Empowering Industrial IoT Systems with GPU-Powered Optimization Algorithms and Cybersecurity Tools

Marwan Fouad Abdelatti

University of Rhode Island, mabdelrazik@uri.edu

Follow this and additional works at: https://digitalcommons.uri.edu/oa_diss

Recommended Citation

This Dissertation is brought to you for free and open access by DigitalCommons@URI. It has been accepted for inclusion in Open Access Dissertations by an authorized administrator of DigitalCommons@URI. For more information, please contact digitalcommons-group@uri.edu.
EMPOWERING INDUSTRIAL IOT SYSTEMS WITH GPU-POWERED OPTIMIZATION ALGORITHMS AND CYBERSECURITY TOOLS

BY

MARWAN F. ABDELATTI

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

IN

INDUSTRIAL AND SYSTEMS ENGINEERING

UNIVERSITY OF RHODE ISLAND

2022
DOCTOR OF PHILOSOPHY DISSERTATION

OF

MARWAN F. ABDELATTI

APPROVED:

Dissertation Committee:

Major Professor  Manbir S. Sodhi
                David G. Taggart
                Resit Sendag
                Brenton L. DeBoef

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2022
Today’s consumers’ demands are continuously increasing, sometimes surpassing commercial and operational advancements. Both individual and business customers expect to get goods faster, more flexibly, and at low or no delivery cost. These high expectations, along with more customized manufacturing, are burdening the supply chain industry. Many report analyses underscore the vulnerabilities of current supply chain systems, particularly in their agility and reaction times, which were so apparent during the COVID-19 pandemic, and the direct succession of the Russo-Ukrainian conflict. These analyses emphasize the necessity of automation with swift and accurate planning. Judging from the facts above, this dissertation follows two directions to improving the supply chain industry, namely, the logistic planning and the manufacturing process. Obtaining high-speed planning algorithms is necessary for nowadays’ logistic systems to meet the increasing consumer demands and quickly respond to unpredicted circumstances, which directly improves the trade flow and, in turn, reflects on the economy. Whereas achieving technological migration inside manufacturing facilities from a functional operation to a comprehensively linked data and network model will help optimize the manufacturing process in terms of speed, quality, consumption of raw material, and energy perspectives.

With regard to improving logistic planning methodologies, this study proposes a novel approach to advance a class of path planning algorithms called the vehicle routing problem (VRP). The proposed approach utilizes the unique surging computing hardware technology and introduces a genetic algorithm (GA)-based approach running entirely on graphics processing units (GPUs). These processing units are known for their capabilities of performing massive arithmetic and logic computations in parallel at a high execution speed. The study introduces
two algorithms, one that runs on a single GPU platform for small and medium size VRP problems. The other algorithm runs on multi-GPU high-performing computers (HPCs) that process large-size problems up to 20,000 customer nodes. The algorithms show improvements in the execution speed compared with their conventional CPU counterparts at a factor reaching 1,700.

The second direction of this study integrates smart sensors, actuators, and controllers into advanced communication protocols where a massive amount of data is exchanged between the devices. The study adopts cutting-edge technologies such as artificial intelligence (AI) and automation in the industrial internet of things (IIoT) framework and introduces a lab-scale smart manufacturing system. This IIoT system involves different manufacturing stations that mimic a real-world manufacturing facility. Despite the merits obtained by IIoT, like controllability and automation, the broadly distributed and entangled nature of the IIoT networks makes them vulnerable to serious security issues. Therefore, communications between units are secured by hashing and digital certification techniques.

Moreover, a comprehensive systematic literature review on cybersecurity approaches in IIoT systems has been developed, and almost 500 research works in the field have been reviewed over the last decade. Based on this review, an intelligent anomaly detection approach for a typical IoT system is discussed. Five machine learning (ML) algorithms are applied to a benchmark IoT dataset to analyze the behavior of different processes and classify it into normal behavior and seven different attack types.
All along my exciting journey to my Ph.D., many people were there to bear a hand to have me complete this work. First of all, I am profoundly grateful to Dr. Sodhi for giving me a chance to work with him as a graduate student. His endless support, guidance, and patience helped me to improve my skills and intuition. Throughout my program, he answered all my questions gracefully and encouraged me to explore new areas and think critically. I am very much indebted to Dr. Sendag and Dr. Hendawi for their support in working with high-performance computing and for their direction and feedback in pursuing my goals. Their collaborations were instrumental in producing the following work. My sincere thanks to Dr. Taggart and Dr. Rivero for their precious time and scientific input.

I am grateful to my parents for seeding and growing up my curiosity and determination. My sincere thanks go to my warmhearted wife — Hadya — and my three children, who provided me with more patience and support than I could ever get alone. Hadya’s efforts in taking care of the kids and strengthening our family bonds are priceless. Finally, a blanket of thanks goes to my peers, who motivated me to stay on track and remain focused.
PREFACE

This dissertation is presented in a manuscript format in accordance with the University of Rhode Island Graduate School Guidelines. The dissertation is composed of five manuscripts that have been either published or prepared for submission. These manuscripts are listed in the following aspects:

- **Manuscript I: An improved GPU-accelerated heuristic technique applied to the capacitated vehicle routing problem**
  This manuscript has been published in the Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO).

- **Manuscript II: Optimizing a GPU-accelerated genetic algorithm for the vehicle routing problem**
  This manuscript has been published in the Proceedings of the 2021 Genetic and Evolutionary Computation Conference Companion (GECCO).

- **Manuscript III: A Multi-GPU Parallel Genetic Algorithm For Large-Scale Vehicle Routing Problems**
  This manuscript has been published in the proceedings of the 2022 IEEE High Performance Extreme Computing (HPEC).

- **Manuscript IV: Lab-scale Smart Factory Implementation Using ROS**
  This manuscript has been published as a chapter in the Robot Operating System (ROS): The Complete Reference (Volume 7 – 2022) Springer.

- **Manuscript V: An Intelligent Anomaly Detection System for Industrial Internet of Things**
  This manuscript has been prepared for publication but is not submitted yet.
# TABLE OF CONTENTS

**ABSTRACT** ................................................................. ii

**ACKNOWLEDGMENTS** ..................................................... iv

**PREFACE** ................................................................. v

**TABLE OF CONTENTS** ................................................... vi

**LIST OF TABLES** ........................................................ x

**LIST OF FIGURES** ....................................................... xi

**MANUSCRIPT**

1 An improved GPU-accelerated heuristic technique applied to the capacitated vehicle routing problem ............... 1

   Abstract ................................................................. 2

   1.1 Introduction ....................................................... 3

   1.2 Related Work .................................................... 5

   1.3 Proposed Algorithm .............................................. 8

      1.3.1 Basic Notations and Definitions .......................... 8

      1.3.2 Algorithm Description .................................... 9

      1.3.3 NVIDIA GPU Architecture ................................. 11

      1.3.4 Algorithm Implementation on GPU ....................... 12

   1.4 Execution Results ............................................... 19

   1.5 Conclusions ..................................................... 26

   List of References ................................................. 28
## 2 Optimizing a GPU-accelerated genetic algorithm for the vehicle routing problem

- **Abstract**
- **2.1 Introduction**
- **2.2 GPU-Accelerated GA for CVRP**
- **2.3 Design of Experiment**
- **2.4 Execution Results and Conclusions**
- **List of References**

## 3 A Multi-GPU Parallel Genetic Algorithm For Large-Scale Vehicle Routing Problems

- **3.1 Introduction**
- **3.2 Related Work**
- **3.3 Problem Definition**
- **3.4 Algorithm Description**
  - **3.4.1 Hardware**
  - **3.4.2 Proposed Algorithm**
  - **3.4.3 GPU implementation**
- **3.5 Results And Discussions**
- **3.6 Conclusions**
- **List of References**

## 4 Lab-scale Smart Factory Implementation Using ROS

- **Abstract**
- **4.1 Introduction**
- **4.2 The Smart Factory Kit**

---

vii
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Parameter settings</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>Numerical results of testing CPU and GPU versions of the algorithm on benchmark problems</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>Numerical results of running the GPU algorithm with and without the 2-opt</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>Results of running the tuned GPU-GA</td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>Execution speeds in secs/generation.</td>
<td>56</td>
</tr>
<tr>
<td>6</td>
<td>Attribute statistics before scaling.</td>
<td>119</td>
</tr>
<tr>
<td>7</td>
<td>Evaluation metrics and speeds</td>
<td>125</td>
</tr>
<tr>
<td>A.8</td>
<td>Solutions obtained by a single GPU for selected benchmarks</td>
<td>133</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>An example solution of a CVRP</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>A typical GPU architecture</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Algorithm implementation on GPU</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>Execution time of CPU vs GPU</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>Solution quality of the proposed algorithm</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Performance With and Without the 2-opt</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>Execution time with and without the 2-opt</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>Algorithm implementation on GPU</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>Solution representation and terms</td>
<td>47</td>
</tr>
<tr>
<td>10</td>
<td>DGX-1 hybrid cube-mesh connection [44]</td>
<td>48</td>
</tr>
<tr>
<td>11</td>
<td>Flowchart of the coarse-grained PGA on n GPUs</td>
<td>52</td>
</tr>
<tr>
<td>12</td>
<td>CUDA API profile</td>
<td>57</td>
</tr>
<tr>
<td>13</td>
<td>Kernel profiling</td>
<td>57</td>
</tr>
<tr>
<td>14</td>
<td>Memory transfer time profiling</td>
<td>58</td>
</tr>
<tr>
<td>15</td>
<td>Memory transfer size profiling</td>
<td>58</td>
</tr>
<tr>
<td>16</td>
<td>The smart factory kit</td>
<td>70</td>
</tr>
<tr>
<td>17</td>
<td>New material process flow</td>
<td>71</td>
</tr>
<tr>
<td>18</td>
<td>New order process flow</td>
<td>71</td>
</tr>
<tr>
<td>19</td>
<td>The HBW flowchart</td>
<td>72</td>
</tr>
<tr>
<td>20</td>
<td>The VGR flowchart</td>
<td>73</td>
</tr>
<tr>
<td>21</td>
<td>The SLD flowchart</td>
<td>74</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>22</td>
<td>Rosserial between Arduino and Raspberry Pi.</td>
<td>76</td>
</tr>
<tr>
<td>23</td>
<td>Physical and logic layers in the system</td>
<td>78</td>
</tr>
<tr>
<td>24</td>
<td>Arduino IDE Preferences</td>
<td>79</td>
</tr>
<tr>
<td>25</td>
<td>Arduino IDE Boards manager</td>
<td>79</td>
</tr>
<tr>
<td>26</td>
<td>Arduino IDE Board Selection</td>
<td>80</td>
</tr>
<tr>
<td>27</td>
<td>Arduino IDE Port Selection</td>
<td>80</td>
</tr>
<tr>
<td>28</td>
<td>MQTT main components</td>
<td>91</td>
</tr>
<tr>
<td>29</td>
<td>Screenshot of the ROS master initiation</td>
<td>95</td>
</tr>
<tr>
<td>30</td>
<td>ROS graph, nodes and topics connections</td>
<td>97</td>
</tr>
<tr>
<td>31</td>
<td>Typical steps and activities in a systematic literature review [4]</td>
<td>105</td>
</tr>
<tr>
<td>32</td>
<td>Sankey diagram of reported literature</td>
<td>108</td>
</tr>
<tr>
<td>33</td>
<td>Publications about anomaly detection</td>
<td>109</td>
</tr>
<tr>
<td>34</td>
<td>Distribution of anomalies in DS2OS</td>
<td>113</td>
</tr>
<tr>
<td>35</td>
<td>Network scan attack</td>
<td>114</td>
</tr>
<tr>
<td>36</td>
<td>Spying attack</td>
<td>114</td>
</tr>
<tr>
<td>37</td>
<td>Spying attack</td>
<td>115</td>
</tr>
<tr>
<td>38</td>
<td>Malicious operation attack</td>
<td>115</td>
</tr>
<tr>
<td>39</td>
<td>Wrong set up attack</td>
<td>116</td>
</tr>
<tr>
<td>40</td>
<td>Malicious control attack</td>
<td>116</td>
</tr>
<tr>
<td>41</td>
<td>Denial of service (DoS) attack</td>
<td>117</td>
</tr>
<tr>
<td>42</td>
<td>Stratification of data</td>
<td>118</td>
</tr>
<tr>
<td>43</td>
<td>Visualization of data</td>
<td>120</td>
</tr>
</tbody>
</table>
MANUSCRIPT 1

An improved GPU-accelerated heuristic technique applied to the capacitated vehicle routing problem

Marwan F. Abdelatti*, Manbir S. Sodhi

Department of Industrial Engineering, University of Rhode Island, Kingston, Rhode Island, 02881

*Corresponding author: mabdelrazik@uri.edu

Published in the Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO), 663–671.
ACM ISBN 978-1-4503-7128-5/20/07
https://doi.org/10.1145/3377930.3390159
Abstract

The capacitated vehicle routing problem (CVRP) is a well-known NP-hard combinatorial problem. Genetic algorithms (GAs) are often used for solving CVRPs. However, the computational effort required to find a feasible solution becomes problematic for very large instances. Parallel-computation technology can significantly improve the performance of CVRP solution algorithms to deal with large problem sets, especially when using GAs. In this paper, an improved genetic algorithm is designed to be entirely executed on an NVIDIA GPU, taking advantage of the special CUDA GPU architecture to solving the CVRP. By distributing array elements over the GPU grid and using GPU kernel functions, the proposed algorithm successfully provides high-quality solutions within reasonable computational times, and near-optimal solutions for smaller benchmark problems. Within this framework, we address the execution speed problem in CVRPs by developing an algorithm that is entirely running on an NVIDIA GPU, investigate how to incorporate local search algorithms with the GA, and develop comparisons between the algorithm performance on both the CPU and the GPU. The utility of improved local search algorithms in the overall performance of the GA is also reported.
1.1 Introduction

Vehicle routing and scheduling problems occupy a special place in distribution management. They are among the most important and challenging optimization problems in the field of operations research because of their complexity and wide applicability. They can model many real-life applications such as assembly lines, postal service planning, logistics distribution systems, waste collection, etc. The vehicle routing problem (VRP) is an NP-hard combinatorial problem for determining optimal delivery/collection routes through a set of locations, subject to some constraints [1]. The capacitated VRP (CVRP) is the most common and the simplest form of VRP where the total demands of the customers assigned to a vehicle cannot exceed the vehicle capacity, and the customers must be visited only once. There are many methods used for solving the CVRP; some are exact methods, and the others are heuristic.

Exact methods mainly depend on direct tree searches, dynamic programming, or linear programming [2, 3]. These methods can provide optimal or near-optimal solutions but require extended computational resources and this limits the problem size they can solve. There are numerous examples of such methods including but not limited to branch-and-bound (B&B), branch-and-cut (B&C) and dynamic programming [4]. The most sophisticated exact methods nowadays can solve problems up to a little more than one hundred customers [5]. One of such method is presented in [6] and employs a partitioning formulation and additional cuts. However, practical problems consist of hundreds or even thousands of customers. For such problems, an exact solution within a reasonable amount of time is hard to obtain [7].

Heuristic methods, on the other hand, can provide high quality near-optimal solutions within acceptable computational times. Many heuristic techniques gen-
erate solutions for the CVRP in a progressive manner [2, 5]. Examples of such techniques include the Clarke and Wright algorithm [8], Tabu Search algorithm [9], local search algorithm (2-Opt) [10], and genetic algorithms (GA) [11]. Genetic Algorithms (GA) are a type of heuristic method that utilize a number of basic reproduction features existing in nature such as: selection, mating, and mutation processes [12]. They are distinguished from other heuristic methods by the fact that they operate on a population of potential solutions which increases the probability of finding a good solution and leads to having them used effectively in many VRP scenarios [11, 13, 14, 15, 16, 17, 18, 19, 20, 21]. Extensive research efforts to incorporate exact methods with GAs have been reported [11, 18, 15, 22]. However, GAs require long execution times, and parallelizing algorithms on multi-core processing units are one approach for finding high quality solutions in a feasible amount of time. With recent hardware advances of graphics processing units (GPUs), these are also being considered for GA applications.

GPUs are multiprocessor devices that can run a large number of instructions in parallel at a high execution speed. They are a key utility in the revolutionary advancement of the processing speed occurring nowadays.

In this paper, a new GA algorithm is proposed to find the best route (with shortest distance) for CVRPs. This algorithm is specifically designed to be entirely executed on the GPU to take advantage of the special NVIDIA compute unified device architecture (CUDA) for parallel computations with very short execution times. The diversity of the population is maintained along the evolution process to prevent the algorithm from getting stuck into a local minimum. Local search algorithms (i.e., 2-opt and nearest neighbor) are utilized to study their effect on the quality of the solutions. They are employed to optimize every single route in each solution in the GA population. Within this framework, we address
the execution speed problem in VRPs by developing an algorithm that entirely runs on an NVIDIA GPU and provide comparisons between the algorithm performance on both CPU and GPU in tables and plots. Moreover, we improve some best-documented values of existing VRP benchmark problems, and investigate the effectiveness of utilizing local search algorithms on the overall performance of the GA. The rest of the paper is organized as follows; Section 1.2 briefly reviews existing works related to VRP, GA, and parallelization. The methodology of the proposed algorithm is introduced in Section 1.3, including an illustration of the GPU technology and the implementation of the proposed algorithm on the GPU. Section 1.4 discusses the results of applying the algorithm on benchmark problems and compares between CPU and GPU runs. Some conclusions are stated at the end of the paper.

1.2 Related Work

A significant body of research on algorithms for solving VRPs is documented. Broadly speaking, these algorithms can be divided into exact methods and heuristic based approaches. Exact methods guarantee that the optimal solution is reached either by searching exhaustively over all possible solutions or by eliminating solutions that lead to non-optimal solutions. Heuristic methods, on the other hand, perform a relatively limited exploration of the search space and typically produce good quality solutions within modest computing times [23]. Genetic algorithms (GAs) are a type of heuristics that operate with a population of solutions to making small changes (mutation) and mixing parts (crossover) in consideration of quality objectives to guide its search toward the optimal solutions [24]. Moreover, most effective GAs are hybridized with local search algorithms [25]. This approach may be viewed as a multi-start local search method, with starting points obtained through a sampling of the search space provided by the genetic algorithm[25].
Some of these algorithms are implemented on traditional computers involving single or multiple CPUs, while others are partially or entirely implemented on the GPU. In [11], a hybrid Genetic Algorithm was applied to a CVRP problem. The chromosomes were represented such that each individual solution specifies the vehicle number to which each customer was assigned. Even though the chromosome representation guaranteed a fixed length for each single solution which simplifies the implementation, the algorithm produced routes with higher excessive demands than the vehicle’s capacity. Another algorithm was introduced in [18] where a 2-opt local search algorithm was utilized with the GA. The chromosomes were represented as a sequence of customers without depot delimiters and a split algorithm was applied to consider the capacity constraint. This chromosome representation has been found more efficient and is adopted by most of the subsequent works on GAs for VRPs in the literature. Duplicate-solutions (clones) were prohibited by a special algorithm. The down side of this algorithm was in the huge computational load (almost 95% of CPU time) consumed by the local search and the clone-restricting algorithms [18]. The work in [26] proposed a more efficient method to restrict cloning but the solutions were still not good enough. An edit distance measure was employed in [15, 22] to improve the clone-restricting algorithm but it involved extended execution times.

Preserving the population diversity is crucial to any algorithm involving GAs. It helps avoiding premature convergence and balances between exploration and exploitation of the search space, yet is very computationally expensive. The high computational cost limits the size of the VRP problems the algorithms deal with. Therefore, many research efforts that utilize GPUs for VRP problems are conducted to speed up the algorithm execution speed. [27] designed a genetic algorithm implemented on the GPU for the Dynamic VRP (DVRP) - which involves
finding VRP solutions with some demands that are revealed after the tour is in progress. It was effective to find good solutions for problems up to 3000 nodes with a speed 35 times faster than the CPU implementation. However, the method assumed that the routes were already planned and targeted only to insert new requests in these routes then genetic operators optimize the resulting solution.

A CPU–GPU integrated perturbation strategy and four different neighborhood local searches implemented purely in GPUs were introduced in [28] for single vehicle routing problem with deliveries and selective pickups (SVRPDSP). The initial solution is generated by two mathematical programming solvers, providing the search algorithm with good starting solutions. The algorithm was able to achieve nearly 15 times speedup over a pure sequential version but a considerable part of the algorithm was executed on the CPU affecting the total execution time. The algorithm was not tested on problems larger than 101 customer nodes. The parallel GPU execution in [29] was only applied to the cost function calculation for CVRP and showed a speed up in the cost calculation function of about 1,000 times than the CPU implementation. It did not, however, report the impact of this implementation on the overall performance of the algorithm.

A parallel GA running on the GPU for the Min-Max Multi-depot VRP (MD-VRP) was proposed in [30]. By assigning the computing tasks for each point in the chromosome (i.e., gene) to independent threads, the algorithm was able to process all the GA operations in parallel. Although the experimental results showed a significant reduction in the computing time that was about 1,000 times less than the CPU implementation for a problem with 200 nodes, the quality of the solution has was not discussed limiting any conclusions on the effectiveness of the proposed algorithm. Also, the initialization of the population was processed on the CPU which affected the execution speed. The work in [31] utilized GPUs to run a Tabu
Search (TS) algorithm for the Distance-constrained VRP and compared it with a CPU version. Even though its computational complexity was higher, the parallel algorithm was able to outperform the classic TS algorithm in terms of speed. However, the TS algorithm is very crude and more evolved models were required for more complex problems [31].

1.3 Proposed Algorithm

We introduce a new hybrid algorithm to find the best route with the shortest distance for the capacitated vehicle routing problem. This algorithm is specifically designed to be entirely executed on the GPU to take advantage of the special NVIDIA CUDA architecture for parallel computations. By distributing array elements over the GPU grid and using GPU kernel functions, the proposed algorithm provides high-quality solutions at a very high computational speeds, and near-optimal solutions for smaller benchmark problems. Before delving into the details of proposed algorithm, the following basic definitions are established.

