FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Parallel Hybrid Metaheuristics for Solving the Firefighter Problem Using the GPU

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Ing. Gajo Gajic, BSc
Matrikelnummer 0828150

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Dipl.-Ing. Christopher Bacher, BSc

Wien, 2. Mai 2018

Gajo Gajic	Günther Raidl

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Parallel Hybrid Metaheuristics for Solving the Firefighter Problem Using the GPU

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Ing. Gajo Gajic, BSc
Registration Number 0828150

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance: Dipl.-Ing. Christopher Bacher, BSc

Vienna, 2nd May, 2018

Gajo Gajic | Günther Raidl

Erklärung zur Verfassung der Arbeit

Ing. Gajo Gajic, BSc
Dürrwienstrasse 8a/1/6, 3021 Pressbaum

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 2. Mai 2018

Gajo Gajic

Kurzfassung

Das Firefighter Problem (FFP) ist ein deterministisches diskretes Zeitmodell, das die Ausbreitung eines Feuers oder eines anderen Problems über einen ungerichteten Graphen simuliert. Es bietet eine Möglichkeit, Brandbekämpfungsstrategien zu entwickeln, indem eine bestimmte Anzahl von Firefightern auf strategischen Punkten in jedem Zeitschritt eingesetzt wird, um so viele Knoten wie möglich vor Feuer zu retten. Das Modell findet Anwendung in zahlreichen Bereichen, wo es um die Verbreitung von verschiedenen Arten von Information geht. Unter Anderem zählen dazu Vakzinationsstrategien, Finanzkapitalflüsse, virales Marketing sowie die Verbreitung von Viren in Computernetzwerken. In mehreren Studien wurde gezeigt, dass das FFP für bestimmte Arten von Graphen [CC17] und eine bestimmte Anzahl der eingesetzten Feuerwehrleute [BCR13] NP-schwer ist. Mit dem Ziel, Strategien für eine effizientere Lösung des FFP zu finden, wird hier eine parallele hybride Metaheuristik auf einer GPU mittels CUDA implementiert. Die hybride Metaheuristik beinhaltet einen parallelen Ant Colony Optimization Algorithmus (ACO), der eine dynamische Kandidatenliste und Heuristik anwendet, welche die Topologie des Graphen zu jedem Zeitschritt berücksichtigen und dadurch den Suchraum reduzieren. Des Weiteren, wird eine parallele Variante der Variable Neighborhood Search (VNS) eingeführt und im Anschluss mit der ACO Implementierung kombiniert. Zusätzlich werden sequentielle Versionen des ACO, der VNS und der hybriden Metaheuristik entwickelt um die Effizienz der parallelen Implementierungen zu testen. Abschließend wird eine Gegenüberdarstellung der entwickelten Algorithmen mit vorherigen Arbeiten [BBGM$^+$14, HWR15] vorgenommen.

Für die Leistungsbewertungstests wird dieselbe Testkonfiguration wie in früheren Arbeiten, welche Benchmark-Instanzen mit 120 Graphen unterschiedlicher Dichte und Größe enthalten, verwendet. Testergebnisse zeigen, dass der vorgeschlagene sequenzielle ACO eine durchschnittliche Verbesserung von 10,47% gegenüber der ursprünglichen ACO Implementierung erreicht. Eine weitere Erkenntnis besteht darin, dass, verglichen mit jedem einzelnen Algorithmus, die Kombination von ACO und VNS eine Verbesserung der Lösungsqualität auf beiden Plattformen liefert. Die Testergebnisse der parallelisierten Algorithmen ergeben, dass jede parallele Implementierung ihre sequenzielle Entsprechung übertrifft, wobei auch die Lösungsqualität verbessert wird. Die erreichten Speed-ups betragen bis zu 141x (ACO), 106x (VNS) und 114x (hybrider Algorithmus).

Abstract

The Firefighter Problem (FFP) is a deterministic discrete-time model which simulates the spread of a fire or other problem over an undirected graph. It offers a possibility of developing fire containment strategies by deploying a given number of firefighters on strategic points at each time step with the goal of saving as many nodes from fire as possible. The model is applied in numerous areas considering the spread of various types of information. These include, among others, vaccination strategies, financial capital flow, viral marketing and the spread of viruses in computer networks. In several studies it has been shown that the FFP is NP-hard for specific types of graphs [CC17] and the number of firefighters [BCR13] involved. With the goal of finding strategies for a more efficient solution to the FFP, a parallel hybrid metaheuristic is implemented on a GPU using CUDA. The hybrid metaheuristic comprises a parallel Ant Colony Optimization algorithm (ACO), which applies a dynamic candidate list and heuristic that take into account the topology of the graph at each time step, thereby reducing the search space. Furthermore, a parallel version of Variable Neighborhood Search (VNS) is introduced and combined with the ACO implementation. In addition, sequential versions of the ACO, the VNS and the hybrid metaheuristic are developed in order to test the efficiency of the parallel implementations. Finally, the developed algorithms are compared with previous works [BBGM$^+$14, HWR15].

For the performance evaluation tests we used the same experimental setup as previous works, which contains a benchmark instance set of 120 graphs with different density and size. Test results show that the proposed sequential ACO achieved an average improvement of 10.47 % compared to the original ACO implementation. Another finding is that the combination of ACO and VNS provides an improvement in the solution quality compared to each algorithm on their own on both platforms. The test results of the parallelized algorithms revealed that each parallel implementation outperforms its sequential counterpart while improving the solution quality. The achieved speed-ups are up to 141x (ACO), 106x (VNS) and 114x (hybrid algorithm).

Contents

CHAPTER 1

Introduction

1.1 Motivation & Contribution

The Firefighter Problem (FFP) was formalized as a graph-based optimization problem in 1995 by Bert L. Hartnell [Har95] as a model for studying the spread and containment of fire. Due to its mathematical structure, variations of this model have found application in numerous fields that analyze the distribution of various kinds of information throughout networks, such as: studying the spread of diseases and developing vaccination strategies [Har04a], analyzing the growth of groups [Mar17], the spread of viruses in computer networks, financial capital flow, as well as viral marketing [CWY09].

The FFP uses a discrete-time model to simulate how a fire or other problem breaks out and spreads over an undirected graph in a discrete time period. The model can be used to investigate strategies for containing a fire or the spread of a similar kind of problem. A strategy consists of deploying a given number of firefighters on strategic points at each time step. Once a firefighter is placed on an unburnt vertex it is protected against catching fire for all time intervals. This process continuous until the fire is contained.

Figure 1.1 depicts an example of an undirected graph with 14 nodes and two firefighters available. At $t = 0$ the fire breaks out on node 1 (red circle). We can save nodes 3 and 8 at $t = 1$. Then the fire spreads to nodes 4, 5 and 6, so that we protect nodes 10 and 11 in the next time step $t = 3$. Since node 9 is the last one to catch fire, there are no unprotected adjacent nodes left, that is, the fire is considered as contained.

Since its initial formulation, the FFP has been the topic of a series of papers, e.g., [FM09, KM10] and [CFvL11], however, with a strong focus on theoretical results. Some extensions of the FFP [Lip17, Mic14] are the focus of some computational studies as well. This thesis focuses on the original formulation of the problem exclusively. Computationally, the original formulation has been tackled only in three recent studies. The first to apply a metaheuristic approach to the FFP is the study conducted by Blum et al. [BBGM$^+$14].

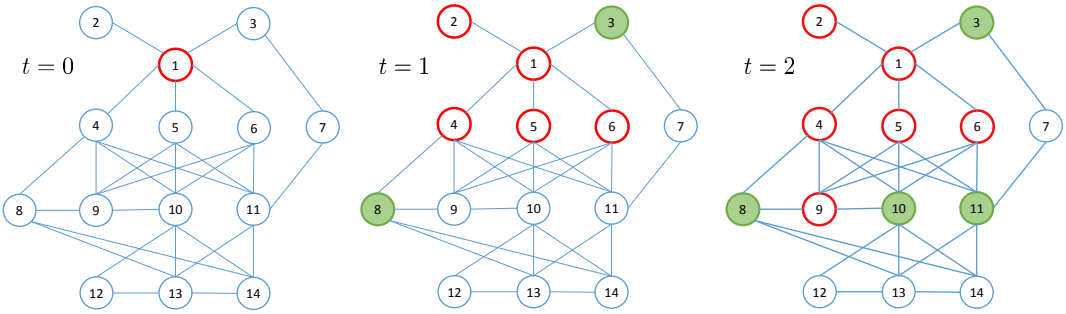


Figure 1.1: Example for a graph with 14 vertices and two firefighters.

It investigates a population-based algorithm based on a MAX–MIN Ant System (MMAS) and a variant hybridized with Integer Linear Programming (ILP). An alternative, single solution based metaheuristic approach with a new, more compact, solution representation was developed by Hu et al. [HWR15]. This article uses a general variable neighborhood search (VNS) approach combined with variable neighborhood descent (VND). The most recent study in the field has been done by García-Martínez et al. [GBRL15]. It combines ideas presented in his and Blum's previous works and proposes nine integer linear programming (ILP) techniques along with six heuristic approaches with random graphs.

