Networks*, 23(8), 661–673.

Talawar, B. and Yelmewad, P. (2017). "Gpu-based iterative hill climbing algorithm to solve symmetric traveling salesman problem." In Grandinetti, L., Mirtaheri, S. L., Shahbazian, R., Sterling, T. L. and Voevodin, V. V., editors, *Big Data and HPC: Ecosystem and Convergence, TopHPC 2017, Tehran, Iran, 24-26 April 2017*, volume 33 of *Advances in Parallel Computing*, IOS Press, 221–241.

Talbi, E.-G. (2009). *Metaheuristics: From Design to Implementation*, Wiley Publishing.

Toth, P. and Vigo, D., editors (2001). *The Vehicle Routing Problem*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Toth, P. and Vigo, D. (2003). “The granular tabu search and its application to the vehicle-routing problem.” *INFORMS J. on Computing*, 15(4), 333–346.

Tsitsiklis, J. N. (1992). “Special cases of traveling salesman and repairman problems with time windows.” *Networks*, 22(3), 263–282.

Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T. and Subramanian, A. (2017). “New benchmark instances for the capacitated vehicle routing problem.” *European Journal of Operational Research*, 257(3), 845 – 858.

Vidal, T., Crainic, T. G., Gendreau, M., Lahrichi, N. and Rei, W. (2012). “A hybrid genetic algorithm for multidepot and periodic vehicle routing problems.” *Operations Research*, 60(3), 611–624.

Vidal, T., Crainic, T. G., Gendreau, M. and Prins, C. (2014). “A unified solution framework for multi-attribute vehicle routing problems.” *European Journal of Operational Research*, 234(3), 658 – 673.

Wei, L., Zhang, Z., Zhang, D. and Leung, S. C. (2018). “A simulated annealing algorithm for the capacitated vehicle routing problem with two-dimensional loading constraints.” *European Journal of Operational Research*, 265(3), 843 – 859.

Wu, B. Y. (2000). “Polynomial time algorithms for some minimum latency problems.” *Information Processing Letters*, 75(5), 225 – 229.

Wu, B. Y., Huang, Z.-N. and Zhan, F.-J. (2004). “Exact algorithms for the minimum latency problem.” *Information Processing Letters*, 92(6), 303 – 309.

Yelmewad, P., Kumar, A. and Talawar, B. (2019). “MMAS on GPU for large TSP instances.” In *10th International Conference on Computing, Communication and Networking Technologies, ICCCNT 2019, Kanpur, India, July 6-8, 2019*, IEEE, 1–6.

Yelmewad, P. and Talawar, B. (2018). “Near optimal solution for traveling salesman problem using gpu.” In *2018 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 1–6.

Yelmewad, P. and Talawar, B. (2019). “Parallel iterative hill climbing algorithm to solve tsp on gpu.” *Concurrency and Computation: Practice and Experience*, 31(7), e4974.

Yelmewad, P. and Talawar, B. (2020). “Gpu-based parallel heuristics for capacited vehicle routing problem.” In *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 1–6.

Yelmewad, P. and Talawar, B. (2020). “Parallel deterministic local search heuristic for minimum latency problem.” *Cluster Computing*.

Yu, V. F., Redi, A. P., Hidayat, Y. A. and Wibowo, O. J. (2017). “A simulated annealing heuristic for the hybrid vehicle routing problem.” *Applied Soft Computing*, 53, 119 – 132.

Zhao, J., Liu, Q., Wang, W., Wei, Z. and Shi, P. (2011). “A parallel immune algorithm for traveling salesman problem and its application on cold rolling scheduling.” *Information Sciences*, 181(7), 1212–1223.

Zhou, Y., He, F., Hou, N. and Qiu, Y. (2018). “Parallel ant colony optimization on multi-core simd cpus.” *Future Generation Computer Systems*, 79, 473 – 487.

Zhou, Y., He, F. and Qiu, Y. (2016). “Optimization of parallel iterated local search algorithms on graphics processing unit.” *The Journal of Supercomputing*, 72(6), 2394–2416.

PUBLICATIONS BASED ON THE RESEARCH WORK

1. Basavaraj Talawar and Pramod Yelmewad (2017). GPU-Based Iterative Hill Climbing Algorithm to Solve Symmetric Traveling Salesman Problem. *Big Data and HPC: Ecosystem and Convergence, TopHPC 2017, Tehran, Iran, 24-26 April 2017*, 33:221-241. [**DOI**: `https://doi.org/10.3233/978-1-61499-882-2-221`]

2. Pramod Yelmewad and Basavaraj Talawar (2018). Near Optimal Solution for Traveling Salesman Problem using GPU. *2018 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 1-6. [**DOI**: `10.1109/CONECCT.2018.8482363`]

3. Pramod Yelmewad and Basavaraj Talawar (2018) Parallel iterative hill climbing algorithm to solve TSP on GPU. *Concurrency and Computation: Practice and Experience (Wiley)*, 31, 7, 1-25. [**DOI**: `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4974`]

4. Pramod Yelmewad, Aniket Kumar and Basavaraj Talawar (2019). MMAS on GPU for Large TSP Instances. *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 1-6 [**DOI:** `0.1109/ICCCNT45670.2019.8944770`]

5. Pramod Yelmewad and Basavaraj Talawar (2020). GPU-based Parallel Heuristics for Capacited Vehicle Routing Problem. *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 1-6 [**DOI:** `10.1109/CONECCT50063.2020.9198667`]

6. Pramod Yelmewad and Basavaraj Talawar (2020). Parallel Deterministic Local Search Heuristic for Minimum Latency Problem. *Cluster Computing (Springer)* . [**DOI**: `https://doi.org/10.1007/s10586-020-03173-4`]

7. Pramod Yelmewad and Basavaraj Talawar (2020). Parallel Version of Local Search Heuristic Algorithm to Solve Capacitated Vehicle Routing Problem. *Cluster Computing (Springer)* [**Revised**]

BIO-DATA

Name:	Pramod Hanmantrao Yelmewad
Date of Birth:	02/04/1992
Gender:	Male
Marital Status:	Single
Father's Name:	Hanmantrao
Mother's Name:	Laxmi
Email Id:	p.yelmewad@gmail.com
Present Address:	Manik Niwas, Talwes Road, Near Laxminarayan Temple, Udgir, Dist. Latur, Maharashtra- 413517
Educational Qualifications:	M.Tech (IT) - Walchand College of Engineering, Sangli, Maharashtra B.E (IT) - PES Modern College of Engineering, Shivaji Nagar, Pune, Maharashtra Diploma (IT) - Govt. Polytechnic College, Nanded, Maharashtra
Areas of Interest:	High Performance Computing.