1.3.1 Basic Notations and Definitions

The following notations are used in the paper. \( \mathbb{R} \) and \( \mathbb{R}_+ \) denote the sets of real and positive real numbers respectively. Similarly, \( \mathbb{I} \) and \( \mathbb{I}_+ \) represent the sets of integers and positive integers respectively. A set of numbers is represented by \( \{n_1, n_2, \ldots, n_k\} \) whereas a set of tuples is represented by \( \{(m_1, m_2), (m_3, m_4), \ldots, (m_p, m_q)\} \). Based on these notations, a typical CVRP can be defined as:

**Definition 1.** Given a depot node \( \{0\} \) at location \((x_0, y_0)\) and a set of \( n \) customer nodes \( \mathcal{C} = \{1, 2, 3, \ldots, n\} \), \( n \in \mathbb{I}_+ \) at locations \((x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots, (x_n, y_n)\) where \( x_0, \ldots, x_n, y_0, \ldots, y_n \in \mathbb{R} \), there exists an undirected graph \( \mathcal{G} = (\mathcal{N}, \mathcal{A}) \), that can be established where \( \mathcal{N} = \{0\} \cup \mathcal{C} \) and \( \mathcal{A} = \{(i, j) : i, j \in \mathcal{N}, i < j\} \) is a set
of edges/connections between the nodes. Each customer $i$ orders a non-negative demand $d_i \in \mathbb{R}_+ \forall i$. It has to be noted that this demand cannot be divided and hence every customer can be visited only once. The cost $c_{ij}$ associated to each edge $(i, j) \in A$ can be defined as the Euclidean distance between node $i$ and node $j$ given as:

$$c_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

(1)

It is required to find the optimal solution $l^*$ that has the minimum cost, i.e.,

$$l^* = \arg\min_{l \in P} \sum_{i,j=0}^{n} c_{ij}, \ \forall i \neq j$$

(2)

where $P$ is the set of all feasible solutions for the problem.

For simplicity, we assume that the vehicles have identical capacities $Q$ where $Q \geq d_i \forall i \in C$ so that any vehicle can be assigned to any route available in the solution to avoid more complexity of the problem. The total cost $f_{i,t}$ of an individual solution $l_{i,t}$ is the sum of the costs $c_{ij}$ associated with all the edges in it i.e.,

$$f_{i,t} = \sum_{i,j=0}^{n} c_{ij}, \ \forall i \neq j$$

(3)

Figure 1 illustrates the capacitated VRP problem by an example where the demand $d_i$ of each of the 12 customers as well as the cost of the connections are shown. Four vehicles are available with capacity $Q = 10$. The feasible routes are represented by $r_1, r_2, r_3, \text{ and } r_4$ with $r_i \in A \forall i$. For example, $r_1 = \{(0, 4), (4, 11), (11, 6), (6, 0)\}$ with a cost of 35 and load of $9 < Q$. The set $l = \{r_1 \cup r_2 \cup r_3 \cup r_4\} \in P$ represents an example solution of this particular problem with a total cost of 114, this is not necessarily the optimal solution.

1.3.2 Algorithm Description

We incorporate a GA and a 2-opt local search into a hybrid algorithm to solve the CVRP defined in Definition (1). At each iteration $t$, population $P_t \subset A$
includes all feasible solutions $l_t$ where each row in $P_t$ is a single solution. These solutions are called individuals and are represented as a sequence of customer nodes $\in C$ delimited by the depot node 0 to count for the capacity constraint. The number of individuals in population $P_t$ is called the population size ($popSize$). The population size is predefined and is fixed along all iterations. At the beginning, the population $P_0$ is initiated by random individuals.

Evolutionary processes like cross-over and mutation are applied to each individual $l_{i,t} \forall i = \{1, \ldots, popSize\}$ to update the population at each iteration $t > 0$ to form $P_t$. The cost value of each individual is computed when the individual is produced based on Equation 3 and the cost table which is generated by the algorithm. The 2-opt algorithm is used for every route inside each individual as a separate travelling salesman problem (TSP).

Two identical algorithms have been developed in this work, one is running on the GPU and the other is running on the CPU for performance comparison purposes. The two algorithms are identical except for the way the functions are coded to run properly on the corresponding hardware.
1.3.3 NVIDIA GPU Architecture

In recent years, much has been made in computers industry that caused a widespread shift to parallel computing. Nowadays, most consumer computers are shipped with graphics processing units (GPUs). The rise of GPU computing introduced a new concept of performing general-purpose computations on it. In 2006, NVIDIA unveiled their first GPU with CUDA architecture, namely, GeForce 8800 GTX. It has been provided by many features that were not existing in GPUs before such as the unified shader pipeline which obviated the need for vertex and pixel shaders that are used to exist before, arithmetic logic units (ALU) complied with IEEE requirements for single-precision floating-point arithmetic, instruction set for general computation rather than specific graphics types, and a shared memory [32]. Thanks to this technology, a standard computational representation have been used and a variety of common programming languages (e.g., C, C++, and Python) have been involved.

As a result of these improvements, many industries and applications have been successfully developed by counting on CUDA technology. These benefits included performance-per-dollar and performance-per-watt improvement compared to similar implementations built on traditional central processing units (CPUs) [33]. Some successful implementations on GPUs included medical imaging, computational fluid dynamics, environmental science, and the revolutionary machine learning frameworks like Tensorflow and pyTorch [34, 35, 36].

A typical NVIDIA GPU consists of a virtual container called (grid) that includes several processors called Streaming Multiprocessors (SMs) that contain hundreds of cores (SPs). These cores are responsible for running the threads that execute the same code in parallel. A block is a unit consisting of one or more warps of 32 threads each. The number of threads can greatly exceed the number of cores
thanks to the automatic thread scheduling policies such as: independent thread scheduling, Round Robin (RR) - Instructions, Thread block-based criticality aware warp scheduling, etc. [37]. Functions executed on the GPU (kernels) can be called either from the GPU by another kernel or from a CPU (host) function by passing the number of blocks and threads that can be configured manually.

GPUs have a special memory structure. The main memory type used by our algorithm is the global memory that is analogous to RAM for CPUs. It is the largest memory in a GPU with size up to 16 GB in some products. It is accessible by all threads as well as the CPU. An illustration of the basic architecture of a typical GPU is shown in Figure 2 where a grid contains several blocks or SMs with each consisting of several cores (SPs), a single shared memory which is a small memory accessed by the threads of a single block at a very high speed, and L1 and L2 cache memories that have much higher bandwidth than their CPU counterparts. All SMs share a single global memory, and an L2 cache [38].

1.3.4 Algorithm Implementation on GPU

In this implementation, the device grid is arranged as a 2D array of $B \times B$ blocks with each block being a 2D array of $T \times T$ threads where $B$ and $T$ are integer numbers. The grid, therefore, has a total of $(B \times T)^2$ threads running in
parallel. Every input array (e.g., the population $P_i$) is mapped into the grid in such a way that its elements are distributed over the threads that perform computations separately. This, in turn, facilitates high throughput memory transfers in the GPU.

Several kernels running on the GPU are devoted to perform different functions of the algorithm. These functions include the random initialization, cost calculation, cross over, capacity constraints, 2-opt local search. After reaching a specific number of generations given by the user, the GPU passes the final solution to the CPU that prints out the best solution as well as the total running time and the time elapsed to produce one generation.

The kernels in the proposed algorithm can be divided into two types based on how they deal with the input arrays. The first type is a scalar function where a single thread operates on a single array element (gene). It handles the entire array operations elementwise with a total of $popSize \times length_{of\_individual}$ operations running concurrently. Examples of this type include cost calculation, selection of parents, and crossover. The second type is a vector function where a single thread operates on the entire row of the array with a total of $popSize$ operations running concurrently. Although it operates in a parallel fashion on the entire array, it operates sequentially on each row. This type includes the kernels that require only one active thread in each warp (i.e., row) to access the entire chromosome such as: duplicate/missing nodes check, capacity constraints algorithm, and mutation.

It is worth mentioning that GPU functions support only numerical arrays with homogeneous widths. This property is not applicable in VRP problems because every single individual in the population array represents a possible solution for the problem and could have longer or shorter length than the others. To overcome this, all individuals are initialized with a fixed length of the longest possible solution. Additional genes inside shorter individuals are assigned to zero. Figure 3 illustrates
Figure 3: Algorithm implementation on GPU.
the implementation of the proposed algorithm on the GPU, the outline of the algorithm described in Section 1.3.2 is detailed as follows:

**Step 1:** the program starts by reading the problem data file that includes a list of node IDs, their $x$ and $y$ coordinates, their demand values, and the vehicle capacity. Then, it saves them into a data array of size $n \times 4$ where $n$ is the number of nodes. The pre-defined number of blocks and threads per block are used by the rest of the steps.

**Step 2:** the distance between every pair of nodes is represented by a hollow symmetric matrix (i.e., cost array) of size $n \times n$ is computed by a cost GPU kernel based on Equation 1. This array includes all the distances $c_{ij}$ between each pair of nodes. The cost array is saved in the GPU global memory since it is too large to be copied to shared memories.

**Step 3:** the initialize kernel is called to randomly initialize a population array $P_0$ of size $\text{popSize} \times \text{individual length}$ where $\text{popSize}$ is pre-selected and the individual length depends on the number of nodes. All random numbers are generated directly by the GPU using xoroshiro128+ pseudorandom number generator [39]. The reason of the random initialization of the population is to generate a pool of solutions that is spread across the entire search space [40]. Each row of $P_0$ represents a possible solution for the problem.

**Step 4:** refining kernels check for duplicate or missing nodes and insert depot nodes to count for capacity constraints if the cumulative customer demand values obtained by Equation 3 is larger than or equal to the vehicle capacity $Q$. Once the initial population is constructed, the evolution process begins.

**Step 5:** a binary tournament kernel is called to create a pool of four candidate arrays by randomly selecting individuals from $P_0$. Each corresponding row in the four arrays represents a prospective parent in the following process. The best two
individuals (the two with the lowest cost) from each corresponding row in the candidate arrays are selected to form two parent arrays.

**Step 6:** every corresponding row of the two parent arrays is used in a *crossover* kernel to form two child arrays. A standard k-point crossover is adopted where \( k \) is a static number for every single problem instance. This number is pre-selected from one to five according to the number of nodes in the problem instance.

**Step 7:** a random swap mutation process is applied to the child arrays by a *mutate* kernel. Two random genes in each individual are chosen to switch their locations with a pre-set occurrence (mutation) probability. Then, refining kernels are re-called to adjust the generated child arrays and relocate the depot nodes according to the capacity limits.

**Step 8:** *2-opt* local search kernel is applied to the child arrays to further improve the generated children. The cost of individuals is re-calculated to count for possible changes made through the entire evolution process, the 2-opt algorithm, and the refining process. Generated children enter the new population \( P_t \) only if they have a lower cost than their parents. This greedy replacement compares the children with their parents only, not with the entire population. If parents have very poor fitnesses and their children have better ones, the children will still be selected even if they have higher costs than some other parents. The adopted elitism concept compensates this issue to guarantee preserving high quality parents in the new population by carrying the top 5% individuals from the old population to the new one [14].

**Step 9:** to keep the diversity of the population, a *unique* kernel is used to remove duplicate individuals and replace them with others from parent and child arrays. The new population is then used for another evolution cycle until reaching a specific number of generations.
Finally, the best individual with the lowest cost in the final population is copied to the host (CPU) for program output. By utilizing these kernels, each thread on the GPU is assigned to work on one point in the chromosome which guarantees a parallel run of the proposed algorithm and hence an increase in the execution speed.

The pseudocodes shown in Algorithms 1 and 2 are example implementations of the cost calculating function on both CPU and GPU. Given the \( x, y \) coordinates of each node, the function calculates the distance \( c_{ij} \) between each pair and outputs an array of distances (costs).

**Algorithm 1** CPU implementation of the cost calculating function

**Input:** \( c_{ij} \) \( \forall i, j \in \mathcal{N} \)

1: function calCCOSTTBF(coord, cost_tbl)
2:    for index, node in enumerate(coord[:, 0]) do
3:        cost_tbl[index, index + 1] =
4:            np.round(np.hypot(np.subtract([coord[index, 2]] * len(coord[index + 1 :, 2]), coord[index + 1 :, 2]),
5:                np.subtract([coord[index, 3]] * len(coord[index + 1 :, 3]), coord[index + 1 :, 3])))
6:    end for
7:    cost_tbl = np.add(cost_tbl, np.transpose(cost_tbl))
8: return (cost_tbl)
9: end function
Algorithm 2 GPU implementation of the cost calculating function

<table>
<thead>
<tr>
<th>Input:</th>
<th>( c_{ij} \ \forall i, j \in \mathcal{N} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>function ( \text{CALC COST TBL}(\text{coord, cost tbl}) )</td>
</tr>
<tr>
<td>2:</td>
<td>( \text{th}<em>\text{row}, \text{th}</em>\text{col} = \text{cuda.grid}(2) )</td>
</tr>
<tr>
<td>3:</td>
<td>( \text{stride}_x, \text{stride}_y = \text{cuda.gridsize}(2) )</td>
</tr>
<tr>
<td>4:</td>
<td>for ( \text{row} ) in range(( \text{th}_\text{row}, \text{coord.shape}[0], \text{stride}_x )) do</td>
</tr>
<tr>
<td>5:</td>
<td>( \text{for col in range}(\text{th}_\text{col}, \text{coord.shape}[0], \text{stride}_y) ) do</td>
</tr>
<tr>
<td>6:</td>
<td>( \text{cost tbl}[\text{row}, \text{col}] = \text{round}(\text{hypot}[[\text{coord}[\text{row}, 2] - \text{coord}[\text{col}, 2], \text{coord}[\text{row}, 3] - \text{coord}[\text{col}, 3])]) )</td>
</tr>
<tr>
<td>7:</td>
<td>end for</td>
</tr>
<tr>
<td>8:</td>
<td>end for</td>
</tr>
<tr>
<td>9:</td>
<td>end function</td>
</tr>
</tbody>
</table>

It is noticed from the GPU implementation how the input and output arrays (i.e., \( \text{coord} \) and \( \text{cost tbl} \)) are distributed over the GPU grid. Line 2 gives a unique thread index within the entire grid. This thread will do the work on the array element which has the same index within the array. Line 3 is useful for arrays of size larger than the grid size. It enables the thread to have more than one unique index and stride over the grid using the \( \text{for} \) loops in Lines 4 and 5 to cover the entire array. This arrangement results in having each thread perform the same operation on parallel which, in turns, increases the execution speed of the algorithm. To compare between the CPU and the GPU implementations, both functions are executed for seven independent runs on a problem of 200 nodes. The results show that the average execution speed of the CPU is \( 8.81 \text{ ms} \pm 21.1 \mu\text{s} \) per loop whereas on the GPU it took \( 305 \mu\text{s} \pm 110 \mu\text{s} \) per loop of average speed. Despite the CPU version is also parallelized over the CPU cores, the GPU execution speed is 29 times faster. This is due to the special architecture that the GPU has which supports parallel intensive calculations as well as the limited number of cores the CPU has compared to the GPU. Interested users can refer to our Git repository for the source code: https://github.com/MarwanAbdelatti/GA_VRP_GPU.git
Table 1: Parameter settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>popSize</td>
<td>100</td>
</tr>
<tr>
<td>number of generations</td>
<td>Range from 2000 to $10^6$</td>
</tr>
<tr>
<td>Total number of blocks ($B \times B$)</td>
<td>5 $\times$ 5</td>
</tr>
<tr>
<td>Number of threads per block ($T \times T$)</td>
<td>20 $\times$ 20</td>
</tr>
<tr>
<td>Selection</td>
<td>Binary tournament</td>
</tr>
<tr>
<td>Crossover</td>
<td>k-point</td>
</tr>
<tr>
<td>Crossover probability</td>
<td>1</td>
</tr>
<tr>
<td>number of crossover points</td>
<td>Range from 1 to 5</td>
</tr>
<tr>
<td>mutation probability</td>
<td>1:15</td>
</tr>
</tbody>
</table>

1.4 Execution Results

The proposed algorithm is coded using CUDA Python with Numba and CuPy library, and runs in CUDA 10.0 environment. The GPU used is an NVIDIA GeForce RTX 2080 that has a Turing architecture and 8 GB global memory, 2944 CUDA cores, and 1024 maximum number of threads per block. The number of blocks ($B \times B$) is 5 $\times$ 5. In order to avoid 100% occupancy [32], each block is configured to have 400 ($20 \times 20$) threads. The population size ($popSize$) is selected to be 100. These arrangements are suitable for problems with individuals of 100 points to be processed concurrently. For larger problems, a stride is used to project the device grid on the input arrays with a $stride_value = B \times T = 100$. Theoretically, the algorithm should work on any GPU with Turing or Pascal architecture and 6 GB or more of shared memory. However, the only confirmed hardware we used is NVIDIA GeForce RTX 2080, GTX 1060 and GTX1070.

Table 1 summarizes the parameter settings. The number of generations and the number of crossover points depend on the problem size. Small problems require about 2000 generations to solve whereas large problems may require up to $10^6$ generations. Similarly, the number of crossover points is selected by trying the algorithm with different numbers.
The proposed algorithm has been tested on 14 common benchmark CVRP problems selected from [41, 42, 43, 44, 45, 46]. For the sake of fairness and to come to accurate conclusions, these benchmark problems have been selected with different sizes ranging from 16 to 76 nodes and different problem sets. However, work with larger problems is going on. For speed comparison purposes, an identical version of the proposed algorithm has been developed to entirely run on the CPU. Each problem set is run 10 times and the average is reported. A comparison between our results and the best-documented results is conducted. Moreover, we test the impact of the 2-opt local search on the proposed algorithm by running the algorithm for each problem set with and without the 2-opt. Two aspects have been considered to measure the performance of the proposed algorithm:

1. The execution speed which tracks the time elapsed by both algorithms to produce one population through the evolution processes.

2. The quality of solution where The \( %\text{Gap} \) [11] is used as a measure of quality for the obtained solution such that:

\[
%\text{Gap} = \frac{l_o - l^*}{l^*}
\]  

(4)

where \( l_o \) is the best solution obtained from our algorithm and \( l^* \) is the well-known published results from the literature. A good obtained solution is recognized by its low \( %\text{Gap} \) value. Table 2 shows the results of executing the algorithm (with the 2-opt included) on both CPU and GPU for the above-mentioned problems. The table shows the time elapsed to generate one population in seconds/loop, the best-known costs in the literature, and the best values obtained by the proposed algorithm (\( %\text{Gap} \) values).

It can be noticed from the table how the execution speed is significantly improved by using the GPU. While the elapsed execution time of the CPU is continuously increasing with the number of nodes, the elapsed time at the GPU is
changing in a small range of values. The table shows speedups of the GPU version over the CPU by 52.7 times for the 21-node problem, 162.6 times for the 31-node problem, 368.6 times for the 40-node problem, 321.5 times for the 50-node problem, 339 times for the 60-node problem, and 454.4 times for the 76-node problem. It also shows how the speed gap between the CPU and the GPU implementations increases with the number of nodes.

The proposed algorithm has shown an adequate performance by reaching the best-known values of some problems typed in bold face and obtaining better values in others indicated by a boldface, negative $\%Gap$ value, and an asterisk. However, some high $\%Gap$ values are also obtained.
Table 2: Numerical results of testing CPU and GPU versions of the algorithm on benchmark problems

<table>
<thead>
<tr>
<th>Problem ID</th>
<th>CPU time (sec/loop)</th>
<th>GPU time (sec/loop)</th>
<th>Best known</th>
<th>Obtained</th>
<th>Obtained %Gap (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-n16-k8</td>
<td>0.79</td>
<td>0.0509</td>
<td>450</td>
<td>450</td>
<td>0</td>
</tr>
<tr>
<td>P-n20-k2</td>
<td>9.23</td>
<td>0.0850</td>
<td>216</td>
<td>216</td>
<td>0</td>
</tr>
<tr>
<td>P-n21-k2</td>
<td>5.37</td>
<td>0.1020</td>
<td>211</td>
<td>211</td>
<td>0</td>
</tr>
<tr>
<td>P-n22-k8</td>
<td>4.08</td>
<td>0.1060</td>
<td>603</td>
<td>590*</td>
<td>-2.16*</td>
</tr>
<tr>
<td>P-n23-k8</td>
<td>12.62</td>
<td>0.0821</td>
<td>529</td>
<td>529</td>
<td>0</td>
</tr>
<tr>
<td>B-n31-k5</td>
<td>20.86</td>
<td>0.1152</td>
<td>672</td>
<td>677</td>
<td>0.74</td>
</tr>
<tr>
<td>P-n40-k5</td>
<td>27.39</td>
<td>0.0743</td>
<td>458</td>
<td>478</td>
<td>4.37</td>
</tr>
<tr>
<td>B-n45-k5</td>
<td>31.14</td>
<td>0.0645</td>
<td>751</td>
<td>762</td>
<td>1.46</td>
</tr>
<tr>
<td>B-n50-k8</td>
<td>21.54</td>
<td>0.0670</td>
<td>1312</td>
<td>1327</td>
<td>1.14</td>
</tr>
<tr>
<td>P-n55-k15</td>
<td>36.00</td>
<td>0.0987</td>
<td>989</td>
<td>975*</td>
<td>-1.42*</td>
</tr>
<tr>
<td>P-n60-k15</td>
<td>34.51</td>
<td>0.1018</td>
<td>968</td>
<td>1061</td>
<td>9.61</td>
</tr>
<tr>
<td>P-n65-k10</td>
<td>25.34</td>
<td>0.1203</td>
<td>792</td>
<td>873</td>
<td>10.23</td>
</tr>
<tr>
<td>P-n70-k10</td>
<td>42.03</td>
<td>0.1312</td>
<td>827</td>
<td>892</td>
<td>7.86</td>
</tr>
<tr>
<td>E-n76-k14</td>
<td>48.80</td>
<td>0.1074</td>
<td>1021</td>
<td>1109</td>
<td>8.62</td>
</tr>
</tbody>
</table>
Figure 4 shows the algorithm execution time on both CPU and GPU. The logarithmic scale is meant to show the subtle differences in the results and to enable the plot to accept the large values resulted from the CPU. The figure shows that the execution time of the CPU keeps increasing with the problem size whereas the GPU execution time is oscillating around 0.1 sec/loop for all problem instances. This does not necessarily mean that the GPU execution time will be held around 0.1 sec/loop for any problem size.