The first two mentioned approaches solving the FFP have each their own set of strengths and drawbacks. More specifically, it has been established that the major advantage of the MMAS is its exploration capability, whereas the strength of the VNS lies in its fast search intensification capability. Therefore [BPRR11] and [Tal09] propose combining the two approaches in order to enhance the results.

Various studies on the complexity of FFP show that it is NP-hard for specific types of graphs [CC17] and the number of firefighters [BCR13] involved. As it is a given that parallelized computing enables us to solve larger or more complex instances of optimization problems more efficiently, the logical next step is to introduce a parallel-hybrid optimization technique for solving the FFP, which is the aim of this thesis.

The development of microprocessor technologies has raised significant questions concerning how parallel programs can be efficiently implemented. New types of hardware have been designed for high performance computing (HPC). Among them, there are Graphics Processing Units (GPUs), which provide great computing acceleration [GGN+08] at an affordable cost but demand extensive programming skills.

This thesis proposes the first parallel hybrid metaheuristic solution using GPU for the FFP. The major contributions are the following:

1. A new sequential ACO algorithm with dynamic metrics that are provided by analyzing the topology of the graph. By using these metrics, we can additionally

develop a candidate list with prioritized nodes including only nodes that have a positive impact on the containment of fire. The higher computational cost of generating the metrics and the list is expected to be compensated by the smaller search space. The proposed algorithm shows promising results compared to previous works.

2. Similar ideas were applied to a VNS algorithm in order to develop an adaptive shaking function improving the accuracy of the node removal phase.
3. By combining the two algorithms we implemented a hybrid metaheuristic, which generated better results than each algorithm on their own.
4. Parallelization of the ACO for the FFP, which is to the best of our knowledge the very first of its kind. In the analysis we compare it to the sequential counterpart and come to the conclusion that the speed-up factor is up to 141 for the largest instances tested.
5. Parallelization of the VNS, with a speed-up factor up to 106 for the largest instances tested while improving the quality of the results for instances sizes ≥ 500.
6. Hybridization of the parallel ACO and the parallel VNS, resulting in a speed-up (114x) and outperforming all the previous results.

1.2 Problem Statement

Given is an undirected graph $G = (V, E)$ where V and E are the vertex set and the edge set of G. The labels $L = \{untouched, burnt, protected\}$ represent the state of a vertex at a particular point in time. Initially, all vertices in V are labeled as *untouched*. The state of a predefined set of vertices $B_{\text{init}} \subseteq V$ changes to *burnt* at time $t = 0$, when the fire breaks out. For each iteration $t \geq 1$ a fixed number D of firefighters have to be placed on *untouched* vertices, which are henceforth labelled as *protected*. Afterwards the labels of all *untouched* vertices adjacent to a *burnt* vertex are set to *burnt*, thereby ending iteration t. For each time step $t = 2, 3, \ldots, |V|$ D firefighters protect each a vertex $v \in V$ that is not burnt while the fire spreads around the graph. The process continues until some iteration t_l where no new vertex is labeled *burnt*. The optimization objective for this problem is to save as many vertices as possible from burning by distributing the firefighters in a strategically optimal way.

INSTANCE: Graph $G = (V, E)$, a set of vertices $B_{\text{init}} \subseteq V$, and a positive integer D.
OBJECTIVE: Maximize the set of (V_u)-unburnt vertices $V_u \subseteq V$.

1.3 Thesis overview

The rest of the thesis is structured as follows. In Section 2, Background, all the used algorithms are defined alongside a description of the architecture and the CUDA programming model for GPUs. This chapter also includes a brief presentation of the optimization

techniques. A short overview of related works is given in Section 4. In Section 5 the six algorithms for solving the FFP are presented. The performance results are discussed in Section 6. The final Section 7 gives a conclusion of the thesis.

CHAPTER 2

Background

This section gives a background for the topic. It describes the used algorithms and gives an overview of the the microarchitecture of modern graphics processing units (GPUs) and the Compute Unified Device Architecture (CUDA) programming model approach for their usage. The focus lies on the Nvidia graphics architectures and the terminology specific to their products. Yet the concepts discussed here are general and can be applied to any similar GPU architecture. The description is based on selected books [KH13, Far12, JC14, NVI17d] and papers [WKP11, HCZ16, MC17].

2.1 Optimization Algorithms

The existing methods for solving combinatorial optimization problems can be classified into exact and approximate methods [Tal09]. Exact methods guarantee to find global optimum solutions and their optimality for any given instance of an optimization problem. When applied to NP-hard problems, however, those methods entail (at least) exponential runtime. Examples of exact methods are Branch & Bound [LW66] and Dynamic Programming [Bel57]. Approximate methods can be divided into two subclasses: approximation algorithms and heuristic algorithms. An approximation algorithm produces solutions that are bound to the global optimum. In other words, it guarantees to obtain a solution quality that is within a certain range of the global optimum. Heuristics, on the other hand, provide no provable guarantees regarding the solution quality. Their performance may only be discovered empirically. However they offer the possibility to deal with large-size problem instances in that they produce solutions close to the global optimum in a reasonable time. The term metaheuristics refers to general heuristic methods that can be used for solving various optimization problems. Their advantage lies in the fact that relatively few adaptations need to be made to render them suitable for a particular problem. Some better known classifications include the following [Tal09]:

- Population-based vs. single-solution based search,
- Deterministic vs. stochastic methods,
- Iterative vs. greedy algorithms,
- Nature-inspired vs. non-nature-inspired,
- Memory-usage vs. memoryless methods.

The decision of whether to apply exact algorithms or heuristics depends on manifold factors. The criteria that are most widely agreed on are the complexity of the problem, the size of the instances. As the exact methods are solely appropriate for moderately sized instances of NP-problems, solving largely sized instances relies on the use of heuristics. For a more exhaustive description of metaheuristics we refer the interested reader to Talbi et al. [Tal09], Gendreau et al. [GP10] and Blum et al. [BPRR11].

In the following subsection the metaheuristics used for solving the FFP are introduced.

2.1.1 Ant Colony Optimization

Ant Colony Optimization(ACO) [DS04] is a nature-inspired metaheuristic based on the swarm behavior of ants. ACO algorithms are population-based, which means that a number of agents cooperate to find an optimal solution. Such algorithms may be applied to a wide range of problems, many of which are graph-theoretic.

Presented in a nutshell, ACO involves the following: Independent artificial ants, which are basically simple agents, generate solutions for a given problem instance in a probabilistic manner by following simulated pheromone trails. Pheromone trails enable indirect coordination between agents via their environment, a mechanism known as stigmergy [DBT00]. Stigmergy is a form of self-organization facilitating the ants to communicate indirectly through a pheromone data structure. This structure is updated, depending on the variant of the algorithm, during or after each ant has created a new solution.

In the past, there have been several variants of ACO. The better known ones are Ant System (AS) [DMC96], Max-Min Ant System (MMAS) [SH00] and Ant Colony System (ACS) [DG97]. The first proposed algorithm was AS. In this variant the pheromone trail structure is updated once all the ants have constructed a solution. In contrast to AS, the main improvement of MMAS lies in the fact that now only the best ant can update the pheromone trail and the pheromone levels are bounded. The Ant Colony System is another improvement over the original AS. In ACS pheromone updates are performed during the construction of a tour (by applying the local updating rule) as well as after all ants have constructed their tour (by applying the global updating rule).

The original formulation of the ACO is based on the traveling salesman problem (TSP) [Flo56]. The following description is slightly adapted for the purpose of this thesis, however, retaining its general aspect and applicability in other contexts.

As described in Algorithm 2.1 the AS algorithm consists of three main stages: Initialization, solution construction and pheromone update. First, an initialization phase, in which the pheromone trails are set to an initial value determined by the particular algorithm variant, is performed. Subsequently, solution construction followed by an optional local search and pheromone update are iterated until a termination condition is met.