We further investigate the impact of the 2-opt local search on the performance of the algorithm on the GPU. For each problem set, we exclude the 2-opt from the algorithm and examine the results. Table 3 shows the obtained value, the %Gaps, and the execution times of running the algorithm with and without the 2-opt. Reaching the best-known values is marked in bold face and obtaining better values is marked by a boldface, negative %Gap value, and an asterisk. It is obvious from the table the important role the 2-opt has in the algorithm. It is shown that taking out the 2-opt deteriorates the performance and negatively affects the solution quality. The %Gap obtained without the 2-opt is sometimes 5 times higher than that of having the 2-opt in the loop despite the reduced execution time. However, having longer execution time by involving the 2-opt is a feasible cost to pay for having a better solution quality in return, especially that increase is only 58.6 ms/loop at a maximum.
Figure 4: Execution time of CPU vs GPU.

Figure 5: Solution quality of the proposed algorithm
Table 3: Numerical results of running the GPU algorithm with and without the 2-opt

<table>
<thead>
<tr>
<th>Problem ID</th>
<th>Best known</th>
<th>Obtained value</th>
<th>%Gap (%)</th>
<th>Obtained value</th>
<th>%Gap (%)</th>
<th>Obtained value</th>
<th>%Gap (%)</th>
<th>Time (sec/loop)</th>
<th>Time (sec/loop)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-n16-k8</td>
<td>450</td>
<td>450</td>
<td>0.00</td>
<td>456</td>
<td>1.33</td>
<td>0.0509</td>
<td></td>
<td>0.0389</td>
<td></td>
</tr>
<tr>
<td>P-n20-k2</td>
<td>216</td>
<td>216</td>
<td>0.00</td>
<td>254</td>
<td>17.59</td>
<td>0.0850</td>
<td></td>
<td>0.0501</td>
<td></td>
</tr>
<tr>
<td>P-n21-k2</td>
<td>211</td>
<td>211</td>
<td>0.00</td>
<td>244</td>
<td>15.64</td>
<td>0.1020</td>
<td></td>
<td>0.0734</td>
<td></td>
</tr>
<tr>
<td>P-n22-k8</td>
<td>603</td>
<td>590*</td>
<td>-2.16</td>
<td>603</td>
<td>0.00</td>
<td>0.1060</td>
<td></td>
<td>0.0474</td>
<td></td>
</tr>
<tr>
<td>P-n23-k8</td>
<td>529</td>
<td>529</td>
<td>0.00</td>
<td>554</td>
<td>4.73</td>
<td>0.0821</td>
<td></td>
<td>0.0696</td>
<td></td>
</tr>
<tr>
<td>B-n31-k5</td>
<td>672</td>
<td>677</td>
<td>0.74</td>
<td>732</td>
<td>8.93</td>
<td>0.1152</td>
<td></td>
<td>0.0790</td>
<td></td>
</tr>
<tr>
<td>P-n40-k5</td>
<td>458</td>
<td>478</td>
<td>4.37</td>
<td>620</td>
<td>35.37</td>
<td>0.0743</td>
<td></td>
<td>0.0603</td>
<td></td>
</tr>
<tr>
<td>B-n45-k5</td>
<td>751</td>
<td>762</td>
<td>1.46</td>
<td>840</td>
<td>11.85</td>
<td>0.0645</td>
<td></td>
<td>0.0558</td>
<td></td>
</tr>
<tr>
<td>B-n50-k8</td>
<td>1312</td>
<td>1327</td>
<td>1.14</td>
<td>1529</td>
<td>16.54</td>
<td>0.0670</td>
<td></td>
<td>0.0361</td>
<td></td>
</tr>
<tr>
<td>P-n55-k15</td>
<td>989</td>
<td>975*</td>
<td>-1.42</td>
<td>1058</td>
<td>6.98</td>
<td>0.0987</td>
<td></td>
<td>0.0898</td>
<td></td>
</tr>
<tr>
<td>P-n60-k15</td>
<td>968</td>
<td>1061</td>
<td>9.61</td>
<td>1154</td>
<td>19.21</td>
<td>0.1018</td>
<td></td>
<td>0.0895</td>
<td></td>
</tr>
<tr>
<td>P-n65-k10</td>
<td>792</td>
<td>873</td>
<td>10.23</td>
<td>1097</td>
<td>38.51</td>
<td>0.1203</td>
<td></td>
<td>0.0796</td>
<td></td>
</tr>
<tr>
<td>P-n70-k10</td>
<td>827</td>
<td>892</td>
<td>7.86</td>
<td>1189</td>
<td>43.77</td>
<td>0.1312</td>
<td></td>
<td>0.0989</td>
<td></td>
</tr>
<tr>
<td>E-n76-k14</td>
<td>1021</td>
<td>1109</td>
<td>8.62</td>
<td>1309</td>
<td>28.21</td>
<td>0.1074</td>
<td></td>
<td>0.0862</td>
<td></td>
</tr>
</tbody>
</table>
Figures 6 and 7 illustrate the results detailed in Table 3 and compare the performance of the algorithm with and without the 2-opt in terms of the solution quality and the execution time respectively.

![Obtained %Gap With and Without 2-Opt](image)

Figure 6: Performance With and Without the 2-opt

![Elapsed GPU time With and Without 2-Opt](image)

Figure 7: Execution time with and without the 2-opt

1.5 Conclusions

This work proposes a new hybrid algorithm running on a GPU to take advantage of the special CUDA parallel processing. In this algorithm, a 2-opt local search is incorporated with a GA to solve capacitated VRP problems. The algorithm uti-
lizes the GPU for handling the entire operations such as initializing population, reproduction, 2-opt local search, and the refining processes. A clone-restricting process is also adopted to maintain the diversity of the population.

In order to conduct a precise experiment, two versions of the algorithm have been applied to the same problem instances: one runs on the CPU and the other on the GPU. Two performance measures have been considered; (1) the execution time in seconds per loop; and (2) the %Gap value between the obtained costs and the best-known values. Corresponding to the first measure, a significant improvement in the execution speed has been observed. Unlike the exponential increase in CPU execution time, the GPU execution time is very low and almost constant. This is helpful when dealing with larger problem instances and when using a large number of generations for reaching higher-quality solutions in a feasible time. Considering the second measure, the proposed algorithm reached the best-known values of some problems and improved some others. High values are obtained for some problems. To study the 2-opt effect, the algorithm is run with and without it. The results show a significant impact of the 2-opt despite the increase in the execution time which is acceptable compared to the obtained improvements.

Our contributions in this work can be summarized as: (1) we address the execution speed problem in VRPs by developing an algorithm that is entirely executed on a GPU; (2) we investigate the utilization of a local search algorithm (i.e., the 2-opt) with the GA; (3) we improve some best-documented values of VRP benchmark problems; and (4) provide a detailed numerical analysis and comparison plots for the algorithm performance. Testing the proposed algorithm with larger problems is ongoing on. Further investigations will be conducted to improve the performance of the proposed algorithm to obtain higher-quality solutions. This will include different crossover operators like Best Cost-Best Route and Cyclical
crossover, different mutation operators, and using modified GA algorithms like Cellular GA (cGAs) and Island model GAs for instance.

**List of References**


Optimizing a GPU-accelerated genetic algorithm for the vehicle routing problem

Marwan F. Abdelatti\textsuperscript{1*}, Abdeltawab Hendawi\textsuperscript{2}, and Manbir S. Sodhi\textsuperscript{1}

\textsuperscript{1}Department of Industrial Engineering, University of Rhode Island, Kingston, Rhode Island, 02881

\textsuperscript{2}Department of Computer Science, University of Rhode Island, Kingston, Rhode Island, 02881

\*Corresponding author: mabdelrazik@uri.edu

\textit{Published in the Proceedings of the 2021 Genetic and Evolutionary Computation Conference Companion (GECCO), 117–118.}

ACM ISBN 978-1-4503-8351-6/21/07

https://doi.org/10.1145/3449726.3459458
Abstract

The capacitated vehicle routing problem (CVRP) is an NP-hard optimization problem with many applications. Genetic algorithms (GAs) are often used to solve CVRPs but require many parameters and operators to tune. Incorrect settings can result in poor solutions. In this work, a design of experiments (DOE) approach is used to determine the best settings for GA parameters. The GA runs entirely on an NVIDIA RTX 3090 GPU. The GPU execution for a 200-node benchmark shows a speed by a factor of 1700 compared to that on an octa-core i7 CPU with 64 GB RAM. The tuned GA achieved a solution for a 400-node benchmark that is 72% better than that of an arbitrarily tuned GA after only 263 generations. New best-known values for several benchmarks are also obtained. A comparison between the performance of the algorithm with different hardware and tuning sets is also reported.
2.1 Introduction

Genetic algorithms (GAs) are commonly used for the capacitated vehicle routing problems (CVRP) to find near-optimal delivery/pick-up routes through a set of locations [1, 2, 3]. Because of the large number of parameters and operators used in the GA, and the limited knowledge about the effect of parameter interactions [4], a two-level $2^k$ factorial design of experiment (DOE) is used to determine the best settings for the GA parameters. Pilot runs are performed on a small problem settings for the parameters are determined. The tuned parameters are then applied to larger problems. The developed GA (GPU-GA) runs entirely on graphics processing units (GPUs) to capitalize their parallel execution capabilities [5]. The tuned algorithm obtained an 8.89% gap from the best-known solution for a 200-node benchmark problem $M-n.200-k17$ [6] after 50,000 generations. A 72% improvement in the gap for a 400-node benchmark compared to arbitrary parameter sets within 263 generations only, and new best-known values for other problems. Compared to sequential executions on CPUs, GPUs achieved an improvement factor of 1700 in the execution speed.

2.2 GPU-Accelerated GA for CVRP

GPU-GA is a GA algorithm with a 2-opt local search. A clone-restricting procedure preserves the diversity of population throughout the evolution process. The GPU grid is arranged with a total of $(B*T)^2$ threads running in parallel where $B \times B$ is the 2D array arrangement of the GPU blocks and $T \times T$ are the thread/block arrangement, $B$ and $T$ are integers. A block diagram of the algorithm is shown in Figure 8 [5]. The algorithm starts by using the CPU for reading problem data, and then creates and copies an array of nodes to the GPU. The GPU creates the cost table array between nodes. Evolutionary processes like random initialization, selection, crossover, and mutation as well as the 2-opt local
Table 4: Results of running the tuned GPU-GA

<table>
<thead>
<tr>
<th>Problem ID</th>
<th>Best Known (s/loop)</th>
<th>%Gap</th>
<th>Speed (s/loop)</th>
<th>%Gap</th>
<th>Speed (s/loop)</th>
<th>%Gap</th>
<th>Speed (s/loop)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-n16-k8</td>
<td>450</td>
<td>0.00</td>
<td>0.1006</td>
<td>0.00</td>
<td>0.0397</td>
<td>0.00</td>
<td>0.7936</td>
</tr>
<tr>
<td>P-n20-k2</td>
<td>216</td>
<td>0.00</td>
<td>0.2392</td>
<td>0.00</td>
<td>0.0837</td>
<td>18.04</td>
<td>9.23</td>
</tr>
<tr>
<td>P-n22-k8</td>
<td>603</td>
<td>0.00</td>
<td>0.1205</td>
<td>-2.16</td>
<td>0.0479</td>
<td>3.02</td>
<td>4.08</td>
</tr>
<tr>
<td>P-n23-k8</td>
<td>529</td>
<td>0.00</td>
<td>0.178</td>
<td>0.00</td>
<td>0.0524</td>
<td>3.48</td>
<td>12.62</td>
</tr>
<tr>
<td>B-n31-k5</td>
<td>672</td>
<td>0.00</td>
<td>0.2145</td>
<td>0.00</td>
<td>0.0905</td>
<td>12.01</td>
<td>20.86</td>
</tr>
<tr>
<td>P-n40-k5</td>
<td>458</td>
<td>3.06</td>
<td>0.1878</td>
<td>0.22</td>
<td>0.0831</td>
<td>40.04</td>
<td>27.39</td>
</tr>
<tr>
<td>B-n50-k8</td>
<td>1312</td>
<td>1.68</td>
<td>0.1992</td>
<td>1.14</td>
<td>0.0827</td>
<td>34.01</td>
<td>21.54</td>
</tr>
<tr>
<td>P-n55-k15</td>
<td>989</td>
<td>-2.93</td>
<td>0.4489</td>
<td>-3.44</td>
<td>0.1598</td>
<td>21.01</td>
<td>36</td>
</tr>
<tr>
<td>P-n65-k10</td>
<td>792</td>
<td>1.77</td>
<td>0.4566</td>
<td>5.05</td>
<td>0.1823</td>
<td>52.10</td>
<td>25.34</td>
</tr>
<tr>
<td>P-n70-k10</td>
<td>827</td>
<td>7.74</td>
<td>0.4655</td>
<td>4.47</td>
<td>0.1885</td>
<td>56.08</td>
<td>42.03</td>
</tr>
<tr>
<td>A-n80-k10</td>
<td>1763</td>
<td>8.91</td>
<td>1.5229</td>
<td>7.26</td>
<td>0.1848</td>
<td>80.15</td>
<td>63.67</td>
</tr>
<tr>
<td>E-n101-k14</td>
<td>1071</td>
<td>20.17</td>
<td>1.8299</td>
<td>7.56</td>
<td>0.2386</td>
<td>100.17</td>
<td>85.51</td>
</tr>
<tr>
<td>M-n200-k17</td>
<td>1373</td>
<td>99.49</td>
<td>2.8063</td>
<td>8.30</td>
<td>0.9387</td>
<td>201.99</td>
<td>1592.14</td>
</tr>
<tr>
<td>Golden_3</td>
<td>11,063.22</td>
<td>36.68</td>
<td>12.3702</td>
<td>13.86</td>
<td>4.7246</td>
<td>401.45</td>
<td>9240</td>
</tr>
</tbody>
</table>

 searches run on the GPU until reaching a stopping criteria: either stagnating at a cost value for a certain number of generations, or, reaching a limit on the number of generations (in case of performance analysis). The final solution is sent back to the CPU for output. Interested readers can clone the GPU-GA from the GitHub repository\(^1\).

### 2.3 Design of Experiment

To obtain high-quality solutions from the proposed algorithm, multiple GA parameters with different values have been collected from various studies [7, 8, 9]. A \(2^k\) factorial design of experiment (DOE) is utilized on a 70-node (\(P-n70-k10\)) benchmark to analyze different parameter combinations and their interactions. Each combination is independently run five times. The parameters considered are:

- **crossover operator**: 1-point or 2-point crossover; **mutation operator**: either

\(^1\)https://github.com/MarwanAbdelatti/GA_VRP_mod
inversion mutation where a random range of genes is inverted, or swap mutation that switches two random genes; population size: 10× or 20× the number of nodes $n$. elitism rate: the percentage of parents transferred to the new generation along with the offspring; and crossover and mutation rates: the occurrence probabilities of crossover and mutation during the evolution process respectively.

It was found from the experiment that the elitism rate has no significant effect on the GA behavior.

2.4 Execution Results and Conclusions

The tuned GPU-GA is used to solve a number of benchmark problems. The %Gap between the obtained and the best-known solution is the measure of solution quality where

$$\% \text{Gap} = \frac{\text{obtained soln} - \text{best known}}{\text{best known}}.$$  

The stopping criteria is reaching or exceeding the best-known value or solution stagnation. The CPU is an Intel Octa-core i7 CPU @ 2.4 GHz and 64 GB RAM, and an NVIDIA RTX 3090 GPU with 24 GB of memory. The GA executes on 35×35 blocks with 20×20 threads per block. The obtained solution quality is compared with solutions obtained with random parameter settings. The optimal parameters for the GA were: 1-point crossover, inversion mutation, a 10×$n$ population size, a crossover rate of 0.8, and a mutation rate of 0.1.

Numerical results show that the tuned parameter set results in faster convergence and no premature termination. The optimized algorithm is run on different hardware (e.g., CPU, NVIDIA RTX 3090 GPU with 10,496 cores and 24 GB memory, and RTX 2080Ti with 4,352 cores and 11 GB memory). The results are shown in Table 4. It is found that the number of CUDA cores has a great impact on the GA performance - more than double the speed is achieved on the 3090 GPU compared to the 2080Ti. Worth mentioning is that for a 200-node problem, the
algorithm executes at a speed of 26 min/generation on the CPU compared with 0.9 sec/generation on the RTX 3090 GPU, reducing the execution time from 2.4 years on the CPU to 12 hours on GPU for a similar result quality.

List of References


Figure 8: Algorithm implementation on GPU.
A Multi-GPU Parallel Genetic Algorithm For Large-Scale Vehicle Routing Problems

Marwan F. Abdelatti\textsuperscript{1*}, Manbir S. Sodhi\textsuperscript{1}, and Resit Sendag\textsuperscript{2}

\textsuperscript{1}Department of Industrial Engineering, University of Rhode Island, Kingston, Rhode Island, 02881
\textsuperscript{2}Department of Electrical And Computer Engineering, University of Rhode Island, Kingston, Rhode Island, 02881

*Corresponding author: mabdelrazik@uri.edu

https://10.1109/HPEC55821.2022.9926363
abstract

The Vehicle Routing Problem (VRP) is fundamental to logistics operations. Finding optimal solutions for VRPs related to large, real-world operations is computationally expensive. Genetic algorithms (GA) have been used to find good solutions for different types of VRPs but are slow to converge. This work utilizes high-performance computing (HPC) platforms to design a parallel GA (PGA) algorithm for solving large-scale VRP problems. The algorithm is implemented on an eight-GPU NVIDIA DGX-1 server. Maximum parallelism is achieved by mapping all algorithm arrays into block threads to achieve high throughput and reduced latency for full GPU utilization. Tests with VRP benchmark problems of up to 20,000 nodes compare the algorithm performance (speed) with different GPU counts and a multi-CPU implementation. The developed algorithm provides the following improvements over CPU or single-GPU-based algorithms: (i) larger problem sizes up to 20,000 nodes are handled, (ii) execution time is reduced over the CPU by a factor of 1,700, and iii) for the range tested, the performance increases monotonically with the number of GPUs.

Keywords: multi-GPU, parallelization, parallel genetic algorithms, local search, vehicle routing problem

3.1 Introduction

Ongoing supply chain shortages have reinforced the need for better management of logistics resources and a more resilient supply chain. The vehicle routing problem (VRP) is crucial for logistics operations and is one of the most widely studied combinatorial optimization problems. It involves finding the optimal set of routes for a fleet of trucks to serve a given set of customers subject to some constraints [1]. The VRP is NP-Hard, and large problems require massive computational resources and support exceeding the capabilities of service providers.
Heuristics and metaheuristic methods exist among the approaches for finding solutions for large-scale VRPs efficiently. These have acceptable execution speeds, and solutions [2]. Typical examples of these include: local search (e.g., 2-Opt) [3], Clarke and Wright [4], Tabu search [5], and genetic algorithms (GA) [6]. GAs are based on computational emulations of natural reproduction features such as selection, mating, and mutation processes. GAs work on a population of feasible solutions (i.e., individuals or chromosomes) that are evaluated by a fitness function. A mating mechanism selects the fittest individuals (parents). Mutations introduce diversity into the generated offspring. Metaheuristics in general and GAs, in particular, have a special property that distinguishes them from many other heuristic methods in that they operate on a population of potential solutions. This improves the efficiency of the algorithm and the likelihood of finding a good solution [6]. Research has shown that integrating GAs with local search methods such as 2-opt improves its performance and the solution quality [6, 2].

A common problem with GAs is that solutions get stuck in local minima, often when solving medium and large-size problems. Also, as with algorithms relying on randomized solutions, such problems are slow to converge to acceptable solutions [7]. Approaches to overcome these issues include optimizing the GA parameters: using fuzzy logic [8], design of experiments (DOE) [9, 10, 11], or using machine learning [12]. Other approaches include parallel versions of GAs (PGA) like fine-grained PGA [13, 14], and coarse-grained PGA [15, 7]. However, as computation power increases, successively larger benchmark problem instances are now appearing in the literature, and marshaling the resources to execute complex algorithms on populations involving thousands of generations is not easy.

Developments in high-performance computing (HPC) using GPUs have opened new levels of accelerated computing. In this work, we utilize the computing
power of multi-GPU computers to run a parallel GA for the vehicle routing problem (VRP). A comprehensive case study is carried out by using a genetic algorithm to solve a VRP. A coarse-grained parallel genetic algorithm (PGA) is integrated with a local search algorithm to execute on a multi-GPU DGX-1 server with eight GPUs. Each GPU is considered a single processing node controlled by a separate CPU thread and performs evolutionary operations on a sub-population. After a pre-set number of generations (i.e., migration interval), all GPUs share a fraction of their solutions (i.e., migration rate) and update their population according to the best solutions found between them.

The rest of this paper is organized as follows: a literature review on the use of HPC for heuristic and metaheuristic algorithms is discussed in Section 3.2 with a focus on VRPs. VRPs characteristics are reviewed in Section 3.3. The developed algorithm and its implementation details for multi-GPU processing are described in Section 3.4. Numerical results are reported and discussed in Section 3.5 followed by a concluding Section 3.6.

### 3.2 Related Work

HPC is now one of the main engines for scientific and engineering research. Apart from its application in artificial intelligence like in computer vision [16], deep reinforcement learning [17], recommender systems [18], and natural-language processing [19], HPC is deployed extensively by numerous heuristic and population-based metaheuristics algorithms for other applications. Salomon et al. [20] presented an efficient approach to finding combinatorial biological patterns (i.e., motifs) in a set of DNA sequences. Hash-based heuristics were used to skip unnecessary computations, and multi-core CPU and single GPU computation models were compared. In [21], heuristics are executed on a cloud HPC for solving a reservation scheduling problem. A multi-GPU implementation of a Min-Min heuristic
algorithm is introduced in [22] for scheduling tasks to heterogeneous computational resources. An HPC-based logistics operation and optimization system was introduced in [23] to increase capacity and throughput while reducing costs at seaport container terminals. A parallel genetic algorithm is designed in [24] on a 46-compute-node cluster for the capacitated p-median problem.