Algorithm 2.1: Sequential AS version

```
1 initialize pheromone trails ;
2 while termination criterion is not met do
3 |   construct solutions using local heuristic and pheromone information ;
4 |   apply local search ;                                          // optional
5 |   update pheromone trails ;
6 end
```

In the construction phase, m ants are initialized with an empty solution. Based on the random proportional rule, which determines the probability of choosing a particular component j out of a set of possible solution extensions N_t^k by an ant $k \in [1, m]$ at time step t, each ant repeatedly chooses a new component at each time step of the construction phase. The probability is given in the following equation:

$$p_{tj}^k(i) = \frac{[\tau_{tj}(i)]^\alpha [\eta_{tj}]^\beta}{\sum_{l \in N_t^k} [\tau_{tl}(i)]^\alpha [\eta_{tl}]^\beta} \text{ if } j \in N_t^k, \tag{2.1}$$

where $\tau_{tj}(i)$ defines the quantity of pheromone being deposited on component j at time step t and iteration i. η_{tj} is the local heuristic value that entails problem-specific information for setting the ants on the right track. α and β are two parameters, which define the relative influences of the pheromone trail and the heuristic information. The numerator of Eq. 2.1 remains constant for every single ant in a single iteration within a solution construction, which improves efficiency by saving this information so as to make it available for all ants. In addition to this, in order to ensure that an ant chooses a component exactly once, each ant k maintains a data structure, M_k, called the *tabu list*, which consists of a chronological ordering of the components already chosen. This data is used to determine the feasible neighborhood, as well as to enable the calculation of the quality of the solution T an ant k generated and the reconstruction of the solution structure in order to deposit pheromone.

Once all the ants have finished constructing their solutions, the last phase of the algorithm takes place, the pheromone update. This phase has two stages, pheromone evaporation and pheromone deposit. To avoid falling into local optima, the pheromone level of every component is first evaporated by a constant factor ρ:

$$\tau'_{tj}(i) = (1-\rho)\tau_{tj}(i), \quad \forall \tau_{tj} \in \mathcal{T}(i), \tag{2.2}$$

where $0 < \rho \leq 1$ is a user-defined evaporation rate and $\mathcal{T}$ the pheromone set. Pheromone evaporation ensures that seldom selected components are forgotten over time. Following the evaporation stage, each ant performs a pheromone deposit for every chosen component.

$$\tau_{tj}(i+1) = \tau'_{tj}(i) + \sum_{k=1}^{m} \Delta\tau^k_{tj}(i), \tag{2.3}$$

where $\Delta\tau^k_{tj}(i)$ denotes the amount of pheromone ant k deposits. This is defined as follows:

$$\Delta\tau^k_{tj}(i) = \begin{cases} C^k(t) & \text{if component } (t,j)^k \text{ belongs to } T^k(i) \\ 0 & \text{otherwise,} \end{cases} \tag{2.4}$$

where $C^k(i)$ denotes the quality of the solution $T^k(i)$ of ant k.

In general, and as can be derived from Eq. 2.3 and 2.4, the process of updating pheromone levels ensures that a qualitatively better solution will result in a larger quantity of pheromone deposited on its components, which in turn increases the chance of the components being selected for the same positions by other ants in one of the next iterations (according to the random proportional rule).

2.1.2 Variable Neighborhood Search

The Variable Neighborhood Search (VNS) is a single-solution based metaheuristic for solving combinatorial and global optimization problems. It was initially proposed by P. Hansen and N. Mladenovic [MH97]. The basic principles can be described as two phases that are applied iteratively. In the first phase local search takes place in order to find the local optimum within a given neighborhood structure. The aim of the second phase is to escape from the local optimum by changing the given neighborhood stochastically or deterministically. The idea of the method goes back to the fact that different neighborhood structures may have different local optima [Jon95].

Naturally, this algorithm has also seen numerous developments and found application in various fields. This thesis applies the General Variable Neighborhood Search (GVNS), which uses the Variable Neighborhood Descent (VND) as its local search method.

Variable Neighborhood Descent

Variable Neighborhood Descent [HM99] is a deterministic version of the VNS algorithm, which uses a predefined ordered set of neighborhood structures N_k $(k = 1, ..., k_{max})$. The

first step is a random generation of an initial solution s_0 (line 1 in Algorithm 2.2). The subsequent step is to apply the local search starting with the first neighborhood (line 4). In case an improvement has been found, the search is reiterated with the improved solution replacing the initial solution while the neighborhood is reset to the first one. If no improvement has been found, the neighborhood is incremented from N_k to N_{k+1} and the process is repeated until termination criteria are met (e.g., the maximum neighborhood k_{max} is reached).

Algorithm 2.2: Variable Neighborhood Descent

Input: neighbourhood structure N_k $(k = 1, ..., k_{max})$
Output: best found solution s

```
1  s ← s_0;
2  k ← 1;
3  while k ≤ k_max do
4  |   find best s' ∈ N_k(s);
5  |   if s' is better than s then
6  |   |   s ← s';
7  |   |   k ← 1;
8  |   else
9  |   |   k ← k + 1;
10 |   end
11 end
12 return s;
```

General Variable Neighborhood Search

General Variable Neighborhood Search [HM01] is a combined deterministic and stochastic variant, which embeds VND as a local search method into the VNS. Hence, GVNS contains two different neighborhood structures N_k $(k = 1, ..., k_{max})$ and N_l $(l = 1, ..., l_{max})$. In contrast to VND, the GVNS contains three steps: shaking, local search and move, and these are repeated until termination criteria are met. In line 5 of Algorithm 2.3 shaking is applied by randomly generating an initial solution s' from the current neighborhood N_l. Then VND is applied with the generated solution s' in order to generate a new solution s''. If the newly found local optimum s'' is better than s (line 16), then s'' substitutes s. Otherwise, the algorithm generates a new solution randomly from the incremented neighborhood only to start the whole procedure anew. The termination criteria for the algorithm can be multiple: time limit, solution quality, total number of steps or a certain number of consecutive steps. A combination of several different termination criteria is possible and is often the case.

Algorithm 2.3: General Variable Neighborhood Search

Input: neighbourhood structures N_k $(k = 1, ..., k_{max})$, N_l $(l = 1, ..., l_{max})$
Output: best found solution s

```
1  s ← s_0;
2  while termination criteria are not met do
3  |  l ← 1;
4  |  while l ≤ l_max do
5  |  |  pick s' ∈ N_l(s) randomly;
6  |  |  k ← 1;
7  |  |  while k ≤ k_max do
8  |  |  |  find best s'' ∈ N_k(s');
9  |  |  |  if s'' is better than s' then
10 |  |  |  |  s' ← s'';
11 |  |  |  |  k ← 1;
12 |  |  |  else
13 |  |  |  |  k ← k + 1;
14 |  |  |  end
15 |  |  end
16 |  |  if s'' is better than s then
17 |  |  |  s ← s'';
18 |  |  |  l ← 1;
19 |  |  else
20 |  |  |  l ← l + 1;
21 |  |  end
22 |  end
23 end
24 return s;
```

2.2 Further Algorithms

Along with the main algorithms introduced in the previous section, there is a number of additional algorithms, which will be incorporated in the metaheuristics. These are described in this section. For the simulation of the fire spread and the evaluation techniques used in VNS we apply a Breadth-first search (BFS) algorithm, which requires exhaustive Scan (or prefix sum) calculations. Furthermore, a radix sort, which also requires Scan operations, is applied to generate neighborhood structures for the VNS algorithm.

2.2.1 Breadth First Search

The Breadth-first search (BFS) algorithm is widely used in graph theory. Some of its many applications are calculating shortest path with unit distances [KGBH16], searching

in peer-to-peer networks [KGZY02], establishing search components for crawlers in search engines [NW01], and searching in social networks [Fay16].

Algorithm 2.4 shows the basic structure of BFS for determining the distance from the source vertex to each vertex. Given is a connected graph $G = (V, E)$ and a source vertex v_s. In order to keep track of the progress, the algorithm starts with initializing a distance array of size $|V|$. This data structure has two functions, one is to save each node's distance (line 10) to the source vertex, the other is to serve as a tabu list in order to avoid cycling (line 9). In line 5 the source vertex v_s is added to a FIFO queue, which in literature is also referred to as frontier queue. After that, the algorithm starts with the examination of all adjacent vertices from the given vertex (line 7-13). The examination determines whether the neighboring nodes have already been visited. In case a node has not been visited, the node's distance is set and the node is added to the frontier queue. After this process, the queue holds the newly determined nodes, which are then used for the examination of the next level, meaning the distance from the source vertex. These nodes are taken as a source for locating the nodes of the next level. This process is repeated on each level until all reachable nodes are located.

Algorithm 2.4: Breadth First Search

Input: connected graph $G = (V, E)$, source vertex v_s
Output: distance array *dist* holding the distance of each vertex to v_s

```
1  foreach v ∈ V do
2  |  dist[v] ← -1;
3  end
4  dist[v_s] ← 0;
5  Q.Enqueue(v_s);
6  while Q ≠ ∅ do
7  |  v ← Q.Dequeue();
8  |  foreach v_n neighbor of v do
9  |  |  if dist[v_n] ≠ -1 then
10 |  |  |  dist[v_n] ← dist[v] + 1;
11 |  |  |  Q.Enqueue(v_n);
12 |  |  end
13 |  end
14 end
15 return dist;
```

2.2.2 Prefix Sums

Prefix Sums is an important parallel building block for numerous parallel applications such as: graph algorithms [BB15], machine learning [GLMN17], and sort algorithms [MG11]. The formulation and its realizable applications were first introduced by Belloch [Ble90].

This paper discussed two variants of Prefix Sums, *scan* (or *inclusive scan*) and *prescan* (or *exclusive scan*).

The *scan* operation is defined as follows [Ble90]:

Definition: The *scan* operation takes an array $[x_0, x_1, \ldots, x_n]$ with n elements, and a binary associative operator $\oplus$, and returns the array $[x0, (x_0 \oplus x_1), (x_0 \oplus x_1 \oplus x_2), \ldots, (x_0 \oplus x_1 \oplus, \ldots \oplus x_{n-1})]$.

Example: Given is an array $[3, 1, 7, 0, 4, 1, 6, 3]$ with 8 elements and an addition operator. The scan operation returns $[3, 4, 11, 11, 14, 16, 22, 25]$ as a result.

Algorithm 2.5: Scan

Input: array $in = [x_0, x_1, \ldots, x_n]$, size n, binary associative operator $\oplus$
Output: array $out = [x0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus, \ldots \oplus x_{n-1})]$

```
1 out[0] ← in[0];
2 for i = 0; i < n; i ← i + 1 do
3 |   out[i] ← in[i] ⊕ out[i − 1];
4 end
5 return out;
```

The *prescan* operation is defined as follows [Ble90]:

Definition: The *prescan* operation takes an array $[x_0, x_1, \ldots, x_n]$ with n elements, and a binary associative operator $\oplus$ with identity I, and returns the array $[I, x_0, (x_0 \oplus x_1), (x_0 \oplus x_1 \oplus x_2), \ldots, (x_0 \oplus x_1 \oplus, \ldots \oplus x_{n-2})]$.

Example: Given is an array $[3, 1, 7, 0, 4, 1, 6, 3]$ with 8 elements and an addition operator. The scan operation returns $[0, 3, 4, 11, 11, 14, 16, 22]$ as a result.

Algorithm 2.6: Prescan

Input: array $in = [x_0, x_1, \ldots, x_n]$, size n, binary operator $\oplus$, identity I
Output: array $out = [x0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus, \ldots \oplus x_{n-1})]$

```
1 out[0] ← I;
2 for i = 0; i < n; i ← i + 1 do
3 |   out[i] ← in[i − 1] + out[i − 1];
4 end
5 return out;
```

Sequentially, both algorithms can be easily implemented as described in Algorithm2.5 and Algorithm 2.7. However, it should be considered, that an efficient parallel version needs further effort for breaking down the sequential nature of the operations.

2.2.3 Radix Sort

Radix Sort is among the fastest sorting algorithms [SKC+10]. The algorithm takes a given sequence of size n and separates each key into d digits. These digits are then sorted according to their significance one digit at a time, either from least to most significant or from most to least significant. It is essential that the algorithm used for sorting the digits belongs to stable sorts like a counting sort or bucket sort [CSRL01]. The running time of radix sort is linear $\Theta(d(n+k))$, where n is the number of elements to be sorted, k is the base and d is the number of digits.

Algorithm 2.7 shows a simple variant of radix sort using a fixed binary representation $k = 2$. The algorithm starts with taking the least significant digit of the first key (line 7) distributing it to the corresponding bucket. These steps are applied for each key of the array. As a result of this phase, a newly sorted array is created by concatenating the first and second bucket, which forms the basis for the next iteration of the next significant digit. This process is repeated until the most significant digit of each key is sorted. After the final iteration, the array is completely sorted.

Algorithm 2.7: Radix Sort

```
   Input: array of integers a[n], length n, key-bit length d
   Output: sorted array a[n]
 1 for i = 0; i < n; i ← i + 1 do
 2 |   bucket_0[i] ← 0;
 3 |   bucket_1[i] ← 0;
 4 end
 5 for shift = 0; shift < d; shift ← shift + 1 do
 6 |   for i = 0; i < n; i ← i + 1 do
 7 |   |   res ← (a[i] ≫ shift) ∧ 1;
 8 |   |   if res = 1 then
 9 |   |   |   bucket_1[b1] ← a[i];
10 |   |   |   b1 = b1 + 1;
11 |   |   else
12 |   |   |   bucket_0[b0] ← a[i];
13 |   |   |   b0 = b0 + 1;
14 |   |   end
15 |   end
16 |   a ← concat(bucket_0, b0, bucket_1, b1);
17 |   b0 = 0;
18 |   b1 = 0;
19 end
20 return a;
```

2.3 GPU Architecture and the CUDA Programming Model

With the rapid evolution of microprocessor technologies, the current trend in computing is to increase parallelism rather than clock rate. Graphics Processing Units (GPUs) are specialized microprocessors that accelerate graphics operations. NVIDIA's newest TITAN V GPU contains up to 5120 cores, enabling the implementation of massive parallel programs. Due to the excessive number of cores, GPUs are suitable for creating supercomputers. For instance, the Titan Supercomputer built at Oak Ridge National Laboratory [TIT18] packs up whole 18,688 NVIDIA GPUs, which puts it on number 5 of the TOP500 list from November 2017 [Top17].

In order to develop efficient parallel programs on the GPU, the significantly different hardware design of the GPU needs to be taken into consideration. In comparison to the GPU, CPUs are designed with more complex control logic such as complicated branch prediction and prefetching and are therefore optimized for task-based computation. On the other hand, the design of GPUs is based on light-weight control logic and is optimized for highly data-parallel computations. Consequently, it is essential to have an in-depth knowledge of the underlying hardware in combination with the programming model. For this reason, this section describes the hardware design of the graphic card (based on the Maxwell architecture), in particular, the hierarchy of the memory, the programming model, and selected optimization techniques.

2.3.1 Hardware Architecture of NVIDIA's GPU

The basic architecture of the NVIDIA chip, given in Figure 2.1, can be described as an array of streaming multiprocessors (SMs) that share an L2 cache. The global memory interface is divided into partitions. The communication with the CPU is made possible by the Host Interface via PCI-Express. The scheduler is distributed in multiple levels. There is at least a global scheduler (GigaThread) at the chip level, which schedules thread blocks to the various SMs in arbitrary order. At the SM level, the thread blocks are divided into a collection of 32 parallel threads named warps. These warps are then scheduled by independent warp schedulers that handle finer-grained scheduling. This decoupled approach results in transparent scalability [1], but comes with limitations that will be discussed in 2.3.4. The capacity and its features may vary according to the graphic card type. NVIDIA's hardware resources and supported features are defined in the compute capability [NVI17d]. An example of the described architecture would be the Maxwell chip (see Fig. 2.1), which can track up to 2048 threads per SM simultaneously, which sums up to 16,384 concurrent threads for this chip.

[1] Transparent scalability refers to the executability of the same application code on varying hardware capacities.

Figure 2.1: Block diagram of NVIDIA's Maxwell chip [NVI16].

2.3.2 NVIDIA's Streaming Multiprocessor

Figure 2.2 depicts the Maxwell streaming multiprocessor (SMM). Its architecture is classified by NVIDIA as single-instruction, multiple-thread (SIMT) [LNOM08]. The SMM contains eight texture units and an on-chip shared memory. The SMM is further divided into four distinct 32 CUDA-core processing blocks, two of which share an instruction cache and a unified texture L1 cache. Each Block contains an instruction buffer, a pipeline where each has a warp scheduler as well as two dispatch units. Furthermore, there is a local register file. The 32 cores are sorted into 4 groups with each 8 units. Each of the 32 cores contains a separate integer arithmetic logic unit (ALU) and a floating point unit (FPU). The FPUs implement the IEEE 754-2008 floating-point standard. There are also 8 load/store units for memory operations and 8 Special Function Units (SFUs) to handle

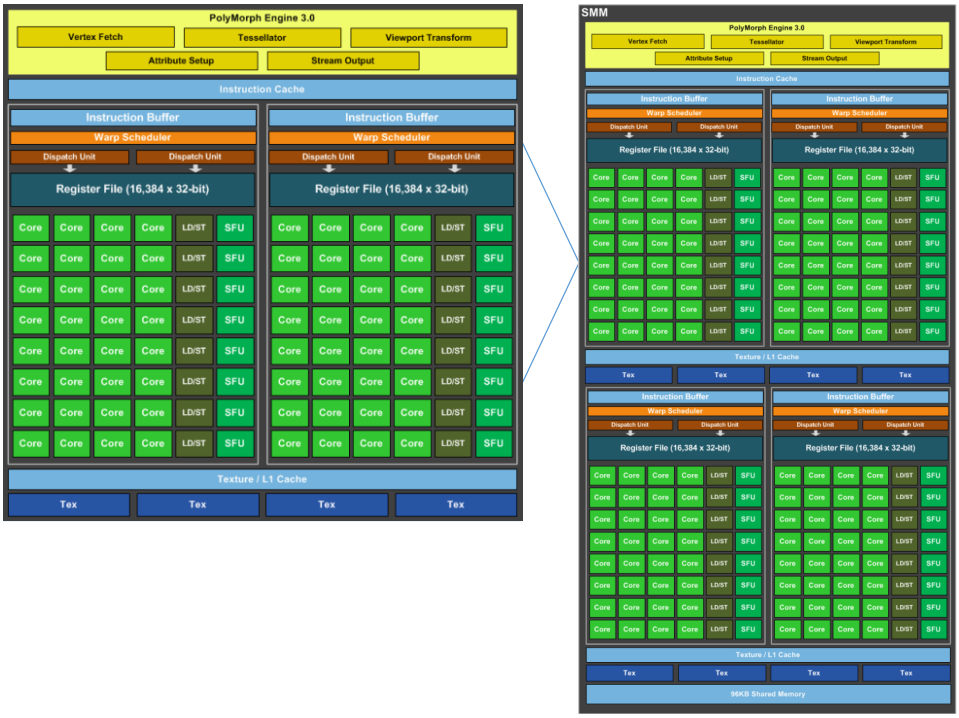


Figure 2.2: Maxwell Streaming Multiprocessor (SMM) (adapted from [NVI16]).

transcendental functions such as sinus, cosines, reciprocal etc.