There is a vast body of literature describing genetic algorithms for solving VRPs [6, 2, 25, 26]. A hybrid of GA and neighborhood search methods were used for the capacitated VRP (CVRP) in [6]. The work adopts a genetic representation where the customers are placed in fixed locations (i.e., e-codons) in the chromosomes, and every e-codon specifies the vehicle number to which the corresponding customer is assigned. The fixed length of chromosomes simplifies the implementation and obtains high-quality solutions, but it fails to meet the capacity constraints in many cases. Another solution representation has been adopted in [2] for the distance-constrained VRP where each chromosome represents a list of routes without depot nodes delimiters, and each e-codons represents a customer node. The algorithm is hybridized with a local search procedure, and a particular splitting procedure is employed to convert GA chromosomes to feasible solutions. However, the split algorithm is computationally intensive. The most common genetic representation of VRPs includes trip delimiters into the chromosomes that hold the node IDs in their e-codons as in [2, 26, 25]. In GAs, preserving the population diversity is crucial to avoid premature convergence and balance between exploration and exploitation of the search space. Clone-restricting algorithms can prevent duplicate solutions (clones), but these result in large computational loads (around 95% of CPU time) [2, 26, 25]. Other techniques include, but are not limited to, restricted mating [27] at the selection stage and elitism [28] at the generation of the offspring stage. However, long execution times remain the challenge.
As a result, PGAs are used to find high-quality solutions in a reasonable amount of time.

By definition, PGAs exploit parallelism, leading to increased efficiency of the search in the solution space. Coarse-grained PGAs improve search-space exploration by dividing the population into separate sub-populations (demes) with different random initializations. Since each deme deals with a sub-population, execution speeds are faster. In [29], a coarse-grained PGA with a hypercube connection topology has been introduced. The algorithm’s performance was evaluated using a function maximization problem and compared with a serial GA. The effect of the migration parameter values on the performance of the PGA was studied in [30]. It was found that the PGA with appropriate migration parameters eventually obtained higher average fitness than the algorithms with no migration. A coarse-grained PGA was proposed in [31] for the VRP with a heterogeneous fleet for problems up to 100 nodes. It has also been used in [32] for solving CVRP problems of sizes up to 80 nodes. Yussof et al. [33] use a coarse-grained PGA for solving shortest path routing, and the effects of migration rates and different network topologies on the solution accuracy are discussed. A hybrid coarse-grained PGA with elitism scheme and Tabu search is used in [34] for a VRP to minimize the total length of routes and balance of route lengths for problems up to 199 nodes.

Several studies have also reported GA implementations for route planning problems on HPC platforms [35, 36, 37, 38]. A PGA on a multi-CPU cluster was introduced in [35] for the route planning problem. Benaini et al. [36] implement a GA on a GPU for the Dynamic VRP (DVRP), which was able to find good solutions for problems up to 3,000 nodes at speed $35 \times$ faster than regular CPU implementations. A partial GPU implementation is reported in [37] for another
VRP. The execution speed was significantly affected by the modules running on the CPU (only a 15× speedup was obtained compared to a pure CPU implementation). The work in [38] utilizes a GPU for the cost function calculation for the CVRP, improving this calculation by a factor of 1,000× compared to the CPU implementation. A GPU-based GA with a local search algorithm is proposed in [39] for the vehicle routing problem with route balancing (VRPRB). The GA was implemented on the CPU while the local search algorithm was executed on the GPU. Previous work, [40] presented an improved GA, incorporated with a local search algorithm for the CVRP. The algorithm was entirely executed on a GPU and effectively provided high-quality solutions within reasonable computational times and near-optimal solutions for small benchmark problems. This framework addressed the execution speed problem in CVRPs, but acceptable solutions for the large-scale problems were not achieved due to tuning issues [40].

Several implementations of GAs utilizing multi-GPUs are reported in the literature. The work in [41] implements a cellular GA on a multi-GPU platform where each thread on the GPU handles an entire individual in the population. The algorithm has been applied to solving three discrete and three continuous functions. Although the study showed that a single GPU utilization resulted in speedups between 8× and 771× over a CPU implementation, using two GPUs increases the computational time by a small fraction compared to a single GPU. The authors attribute this to the overhead caused by data transfers between the GPUs. A coarse-grained PGA is implemented on a multi-GPU cluster for the knapsack problem in [42]. The speedups obtained by this 14-GPU implementation were 35× compared to four CPUs, 194× compared to one CPU, and 781× compared to a single-thread implementation.
3.3 Problem Definition

The Capacitated Vehicle Routing Problem (CVRP) seeks the optimal set of routes for a fleet of trucks to serve a given set of customers, each with a demand $d_i$, such that the total demands of the customers assigned to identical trucks cannot exceed the truck capacity, $Q$, and the customers cannot be split [43]. The number of trucks required is not necessarily known before solving the problem [2]. A typical solution representation of a CVRP problem in GA is depicted in Figure 9 where the first digit represents the generation number where solutions belong. The zeros represent a visit to the depot. The last digit represents the solution fitness value, and the rest of the numbers are the customer node IDs.

GAs are an iterative process, and at iteration $t$, each row in the population array represents a feasible solution called an individual or a gene $l_t$. In this work, an individual represents a sequence of visited customers delimited by the depot node to factor in the capacity constraint. The number of rows of the population array $P$ called the population size (popSize) is static and set at the start. The population array is initialized randomly, and subsequent populations are generated by evolutionary processes such as cross-overs and mutations. The objective of the algorithm is to find the solution with the least cost $f_{i,t}$ where $f_{i,t}$ is the sum of the costs $c_{ij}$ of the associated edges $i$, and $j$ at iteration $t$ i.e., $f_{i,t} = \sum_{i,j=0}^{n} c_{ij}, \forall i \neq j$, and $n$ is the number of nodes. The 2-opt local search algorithm improves every route $r_i$ in every solution $l_{i,t}$.

Because the VRP is NP-hard, we propose a parallel genetic algorithm implemented on a multi-GPU server to reduce the execution times. Following the nomenclature typically used for genetic descriptions, a complete representation of the genome (Figure 9), called a “gene”, represents one possible solution individual in our CVRP context. One gene consists of smaller units called “chromosomes”
that represent single stops in the solution, and these consist of units called “e-
codons”.

3.4 Algorithm Description

We incorporate a parallel coarse-grained GA (PGA) with a local search algo-
rithm (2-opt to be more specific) to solve large capacitated vehicle routing problem
instances.

3.4.1 Hardware

The algorithm is designed to run entirely on a DGX-1 server with eight
NVIDIA Tesla v100 GPUs. Each GPU has a 32 GB shared memory and 5,120
CUDA cores. The eight GPUs are connected with a hybrid cube-mesh topology
through NVLink connection of bandwidth 25 GB/s in each direction as depicted in
Figure 10. The figure shows two clusters of four fully connected GPUs; these clus-
ters are connected via NVLink. It has been shown that this topology provides the
highest bandwidth of any 8-GPU NVLink topology and provides the flexibility of
utilizing subdivisions of fully connected halves or inter-connected pairs [44]. Using
NVLink connections to span the gap between the two clusters guarantees direct
data flow between the GPUs. This relieves the pressure on the PCIe bus and the
CPUs interconnections and avoids intense usage of system memory [44]. More-
over, the NVLink has higher bandwidth and lower latency than PCIe links. The
DGX server’s GPUs support the Peer-to-Peer (P2P) communication, which allows
program kernels to exchange data from one GPU’s memory to another directly
3.4.2 Proposed Algorithm

Every GPU is controlled by one CPU thread [45] and is considered a separate processing node responsible for processing a deme subbed from the main GA population. In addition to the standard GA parameters like population size, crossover and mutation rates, and operators, the coarse-grained GA has three additional operators: (i) the network topology operator that defines the connectivity between the processing nodes, which are set to be the GPU connection topology (i.e., the hybrid cube-mesh connection), (ii) the percentage of sub-populations exchanged between the nodes, the migration rate, and (iii) the number of generations at which this exchange occurs, the migration interval [7].

As illustrated in the flowchart in Figure 11, after reading the problem data file, the calculateLinearCost() kernel is called on each GPU separately. The cost table representing the Euclidean distance between each customer node is computed...
and saved in the GPU global memory since this table is too large (of size $n^2$) to be copied to shared memories. It is worth mentioning that since the customer graph is undirected, the cost table is a symmetric matrix with zeros at the diagonal. The upper triangle of the cost table (in blue) is flattened to save memory space. Given the following 2D symmetric matrix:

\[
\begin{bmatrix}
  x_{00} & x_{01} & x_{02} & \ldots & x_{0n} \\
  x_{10} & x_{11} & x_{12} & \ldots & x_{1n} \\
  x_{20} & x_{21} & x_{22} & \ldots & x_{2n} \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  x_{n0} & x_{n1} & x_{n2} & \ldots & x_{nn}
\end{bmatrix}
\]

the following formula is used to flatten the upper triangle into the 1D vector:

\[
\begin{bmatrix}
  x_{01} & x_{02} & \ldots & x_{0n} & x_{12} & x_{13} & \ldots & x_{n-1,n}
\end{bmatrix}
\]

\[
k = j - i (0.5i - n + 1.5) - 1 \tag{5}
\]

where $i$ and $j$ are row and column indices, respectively, $k$ is the derived index, and $n$ is the number of nodes. Each GPU randomly initializes its sub-population (deme) using the \texttt{initializePop()} kernel and \texttt{xoroshiro128+} pseudo-random number generator. The generated demes are checked for duplicate or missing nodes as well as the capacity constraints by the \texttt{refinePopulation()} process. The total cost of each gene is computed by \texttt{computeFitness()} kernel, which sums the distance between each pair of nodes as discussed in Section 3.3. Evolution processes start separately on each GPU with a binary tournament selection and create pool arrays of size ($deme\_size \times 4$) holding the row indices of candidates randomly selected. The two fittest individuals with the lowest costs are selected for a standard 1-point crossover using \texttt{crossover()} kernel to form two children (offspring) arrays. The inversion mutation performed by the \texttt{mutate()} kernel is applied to the offspring produced. The offspring are then checked for duplicate
or missing nodes and capacity constraints. Subsequently, the 2-opt local search kernel is applied to improve the quality of the offspring, and the fitness is then calculated. A greedy replacement strategy is adopted to ensure that the children replace the parents only if they have better fitnesses.

**Remark 3.4.1.** *It is worth mentioning that the greedy replacement compares the children with their parents only. Children may replace their parents even if these children have lower fitnesses than other parents. The elitism concept compensates for this issue by taking the best 5% parents into the new generation to preserve high-quality properties [46].*

**Remark 3.4.2.** *Since the probability of getting two identical solutions in a large GA population is very low, the clone-restricting algorithm from [40] is dropped. This speeds up the execution by a factor of 4 without negatively affecting the solution quality.*

The new generation is then utilized in the next iteration to produce new offspring and so forth until reaching the migration interval (selected to be 1,000 for small runs and 10,000 for big runs), where all GPUs exchange a fraction of their solutions. At this stage, `migratePop()` sets the population at GPU 0 to receive the best individuals from the populations on other GPUs by replacing lower-quality individuals at GPU 0. Once the population is updated, it is communicated back to the other GPUs by `broadcastPop()` to start a new iteration. Upon reaching the termination iteration count, the migration mechanism is performed one last time to update the population on GPU 0, and the individual with the lowest cost is copied to the host (CPU) for output. It should be noted that when working with eight GPUs, GPUs 5, 6, and 7, as shown in Figure 10, do not have a direct connection to GPU 0. To have them exchange data with GPU 0 avoiding the PCIe bus, we indirectly transfer data to intermediate GPUs 1, 2, and 3, which
are directly connected to them and to GPU 0 as well. This transfer mechanism is referred to as "hops" and is utilized for both the migration and the broadcast processes.

### 3.4.3 GPU implementation

Many GA implementations in the literature assign one thread for each individual in the population, where each thread is treated as a sequential processing node that executes regular FOR... loops. This does not make full use of the GPU capabilities and limits the number of customer nodes the algorithm can handle. Our proposed algorithm assigns one thread for every array element (i.e., customer node) to fully utilize the GPU cores and achieve maximum parallelism. A 2D block grid is initialized where each block is represented as a 2D array of threads, all running in parallel. All algorithm arrays are mapped onto the grid to distribute their elements over the threads. As such, any node index in the algorithm arrays can be mapped to a thread index, so most of the computations are performed at the node level achieving per-thread parallelism rather than the whole individual level or per-warp parallelism. This mechanism facilitates high throughput memory transfers in the GPU and reduces sequential FOR... loops.

For example, the refining process includes different stages handled by a number of kernels. The process starts by reading the considered array (such as population or offspring) and replaces any duplicate nodes in every row with a flag (r_FLAG) to maintain the order of nodes in the solution. Another kernel searches for missing nodes in two steps: first, it adds the indices of the existing nodes in an auxiliary array, then, it derives the missing nodes from the auxiliary array. Every thread element in the auxiliary array simultaneously checks its value. A value of zero means that a customer node at this index in the data array is missing and is, therefore, added to the population array. This two-stage mechanism guarantees
Figure 11: Flowchart of the coarse-grained PGA on $n$ GPUs
a high-speed execution as it avoids running searching algorithms over massive arrays. The capacity constraints are checked by another kernel that inserts zeros, representing the depot return, wherever the accumulated demand exceeds the truck capacity.

A comparison is performed between a per-warp GPU implementation (Algorithm 3) and the per-thread GPU implementation in Algorithm 4 for finding the missing nodes. In Algorithm 3, each thread row iterates through the entire row, comparing the nodes with the node list from the problem data. Although all the rows are processed in parallel, two sequential loops still consume a significant portion of the execution time. In Algorithm 4, on the other hand, each thread checks for the node at the corresponding index and adds it to an auxiliary array holding the existing nodes. Indices of missing nodes at the auxiliary array have a value of zero. The column IDs of the nodes are used to derive a list of non-existing nodes. Interested readers can find more details in the code at (https://github.com/MarwanAbdelatti/GA_VRP_mGPU.git). The two algorithms have been applied to a CVRP benchmark from [47] with 219 nodes and have been executed 1,000 times; subsequently, statistics have been collected. The per-warp algorithm took an average of 47.30 ms ± 23.4811 ms to execute, whereas the per-thread algorithm took an average of 1.358 ms ± 0.8 ms which is almost 35 times faster.

3.5 Results And Discussions

The execution speed of the algorithm in secs/generation is reported on selected benchmark problems taken from [47, 48, 49, 50] with sizes ranging from 420 and 20,000 nodes. The GA parameters are taken from the design of experiment (DOE) performed in [11] – they may not be optimal for this particular setting. The population size is maintained at 20× the number of nodes n, the inverse mutation
Algorithm 3 Finding missing nodes per-warp algorithm

Input: missing_nodes
1: Initialize Blocks number and Threads per Block
2: Function find_missing_nodes(list_of_nodes, population)
3: For each thread row:
4: For every element in the list_of_nodes:
5: Traverse the population array to find the node.
6: If found:
7: Set missing_nodes[row, node_id] to zero.
8: Break.
9: Else:
10: Set missing_nodes[row, node_id] to node value. return (missing_nodes)

has a 0.3 probability, and a 1-point crossover has a 0.6 probability. Additionally, the migration rate where every GPU deme contributes with a portion of its population is $1/gpu\_count$ of its population. The migration interval is set to 1,000 generations between each exchange of data, and the demes are not synchronized at the time of the migration interval to avoid further delays during the execution. The algorithm was executed five times for different GPU arrangements for 5,000 generations on each benchmark problem. The average execution speeds are reported in Table 5 where the first column represents the instance name, followed by the number of customer nodes, and the secs/generation for different hardware arrangements are reported in the body of the table. The results show the dramatic speedups obtained by utilizing GPUs compared with a parallel CPU implementation (specifically, single instruction multiple data (SIMD) vectorization). It is also shown that the modifications and the improvements in this algorithm have improved its efficiency compared to the previous GPU implementation discussed in [11]. The current algorithm, as shown in Table 5, is able to handle a problem of 420 nodes at a speed of 0.78 secs/generation, whereas the older version handled a 400-node problem at 4.72 secs/generation on a single GPU. Additionally, the 8-GPU arrangement is found to have the highest capacity and the fastest execution.
Algorithm 4 Finding missing nodes per-thread algorithm

**Input:** auxiliary_arr

1: **Initialize** Blocks number and Threads per Block
2: **Function** reset_auxiliary_array(list_of_nodes, auxiliary_arr)
3: **For** each thread:
4:   Set index = thread index
5:   If index <= length of list_of_nodes then:
6:     Set auxiliary_arr[index] to 1.
7:   Else:
8:     Set auxiliary_arr[index] to 0. **return** (auxiliary_arr)
9: **Function** find_existing_nodes(auxiliary_arr, population)
10: **For** each thread:
11:   Set index = thread row index, thread column index
12:   Set auxiliary_arr[row index, node_ID] = population[index]. **return** (auxiliary_arr)
13: **Function** derive_missing_nodes(auxiliary_arr)
14: **For** each thread:
15:   Set index = thread index
16:   If auxiliary_arr[index] == 0 then:
17:     # Node ID is equal to column index +1.
18:     Set auxiliary_arr[index] to column index + 1.
19:   Else:
20:     Set auxiliary_arr[index] to 0. **return** (auxiliary_arr)

(bold-face text) compared to the other arrangements despite the additional “hops” discussed in Section 3.4.2 that are resulted from the indirect data transfer from GPUs 5, 6, and 7 to GPU 0 through GPUs 1, 2, and 3 respectively. These results support the premise that more GPUs increase the overall performance despite the additional hops that might occur. The profiling results reinforce this aspect.

**Remark 3.5.1.** It is necessary that both the GPU server and the CUDA version support the Peer-to-peer (P2P) memory copy, and all GPUs are connected by NVLink connections. Otherwise, a reduced performance may result when compared with a single-GPU algorithm.

We perform analytical profiling that shows the CUDA API summary; then, we track the effects of each kernel on the execution time. In order to inspect the effects
Table 5: Execution speeds in secs/generation.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Node Count</th>
<th>CPU</th>
<th>GPU arrangement</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-n420-k130.vrp</td>
<td>420</td>
<td>360</td>
<td>0.78</td>
<td>0.5</td>
<td>0.46</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td>Li_30.vrp</td>
<td>1,040</td>
<td>1,800</td>
<td>18.67</td>
<td>9.03</td>
<td>4.71</td>
<td>1.05</td>
<td></td>
</tr>
<tr>
<td>Li_31.vrp</td>
<td>1,120</td>
<td>X</td>
<td>12.18</td>
<td>6.11</td>
<td>3.18</td>
<td>1.22</td>
<td></td>
</tr>
<tr>
<td>Li_32.vrp</td>
<td>1,200</td>
<td>X</td>
<td>9.92</td>
<td>4.89</td>
<td>2.67</td>
<td>1.80</td>
<td></td>
</tr>
<tr>
<td>33.vrp</td>
<td>2,401</td>
<td>X</td>
<td>80.28</td>
<td>29.17</td>
<td>12.25</td>
<td>6.71</td>
<td></td>
</tr>
<tr>
<td>34.vrp</td>
<td>3,601</td>
<td>X</td>
<td>253.94</td>
<td>145.51</td>
<td>54.62</td>
<td>21.47</td>
<td></td>
</tr>
<tr>
<td>35.vrp</td>
<td>6,001</td>
<td>X</td>
<td>1,067.1</td>
<td>1,062.6</td>
<td>397.89</td>
<td>163.68</td>
<td></td>
</tr>
<tr>
<td>Ghent1.vrp</td>
<td>10,000</td>
<td>X</td>
<td>X*</td>
<td>X*</td>
<td>X*</td>
<td>X*</td>
<td>1103.8</td>
</tr>
<tr>
<td>Flanders1.vrp</td>
<td>20,000</td>
<td>X</td>
<td>X*</td>
<td>X*</td>
<td>X*</td>
<td>X*</td>
<td>2318.9</td>
</tr>
</tbody>
</table>

\(^a\) [47], \(^b\) [48], \(^c\) [49], \(^d\) [50]

* The problem is too big for this arrangement

of the hops, we detail the results of memory transfer, in terms of size and time, between the GPUs (DtoD) and between the GPUs and the CPU back and forth (HtoD, and DtoH). Aiming for this, we utilize NVIDIA® Nsight™ Systems profiling tool. The algorithm profiling results are divided into four parts and illustrated by Figures 12 – 15. Figure 12 summarizes the execution profile of the standard CUDA API library in the algorithm; the figure reports only the top five functions that consume the bulk of the execution time. The rest require less than 0.1% of time and have been removed from the figure. The function cuStreamSynchronize(), highlighted in red, consumes 99% of the reported time. However, this function is executed on the host CPU, and this time is concurrent to the code running on the GPU and should not be added to the GPU time [51]. It is worth noting that the cudaMemcpyPeer() function responsible for transferring data between the GPUs — highlighted in blue — does not consume much time during the execution (~0.1%) or (~1.12%) excluding the cuStreamSynchronize() time.

The kernel profiling report, in Figure 13, shows the five major kernels con-
assuming most of the GPU time; these are: \texttt{find\_duplicates()}, \texttt{twoOpt()}, \texttt{add\_missing\_nodes()}, \texttt{shift\_r\_flag()}, and \texttt{cap\_adjust()}. These kernels involve per-warp (sequential) operations on the GPU, which consume more time in executive loops than per-thread kernels that consume less than 0.1% each and have been removed from the figure. This part of the profiling report also shows that the kernels responsible for exchanging data between the GPUs are not in the top five time-consuming kernels.

The the time and size of memory transfers are shown in Figures 14 and 15, respectively. Figure 14 shows that the time consumed by transfers of data from the GPU (device) to the CPU (host) is 69.5% which is much greater than those between the GPUs (22.2%) including the hops. This is a large difference despite the large size of the data transferred between the GPUs (2,540.21 GB DtoD) compared to those transferred between the CPU and GPU back and forth (25.4 GB DtoH, and 25.39 GB HtoD), as shown in Figure 15. This is because of the high bandwidth of the NVLink between the GPUs compared to its PCIe counterpart and the less frequent transfers between the GPUs (156,100 DtoD instance counts) compared to the transfers to the CPU (11,251,871 + 55 instance counts), which are needed for logging the execution progress. Although these memory numbers look huge, they consume only 1.12% of the execution time, as shown in Figure 12.
The quality of parallelization is measured by its efficiency, $\epsilon_N = T_1 / (N * T_N)$ where $T_1$ and $T_N$ are the execution secs/generation for one and $N$ GPUs, respectively, and $N = 8$ is the number of GPUs in our case. The obtained efficiencies are: 24.38%, 222.26%, 124.80%, 69.03%, 149.47%, 147.87%, and 81.49% for the problems with 420, 1,040, 1,120, 1,200, 2,401, 3,601, and 6,001 nodes, respectively. Despite the superlinear speedup obtained in the majority of the problems, the large variance in $\epsilon_8$ is related to the different nature of each problem set that directly affect the cache performance and the communication overhead [52].