2.3.3 CUDA Programming Model

The CUDA framework comprises a small set of extensions to various programming languages including ANSI C, C++, Python, and Fortran. The CUDA programming model is a relaxed variant of the Single Program, Multiple Data (SPMD) parallel computation paradigm [AF98]. In the context of the SIMT architecture, this means that the same instruction that operates on different data is executed on multiple threads. The main difference is that the SIMT architecture permits independent branching, that is, threads within the same warp can execute different control flows. However, this flexibility comes with the drawback in that it may lead to a serialization of the branches.

The programming model enables the development of heterogeneous programs, which can run simultaneously on the CPU (host) and GPU (device). A CUDA program therefore consists of a CPU Code and a GPU Code. The CPU code can launch or invoke GPU subroutines (so called parallel kernels) asynchronously, that are executed on the GPU with a large number of threads in SPMD style. Typical CUDA kernels are executed by thousand to millions of threads. Starting from CUDA 5.0 and compute capability 3.5, CUDA enables invoking kernels also from device threads dynamically at runtime. This

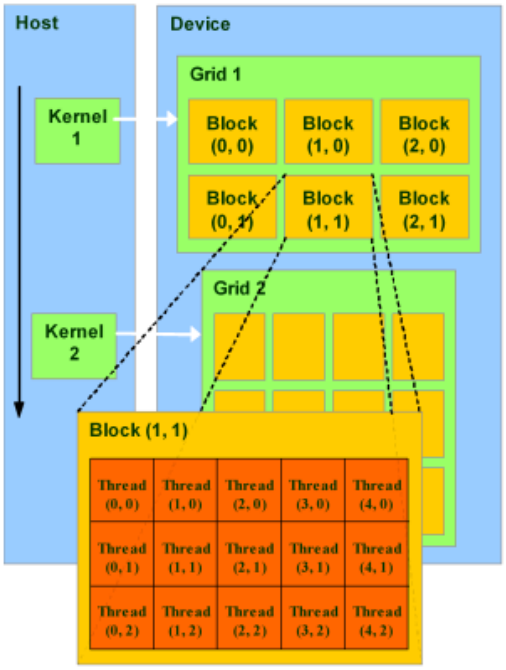


Figure 2.3: A schematic overview of the thread organization [Whi09].

feature is called CUDA Dynamic Parallelism (CDP) [NVI17d].

As can be seen in Figure 2.3, threads are organized into a two-level hierarchy. In general, multiple threads are grouped into 3 dimensional thread blocks. Further thread blocks are organized into 2 dimensional grids, where each grid executes a unique kernel. The exact organization of a grid and its thread blocks is set by the programmer or compiler. Thread blocks and threads have unique coordinates that enable to identify themselves and their domains at runtime. These coordinates, which are assigned by the CUDA runtime system, are also used to identify the appropriate memory locations.

As already described above, threads within a thread block run on a single SMM, so within a block threads have the ability to synchronize via barrier synchronization [2] and/or shared memory by using atomic operations. However, there is no CUDA support for threads in different thread blocks to synchronize with each other. This limitation comes from the decoupled schedulers and independent thread blocks. One possibility to achieve synchronization between thread blocks can be through global memory via atomic operations [XcF10] or L2 cache [LA16] by using (PTX) assembly instructions. Another approach, as described in [KH13], is to simply decompose a given problem into multiple kernels. That would mean each time to terminate the kernel and relaunch a new

[2]Barrier synchronization refers to synchronization primitives implemented in numerous parallel programming languages.

kernel after the synchronization point. It should, however, be considered, that approaches involving atomic operations on global memory are always connected with an increased latency. Furthermore, relaunching kernels incurs significant overhead.

2.3.4 CUDA Memory Model

CUDA threads have access to different memory spaces with different scope, size, lifetime and latency. Understanding their properties is essential for designing efficient algorithms for GPUs.

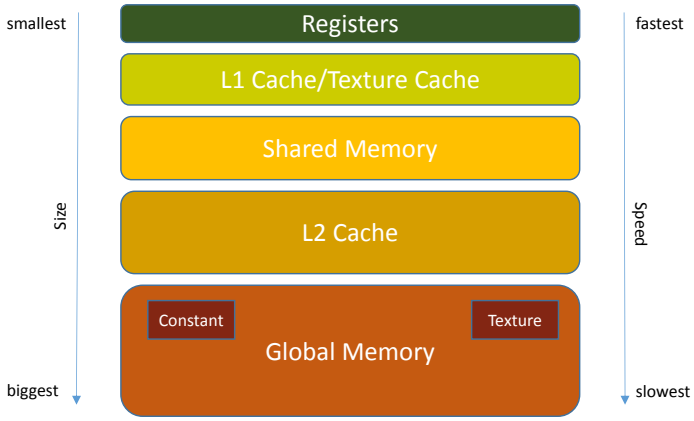


Figure 2.4: An overview of the memory hierarchy (adapted from [JC14]).

Registers

Registers are the fastest memory on the GPU. All automatic [3] local scalar variables declared in device subroutines are automatically placed into registers by the compiler. Non-scalar [4] automatic variables can be placed on registers or local memory. Local memory is an abstract logical memory type defined by NVIDIA for transparency purposes. The implementation of the local memory depends on the architecture of the GPU. For instance, the Maxwell architecture encompasses a local memory range from L1 to Global Memory. The local memory is only used in the following cases:

1. If very large automatic arrays are allocated or regular arrays with dynamic indexes [5] are declared.
2. If register spilling occurs, i.e., in case more registers are required than available, the variables will be hierarchically assigned to the local memory beginning with L1 cache [JC14].

[3] Automatic variables are declared without specific CUDA qualifiers like *__constant__* or *__device__* etc.
[4] Variables that are not arrays are referred to as scalar variables in the literature.
[5] A dynamic index is one that the compiler cannot determine at compile time.

The scope of both variables, scalar and non-scalar, is limited to the threads within a block.

L1 Cache/Texture Cache

The L1 cache and the texture cache is the same physical unit. By default it serves as a cache for local memory accesses. However, it can be configured by the programmer in order to act as a cache for global memory access loads with 32 Byte load granularity in texture cache or 128 Byte load granularity in L1 cache. The choice of configuration depends on the given application. A latency bound application may benefit from increasing the cache hit rate instead of increasing occupancy, that is, parallelism. In other words, a higher cache hit rate may hide the latency more efficiently than an increased number of threads. In the case of a misaligned or unpredictable global memory access a shorter cache-line is more effective than a longer cache-line. On the other hand, a longer cache-line offers higher performance, if the data is accessed in coalesce manner. In opposition to the L2 cache, the L1 cache is not coherent, i.e., it does not reliably display the latest status of the variables it contains, and is designed for spatial locality.

Shared Memory

The on-chip shared memory has a significantly higher bandwidth and lower latency than global memory and has approximately 100x faster latency. Shared memory enables inter-thread communication as threads within a block have access to the same shared memory. The amount of shared memory is defined at the kernel launch time but its scope is limited to the threads within a thread block. Its latency period is 28 cycles [MC17].

L2 Cache

The L2 cache is a coherent read/write cache with a Least Recently Used (LRU) replacement policy [MC17]. It is directly connected to the global memory (Figure 2.4). All transactions to global memory go through L2, including copies to/from CPU host. Further, the cache is completely transparent to the device code. There are only some compiler options for optimization purposes.

Global Memory

The global memory is situated underneath the L2 cache. The off-chip GDDR5 DRAM is the largest and slowest memory unit in the hierarchy. It has the highest latency starting at 230 up to 2766 cycles and the smallest throughput [MC17]. Its cells can be accessed on the device from any SM throughout the lifetime of the application.

2.3.5 Performance Considerations

This section focuses on execution optimization. It deals with a few of the best practice optimization techniques for writing efficient parallel programs as described in [TGEF11,

NVI17b] concentrating on the most important techniques.

Coalesced Access to Global Memory

The global memory is transferred in transactions on the GPU. To maximize the bandwidth and to keep the latency low, each thread in a warp should always try to access a continuous segment in the global memory. If this is successful, the individual queries are combined into as few transactions as possible. The size of the transaction depends on the cache line. Otherwise the queries are worked out sequentially in which case the number of transactions depends on the memory access pattern as well as on the cache line. For scattered access patterns, to reduce overfetch, it can sometimes be useful to enable caching in L1, which caches shorter 32-byte segments.

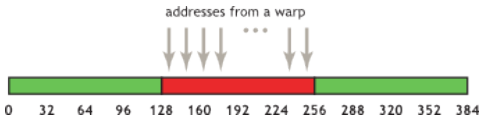


Figure 2.5: Coalesced access - all threads access one cache line [TGEF11].

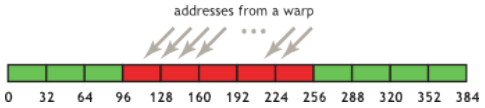


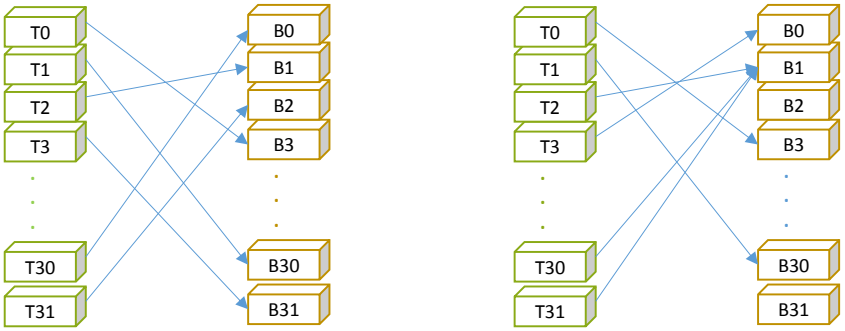
Figure 2.6: Misaligned sequential addresses that fall within five 32-byte L2-cache segments [TGEF11].

Shared Memory Accessing

The shared memory is subdivided into equally sized banks. One bank can be used by only one thread at the same time. Access can be provided in parallel only if all threads refer to different banks. If different threads refer to words within the same bank, the access is serialized causing a so called bank conflict. Figure 2.7a shows conflict-free access where all threads refer to different banks. In this case, the permutation is irrelevant and therefore serviced simultaneously. Figure 2.7b shows conflict-free broadcast access if threads 2, 3, 30, and 31 access the same word within bank B1. Otherwise this can cause up to a four-way bank conflict, depending on how many distinct words within this bank are accessed.

Occupancy

Occupancy is an indicator that is used for hiding latency. Basically, the idea is to use the physical characteristics of the GPU optimally. Occupancy is defined as the number of active warps divided by the maximum number of active warps. A warp is considered active if its corresponding block including the resources (e.g., registers and shared memory) has been allocated. The factors with potential for limiting occupancy



(a) Conflict-free access. (b) Broadcast or four-way bank conflict.

Figure 2.7: Shared memory access patterns.

are: register usage, shared memory usage and block size [Vol10]. However, keeping the occupancy high does not necessarily mean that the latency is completely hidden; some further performance limiters, such as low cache hit rates causing high latency, may need further consideration [Ton15].

Branching and Divergence

As previously discussed in subsection 2.3.1, thread blocks are divided into a collection of 32 parallel threads called warps. Due to the SIMT architecture, all threads within a warp share a single program counter. This means that any control flow instructions [6] may lead to the serialization of the branches. Thus, for higher performance the divergent warps should be kept to a minimum.

[6] In computer science, control flow (or alternatively, flow of control) refers to the order in which the individual statements, instructions, or function calls of an imperative or a declarative program are executed or evaluated [con].

CHAPTER 3

Related Work

This chapter provides an overview of the existing literature on the firefighter problem as well as the most recent and relevant parallel algorithms used in this thesis.

3.1 The Firefighter Problem

An extensive survey focusing on theoretical aspects of the FFP was conducted by Finbow et al. [FM09]. This study provides an overview of the many variants of the existing algorithms and their complexity, as well as various open problems and possible future developments.

Several studies have investigated the complexity of the FFP showing on different graph structures that even for a single firefighter NP-completeness is established. MacGillivray and Wang [MW03] proved NP-completeness for bipartite graphs, whereas Finbow et al. [FKMR07] showed NP-completeness for trees of maximum degree three. Building on the results of the previous article, King and MacGillivray [KM10] established NP-completeness for cubic graphs. Those complexity results indicate the need for heuristic approaches such as ACO and VNS, which are the topic of this work.

Bazgan [BCR13] analyzed instances where firefighters are greater than one. In addition, Costa [CDD$^+$13] studied the case where a fire breaks out on multiple nodes. Cygan et al. [CFvL11] examined various parameterized versions of the FFP on various graphs determining their complexity. This study showed that, when parameterized by the number of burned vertices, the FFP is fixed-parameter tractable on general graphs.

Further studies focused on the containment of fire on grids. In 2002, Fogarty [Fog03] determined that a finite outbreak of fire is controllable for grids of dimension two, with more than two firefirefighers. Feldheim and Hon [FH13] provided lower and upper bounds of firefighters required for fire containment on planar grids. For fire containment on

d-dimensional square grids, Develin and Hartke [DH07] showed that the presence of $2d-1$ firefighters per time step is necessary.

Cai and Wang [CW09] introduced the definition of surviving rates, which is the average percentage of unburnt vertices in cases of fire breakouts on random nodes. Wang et al. [WFW10] analyzed the surviving rates of virus spread for planar networks, series-parallel networks and d-degenerate networks. Subsequently, Kong et al. [DH07] showed improved results for planar grids.

A separate study focusing on approximation algorithms for trees proposed a $\frac{1}{2}$-approximation greedy algorithm [HL00]. Hartke [Har04b] developed linear programming relaxation algorithms on trees based on the integer program approach of MacGillivray and Wang [MW03]. Furthermore, Cai et al. [CVY08] proposed a polynomial-time $(1-\frac{1}{e})$-approximation algorithm, which Iwaikawa et al. [IKM11] later improved by implementing enumeration and backward induction techniques.

Recently, metaheuristic approaches for the FFP have been examined in different studies. Blum et al. [BPRR11] developed a MAX–MIN Ant System (MMAS) and a hybrid variant with a mathematical programming solver (CPLEX). An alternative general variable neighborhood search (VNS) approach combined with variable neighborhood descent (VND) using a more compact representation was presented by Hu et al. [HWR15]. García-Martínez et al. [GBRL15] presented nine Integer Linear Programming (ILP) formulations along with six heuristic approaches on random graphs.

Lately, several extensions to the FFP have been presented. An evolutionary algorithm was introduced to solve a multi-objective version of the firefighter problem [Mic14]. A further extension is the Bi-Firefighter Problem introducing an additional spreading element occurring independently [Lip17]. Finally, a nondeterministic model for the fire spread was introduced in the study of Krzysztof and Knowles [MK16].

3.2 Ant Colony Optimization

An in-depth survey on different parallel ACO implementations strategies along with a new taxonomy for their classification can be found in the study of Pedemonte et al. [PNC11].

A number of preliminary parallel implementations on the GPU used a coarse grained (or task-based) approach based on the work of Stützle [Stü98]. In the context of GPU computing, this means that each ant is mapped directly to a single thread. An entire MMAS implementation [BOL+09] on the GPU for solving the Travelling Salesman Problem (TSP), which is based on the coarse grained approach, achieved competitive results. However, the speed-up factor did not exceed 2 in contrast to a sequential CPU implementation. Alternatively, a heterogonous task-based MMAS variant [WDZ09] for solving the TSP mixing CPU and GPU function calls for the different stages of the algorithm. The achieved speed-up factor was 1.5 when tested against the sequential variant on the CPU.

Cecilia et al. [CGN+13] concluded that the task-based approach is essentially unfitting for the GPU. For improvement of the GPU utilization, a novel data-parallel approach has been introduced in addition to a systematic analysis of efficient strategies for implementing ACO for the TSP. The algorithms were tested using a standard subset of benchmark instances from the TSPLIB library [Rei91]. The speed-up factor achieved at the tour construction was up to 21 while the speed-up factor at the pheromone update was approximately 20 compared to a sequential CPU implementation.

An approach similar to Cecilia et al. [CGN+13] in which single ants are assigned to CUDA Blocks was implemented by Del evacq et al. [DDGK10].

In contrast to the previously presented papers, Uchida et al. [UIN12] did not choose a data-parallel approach, but decomposed the problem into several kernels. Their implementation aimed at the maximization of the global memory bandwidth on the GPU by dynamically and efficiently rearranging the various data. Compared to a sequential counterpart, this approach achieved a speed-up factor of 43.47 overall performance.