The solution quality is another key measure of algorithm efficiency. In this paper, we do not focus on the final quality of the solution but examine the effect of multi-GPU parallelism on the performance of the GA only. Additional computational steps for improving the VRP solutions by clustering and pruning are not incorporated. To demonstrate the ability of the algorithm to find good solutions eventually, it is executed on the eight GPUs for 100,000 generations for the X-n420-k130.vrp problem. The obtained gap is 4.98% compared with the best-known solution in the literature where: $\text{gap} = (\text{obtained} - \text{best}) / \text{best}$. 

\[ \text{gap} = (\text{obtained} - \text{best}) / \text{best} \]
3.6 Conclusions

This work utilizes an HPC of GPUs for a real-world problem that directly impacts logistics operations. A parallel genetic algorithm is incorporated with a 2-opt local search onto a multi-GPU server for large-scale CVRP. The proposed algorithm adopts the per-thread approach to fully utilize the GPU cores for maximum parallelism. The algorithm performance is tested on different hardware arrangements (1, 2, 4, and 8 GPUs) and a CPU. The results show that utilizing multiple GPUs for the developed algorithm has significant improvements over CPU or single-GPU utilization. Our contributions in this work are in the following aspects: (i) larger problem sizes up to 20,000 nodes can be handled, (ii) the execution time is reduced over the CPU by a factor of 1,700; and (iii) for the range tested, the performance increases monotonically with the number of GPUs. To demonstrate the effectiveness of the proposed algorithm, we run the algorithm for 100,000 generations on a 420-node problem, and a solution gap of 4.98% from the best-known solution is obtained. The migration mechanism reduces premature convergence and improves the quality of solutions by exchanging high-quality individuals between the processing nodes. Future work will include improving the algorithm to execute on multi-GPU clusters with varying connection topologies.

List of References


“Nvidia dgx-1 with tesla v100 system architecture,” http://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf.


MANUSCRIPT 4

Lab-scale Smart Factory Implementation Using ROS

Marwan F. Abdelatti1*, and Manbir S. Sodhi1

1Department of Industrial Engineering, University of Rhode Island, Kingston, Rhode Island, 02881

*Corresponding author: mabdelrazik@uri.edu

Published as a chapter in the Robot Operating System (ROS): The Complete Reference (Volume 7 – 2022) Springer.

ACM ISBN XXX-X-XXXX-XXXX-X/XX/XX

https://doi.org/XX.XXX/XXXXXX.XXXXXX
Abstract

The central concepts underlying Industry 4.0 are the inter-connectivity between smart machines, production equipment, storage systems, algorithms with autonomous data exchange, action triggers, and, autonomous process controls. The goal of Industry 4.0 is to improve monitoring and control of manufacturing processes and systems, leading to high-quality parts, lower inventory, and reduced throughput times. However, the limitations and incompatibilities of legacy systems present significant challenges for the introduction of Industry 4.0. This use case chapter describes the implementation of a fully functioning, lab-scale, Industry 4.0 smart factory to demonstrate how legacy systems can be upgraded without major infrastructure changes. Using existing and proven frameworks and protocols such as robot operating system (ROS), and telemetry transport protocol (MQTT), Industry 4.0 features can be introduced to legacy systems by adding a single layer of hardware. Additionally, a supervisory control and data acquisition (SCADA) system is integrated with the IIoT to monitor and control the entire system processes and to provide analytical reports from the collected data. By utilizing ROS, the system has standardized functionalities including unified high-speed communication, hardware interfacing, and multi-thread handling, to name a few. We present the ft_industry4_raspi ROS Melodic package that handles all the distributed nodes over the multiple controllers in the factory. The reconfigured system is upgraded to demonstrate Smart Manufacturing, Cybersecurity, and Cloud-based Manufacturing. This system can function as a testbed for evaluating Cybersecurity, Cyber-Physical Systems concepts, and new IIoT hardware and SCADA software, as well as for embedding a variety of operations control approaches.

Keywords: Industry 4.0, Internet of Things, ROS, SCADA, Smart factory, Embedded systems
4.1 Introduction

The factory of the future, as conceptualized in Industry 4.0, is an integrated system where production machines, material handling systems, part input and output processes, and humans interact within a framework of distributed and centralized controls [1, 2]. Machines exchange parts and data autonomously and algorithms act on this data to control processes and part movements. The data also gives insight into the condition of machines, and maintenance is invoked in a predictive manner. Much of this data is generated by components of the Internet of Things (IoT) [3, 4]. One of the challenges in implementing an Industry 4.0 system is assembling a collection of heterogeneous devices in a common framework. Using the architecture of Robot Operating System (ROS), it is possible to integrate the communications of a variety of production devices. In this application, ROS has the advantage that many of the material handling devices such as robots and other material transfer devices such as AGVs may already have been set up to operate with ROS, facilitating their integration expeditiously.

Small-scale factories or simplified physical models have been commonly used in research involving operations research and maintenance. The main purpose of using such systems is to generate representative data for modeling and testing since it is difficult to get this data from industrial settings by disrupting production, and furthermore, when available, it is not publicly accessible [5]. This cyber-physical approach has many benefits over using a virtual model of a factory or machine since it allows training in real environments and allows replication of details related to industrial practice that can be difficult to simulate virtually [6]. Another advantage of using lab-scale implementations is that these can be built using low cost, non-industrial components. There are several platforms available nowadays including the Smart-LEGO Factory, the Fischertechnik plant model for Industry 4.0 [7] a
Fischertechnik punching workstation [8], the Festo Learning Factory [9] to name a few.

There is an increasing interest in the use of small-scale factories, especially for exploring Industry 4.0 concepts. [10] show how a learning factory can be used in an engineering course on reconfigurable manufacturing in a blended and problem-based learning approach. [11] present an in-house developed learning factory to meet various manufacturing requirements and proving their value in a production environment. A mechanism of how a generic client can access data generated from a workstation has been demonstrated in [8]. Three application cases from business process management (BPM) have been used in [12] to address challenges in Industry 4.0 using DFKI Smart Lego Factory. A physical Fischertechnik factory model equipped with several sensors has been introduced in [5] as an approach for the generation of predictive maintenance data. The development of an ontology to represent a Fischertechnik manufacturing simulation model by reusing existing ontologies has been investigated in [13]. A fully automated factory prototype built out of LEGO bricks has been introduced in [14] to demonstrate artificial intelligence applications in Industry 4.0. However, these all use proprietary software that makes the process of expanding the factory challenging.

In this chapter, we present a laboratory-scale smart factory that can be assembled with off-the-shelf components and IoT devices. The system has been created to give students a facility to learn about Industry 4.0 concepts without the expense, complexity and risk of working in life-scale installations. Another intended use of this system is as a sand-box for testing different frameworks, such as ROS, with a variety of Manufacturing Execution Systems (MESs) to detect performance issues at low expense. A video showing the system in action can be viewed at: https://youtu.be/mH5j1zI3Shw. Our contributions in this work can be listed in
the following aspects:

- We introduce the “University of Rhode Island Lab-scale Factory for Industry 4.0 Kit” (i.e., URILF4.0 for short) as an open-source system for manufacturing research. All proprietary interconnections including hardware, software, and protocols have been replaced by open-source components.

- We develop the ft_industry4_raspi ROS package to handle the industrial processes of the factory. This will give the learners the opportunity to get practical training on ROS for industrial applications.

- We efficiently developed a SCADA system to monitor and control industrial processes by mapping between MQTT and ROS topics.

This chapter is organized as follows, a detailed description of the factory kit is provided in Section 4.2 where various functioning modes are explained with diagrams and flowcharts. Section 4.4 discusses the proposed ROS package and details its key files and environment configurations, and other software architecture. The ft_industry4_raspi system is tested in Section 4.6. Section 4.5 discusses the SCADA system concept and the use of the bridge package to map between ROS and MQTT topics. Section 4.7 concludes this work with some remarks on possible future developments.

4.2 The Smart Factory Kit

The URILF4.0 is a physical simulation of a “smart factory” designed to familiarize students with Industry 4.0 concepts and to serve as a test-bed for exploration and experimentation. The current configuration uses the basic modules purchased from Fischertechnik as part of the Industry 4.0 Learning Factory, but all proprietary interconnections including hardware, software and protocols have
been replaced by open components. URILF4.0 simulates the storing of raw materials, as well as product manufacturing and delivery according to the orders placed by customers. Figure 16 shows the different components of the kit, it consists of factory modules like automatic storage and retrieval station (ASRS), vacuum gripper robot (VGR), high-bay warehouse (HBW), multi-processing station with kiln (MPS), and a sorting line with color detection (SLD).

![Figure 16: The smart factory kit](image)

URILF4.0 supports two main processes: (i) Storing new (incoming) materials. When a part arrives at the pickup location, the VGR picks it and transports it to the ASRS. The ASRS checks for the first free location in the HBW and places the material accordingly. This is shown in Figure 17. (ii) Processing new orders, as documented by Figure 18. This process is activated by receiving an order from the SCADA system representing a customer request. The customer may request a part with one of three colors: red, white, or blue within a unique processing sequence. The ASRS checks for the first available material location in the HBW and moves to it then places the part where the VGR can pick it and transport it to the MPS. The material moves through the MPS and passes along a conveyor
belt to the sorting line where it is processed according to its color. Finally, the VGR picks up the finished product and places it into the delivery outlet where it is ready for customer pick up. The discussion below details the functionalities of each unit component in the kit:

**High-Bay Warehouse**

As shown in the flowchart in Figure 19, there are two different processes associated with the HBW, namely, new material reception and new order processing. With regards to new material reception, an inventory piece is moved from the vacuum gripper robot (VGR) into storage, whereas in the new order process an inventory piece is moved out of storage and back to the custody of the vacuum gripper robot. It is worth mentioning that the high-bay warehouse (HBW) communicates both location and inventory color information for access by both the SCADA and the VGR as will be detailed in the ROS section of this chapter.
The vacuum gripper robot (VGR) is equipped with a suction end to pick and move materials. The role of the VGR is to transport the materials between the different stations and can reach any station in the system. Figure 20 shows the flowchart for the actions performed by the VGR for the two main processes in the system. When a new part arrives at the pickup inlet, the photosensor sends a triggering flag to the VGR to initiate a pickup and subsequent placement at the color detection unit where the color data is created and published along with a triggering flag to the ASRS. The ASRS in turn reads HBW status then moves and picks up an empty box which is transported to the HBW conveyor belt to the VGR dropping side. Meanwhile, the VGR moves the material towards the HBW conveyor belt and waits for the arrival of the empty box where the VGR drops it.

The second process, which defines how parts are processed in the system, starts by placing a new order through the SCADA system. The customer can indicate the part color and the number of parts desired. The VGR moves to the HBW conveyor while the ASRS moves to the appropriate location at the HBW to pick a material and place it on the conveyor belt. The VGR then picks the
material and transports it to the multi-processing station (MPS). The VGR waits at its home location until the material processing is completed and the sorting line (SLD) moves the material to its pickup location based on its color. Finally, the VGR picks the material from SLD to its delivery outlet.

**Multi-Processing Station**

The Multi-Processing Station (MPS) includes multiple processing units like klin, and a milling machine. These units communicate through I2C protocol to simulate a legacy system. Various conveying techniques are used such as a conveyor belt, a turntable, and a vacuum suction gripper.

**Sorting Line Station**

The sorting line (SLD) is responsible for sorting the completed parts by color. Its function lies in the new order process only where it is the last station before the finished product is picked up by the (VGR) and marked as ready for delivery. The flowchart in Figure 21 illustrates the SLD process which is activated once a finished material is dropped from the MPS station. A conveyor belt moves the material to a color detecting unit then one of three pneumatic valves is activated,
Figure 21: The SLD flowchart

according to the detected color, moving the processed part onto the corresponding chute. A trigger signal accompanied by the color is published to activate the VGR to transport the finished piece to the delivery outlet.

4.3 Legacy System Upgrade

The original Industry 4.0 kit is available from Fischertechnik company with multiple options such as basic running voltages of 9v or 24v, proprietary TXT control units or PLCs etc. Technical details about their product and the different purchasing options are available on their website: [https://www.fischertechnik.de/en/products/simulating/training-models](https://www.fischertechnik.de/en/products/simulating/training-models). The base kit used in this project is the 9v kit with TXT controllers. It was delivered pre-assembled and was equipped with proprietary controllers, and software accessible through the company’s website. To increase the flexibility and to support expansion of the system using hardware from different suppliers without compatibility issues, the controllers were replaced by open-source models. We chose Arduinos and Raspberry Pis because of the large established user community, relatively low cost components, and free and broad ranges of libraries of codes. Furthermore, there is increasing interest in using ruggedized versions of these controllers in factory settings. Since the original system utilizes separate controllers for each processing unit, we follow the same
principle in our upgrade and expand it into a distributed system that can: (1) provide the system with reliability and high fault tolerance so that a crash on one unit does not affect the others; (2) enable a scalability level so that more machines can be added in the future with a flexibility to install, implement and debug new services; and (3) reduce the computational load if central PCs are used.

To further support the open-source idea and facilitate reproducibility of the proposed system, ROS is adopted as the basic integration tool. Along with being general and new parts can be added easily in the system, ROS can be implemented with sparse storage and computational resources which makes it suitable for use on embedded computers. Additionally, there is a large library of community developed packages. For example, instead of writing an exclusive master-slave program to interface between the Raspberry Pi’s and Arduino boards in our system, the rosserial package is used for the task, which reduces development and maintenance effort. Moreover, ROS is language agnostic which means that the system can run with parts of codes written in different languages, increasing the scalability of the system.

In order to support the idea of adding a single layer of software and hardware to the pre-built legacy system, the SCADA software interface was retained in its original condition without altering the communication protocol supporting it. This reduces the modifications needed to the original system. Since the SCADA software being used in the system uses the MQTT protocol and does not support ROS messaging, the ROS bridge package is used to map between the two protocols as explained in the sequel. MQTT itself was not utilized to interface to the embedded computers or in the sensor/actuator layer for the same reasons as discussed above. As illustrated by Figure 22, the Arduino controller is used to perform low-level control such as reading inputs from sensors and sending
outputs to motors and pneumatic valves, and handling I2C communication to other machine controllers. Whereas the Raspberry Pi takes care of the WiFi communication and integrates Arduino controllers in the network loop through USB connectivity and the rosserial package.

rosserial is a communication protocol to transfer data through a serial interface or network socket [15]. It's based on a client-server model where a rosserial service runs on a computer running ROS (the Raspberry Pi in our implementation) and a rosserial client on the Arduino side. Data is transferred in the form of ROS messages between them. rosserial_client package is available for several microcontroller types including Arduino, STM32, Raspberry Pi to name a few. Rosserial packages are available for both Python or C++.

4.3.1 ROS components and network

Since our system mimics a real factory with multiple machines involved, the proposed ROS framework is based on a distributed package concept. ROS nodes are run on different machines and communicate over a network. Every machine is responsible for specific tasks, as explained earlier and, therefore, they respond to different ROS topics. The system has one PC functioning as a Human Machine Interface (HMI) with the SCADA system running on it.

Raspberry Pi 3 B+ boards were selected to handle the ROS communication between the local Arduino board as well as with the ROS core over WiFi. The
1.4 GHz processor, 1 GB RAM, and the installed Linux operating system (Ubuntu 18.04 server) were sufficient for this project. Four processing stations were interfaced to the Raspberry Pi’s: three of them, called “unit Pis”, take care of the ROS nodes of the factory units and necessary packages, and the fourth, called “main Pi”, hosts the ROS core, MQTT server, and other configuration files necessary to the system. To be more specific, each unit Pi runs rosserial_server package to communicate with its Arduino board, and the proposed ft_industry4_raspi package to handle the factory tasks. In addition to these packages, the main Pi runs the ROS core, and the mqtt_bridge package to map between the ROS topics and the MQTT topics as detailed in the following sections as well as the Mosquitto MQTT server. The HMI PC has a SCADA software installed. The PC does not directly interact with ROS since the SCADA software supports MQTT rather than ROS, the mapping performed by the mqtt_bridge package installed on the main Pi is responsible for this PC interface.

The remainder of basic control operations that require direct physical link to sensors or actuator drivers like color detection, robot manipulation, and servo motor and solenoid control are performed on the Arduino boards. The board used is an Arduino-Leonardo compatible board called ftDuino. It has built in motor drivers, eight universal inputs that can be utilized for analog or digital readings, four counter inputs, and eight outputs in addition to I2C, and USB connections. More information about the board can be found in this link: https://harbaum.github.io/ftduino/www/de/. It fits perfectly into the Arduino ecosystem for handy development, as discussed in Section 4.3.2, but the lack of network connectivity in ftDuino was one of the motivators behind using Raspberry Pi as another board to grant network access. The communication network uses a router and all the devices are connected via WiFi. The formed network
consists of one PC, four Raspberry Pis 3 B+, and six ftDuinos. Every ftDuino is connected to one Raspberry Pi for ROS connectivity, the two additional ftDuinos are for the local Multi-processing station that rely on sensor based detection and I2C master-slave communication for operation. The network components are detailed in the architecture given in Figure 23 where the Raspberry Pis are connected to a WiFi router gateway with unique static IP addresses, the PC is remotely connected to the main Pi via internet. The main Pi functions as the ROS master that handles the communications between the factory components on one hand, and on the other hand acts like an MQTT broker that talks to the HMI PC to present data on the SCADA. The \texttt{mqtt\_bridge} package on the main Pi simultaneously maps between the ROS messages and the MQTT messages in a bidirectional form. The MQTT communication is done through port 1883 since encrypted messages are not handled in this version of the system.

4.3.2 ftDuino Integration into the Arduino IDE

In order to access the ftDuino through the Arduino IDE, the following steps need to be followed: after connecting the ftDuino to the PC where the Arduino IDE is installed, the ftDuino board must be installed in the IDE. At the \texttt{File \rightarrow Preferences} menu, the following URL is added in the “Additional Boards Manager URLs:” field:
https://harbaum.github.io/ftduino/package_ftduino_index.json. Then press “OK” as shown in the screenshot in Figure 24.

Once the link is provided, the board package can be installed from the Boards Manager at Tools -> Board: ... -> Boards Manager as shown in Figure 25. The ftDuino can now be selected by choosing ftDuino from Tools -> Board: ... -> ftDuino Boards as illustrated in Figure 26. If the board is already connected to the PC, it can now be selected under Tools -> Port as in Figure 27.
4.4 ROS environment configuration

The ROS version used for this project is ROS Melodic on Ubuntu 18.04 server connected to four ROS machines that are running on a network with the same ROS master. All necessary files including the configuration, and the nodes as well as the launch files are available in the package repository in the following link: https://github.com/MarwanAbdelatti/ft_industry4_raspi. The package should be cloned inside the src folder in the catkin_ws workspace. The src folder inside the package includes five folders for the four system machines as well as the “main Pi”. Some of the key files inside this package are discussed below:

- Every unit Pi has a shell file to configure its launch processes and remote environment, namely, env_hbw.sh, env_vgr.sh, and env_sld.sh. The code below shows an example of these files:

```bash
#!/bin/bash

export ROS_MASTER_URI=http://192.168.0.10:11311
```
where the ROS master remote IP is defined in line 3 followed by the local host IP definition. The environment is configured in lines 6 and 7.

- **src/main/industry4.launch**: is the main launch file that initiates the nodes and sub-launch files for the system. The file can be divided into two parts: (1) the machine declaration, where the parameters of the "main Pi" and every "unit Pi" are defined; and (2) the node declaration, where every node required to run on the system is defined. The following code shows part of the file:

```xml
<launch>
  <!-- Declaring Machines -->
  <group>
    <machine
      name="main"
      address="192.168.0.10"
      env-loader="/home/ubuntu/catkin_ws/devel/setup.bash"
      default="true"
      user="ubuntu"
      password="RandomPassword">
    </machine>
  </group>
  <!-- Include Files -->
</launch>
```
In this file, we start with a `<group>` tag since we are using multiple launch files in the “main” machine. Then, we define the machine parameters using the `<machine>` tag where we provide the machine name, its local IP, the bash environment loader file, and the username and password for the ssh remote
connection. Although explicitly providing passwords in a readable file is a security risk, we provide it here for clarification and simplicity. A secure version of the launch file utilizing public key authentications is detailed in the text file https://github.com/MarwanAbdelatti/ft_industry4_raspi/blob/main/Secured-Approach-for-Launch-Files.md in the package repository link on Github. At line 14 in the file we currently discuss, another launch file is included. This file is responsible for launching the `mqtt_bridge` nodes that interface between MQTT and ROS messaging. The file includes node configurations, and the conversion map between ROS and MQTT topics which is available in the data file `ft_industry4.params.yaml`. Similarly, the other machines are declared such as the HBW in line 17 but there’s no `<group>` tag used here since there are no sub-launch files to include, the other parameters are similar like the IP address, the environment loader file, and so on.

The second part of this file is the node declaration where we launch the `rosserial_python` node on each machine using the `<node machine>` tag to enable the communication between each Raspberry Pi and the ftduino connected to it. The link is enabled on the USB port with ID `ttyACM0` and with communication speed 57600 bps. In addition to this, the `status_update` node is declared on the main Pi which is responsible for updating the HBW occupancy status of the shelves as well as the material color available. This data is saved in the `data.json` status file. The dots in lines 21 and 35 are to indicate that the file is clipped for the sake of space and that the clipped parts are similar.

- The rest of files contain Arduino codes (.ino files) that are meant to run on
the ftDuino’s of the units and contain the remainder of nodes that subscribe or publish to the different ROS topics in the system. The following is a sample of the VGR Arduino code:

```cpp
// Use the following line if you have a Leonardo, FtDuino or MKR1000

#define USE_USBCON

#include <ros.h>
#include <std_msgs/String.h>
#include <std_msgs/Bool.h>
#include <Ftduino.h>

ros::NodeHandle nh;
std_msgs::String str_msg;
std_msgs::String str_loc_msg;

ros::Publisher vgr_new_material("vgr_new_material", &str_msg);
ros::Publisher provide_empty_loc("provide_empty_loc", &str_loc_msg);

char empty_loc_content[3] = "Z4";
char detected_color[6] = "NA";
bool move_cmd = true;
bool in_processing_object = false;

void messageCb_red(const std_msgs::Bool &payload_msg){
  if (payload_msg.data && move_cmd){
    nh.loginfo("VGR Received an Order at the RED Topic");
    move_cmd = false;
    //move to Sorting Line Drop-off area (red)
    ftduino.motor_counter(Ftduino::M1, Ftduino::LEFT, Ftduino::MAX, 390);
    while(ftduino.motor_counter_active(Ftduino::M1)); /*no-op*/
  }
}
```
//pick up material (red)
ftduino.motor_counter(Ftduino::M3, Ftduino::LEFT, Ftduino::MAX, 450);
while(ftduino.motor_counter_active(Ftduino::M3)); /*no-op*/

ftduino.motor_counter(Ftduino::M2, Ftduino::LEFT, Ftduino::MAX, 850);
while(ftduino.motor_counter_active(Ftduino::M2)); /*no-op*/

ftduino.output_set(Ftduino::O7, Ftduino::HI, Ftduino::MAX);
ftduino.output_set(Ftduino::O8, Ftduino::HI, Ftduino::MAX);
delay(2000);

//move to end point

....

//drop off material at end point
digitalWrite(LED_BUILTIN, HIGH);
dropMaterial();
delay(2000);
digitalWrite(LED_BUILTIN, LOW);

//move back home
goHome();
}
}

ros::Subscriber<std_msgs::Bool> vgr_blue_sub("sld_blue_pub",
&messageCb_blue);
ros::Subscriber<std_msgs::Bool> vgr_red_sub("sld_red_pub",
&messageCb_red);
ros::Subscriber<std_msgs::Bool> vgr_white_sub("sld_white_pub",}
ros::Subscriber<std_msgs::String> new_order_sub("new_order",
&material_processing);
ros::Subscriber<std_msgs::String> provide_empty_loc_sub("provide_empty_loc_back", &empty_locCb);

void setup()
{
    nh.initNode();
    nh.subscribe(vgr_blue_sub);
    nh.subscribe(vgr_red_sub);
    nh.subscribe(vgr_white_sub);
    nh.subscribe(new_order_sub);
    nh.subscribe(provide_empty_loc_sub);

    nh.advertise(provide_empty_loc);
    nh.advertise(vgr_new_material);

    pinMode(LED_BUILTIN, OUTPUT);
    ftduino.init();
    //Vertical motor inputs
    ftduino.input_set_mode(Ftduino::I2, Ftduino::SWITCH);
    ftduino.counter_set_mode(Ftduino::C2, Ftduino::C_EDGE_RISING);
    //Horizontal motor inputs
    ftduino.input_set_mode(Ftduino::I3, Ftduino::SWITCH);
    ftduino.counter_set_mode(Ftduino::C3, Ftduino::C_EDGE_RISING);
    //turn encoder
    ftduino.input_set_mode(Ftduino::I1, Ftduino::SWITCH);
    ftduino.counter_set_mode(Ftduino::C1, Ftduino::C_EDGE_RISING);
    //in-processing sensor
    ftduino.input_set_mode(Ftduino::I7, Ftduino::SWITCH);
The first six lines are related to the rosserial_arduino package and enable the ftDuino to connect to ROS topics. Line 7 imports the ftDuino library to communicate with the IO ports to read sensors and control devices connected to it. Lines 9-14 declare the ROS node handle, message objects used in the code, and publisher objects to different topics. Some global variables are defined in lines 16-19. The messageCb_red() callback function is defined in lines 21-54 to be invoked when a red part is detected at the SLD chute, the dots in line 43 are to cut some text to save space. ROS subscriber objects are defined in lines 56-60 where the corresponding callback functions are tied to them. The node is initialized and ROS objects are connected to the topics by the setup() function declared in lines 62-92 where board-related I/O definitions are also defined. The main loop is defined in lines 94-104.
4.4.1 Prerequisite Packages

In order for the `ft_industry4_raspi` package to initiate and run successfully, some packages are necessary. The communication between each Raspberry Pi and the ftDuino board connected with requires the `rosserial` package to be available on each Raspberry Pi and ftDuino. To handle the communication between the main Pi and the SCADA HMI, an MQTT server and the `mqtt_bridge` package are required on the main Pi to exchange the messages between ROS and MQTT back and forth. Steps to properly install and configure these packages are summarized below:

The rosserial Package

is a metapackage that references multiple packages required to handle the communication over the USB port in our system. This package includes `rosserial_arduino`, `rosserial_python`, `rosserial_msgs`, and `rosserial_client`. More information can be found in ROS Wiki: [http://wiki.ros.org/rosserial](http://wiki.ros.org/rosserial). To install the package on ROS Melodic for Arduino-compatible boards, the following steps should be followed:

These two commands install the packages on each Raspberry Pi:

```
$ sudo apt-get install ros-melodic-rosserial-arduino
$ sudo apt-get install ros-melodic-rosserial
```

Now that the necessary libraries are installed on the Raspberry Pi, packages are built using `catkin_make` command. Header files and libraries necessary for Arduino programs to interact with ROS are built using the following commands:

```
$ cd ~/Arduino/libraries
$ rm -rf ros_lib
$ rosrun rosserial_arduino make_libraries.py
```

88
The goal of the first two commands is to remove any old ROS libraries installed within the libraries folder, which exists inside the installation folder of the Arduino IDE. Then, ROS libraries are built with the third command which requires a target location for the libraries, and is specified by the last (dot) in the line.

The mqtt_bridge Package

is a package that maps between ROS and MQTT messages in a bidirectional form. Messages from ROS are serialized by a messagepack json file for MQTT, and messages from MQTT are deserialized for ROS topic. More information is available in the ROS Wiki: http://wiki.ros.org/mqtt_bridge. The mqtt_bridge and its components are required on the main Pi only. First the MQTT broker (i.e., server) is installed:

```
$ sudo apt install mosquitto mosquitto-clients
```

Then the following commands install the rest of the prerequisites before installing the package:

```
$ sudo apt install python-rospkg
$ pip install rospkg
$ sudo apt install ros-melodic-rosbridge-library
```

The mqtt_bridge package is installed by the following command:

```
$ sudo apt install ros-melodic-mqtt-bridge
```

4.5 SCADA System

Supervisory Control and Data Acquisition (SCADA) system, as defined in [16], is:

"A sub-class of industrial control systems (ICSs) in which control is performed over multiple, distributed individual lower-level control systems (hence the word
“supervisory”.

A typical SCADA control center monitors and manages automation processes across remote areas through smart controllers such as remote terminal units (RTUs) and intelligent electronic devices (IEDs). The system status is presented on human-machine interface (HMI) that represents the system graphically and displays events and alarms lists. In addition to status monitoring, human operators can send control commands via the HMI software to different ICS components. Such control commands include but are not limited to changing a set point of a motor speed, start/stop a compressor, or open/close valves. In order for a SCADA software to communicate with the remote site, appropriate communication drivers must be available in the software. The SCADA software in our proposed system supports MQTT data transmission and functions as an MQTT client connecting to the MQTT broker located in the main Pi.

4.5.1 MQTT Protocol

According to the official MQTT specifications in the OASIS standard [17], MQTT is defined as:

"MQTT is a Client Server publish/subscribe messaging transport protocol. It is light weight, open, simple, and designed so as to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for communication in Machine to Machine (M2M) and Internet of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium."
MQTT was developed in 1999 by Andy Stanford-Clark (IBM) and Arlen Nipper (Arcom, now Cirrus Link) to connect to oil pipelines via satellite with minimal battery loss and minimal bandwidth. In contrast to client/server architecture where a client communicates directly with an endpoint (server) and the server responds back, MQTT adopts publish/subscribe architecture where there are three key components illustrated in Figure 28 forming the communication process, i.e. the publisher, the subscriber and the broker. The broker decouples the client (the publisher) that sends a message from the client or clients (the subscribers) that receive the messages. The publishers and subscribers never have a direct contact to each other. The broker filters all incoming messages and ensures they are correctly distributed to subscribers [18].

MQTT has been gaining popularity along with the proliferation of the Internet of Things (IoT) that needs the capability to work with low-powered devices. According to ISO/IEC 20922, MQTT is an ideal communication protocol in Machine to Machine (M2M) communications and within the IoT. The publish/subscribe architecture showed more efficiency than the conventional client/server model for industry 4.0 systems. Therefore, many systems are adopting MQTT. The publish-subscribe architecture along with the quality of service (QoS), scalability, message filtering, as well as the last will and testament (LWT) features give more flexibil-
ity to the connection of low-bandwidth devices with limited CPUs and with very little overhead [19]. MQTT has already been successfully implemented in many smart applications like home automation, surveillance, healthcare, transportation, industry, and logistics.

### 4.5.2 ROS-MQTT Bridge Configuration

As discussed in Section 4.4, the file `ft_industry4.params.yaml` includes the MQTT configurations, and the conversion map between ROS and MQTT topics. The contents of the file are shown in the listing below which includes the mapping between different ROS and MQTT topics in both directions.

```yaml
mqtt:
  client:
    protocol: 4 # MQTTv3.11
  connection:
    host: localhost
    port: 1883
    keepalive: 60
  private_path: device/001
  bridge:
    # new_order
    - factory: mqtt_bridge.bridge:MqttToRosBridge
      msg_type: std_msgs.msg:String
      topic_from: mq_new_order
      topic_to: /new_order
    # /vgr_new_material
    - factory: mqtt_bridge.bridge:RosToMqttBridge
      msg_type: std_msgs.msg:String
      topic_from: /vgr_new_material
```
The lines 1-8 contain the MQTT connection configuration for the bridge including the broker name (here is localhost since the bridge and the broker are on the same machine), and the port number. The mapping bridge configurations start at line 11, the first block of configuration serves the order placing feature from the HMI which sends the order to an MQTT topic called mq_new_order then the bridge forwards it to the ROS topic /new_order. Line 13 tells the bridge to map from MQTT to ROS. Line 14 specifies the message type based on ROS message types. Lines 15 and 16 specifies the source and destination topics respectively. The same applies to the blocks at lines 17-36 but the difference is that the mapping is in the other direction (i.e., from ROS to MQTT), the source and destination topics, as well as the message type for the last three topics (lines 22-36).
4.6 Test of the System

In this section, steps for properly running the system are provided. A graph with the system nodes and topics is given for better understanding how the system works and the node running sequence. Also, an application video is linked at the end of this section.

The system is initiated by running the nodes in related packages through a single `.launch` file that also contains other `.launch` files for easy operation. The launch file is run by the following command:

```
$ roslaunch ft_industry4_raspi industry4.launch
```

where `ft_industry4_raspi` is the package name and `industry4.launch` is the launch file. By running this command on the main Pi, the ROS master connects to the different unit Pi machines through `SSH` connections and uses the environment configuration `.sh` files as shown in the first part of Figure 29. The different roslaunch components including the nodes, machines, as well as the connection parameters of both `rosserial` and `mqtt_bridge` are listed in the summary part in the figure. The last part in the figure shows the successful initiation of all the nodes in the system indicating that the system is up and ready to work.

Although part of the system we present works using the client/server mechanism in the `rosserial` package, the system mainly depends on the pub/sub mechanism of ROS. Most the node communications are based on topics that show up on the `$ rostopic list` command once the nodes are run. These topics are:

```
/back
/diagnostics
/echo
/new_order
```
Figure 29: Screenshot of the ROS master initiation
While some topics are feedback topics that return some information to nodes for processing, others are command topics that are intended to directly initiate an action in the system (e.g., moving the robot). The ROS graph in Figure 30 depicts the system nodes (as ellipses) and their corresponding connections to the topics that are shown as rectangles. Now that the system is up and ready, we have two modes of operations as discussed earlier in Section 4.2, namely, (i) storing new part, and (ii) processing materials for new orders. Storing new material mode is automatically initiated once the system detects a new material at its pickup inlet by the photosensor connected to the ftDuino which runs the vgr_node. The vgr_node, as the graph shows, publishes an inquiring message to the /provide_empty_loc topic where the status_update node subscribes and checks the HBW shelf status i.e., data.json file for empty locations then returns this location back to the VGR through the /provide_empty_loc_back.

The VGR moves the material to the color detection station then publishes to the /vgr_new_material topic its color and the intended location to be placed in.
The **hbw_node** which subscribes to this topic moves its ASRS robot to pick the material from the VGR and places it accordingly. Once the new material is placed successfully, the node publishes to the `/what_is_occupied` topic the updated location and color for the **status_update** node to update the shelf status file. The typical `data.json` file looks like the following, where A1, A2,...,C3 are pre-defined locations at the HBW:

```json
{
    "A1":"white",
    "A2":"red",
    "A3":"blue",
    "B1":"white",
    "B2":"red",
    "B3":"blue",
    "C1":"white",
    "C2":"red",
    "C3":"blue"
}
```

The second operation mode, on the other hand, responds to new order request from customers. The mode is initiated by the following command from terminal or by the SCADA HMI:

```
$ rosrun ft_industry4_ftduino place_order.py red
```

The **rosrun** command runs the **place_order** node which takes the requested color.
as an argument, searches for an HBW location having a material with the corresponding color, then publishes this location to the topic /new_order as shown in the graph of Figure 30. Both the **hbw_node** and **vgr_node** subscribe to this topic and respond to it at the same time such that the HBW ASRS robot moves to the target location, picks the material, and places it where the VGR robot can pick. The ASRS brings the empty box back to the HBW shelf and the the **hbw_node** node publishes to the /what_is_empty topic the empty location for the **status_update** node to update the shelf status file. Meanwhile, the VGR moves the material to the MPS station for processing and returns back home listening to the topics: /sls_blue_pub, /sls_red_pub, and /sls_white_pub. Once the material processing is done, it’s kept at one of the three SLD locations based on its color where the **sld_node** publishes to one of the three above-mentioned topics so that the VGR can pick and place it at the delivery outlet.

In order for the SCADA to monitor the processes properly, the ROS node of name **mqtt_bridge** subscribes to relevant topics namely, /vgr_new_material, /sls_blue_pub, /sls_red_pub, and /sls_white_pub and publishes to /new_order as illustrated with green connections in Figure 30. Future work will include other topics that provide manual override to VGR and ASRS robots to forcefully send them to their home positions in case of emergencies.

4.7 Conclusion

URILF4.0 is designed as a laboratory-scale manufacturing test-bed for exploring and experimenting with Industry 4.0 concepts. It is built with low cost, open-source controllers and software. The sensors interface to Arduinos, connected with Raspberry Pi 3 B+’s, which provide network communications and serve as ROS nodes. An additional Pi is used as the ROS master node. The ROS network is connected to a SCADA system and a widely used proto-
col, MQTT, is used to link SCADA and ROS. The SCADA gives full access to the low level hardware and enables user inputs to initiate (and process) orders through the manufacturing test-bed and to receive and store materials in an ASRS. Work on expanding the manufacturing test-bed is ongoing. A digital twin of this system is also under development. The source code has been shared at https://github.com/MarwanAbdelatti/ft_industry4_raspi, and the authors can also be contacted directly for any additional information.

**Authors Biographies**

**Marwan F. Abdelatti** received the M.Sc. degree in Control Engineering from Cairo University, Egypt in 2010. He is currently a Ph.D candidate in the department of Industrial Engineering at the University of Rhode Island, USA. His current research focuses on smart manufacturing system, intelligent agents, and parallel GPU computing. Marwan is involved in research projects including GPU implementations of optimization algorithms for route-planning problems, and autonomous robots, specifically, ground vehicles.

**Manbir S. Sodhi** obtained his undergraduate degree from Jadavpur University, Kolkata, and graduate degrees from the University of Arizona. He is the Professor of Industrial and Manufacturing Engineering at the University of Rhode Island. He has published many engineering research articles involving manufacturing, sustainability, driving distractions and autonomy. For over a decade, he has hosted the Global Product Sustainability Systems Meeting in Rhode Island. He has also worked as a visiting scientist with NATO in Italy, and a visiting Professor at the Technical University of Braunschweig, Germany.
List of References


An Intelligent Anomaly Detection System for Industrial Internet of Things

Marwan F. Abdelatti

Department of Industrial Engineering, University of Rhode Island, Kingston, Rhode Island, 02881

*Corresponding author: mabdelrazik@uri.edu

Drafted for submission, XXX-XXX.

ACM ISBN XXX-X-XXXX-XXXX-X/XX/XX

https://doi.org/XX.XXX/XXXXXXX.XXXXXX
Abstract

The industrial internet of things (IIoT) aims to improve the controllability of industrial systems by exchanging vast amounts of information between different components. However, due to the broadly distributed and entangled nature of the IIoT communication networks, most existing industrial structures are vulnerable to serious security issues. Therefore, finding anomaly detection solutions is needed as a critical post-attack process in IIoT cybersecurity. In this work, we follow a standardized, systematic literature review methodology for IIoT cybersecurity research done in the last decade with more focus on anomaly detection approaches. We summarize our review outcome using advanced illustration techniques and show the correlation between different IIoT cybersecurity research fields. Then, we study multiple supervised machine learning (ML) algorithms and develop anomaly detection systems that help decision-makers early detect potential attacks and take the necessary responses. To be more specific, we train five ML models on a commonly used application-layer dataset existing in the literature, namely DS2OS. We analyze and evaluate the performance by standard metrics. This work represents a basic core to our future work in designing a new dataset and an intelligent machine-learning anomaly detection system for our smart manufacturing system.

Keywords: cyberattacks, anomaly detection, machine learning, industrial internet-of-things

5.1 Introduction

Modern industrial control systems (ICS) nowadays are capitalizing on the valuable features of the fourth industrial revolution (Industry 4.0) and the internet of things (IoT) to efficiently leverage their performance. Such new systems are called the industrial internet of things (IIoT) and are characterized by gaining remote connectivity and more autonomous decision-making over human intervention.
These IIoT features, achieved by the elevated use of smart sensors and actuators [1], are of great interest in research and commercial circles in the last decade with a global forecast projected around $950 billion by 2025 [2]. However, the huge amount of data transfer in the new technology demands a high level of security against cyberattacks that increasingly jeopardize the entire process.

In this article, we study the performance of standard machine learning (ML) algorithms to predict anomalies in typical IoT systems. The dataset we use is presented in [3], where data from three smart homes and a smart office was recorded during normal behavior and under seven different types of attacks. The historical sensory data is collected from the application layer, where the smart controllers send and receive communication requests by smart sensors and actuators integrated into the system. The ML models are meant to learn the normal behavioral patterns of system processes and the possible patterns of different attacks. Although this procedure seems simple so that any simple database system can do it, the many devices integrated into it, and the high permutations between different components make it more sophisticated.

The remainder of the paper is organized as follows: a literature review of cybersecurity of IIoT with a focus on anomaly detection is discussed in Section 5.2. The benchmark DS2OS dataset is explained in Section 5.3, where we discuss the main features and their meaning, as well as the procedures we followed to prepare the dataset for training and testing on the models. The algorithms and the metrics considered to evaluate them are discussed in Section 5.4. Experimental results are reported and discussed in Section 5.5 followed by a conclusion driven in Section 5.6.

5.2 Literature Review

Cybersecurity may be a weak link that could obstruct capitalizing the full potential of industry 4.0. Many research efforts have been reported in attempts to
find a reliable security framework in the race toward IIoT process automation. In order to study the tendency of recent research to IIoT security, we surveyed around 500 recent works in the literature and developed a Sankey graph to illustrate the analysis. The survey strategy we follow is similar to the procedure adopted in [4] as shown in Figure 31. While we follow the four main steps of the systematic literature review, part of the activities in each stage are followed to fit the need of this work.

As a consequence, we identify our objectives and the selection criteria through the review planning and search execution steps:

1. **Definition of analysis**: we aim to find published works about cybersecurity for industrial IoT systems in the last decade and perform analysis and evaluation.

2. **Selection of sources of information**: the research process was realized using two of the most comprehensive sources of publication databases, namely, Scopus ([www.scopus.com](http://www.scopus.com)) and Google Scholars ([scholars.google.com](http://scholars.google.com)).

3. **Selection of search keywords and queries**: we base our search on the following keywords: “cybersecurity”, “information security”, “industrial internet of things”, “IIoT”, “Industry 4.0”, “smart manufacturing”, and “smart fac-
Consequently, the following search query was established: (cybersecurity OR security) AND (“industrial internet of things” OR IIoT OR “Industry 4.0” OR “smart manufacturing” OR “smart factory”). These search criteria resulted in 538 published or approved-to-publish articles as of July 2022.

4. **Filtering and selection**: these articles were then filtered so that only English documents were selected, and poster presentations were excluded.

5. **Applying qualitative mechanisms for content selection**: through skimming the abstract of each article, some have been filtered out due to poor quality or irrelevant research objectives.

As a result of the above procedures, we create a Sankey diagram that summarizes the relevant research articles we obtained and sorts them into 30 topics. Sankey diagrams help represent quantitative results and visualize their correlations in an input/output form, as we found many articles discussing multiple topics inside. Figure 32 shows the Sankey diagram of the survey results where 30 categories and subcategories are represented by nodes of a specific color with sizes representing the number of reported articles. The figure can be seen in three areas; the left area shows six main categories found as the core idea in the reported articles: data security, authentication, anomaly detection, resource management, data sets/testbeds, and surveys. The middle area represents subcategories as a branch category from the main one or an additional feature commonly found within. This area includes 14 subcategories: data storage, authorization, encryption algorithms, data sharing, threat analysis, behavioral anomaly, intrusion detection, security framework, threat analysis, security framework, asset management, computing reservation, network resources, and testbeds. The right side of the figure
shows ten main techniques used for the adopted approaches, such as artificial intel-
ligence (AI)/neural networks (NN)-based algorithms, post-quantum or quantum
cryptography, parallel GPU acceleration, and blockchain, to name a few. The links
between the nodes indicate the number of articles discussing topics from the figure.
Figure 32: Sankey diagram of reported literature
It is shown from the figure that the top three main topics explored in the literature are data security, including data storage techniques, data sharing methods, and data encryption (138 publications). Surveys are done in 123 publications. Anomaly detection schemes, our focus of research, are discussed in 81 publications; 17 involved AI or NN algorithms approaches, and six used blockchain techniques at the network and the application layers. Fourteen publications discussed behavioral anomalies on the application layer. The intrusion detection subcategory has been investigated in 23 publications, where one of these articles involved evolutionary algorithms (EA) and heuristic techniques. Anomaly detection through network traffic analysis has been discussed in 11 publications, nine introduced security frameworks, and one discussed pre-attack threat analysis for anomaly detection. Figure 33 shows a pie chart of the publications related to anomaly detection. As the figure shows, many different anomaly detection techniques are presented in the literature. However, we focus on techniques based on machine learning (ML) [5, 6, 7, 8, 9, 10, 11] and deep learning (DL) [12, 13, 14]. These techniques can be classified into supervised [5, 6, 7], semi-supervised [8, 9], and unsupervised categories [10, 11].