Based on the work of Cecilia et al., Dawson and Stewart [DS13] proposed an adapted tour construction implementation named Double-Spin Roulette (DS-Roulette), which achieved speed-up factors up to 8.5 in comparison to the study of Cecilia et al.

Recently, Rafal Skinderowicz [Ski16] has proposed three novel parallel ACS implementations for the TSP on the GPU based on the data-parallel approach. At the onset, the algorithm computes a static candidate list with a maximum capacity of 32 elements containing the nearest neighboring nodes for each node. This limitation enables a computation of the fitness proportionate selection on a single warp and its efficient warp functions. The performance was tested on several TSP instances selected from the TSPLIB library. The performance evaluation showed a maximum speed-up factor of 24.29 for the fastest variant, however, the variant with a solution quality approximately identical to the sequential version reaches speed-up factors up to 6.43, when compared to the CPU implementation.

3.3 Variable Neighborhood Search

Several strategies for parallel Variable Neighborhood Search are presented in the study of Moreno-Pérez et al. [PHM05]. However, studies of parallel VNS on the GPU have been seldom conducted.

Thé Van Luong et al. [LMT10] studied different neighborhood structures for Local Search (LS) algorithms on the GPU. Additionally, the authors presented a template for LS algorithms focusing on assigning techniques between different neighborhood structures and threads. This approach was evaluated with three neighborhoods of variable sizes for binary problems, resulting in a speed-up factor up to 26.3.

The previous study was extended by proposing a methodology for implementing Local Search Metaheuristics on GPU [LMT13]. The work concentrates on different aspects

of the following GPU implementation techniques: optimization of data transmission between the CPU and GPU, efficient assignment strategies for neighborhood structures and threads, memory management, and occupancy maximization using dynamic heuristics for optimal configuration of threads per block and the total number of blocks.

In a recent study, Nikolaos Antoniadis and Angelo Sifaleras proposed a hybrid CPU-GPU implementation [AS17] of the VNS for inventory optimization problems, which uses a combination of OpenMP [1] [DM98] on the CPU and OpenACC [2] [CAP11] on the GPU. Without presenting the speed-up, the result of their work reported higher quality solutions achieved by the hybrid implementation in comparison to the sequential approach.

3.4 Breadth First Search

The first approach for developing BFS on the GPU using CUDA was introduced by Harish and Narayanan [HN07]. Their work used a static task-based approach, meaning that each vertex is statically assigned to a thread. This paper reports a speed-up factor of 20–50 over the single-threaded variant for random input graphs, which have $O(n^2 + m)$ work complexity [3] for BFS where n is the number of vertices and m is the number of edges. Nevertheless, with the work complexity, the speed-up drops to a minimum when using real world data. The reason is twofold: firstly, the static mapping approach leads to load imbalance when the graph is irregular, and secondly, due to the absence of memory optimizations, the implementation is subjected to high latency memory access.

These characteristics were observed by Hong et al. [HKOO11] who thus proposed a novel virtual warp-centric programming method. Instead of mapping single threads, the method assigns a number of threads $v \in \{x \mid x = 2^n, n \in \mathbb{N}\}$, referred to as virtual warps, to single vertices statically. This approach improved GPU utilization by decreasing load imbalance and increasing coalesced memory access, which in sum results in a speed-up factor up to 15.1, compared to the implementation of Harish and Narayanan.

A linear work complexity approach [LWH10] was achieved by applying a hierarchical queue management technique and a three-layer kernel arrangement strategy. Yet, the maximum achieved speed-up factor was measured 10.3.

Merrill et al. [MGG15] introduced a semi-dynamic mapping approach [BB17a] that achieved asymptotically linear runtime. They were the first to apply a parallel prefix-scan for calculating the number of vertices for the inspection on the upcoming levels. Since their proposal, the parallel prefix-scan has been used as a basis for frontier propagation in numerous recent BFS implementations [FDB+14, LH15, BB15].

[1] A shared memory parallel programming model.

[2] A parallel programming model for many-core and multi-core processors.

[3] The work complexity of a multithreaded algorithm is the sum of the processing time (i.e., the sequential work) of each thread.

Furthermore, distributed BFS algorithms on GPU Clusters are proposed in [BBM16, BCM+15, FDB+14], and various BFS algorithms on heterogeneous platforms are presented in [RGAN16, DNM14].

3.5 Prefix Sums

The existing attempts at designing parallel prefix sums on the GPU follow the same parallelization model. The first step of the model is to segment a given array, which is then distributed to different CUDA blocks. The second step is to apply a composite of intra-block and inter-block computation.

Whereas Harris et al. [HSO07] implemented the first parallel prefix sum (scan) with linear work complexity on CUDA, Sengupta et al. [SHZO07] proposed the first parallel segmented scan on CUDA. Based on these two works, Sengupta et al. [SHG08] introduced a new three-phase approach. In the first phase, a given array with size n is decomposed into subsets s_i of size b and distributed to CUDA blocks. Then, an initial intra-block computation in form of local [4] scan on each block is performed. Subsequently, the last element of each scanned s_i is inserted in a new array A of size $\lceil n/b \rceil$. In the following phase, an exclusive scan operation on A is performed. Finally, the resulting individual sums of A are added to each corresponding s_i to complete the scan operation in its last phase. This implementation requires a minimum of $4n$ global memory accesses; in addition, a global barrier synchronization is necessary to complete each phase.

A novel matrix-based scan reducing the number of global memory access to $3n$ was proposed by Dotsenko et al. [DGS+08]. The three-phase approach starts with a reduction operation, and continues with an exclusive scan using the results of the aforementioned reduction. Thus, in the final phase, the partial sums of the preceding operation are scanned with the corresponding subsets of the input array.

Shengen Yan et al. [YLZ13] evade global barrier synchronization and further reduce the number of global memory access to $2n$. Their proposed implementation starts with an intra-block reduction where each block b_i reduces its subset s_i. Afterwards, an inter-block computation is performed by sequentially passing the sum r_i of each block b_i plus the accumulated result r_{i-1} of the previous block b_{i-1} to the adjacent block b_{i+1}. The interaction between the thread blocks (inter-block synchronization) is achieved via busy waiting on global memory. Finally, to complete the scan operation, a last intra-block scan is performed after the synchronization phase.

The most recent implementation [LA16] applies the fastest features of the NVIDIA Kepler [NVI12] architecture. Liu et al. use warp shuffle functions for the intra-block computations and apply parallel thread execution (PTX) assembly instructions on the coherent L2 cache for enhancing efficient inter-block synchronization. This approach outperformed the leading GPU libraries CUDPP [HSO07], NVIDIA Thrust [BH12],

[4] A local operation is one in that only local resources like register and shared memory are used.

NVIDIA ModernGPU [NMG17], NVIDIA CUB [NVI17a] as well as the Intel Threading Building Blocks (TBB) [Phe08].

3.6 Radix Sort

The first implementation on parallel radix sort on the GPU was presented by Harris et al. [HSO07] who was the first to apply parallel scan algorithms for radix sort. The algorithm starts with decomposing an input array into subsets and separating each of its keys into single bits. Then, the subsets are sorted bitwise in three consecutive operations: binary splits, prefix sums, and scatter. After the sorting phase, the elements of the aforementioned subsets are combined by applying a parallel bitonic sort [Bat68].

A most significant digit (MSD) radix sort on the GPU was proposed by He et al. [HGLS07]. Similar to Harris et al., the input data gets partitioned into subsets. However, instead of sorting single bits, they sort a constant size of five bits at a time for the reduction of scatter operations on global memory. In comparison to the sequential implementation, the overall performance of their approach achieved a speed-up factor of 2.

Satish et al. [SHG09] separate each key into 2^b digits and perform a counting sort on each subset iterating 1-bit split b times in shared memory. The sorting results are then saved in a histogram h of size 2^b. Following this, a prefix sum is performed to accumulate the partial sums of h. Then, the sorted elements are distributed to their corresponding block. The whole procedure is repeated until all digits are sorted. The approach reached a speed-up factor of 2 in comparison to previous GPU implementations.

Further strategies that aim for a maximal utilization of the GPU were developed by Merrill et al. [MG11]. The presented methods comprise optimized split operations and reduced data movement through global memory, for which the number of kernel calls were decreased by applying kernel fusion and thread block serialization. The runtime of this implementation was 3.8 times faster than existing GPU sorting algorithms.

In 2017, Elias Stehle and Hans-Arno Jacobsen proposed a heterogeneous implementation of MSD radix sort [SJ17]. Their approach combines counting sort with local sort that enables the sorting of eight bits per pass, thus reducing the transfer of data by a factor not less than 1.6. Furthermore, for sorting instances too large for the GPU, they proposed a heterogeneous approach, where the input array is divided into several subsets, which are sorted on the GPU individually. During this process, the sorted subsets are returned to the CPU where they are merged, which enables a partially simultaneous computation on both architectures.