Figure 33: Publications about anomaly detection

A supervised hybrid system of machine learning models and a genetic algo-
Algorithm (GA) operating on the network layer has been introduced in [5] and was tested on the UNSW-NB15 IIoT dataset. The algorithm relies basically on the GA to select the best combination of features (feature vector) that achieves a specific level of accuracy. The GA's fitness function is the accuracy obtained by different ML models like Linear Regression (LR), Naïve Bayes (NB), Decision Tree (DT), Extra-Trees (ET), and Extreme Gradient Boosting (XGB). The resulting system was compared to other developed approaches in the literature like [15, 16]. However, this algorithm takes a long time to execute since the ML model splits, trains, validates, and tests the given dataset once again for every generation in the GA. An explainable AI (XAI) model was introduced in [6] with a case study on IIoT security. The algorithm was tested on three datasets [17, 18, 19] and showed a good level of accuracy, but it had overfitting issues [6]. The authors in [7] have proposed a framework for data collection and attacks detection at the network layer using ML techniques and shallow neural networks (NN) and applied it on KDDCup'99 and DARPA29F datasets. However, the detection algorithm was computationally expensive as parallel computations were not utilized [7].

A semi-supervised learning scheme was introduced in [8] where auto-encoders on different clients are trained on unlabeled data from each IIoT device. The trained models are then aggregated into a global AE. Finally, a supervised neural network is trained on the resulting model using a small amount of labeled data. However, the local training data cause much overhead in communication. Even with the random selection of clients, the training time and communication cost are increased due to the unselected clients [8]. A deep learning semi-supervised scheme was proposed in [9] to train on data collected from remote telemetry unit (RTU) streams of gas pipelines where auto-encoders and deep NN were also used. However, the authors did not compare their algorithm performance with other
approaches in the literature in terms of speed or accuracy. A Gaussian Mixture-based model is proposed in [10] to discover zero-day attacks in IIoT edge networks. The proposed system was tested on NSL-KDD and UNSW-NB15 datasets and achieved higher accuracy and faster processing compared with five anomaly detection techniques in the literature. The authors, however, did not discuss the system’s performance against other types of attacks. A multi-feature data clustering optimization model was introduced in [11] for intrusion detection of industrial networks. The classification process is based on the priority threshold of data features for each node in the network. The algorithm was tested on NSL-KDD dataset and six different attack methods, but it required more features to perform better compared to other algorithms in the literature [11].

5.3 The DS2OS Dataset

5.3.1 Description of the Data

The distributed smart space orchestration system (DS2OS) dataset [20] is an anomaly detection dataset for IoT systems. It was introduced in [3] as a benchmark with features from the system’s application layer where service IDs and node identifiers are utilized rather than IP addresses and protocol packets. It contains network flow information about the communication between every two endpoints in the system, where a number of smart sensors, including lighting sensors, motion detectors, and temperature sensors, are distributed over 21 locations. The dataset includes 12 attribute features that describe the current operation occurring in the system. These features are [3]:

- **sourceID**: is a unique name for each accessing service controller. This is analogous to the hostname or machine name in MQTT and ROS protocols contexts.
• **sourceAddress**: is the address from which an accessing service is sending requests. Since the dataset is based on a publish-subscribe model of communication systems, this attribute points to the *publishing* node name.

• **sourceType**: every registered service must have a type that defines its category in the system (e.g., “motion sensor” type).

• **sourceLocation**: is the physical location of the registered accessing service (e.g., room number).

• **destinationServiceAddress**: is the address to which requests are sent. This corresponds to the *subscribing* node name in a publish-subscribe communication systems model.

• **destinationServiceType**: is the type defining the registered destination service.

• **destinationServiceLocation**: is the physical location of the registered destination service.

• **accessedNodeAddress**: is the topic name that holds the communicated message between the publishing and subscribing services.

• **accessedNodeType**: the message type supported by the topic could be of a standard type like “Boolean” or a custom type like “\lightController”. This is the same definition of message type in the ROS context.

• **operation**: represents the type of operation performed. Some operation types in the dataset are read/write, subscribe/unsubscribe, notify-callback, register/unregister service, and lock the whole subtree of nodes.
• **value**: is the value being read or written between services (e.g., temperature value).

• **timestamp**: the time at which a certain service has access to another (e.g., a request to switch off the light).

![Pie chart showing the distribution of anomalies in DS2OS](image)

Figure 34: Distribution of anomalies in **DS2OS**

With a total of 357,952 record indices, including benign behaviors, seven attack scenarios considered in the dataset are distributed as shown in Figure 34, and are described as follows [3]:

**Network scan:**

When an attacker scans some active services in the system. It can be seen in the current dataset as too many requests from the same source service to many random destination services, as shown in Figure 35.

Simulating this attack on our smart manufacturing system involves a single source service like `/sld_node` accessing multiple random destination services like `/sld_red_pub`, `/sld_white_pub`, `/sld_blue_pub`, and `/vgr_node`. 
Spying:

An attacker attempts to read the operation values of active services. The **DS2OS** dataset shows this type of attack when too many *read* requests from one source to many destination services *of the same type*, not random like the scan attack, as shown in Figure 36.

This attack can be simulated on the smart manufacturing system when, for example, the */hbw node* that has no relationship with the SLD station reads data from */sld_red_pub*, */sld_white_pub*, */sld_blue_pub*, and */vgr_node* services.

**Data types probing:**

It involves sending data of different types than the one assigned to the target. Per contra to the last two types, this attack involves only *one destination service*
accessed by one source for only write operations to write data of type different from the original destination service type. As shown in Figure 37, the normal behavior involves data of types /basic/text or /basic/number, whereas the attack involves another different type, namely, /basic/composed.

For the smart manufacturing system, this can be represented when, for example, the /place_order service writes a numerical value or a random string to the /new_order service rather than pre-set values like red, white, and blue for instance.

**Malicious operation:**

This attack occurs when one source service attempts to access a destination service with improper operation type, like writing to a sensor service, while sensor services are created for reading only. As shown in Figure 38, a /lightcontrol service is trying to write values to a /movementSensor service.

Similarly, a malicious operation in the smart factory context is described when, for example, a service tries to subscribe (i.e., read) to the /place_order topic,
whereas this topic only writes to other services to initiate a new order process.

Wrong set up:

It is an access attempt from one source to a destination service in the wrong location. Figure 39 shows an example of a wrong setup where a /lightcontrol service in the parents’ room reads sensory data from the children’s room.

In the smart factory context, this could happen when the /new_order service tries to read the empty locations on the high-bay warehouse station via /what_is_empty topic.

Malicious control:

Malicious control occurs when one request from a source service is sent to an irrelevant destination service. This attack type is a generalized version of the wrong setup attack where the irrelevance is not necessarily in the source/destination location. For example, when a /washingmachine service attempts to open the /entranceDoor as shown in Figure 40.

This can be simulated in the smart factory when the /vgr_node tries to acti-
Denial of service (DoS):

DoS occurs when a single source service sends too many requests to the same destination service over and over. Figure 41 shows an example of a DoS from the dataset, where the timestamp column shows the high-frequency rate of these requests (almost every second).

A DoS attack on the smart factory system can occur when the `/sld_node` keeps sending too many requests to `/sld_red_pub`, which may destroy the pneumatic valves and halt the system.

5.3.2 Data Pre-processing

The data contains 12 columns, including features like source and destination IDs, addresses, locations, and data types, and an additional column labeling the attack type. All have a string object data type except for the “timestamp”. Some of these features have empty records: 2,050 empty entries in the “value” feature and 148 in the “accessedNodeType”. These features are adjusted, as discussed in the sequel.

The first step is to create an untouched test set in this very early stage. Since the random selection of data instances may underrepresent some categories [21], we instead perform stratified sampling where the data entries are divided into homogeneous subgroups, i.e., “strata”, and a number of instances are sampled from
each subgroup to guarantee that the whole population is represented appropriately in the test set. A 20% of the whole data is sampled on a stratified basis according to the attack type. Figure 42 shows the code used for stratification and the resulting percentage of each category in the test set and the original data. The percentage shows that each category has a very close percentage of representation as the original dataset.

```
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=1)
for train_index, test_index in split.split(cybersec, cybersec['normality']):
    strat_train_set = cybersec.loc[train_index]
    strat_test_set = cybersec.loc[test_index]

print('Frequency of each category in the test data:
')

print('Frequency of each category in the original data:
')
```

![Figure 42: Stratification of data](image)

Most machine learning algorithms perform poorly with missing features. Since our data is mainly of a string type, filling empty values with the average or median is not doable. Therefore, we remove the data rows that have empty entries. For the same reason, we convert all the features’ text data into numerics using the “ordinal-encoding” concept, where a unique number is given to each text. To be more specific, a list of unique text entries for each feature is prepared with each element represented by an integer [21]. The class labels are also encoded into eight numbers (from 0 to 7) to represent the seven available attacks and the normal condition. The statistical analysis of the encoded attributes shows that the class labels have various scales. Table 6 shows the statistics of each attribute where
<table>
<thead>
<tr>
<th>Attribute</th>
<th>mean</th>
<th>std</th>
<th>min</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourceID</td>
<td>38.19</td>
<td>29.98</td>
<td>0</td>
<td>15</td>
<td>29</td>
<td>71</td>
<td>83</td>
</tr>
<tr>
<td>sourceAddress</td>
<td>45.14</td>
<td>27.79</td>
<td>0</td>
<td>16</td>
<td>45</td>
<td>73</td>
<td>88</td>
</tr>
<tr>
<td>sourceType</td>
<td>2.74</td>
<td>2.21</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>sourceLocation</td>
<td>8.61</td>
<td>5.51</td>
<td>0</td>
<td>5</td>
<td>8</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>destination Service Address</td>
<td>42.00</td>
<td>26.58</td>
<td>0</td>
<td>15</td>
<td>42</td>
<td>71</td>
<td>83</td>
</tr>
<tr>
<td>destination Service Type</td>
<td>2.27</td>
<td>1.77</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>destination Service Location</td>
<td>8.69</td>
<td>5.27</td>
<td>0</td>
<td>5</td>
<td>7</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>accessedNode Address</td>
<td>84.35</td>
<td>52.68</td>
<td>0</td>
<td>31</td>
<td>84</td>
<td>140</td>
<td>166</td>
</tr>
<tr>
<td>accessedNode Type</td>
<td>3.48</td>
<td>2.89</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>operation</td>
<td>0.92</td>
<td>1.38</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>value</td>
<td>5,168</td>
<td>4,482</td>
<td>0</td>
<td>2</td>
<td>5,187</td>
<td>10,594</td>
<td>10,622</td>
</tr>
<tr>
<td>normality</td>
<td>6.88</td>
<td>0.84</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

The columns show the number of records, the mean, the standard deviation, the minimum, three quartiles, and the maximum value, respectively. It shows that some attribute values range between 0 to 10 or less, whereas others may be up to several hundred or thousands. This inconsistency in the numeric data can severely degrade the performance of the ML models. Therefore, a “min-max” normalization is utilized for each feature so that each has a range between [0, 1] based on the following equation:

\[
x' = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}
\]

where \(x\) is the original value, \(x'\) is the resulted normalization, and \(x_{\text{min}}\) and \(x_{\text{max}}\) are the minimum and maximum values, respectively [21].
5.3.3 Visualizing Data

In order to gain more insights about the data in hand, we produce a separate version of the data with reduced dimensionality so that we can plot it. We apply a principal component analysis (PCA) procedure based on singular value decomposition (SVD) of data to project it to a lower dimensional space. We reduce the features into three categories to be plotted in a three-dimensional scatter plot as depicted in Figure 43, where each color represents a type of attack obtained from the label column in the dataset. The plot clearly shows that the data is, by default, separable so that learning models can quickly identify anomalies and produce high evaluation scores unless overfitting occurs, as shown in Section 5.5.

![Figure 43: Visualization of data](image)

5.3.4 Cross-validation of Data

We divide our training data into chunks “folds” where our models train on some of them and get evaluated on others. The training process keeps switching
folds between training and evaluation until the whole training set is consumed. This process is called “cross-validation” and is helpful to validate and improve the model’s accuracy and avoid overfitting [21]. Our study uses the “k-fold” cross-validation algorithm where the training set is randomly shuffled and then divided into \( k = 5 \) folds.

5.4 Proposed Approach

We study the performance of different ML models in detecting and classifying the IoT attacks based on the DS2OS dataset. In order to perform a more subjective evaluation for these models, metrics like precision, recall, and F1-score have been considered. Although accuracy sounds like a common metric for ML models, it is generally not the preferred metric for classifiers, especially when some classes are much more frequent than others [21] (as the normal behavior compares to other attacks in our dataset). One reason for that is that the accuracy is the ratio between the “True Positive” and “True Negative” classifications and the entire output of the model. If \( TP, TN, FP \), and \( FN \) are the “True Positive”, “True Negative”, “False Positive”, and “False Negative”, respectively, accuracy is calculated as:

\[
\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{7}
\]

On the other hand, precision represents the correct positive classifications. It is the ratio of the “True Positive” to the total positive classifications. The following equation can calculate precision:

\[
\text{precision} = \frac{TP}{TP + FP} \tag{8}
\]

The recall, sensitivity, or the true-positive rate (TPR) is the ratio of “True Positive” classifications to the total number of actual positives. The following equation
computes the recall:

\[ recall = \frac{TP}{TP + FN} \]  

(9)

The F1-score is the harmonic mean of the precision and the recall. Classifiers will only get a high F1 score if both recall and precision are high. It is expressed as:

\[ F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \]  

(10)

**Remark 5.4.1.** Although precision shows more particularity than accuracy, it ignores the FN attacks, which are critical in measuring the performance of a classifier of such systems. If a trade-off happens between precision and recall, the recall metric is more desirable as it considers the FN even if FP alarms may arise.

### 5.4.1 Learning Models

We consider the following five models:

**k-Nearest Neighbor:**

The k-nearest neighbors (kNN) algorithm is a supervised classification ML algorithm which, given training labeled data, outputs the class labels of input test set [22] based on the labels of its \( k \) nearest neighbors. The neighborhood of points is determined by the distance between points, Euclidean distance, for instance, and the number of nearest neighbors \( k \) is a hyperparameter previously defined.

**Decision Tree**

Based on the given attribute features, the decision tree (DT) algorithm builds a tree model where a basic node is split into multiple sub-nodes based on correlations between features until leading to the class labels (leaves). This split operation keeps taking place until specific criteria are met (e.g., maximum tree depth).
Random Forest

The random forest (RF) algorithm is an ensemble meta-algorithm involving multiple decision trees. The RF algorithm trains its model on different parts of the features for $n$ training loops to build $n$ decision trees to control overfitting.

Artificial Neural Networks

Artificial neural networks (ANN) are the core component of the deep learning algorithms that are used in highly complex ML tasks such as computer vision (e.g., Google Images) and speech recognition services (e.g., Apple’s Siri), to name a few. The simplest form of an ANN is the multi-layer perceptron (MLP) that learns an approximation function that maps between input features and output labels. There can be one or more nonlinear layers between the input and the output layer, called hidden layers.

Support-Vector Machines

Support-vector machine (SVM) algorithm attempts to build a hyperplane between data points after getting labeled training data for each category. The SVM algorithm is computationally expensive, especially in high dimensional data (data of many features). However, the functions used to compute the hyperplane (i.e., the kernels) are optimized to reduce the complexity.

5.5 Experimental Results

The ML algorithms described in Section 5.4 are trained on the training set. As discussed in Section 5.3.2, the training set is obtained after separating the testing set from the data. It consists of pre-processed features that describe particular properties in the system by which the behavior is classified as normal or abnormal, with seven attack types. Classification labels of the system behavior are provided in the training data for the models to train on. The cross-validation step discussed
in Section 5.3.4 divides the training set into five folds to train and validate the models on them. Once the models are trained, the normalized test set, based on eq. (6), is applied to the trained model for evaluation. Table 7 shows the evaluation results in both the training and testing stages. The first five rows represent the training process’s precision, recall, F1 score, accuracy, and execution speed, whereas the following five rows represent their testing stage counterparts. Columns 3–7 represent the five utilized machine learning (ML) algorithms.

It is shown from the table that all algorithms have almost the same accuracy, whereas different precision, recall, and F1 metrics are found, especially in the testing stage. This enforces our discussion in Section 5.4 that accuracy is not a precise metric. It is also noted that although the performance of the SVM model is higher than 0.93 in all metrics in the training process, it has a poor performance during testing (metrics are between 0.75 to 0.87). This means that the model overfits the training data and fails to classify the testing set accurately. Moreover, the SVM takes a very long time in both training and testing compared to other techniques. The ANN achieves the fastest anomaly detection (8 mSec on the test set), but it takes 70.64 seconds to train. The low-speed training process makes it less favored in applications requiring fast training, despite its high recall scores. A similar conclusion is driven for the RF algorithm. DT and kNN algorithms have outstanding performances, but DT could be more favored in applications requiring fast detection.

5.6 Conclusions and Future Work

Industrial internet of things (IIoT) systems involve many smart devices, including sensors and controllers, that aim to improve the system performance to optimize the industrial process. These smart components continuously talk to each other and exchange a huge amount of data. Although this provides numerous
advantages to the system, it is more vulnerable to severe security attacks due to the broadly distributed communication and the potential inconsistency between different devices in the same network. Therefore, anomaly detection systems are necessary for any IIoT system to perform safely and avoid interrupting services. Our work was divided into two stages; first, we performed a detailed literature review of cybersecurity research for IIoT systems with more focus on anomaly detection approaches. We followed a standardized, systematic procedure to perform this review and used advanced Sankey illustrations to demonstrate the conclusion of our review and to show the correlation between the different fields of research on IIoT cybersecurity. Second, we studied the performance of different supervised machine learning algorithms by testing them against an anomaly benchmark dataset for IoT systems. More specifically, we performed pre-processing operations on the dataset, such as removing empty fields, encoding text entries, and scaling features to improve the performance of the models. Five ML models have been considered: kNN, DT, RF, ANN, and SVM. It was found that SVM took a significantly long time to train and test and showed poor precision when tested. Although the ANN had the fastest detection time during testing and demonstrated high preci-

<table>
<thead>
<tr>
<th>Model</th>
<th>SVM</th>
<th>ANN</th>
<th>RF</th>
<th>DT</th>
<th>kNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Precision</td>
<td>0.9949</td>
<td>0.9958</td>
<td>0.9999</td>
<td>0.9984</td>
<td>0.9987</td>
</tr>
<tr>
<td>Recall</td>
<td>0.9325</td>
<td>0.9999</td>
<td>0.9995</td>
<td>0.9985</td>
<td>0.9999</td>
</tr>
<tr>
<td>F1</td>
<td>0.9535</td>
<td>0.9979</td>
<td>0.9997</td>
<td>0.9985</td>
<td>0.9993</td>
</tr>
<tr>
<td>Accuracy</td>
<td>0.9942</td>
<td>0.9996</td>
<td>0.9999</td>
<td>0.9999</td>
<td>0.9999</td>
</tr>
<tr>
<td>Speed (sec)</td>
<td>236.944</td>
<td>70.64</td>
<td>27.06</td>
<td>1.78</td>
<td>0.952</td>
</tr>
<tr>
<td>Testing Precision</td>
<td>0.7511</td>
<td>0.9861</td>
<td>1.0000</td>
<td>0.9923</td>
<td>0.9968</td>
</tr>
<tr>
<td>Recall</td>
<td>0.8674</td>
<td>0.9999</td>
<td>1.0000</td>
<td>0.9483</td>
<td>0.9999</td>
</tr>
<tr>
<td>F1</td>
<td>0.7891</td>
<td>0.9928</td>
<td>1.0000</td>
<td>0.9652</td>
<td>0.9984</td>
</tr>
<tr>
<td>Accuracy</td>
<td>0.9942</td>
<td>0.9996</td>
<td>0.9999</td>
<td>0.9999</td>
<td>0.9999</td>
</tr>
<tr>
<td>Speed (sec)</td>
<td>11.361</td>
<td>0.008</td>
<td>0.915</td>
<td>0.168</td>
<td>8.74</td>
</tr>
</tbody>
</table>

Table 7: Evaluation metrics and speeds
sion, it took the longest to train, making it less favored in applications requiring fast training. The decision tree (DT) algorithm performed best in precision metrics and training and testing speeds. The data visualization plot shows that the data points are separable by their nature which explains the very high precision scores we obtained by utilizing basic ML algorithms. This data perfection may not exist in other IIoT complex systems where hundreds or even thousands of devices are communicating together, and more sophisticated processes are involved. Therefore, our future work will involve designing an IIoT dataset for the smart manufacturing system discussed in Manuscript 4. Different ML algorithms will be tested on that system, and new anomaly detection algorithms may be proposed.

List of References


national Conference on Knowledge and Systems Engineering (KSE). IEEE, 2019, pp. 1–6.


Conclusions & Future Work

In a step toward secure and efficient supply chain systems that respond quickly to unpredicted events, this study follows two directions to improve vital parts in typical supply chains: logistics planning and manufacturing systems. The first direction is developing a high-speed algorithm for the vehicle routing problem (VRP). The study proposes a novel hybrid algorithm running entirely on different graphics processing units (GPU) platforms to take advantage of their exceptional high-speed parallel execution. In this algorithm, a genetic algorithm (GA) is integrated with a 2-opt local search to solve capacitated VRP (CVRP) problems.

The dissertation discussed the implementation of the algorithm on single GPU platforms where the algorithm could handle small and medium benchmark problems. The works compare the proposed algorithm with conventional central processing units (CPU) counterparts to demonstrate the fast execution features of the GPU implementation. Since GAs are subject to issues like premature convergence and getting stuck in local minima, the works discussed a design of experiment (DOE) approach to statistically and statically tune the GA parameters. Numerical results show that the tuned parameter set results in faster convergence without premature termination. Moreover, the execution speed has increased 1,700 times compared to the CPU counterpart, reducing the execution time of some problems from 2.4 years to almost 12 hours.

Further to the first direction, the study utilized high-performing computing (HPC) platforms to run the GA algorithm on multiple GPUs. Besides running simultaneously, GPUs collaborate and exchange solutions with each other. The proposed algorithm implements per-thread rather than per-warp executions to fully
utilize the GPU cores for maximum parallelism. Different hardware arrangements have been studied to prove the scalability of the proposed algorithm. The results showed the benefits obtained from executing multiple GPUs compared to CPUs or single GPU implementations. Larger problem sizes of up to 20,000 nodes were handled, and the performance increased monotonically with the number of GPUs maintaining high parallel efficiency. The numerical results have been enforced by profiling analysis of the algorithm to study the behavior of executive functions and memory transfers in terms of speed and size. The data exchange between GPUs reduced premature convergence and improved the quality of the obtained solutions. Future work in this direction will focus on developing a more genuine GA algorithm to execute on multi-GPU clusters with varying connection topologies, which will enable the utilization of more GPUs.

The second direction of this study discussed the utilization of the industrial internet of things (IIoT) for manufacturing systems where smart devices like sensors, actuators, and controllers are interconnected through sophisticated communication protocols to exchange a vast amount of data. An IIoT lab-scale smart manufacturing system called “URILF4.0” was introduced. Hardware and open-source software were utilized at an affordable cost. This IIoT system is a testbed for exploring and experimenting with Industry 4.0 concepts involving different manufacturing stations that mimic real-world manufacturing facilities. The dissertation discussed the details required to build and maintain such a system. Future work in this direction involves expanding the “URILF4.0” to include more manufacturing stations, digital twins, and simulation peripherals.

Due to the broad communication network and the potential inconsistency between devices from different manufacturers, modern IIoT systems are vulnerable to cybersecurity attacks. Therefore, communications between the “URILF4.0” units
are secured by hashing and digital certifications techniques. Moreover, a comprehensive systematic literature review on cybersecurity approaches in IIoT systems has been developed. Almost 500 research works in the field have been reviewed for the last decade. Based on this review, an intelligent anomaly detection approach for a typical IoT system is discussed. Five machine learning (ML) algorithms are applied to a benchmark IoT dataset to analyze the behavior of different processes, classify it into normal behavior, and seven different attack types. Future work will involve designing an IIoT dataset for the “URILF4.0” smart manufacturing system, testing it with different ML algorithms, and proposing new anomaly detection algorithms for such systems.
APPENDIX A

Supplementary Material – More CVRP Solutions

Table A.8 shows numerical results for the GA algorithm discussed in manuscripts 1 and 2 where a single GPU is utilized to execute the statically-tuned algorithm to solve more benchmark problems. Eight columns structuring the table represent the problem ID, the number of customer nodes, the best-known solution in the literature, the solution obtained by the proposed algorithm, the execution speed in (sec/generation), the total number of generation required to obtained this solution, the total execution time, and the gap between the obtained solution and the best-known one in the literature, respectively. The results show that the algorithm is able to obtain high-quality solutions in a feasible time for most small and medium size problems (up to 500 nodes). However, the quality degrades with larger problem sizes.
<table>
<thead>
<tr>
<th>Problem ID</th>
<th>n</th>
<th>Best known solution</th>
<th>Obtained solution</th>
<th>Sec/generation</th>
<th>Number of generations</th>
<th>Total time hh:mm:ss</th>
<th>Gap with literature</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-n101-k10</td>
<td>101</td>
<td>820</td>
<td>886.98</td>
<td>0.0264</td>
<td>17,000</td>
<td>0:07:29</td>
<td>0.08</td>
</tr>
<tr>
<td>M-n121-k7</td>
<td>121</td>
<td>1,034</td>
<td>1,192.73</td>
<td>0.0471</td>
<td>28,000</td>
<td>0:21:59</td>
<td>0.15</td>
</tr>
<tr>
<td>M-n151-k12</td>
<td>151</td>
<td>1,015</td>
<td>1,108</td>
<td>0.0364</td>
<td>34,500</td>
<td>0:20:56</td>
<td>0.09</td>
</tr>
<tr>
<td>M-n200-k16</td>
<td>200</td>
<td>1,274</td>
<td>1,486</td>
<td>0.0639</td>
<td>16,100</td>
<td>0:17:09</td>
<td>0.17</td>
</tr>
<tr>
<td>M-n200-k17</td>
<td>200</td>
<td>1,275</td>
<td>1,420</td>
<td>0.0579</td>
<td>44,000</td>
<td>0:42:28</td>
<td>0.11</td>
</tr>
<tr>
<td>Golden_1</td>
<td>240</td>
<td>5,623.47</td>
<td>5,928.47</td>
<td>0.0778</td>
<td>19,800</td>
<td>0:25:40</td>
<td>0.05</td>
</tr>
<tr>
<td>Golden_2</td>
<td>320</td>
<td>8,404.61</td>
<td>9,075.53</td>
<td>0.5755</td>
<td>26,800</td>
<td>4:17:03</td>
<td>0.08</td>
</tr>
<tr>
<td>Golden_3</td>
<td>400</td>
<td>10,997.80</td>
<td>1,2540.26</td>
<td>0.5121</td>
<td>20,700</td>
<td>2:56:40</td>
<td>0.14</td>
</tr>
<tr>
<td>Golden_4</td>
<td>480</td>
<td>13,588.60</td>
<td>1,5636.85</td>
<td>0.7312</td>
<td>58,500</td>
<td>11:52:55</td>
<td>0.15</td>
</tr>
<tr>
<td>Golden_5</td>
<td>200</td>
<td>6,460.98</td>
<td>7,214.5</td>
<td>0.192</td>
<td>19,700</td>
<td>1:03:02</td>
<td>0.12</td>
</tr>
<tr>
<td>Golden_6</td>
<td>280</td>
<td>8,400.33</td>
<td>9,682.65</td>
<td>0.1076</td>
<td>16,600</td>
<td>0:29:46</td>
<td>0.15</td>
</tr>
<tr>
<td>Golden_7</td>
<td>360</td>
<td>10,102.70</td>
<td>1,1307.03</td>
<td>0.4</td>
<td>72,500</td>
<td>8:03:20</td>
<td>0.12</td>
</tr>
<tr>
<td>Golden_8</td>
<td>440</td>
<td>11,635.30</td>
<td>1,3101.5</td>
<td>0.7301</td>
<td>59,200</td>
<td>12:00:22</td>
<td>0.13</td>
</tr>
<tr>
<td>Golden_9</td>
<td>255</td>
<td>579.70</td>
<td>635.38</td>
<td>0.1721</td>
<td>57,500</td>
<td>2:44:56</td>
<td>0.10</td>
</tr>
<tr>
<td>Golden_10</td>
<td>323</td>
<td>735.43</td>
<td>813.86</td>
<td>0.6217</td>
<td>50,900</td>
<td>8:47:25</td>
<td>0.11</td>
</tr>
<tr>
<td>Golden_11</td>
<td>399</td>
<td>911.98</td>
<td>1,014.95</td>
<td>1.0675</td>
<td>86,900</td>
<td>25:46:06</td>
<td>0.11</td>
</tr>
<tr>
<td>Golden_12</td>
<td>483</td>
<td>1,101.50</td>
<td>1,282.85</td>
<td>0.9854</td>
<td>95,300</td>
<td>26:05:09</td>
<td>0.16</td>
</tr>
<tr>
<td>Golden_13</td>
<td>252</td>
<td>857.19</td>
<td>914.34</td>
<td>0.4321</td>
<td>44,000</td>
<td>5:16:52</td>
<td>0.07</td>
</tr>
<tr>
<td>Golden_14</td>
<td>320</td>
<td>1,080.55</td>
<td>1,166.07</td>
<td>0.766</td>
<td>57,600</td>
<td>12:15:22</td>
<td>0.08</td>
</tr>
<tr>
<td>Golden_15</td>
<td>396</td>
<td>1,337.27</td>
<td>1,499.7</td>
<td>0.513</td>
<td>50,000</td>
<td>7:07:30</td>
<td>0.12</td>
</tr>
<tr>
<td>Golden_16</td>
<td>480</td>
<td>1,611.28</td>
<td>1,848.83</td>
<td>0.9078</td>
<td>50,000</td>
<td>12:36:30</td>
<td>0.15</td>
</tr>
<tr>
<td>Golden_17</td>
<td>240</td>
<td>707.76</td>
<td>776.49</td>
<td>0.088</td>
<td>47,900</td>
<td>1:10:15</td>
<td>0.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-----</td>
<td>--------</td>
<td>--------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>Golden_18</td>
<td>300</td>
<td>995.13</td>
<td>1,084.12</td>
<td>0.6803</td>
<td>50,000</td>
<td>9:26:55</td>
<td>0.09</td>
</tr>
<tr>
<td>Golden_19</td>
<td>360</td>
<td>1,365.60</td>
<td>1,514.63</td>
<td>1.0707</td>
<td>90,900</td>
<td>27:02:07</td>
<td>0.11</td>
</tr>
<tr>
<td>Golden_20</td>
<td>420</td>
<td>1,817.59</td>
<td>2,070.88</td>
<td>0.7223</td>
<td>98,300</td>
<td>19:43:22</td>
<td>0.14</td>
</tr>
<tr>
<td>X-n219-k73</td>
<td>219</td>
<td>117,595</td>
<td>118,360</td>
<td>0.0757</td>
<td>50,000</td>
<td>1:03:05</td>
<td>0.01</td>
</tr>
<tr>
<td>X-n266-k58</td>
<td>266</td>
<td>75,478</td>
<td>79,484.14</td>
<td>0.1726</td>
<td>43,200</td>
<td>2:04:16</td>
<td>0.05</td>
</tr>
<tr>
<td>X-n317-k53</td>
<td>317</td>
<td>78,355</td>
<td>79,932.85</td>
<td>0.1162</td>
<td>46,900</td>
<td>1:30:50</td>
<td>0.02</td>
</tr>
<tr>
<td>X-n336-k84</td>
<td>336</td>
<td>139,111</td>
<td>146,596.07</td>
<td>0.2417</td>
<td>48,000</td>
<td>3:13:22</td>
<td>0.05</td>
</tr>
<tr>
<td>X-n376-k94</td>
<td>376</td>
<td>147,713</td>
<td>149,684</td>
<td>0.1663</td>
<td>46,300</td>
<td>2:08:20</td>
<td>0.01</td>
</tr>
<tr>
<td>X-n384-k52</td>
<td>384</td>
<td>65,928</td>
<td>72,087</td>
<td>0.2413</td>
<td>48,500</td>
<td>3:15:03</td>
<td>0.09</td>
</tr>
<tr>
<td>X-n420-k130</td>
<td>420</td>
<td>107,798</td>
<td>112,265</td>
<td>0.3656</td>
<td>49,100</td>
<td>4:59:11</td>
<td>0.04</td>
</tr>
<tr>
<td>X-n429-k61</td>
<td>429</td>
<td>65,449</td>
<td>71,729.47</td>
<td>0.7063</td>
<td>45,700</td>
<td>8:57:58</td>
<td>0.10</td>
</tr>
<tr>
<td>X-n469-k138</td>
<td>469</td>
<td>221,824</td>
<td>230,284.4</td>
<td>0.5445</td>
<td>46,800</td>
<td>7:04:43</td>
<td>0.04</td>
</tr>
<tr>
<td>X-n480-k70</td>
<td>480</td>
<td>89,449</td>
<td>96,220.47</td>
<td>2.2194</td>
<td>47,200</td>
<td>29:05:56</td>
<td>0.08</td>
</tr>
<tr>
<td>X-n548-k50</td>
<td>548</td>
<td>86,700</td>
<td>95,050.92</td>
<td>0.9108</td>
<td>48,700</td>
<td>12:19:16</td>
<td>0.10</td>
</tr>
<tr>
<td>X-n586-k159</td>
<td>586</td>
<td>190,316</td>
<td>199,562.5</td>
<td>0.7979</td>
<td>49,000</td>
<td>10:51:37</td>
<td>0.05</td>
</tr>
<tr>
<td>X-n599-k92</td>
<td>599</td>
<td>108,451</td>
<td>119,985.06</td>
<td>2.4354</td>
<td>66,400</td>
<td>44:55:11</td>
<td>0.11</td>
</tr>
<tr>
<td>X-n655-k131</td>
<td>655</td>
<td>106,780</td>
<td>110,246</td>
<td>0.5376</td>
<td>49,800</td>
<td>7:26:12</td>
<td>0.03</td>
</tr>
<tr>
<td>X-n733-k159</td>
<td>733</td>
<td>136,187</td>
<td>167,601</td>
<td>4.9159</td>
<td>49,900</td>
<td>68:08:23</td>
<td>0.23</td>
</tr>
<tr>
<td>X-n749-k98</td>
<td>749</td>
<td>77,269</td>
<td>107,101</td>
<td>1.0728</td>
<td>36,000</td>
<td>10:43:41</td>
<td>0.39</td>
</tr>
<tr>
<td>X-n819-k171</td>
<td>819</td>
<td>158,121</td>
<td>174,133</td>
<td>1.8115</td>
<td>50,000</td>
<td>25:09:35</td>
<td>0.10</td>
</tr>
<tr>
<td>X-n837-k142</td>
<td>837</td>
<td>193,737</td>
<td>227,936.8</td>
<td>1.8115</td>
<td>49,300</td>
<td>24:48:27</td>
<td>0.18</td>
</tr>
<tr>
<td>X-n856-k95</td>
<td>856</td>
<td>88,965</td>
<td>96,335.26</td>
<td>0.9954</td>
<td>49,600</td>
<td>13:42:52</td>
<td>0.08</td>
</tr>
<tr>
<td>X-n916-k207</td>
<td>916</td>
<td>329,179</td>
<td>351,907</td>
<td>3.2831</td>
<td>49,700</td>
<td>45:19:30</td>
<td>0.07</td>
</tr>
<tr>
<td>X-n957-k87</td>
<td>957</td>
<td>85,465</td>
<td>96,034</td>
<td>1.4275</td>
<td>49,900</td>
<td>19:47:12</td>
<td>0.12</td>
</tr>
<tr>
<td>Li_21</td>
<td>560</td>
<td>16,212.83</td>
<td>20,060.88</td>
<td>0.3901</td>
<td>42,200</td>
<td>4:34:22</td>
<td>0.24</td>
</tr>
<tr>
<td>Li_22</td>
<td>600</td>
<td>14,499.04</td>
<td>17,064.71</td>
<td>1.3979</td>
<td>32,000</td>
<td>12:25:33</td>
<td>0.18</td>
</tr>
<tr>
<td>Li_23</td>
<td>640</td>
<td>18,801.13</td>
<td>24,735.21</td>
<td>0.551</td>
<td>47,400</td>
<td>7:15:17</td>
<td>0.32</td>
</tr>
<tr>
<td>Li_24</td>
<td>720</td>
<td>21,389.43</td>
<td>27,119.87</td>
<td>0.7659</td>
<td>27,400</td>
<td>5:49:46</td>
<td>0.27</td>
</tr>
<tr>
<td>Li.25</td>
<td>760</td>
<td>16,665.70</td>
<td>18,305.93</td>
<td>0.7973</td>
<td>50,000</td>
<td>11:04:25</td>
<td>0.10</td>
</tr>
<tr>
<td>Li.26</td>
<td>800</td>
<td>23,977.73</td>
<td>33,410.5</td>
<td>1.1634</td>
<td>45,000</td>
<td>14:32:33</td>
<td>0.39</td>
</tr>
<tr>
<td>Li.27</td>
<td>840</td>
<td>17,320.00</td>
<td>19,660.46</td>
<td>1.1388</td>
<td>47,300</td>
<td>14:57:45</td>
<td>0.14</td>
</tr>
<tr>
<td>Li.28</td>
<td>880</td>
<td>26,566.03</td>
<td>34,203.43</td>
<td>1.3464</td>
<td>35,000</td>
<td>13:05:24</td>
<td>0.29</td>
</tr>
<tr>
<td>Li.29</td>
<td>960</td>
<td>29,154.34</td>
<td>46,531.04</td>
<td>1.8003</td>
<td>5,000</td>
<td>2:30:02</td>
<td>0.60</td>
</tr>
<tr>
<td>Li.30</td>
<td>1,040</td>
<td>31,742.64</td>
<td>44,413</td>
<td>2.3337</td>
<td>20,200</td>
<td>13:05:41</td>
<td>0.40</td>
</tr>
<tr>
<td>Li.31</td>
<td>1,120</td>
<td>34,330.94</td>
<td>48,702</td>
<td>2.907</td>
<td>50,000</td>
<td>40:22:30</td>
<td>0.42</td>
</tr>
<tr>
<td>Li.32</td>
<td>1,200</td>
<td>37,159.41</td>
<td>53,689.59</td>
<td>3.3217</td>
<td>49,800</td>
<td>45:57:01</td>
<td>0.44</td>
</tr>
</tbody>
</table>
The Capacitated Vehicle Routing Problem

The capacitated VRP (CVRP) is the most common and basic form of VRP where the total demands of the customers assigned to identical trucks cannot exceed its capacity, and the customers cannot be split [1]. Based on the above notations, a typical CVRP can be defined as [2]:

**Definition 2.** Given a depot node \( \{0\} \) at location \((x_0, y_0)\) and a set of \( n \) customer nodes \( C = \{1, 2, 3, \ldots, n\}, n \in \mathbb{I}_+ \) at locations \( \{(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots, (x_n, y_n)\} \) where \( x_0, \ldots, x_n, y_0, \ldots, y_n \in \mathbb{R}, \mathbb{I}_+ \) and \( \mathbb{R} \) are sets of positive integers and real numbers respectively, there exists an undirected graph \( G = (\mathcal{N}, \mathcal{A}) \), that can be established where \( \mathcal{N} = \{0\} \cup C \) and \( \mathcal{A} = \{(i, j): i, j \in \mathcal{N}, i < j\} \) is a set of connections (i.e., edges or vertices) between the nodes. Each customer \( i \) orders a non-negative demand \( d_i \in \mathbb{R}_+ \). This demand cannot be divided and hence every customer can be visited only once. The cost \( c_{ij} \in \mathbb{R}_+ \) associated to each edge \((i, j) \in \mathcal{A}\) can be defined as the Euclidean distance between node \( i \) and node \( j \) given as:

\[
c_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}
\]  

It is required to find the optimal solution \( l^* \) that has the minimum distance cost, i.e.,

\[
l^* = \arg\min_{l \in \mathcal{P}} \sum_{i,j=0}^{n} c_{ij}, \ \forall \ i \neq j
\]  

where \( \mathcal{P} \) is the set of all feasible solutions for the problem. For simplicity, we have the following assumption[3, 4, 5]:

136
Assumption 1. A number of $K$ homogeneous trucks are available with capacity $Q$ where $Q > d_i, \forall i \in C$. The number of trucks is not necessarily known before solving the problem.

So that any truck can be assigned to any route available in the solution to avoid more complexity of the problem. A typical CVRP problem can be formulated as follows [6]:

$$\min \sum_{k \in K} \sum_{i \in N} \sum_{j \in N, i \neq j} c_{ij} x_{ijk}, \quad \text{s.t.,} \quad (B.13a)$$

$$\sum_{k \in K} \sum_{i \in N, i \neq j} x_{ijk} = 1, \quad \forall i \in C \quad (B.13b)$$

$$\sum_{i \in C} d_i \sum_{j \in N, i \neq j} x_{ijk} \leq Q, \quad \forall k \in K \quad (B.13c)$$

$$\sum_{j \in N} x_{0jk} = 1, \quad \forall k \in K \quad (B.13d)$$

$$\sum_{i \in N} x_{ijk} - \sum_{j \in N} x_{jik} = 0, \quad \forall j \in C, \forall k \in K \quad (B.13e)$$

$$\sum_{k \in K} \sum_{i \in S} \sum_{j \in N, i \neq j} x_{ijk} \leq |S| - 1, \quad \forall S \subseteq C, |S| > 2 \quad (B.13f)$$

$$x_{ijk} \in \{0, 1\}, \quad \forall i, j \in N, \forall k \in K \quad (B.13g)$$

Eq.(B.13a) states that the objective of this problem is to minimize the total cost of routes, which is the total cumulative Euclidean distance between customer nodes. Whereas eq.(B.13b) ensures that each customer is visited only once, eq.(B.13c) makes sure that the vehicle capacity is not exceeded. Equations (B.13d), and (B.13e) maintains the conditions that all vehicles must leave the depot and the customers are visited only once, respectively. Eq.(B.13f) takes care of subtour elimination where any routes that are not connected to the depot are removed from the solution according to the classical Dantzig-Fulkerson-Johnson (DFJ) formulation [6]. For example, if the number of nodes in subtour $S \subseteq C$ is equal to the
edges between them, then $S$ is a subtour and at least one edge should be removed forcing the open nodes to connect to others outside $S$. Eq.(B.13g) is for integrality.

A typical CVRP problem with a feasible solution is depicted in the undirected graph in Figure 1 where 12 customer nodes and a depot node \{0\} are distributed in the problem space. The lines between the pairs of points represent a connection path between them. The demand $d_i$ of each of the customers as well as the cost of the connections $c_{ij}$ are shown. The truck capacity is limited to $Q = 10$. Feasible routes are represented by $r_1, r_2, r_3, \text{ and } r_4$ with $r_i \subset A \forall i$ enabling four identical trucks to be used. The first route, for example, $r_1 = \{(0, 4), (4, 11), (11, 6), (6, 0)\}$ has a cost of 35 and load of $5 + 3 + 1 = 9 < Q$. The set $l = \{r_1 \cup r_2 \cup r_3 \cup r_4\} \subset P$ with a total cost of 114 represents an example solution of this particular problem which may not necessarily represent the optimal solution.

**List of References**