CHAPTER 4

Implementation

The central aim of this thesis is the design of a parallel hybrid metaheuristic for the firefighter problem. In Section 4.1 a hybrid encoding is presented, which is used for the development of the algorithms. The sequential algorithms are introduced in Section 4.2 whereas the subsequent Section 4.3 provides the description of the parallelization of the sequential approach.

4.1 Solution Representation

The hybrid approach alternates between two encodings during its various phases. For the generation of the solutions for the MMAS and the VNS we follow the strategy of Hu et al. [HWR15] and use a bitvector encoding $\mathcal{P} = \langle p_1, \ldots, p_n \rangle$, where

$$p_v = \begin{cases} 1 & \text{if vertex } v \text{ should be protected,} \\ 0 & \text{otherwise,} \end{cases} \qquad \forall v \in V \tag{4.1}$$

and $n = |V|$. The main reason behind this choice of encoding lies in its compactness, as the solution space is reduced in comparison to a permutation based representation. Furthermore, it enables the use of more compact data structures for the computations.

The encoding does not provide explicit information on which vertices are to be protected at a particular time step. Consequently, for the pheromone update stage the binary representation needs to be decoded into an explicit solution representation containing the missing time information. The decoding function $g : G \times \mathcal{P} \mapsto \mathcal{S}$ takes a graph $G = (V, E)$ and an encoding $\mathcal{P}$ and maps the pair to an encoding $\mathcal{S}$.

The representation $\mathcal{S} = \langle s_1, \ldots, s_n \rangle$ is defined as follows:

$$s_v = \begin{cases} t & \text{if vertex } v \text{ should be protected at ime step t,} \\ 0 & \text{otherwise.} \end{cases} \qquad \forall v \in V \tag{4.2}$$

The mapping can be achieved by simulating the fire spread over the graph while saving the time steps t of the protected nodes in $\mathcal{P}$. This process can be accomplished implicitly during the evaluation function in the VNS and hence an explicit decoding is not necessarily required.

4.2 Sequential Implementation

The presented algorithm combines a $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System (MMAS) with a General Variable Neighborhood Search (GVNS) implemented with a Variable Neighborhood Descent (VND) as local search. The MMAS is based on the work of Blum et al. [BBGM$^+$14] implemented in the Hyper-Cube Framework (HCF) [BRD01, BD04]. The VNS implementation is based on the work of Hu et al. [HWR15].

4.2.1 Sequential Hybrid ACO

Algorithm 4.1 shows the basic structure of the sequential hybrid approach. The algorithm saves three different solutions: the iteration-best solution $\mathcal{S}_{\text{ib}}$, the restart-best solution $\mathcal{S}_{\text{rb}}$, and the best-so-far solution $\mathcal{S}_{\text{bs}}$. At the onset all the variables and the pheromone values are initialized (lines 2-3). In the second step, each ant constructs a solution applying the ConstructACOSolution() method. After that, depending on the current solution P_{cr} and the threshold defined by parameter $B \in [0, 1]$, the solution may be improved by applying the ConstructVNDSolution(P_{cr}) procedure. During this phase the solution is implicitly transformed by mapping $\mathcal{P}_{\text{cr}}$ to $\mathcal{S}_{\text{cr}}$. If VNS is not applied, the transformation has to be done explicitly (line 10). The three saved solutions $\mathcal{S}_{\text{ib}}$, $\mathcal{S}_{\text{rb}}$, and $\mathcal{S}_{\text{bs}}$ are then updated in case a better solution was found by the previous construction process. After the construction phase, the ApplyPheromoneUpdate(*cl*, *bs_update*, $\mathcal{T}$, $\mathcal{S}_{\text{ib}}$, $\mathcal{S}_{\text{rb}}$, $\mathcal{S}_{\text{bs}}$) method is applied, which serves to update the pheromone values using $\mathcal{S}_{\text{ib}}$, $\mathcal{S}_{\text{rb}}$, and $\mathcal{S}_{\text{bs}}$. Then, based on the pheromone values, a new convergence factor *cl* is computed in the ComputeConvergenceFactor($\mathcal{T}$) method. This factor in combination with the Boolean variable *bs_update* defines whether the pheromone updates and the restart-best solution are to be reset to their initial value. These steps are iterated until some predefined termination conditions are met.

The salient methods of this algorithm are further described in the following subsections.

4.2.2 ACO Solution Construction

In order to reduce the search space, Algorithm 4.2 uses a dynamic candidate set for the node selection. This set contains solely nodes that may at least postpone the fire spread. To determine the set, the topology of the graph is analyzed at each time step. Based on the current state of a given graph (burnt and protected nodes) the algorithm simulates how the fire spreads over the remaining nodes. During this procedure metrics, i.e., the indegree and the time step of a node catching in the simulation, are collected for each node, which are then used to create the set and calculate the heuristic information.

Algorithm 4.1: Sequential hybrid ACO for the firefighter problem (based on [BBGM+14])

Input: An undirected graph $G = (V, E)$, number of ants m, number of firefighters D, and parameter B
Output: The best found solution $\mathcal{S}_{\text{bs}}$.

1 $\mathcal{S}_{\text{rb}} \leftarrow \varnothing$, $\mathcal{S}_{\text{bs}} \leftarrow \varnothing$, $cf \leftarrow 0$, $bs\_update \leftarrow$ false ;
2 $\tau(t, j) \leftarrow 0.5, \quad \forall (t, j) \in \mathcal{T}$;
3 **while** *termination criterion is not met* **do**
4 $\mathcal{S}_{\text{ib}} \leftarrow \varnothing$;
5 **for** $i = 0; i < m; i \leftarrow i + 1$ **do**
6 $\mathcal{P}_{\text{cr}} \leftarrow$ ConstructACOSolution() ;
7 **if** $f(\mathcal{P}_{cr}) > (f(\mathcal{S}_{bs}) \cdot B)$ **then**
8 $\mathcal{S}_{\text{cr}} \leftarrow$ ConstructVNDSolution($\mathcal{P}_{\text{cr}}$) ;
9 **else**
10 $\mathcal{S}_{\text{cr}} \leftarrow g(\mathcal{P}_{\text{cr}})$, where $g : \mathcal{P}_{\text{cr}} \mapsto \mathcal{S}_{\text{cr}}$;
11 **end**
12 **if** $f(\mathcal{S}_{cr}) > f(\mathcal{S}_{ib})$ **then** $\mathcal{S}_{\text{ib}} \leftarrow \mathcal{S}_{\text{cr}}$;
13 **if** $f(\mathcal{S}_{cr}) > f(\mathcal{S}_{rb})$ **then** $\mathcal{S}_{\text{rb}} \leftarrow \mathcal{S}_{\text{cr}}$;
14 **if** $f(\mathcal{S}_{cr}) > f(\mathcal{S}_{bs})$ **then** $\mathcal{S}_{\text{bs}} \leftarrow \mathcal{S}_{\text{cr}}$;
15 **end**
16 ApplyPheromoneUpdate(cf , bs_update, $\mathcal{T}$, $\mathcal{S}_{\text{ib}}$, $\mathcal{S}_{\text{rb}}$, $\mathcal{S}_{\text{bs}}$) ;
17 $cl \leftarrow$ ComputeConvergenceFactor($\mathcal{T}$) ;
18 **if** $cf > 0.99$ **then**
19 **if** $bs\_update =$ true **then**
20 $\tau(t, j) \leftarrow 0.5, \forall (t, j) \in \mathcal{T}$;
21 $\mathcal{S}_{\text{rb}} \leftarrow \varnothing$;
22 $bs\_update \leftarrow$ false ;
23 **else**
24 $bs\_update \leftarrow$ true ;
25 **end**
26 **end**
27 **end**
28 return $\mathcal{S}_{\text{bs}}$;

This process is formalized in Algorithm 4.3 and Algorithm 4.4. For the creation of the candidate set two procedures, a top-down BFS and bottom-up BFS, are applied consecutively. Algorithm 4.3 presents the top-down approach, which is responsible for the calculation of metrics and the simulation of fire spread. For each remaining time step t the algorithm creates a set of nodes that are reached at t (lines 5-8) and calculates the indegree of each of them (lines 9-11). The design of Algorithm 4.4 considers two purposes: to create the candidate set $\mathcal{C}$ and calculate the heuristic information η using the collected information of the previous BFS. Starting from the latest time step $i = t_{\max}$,

Algorithm 4.2: ConstructACOSolution()

Input: An undirected graph $G = (V, E)$, a set B_{init}, number of firefighters D.
Output: Solution $\mathcal{P}_{\text{cr}}$.

```
1  t ← 1 ;
2  burningNodes ← |B_init| ;
3  while burningNodes > 0 do
4  |   ApplyTopDownBFS() ;
5  |   ApplyBottomUpBFS() ;
6  |   PlaceFirefighters() ;
7  |   burningNodes ← PropagateFire() ;
8  |   t ← t + 1 ;
9  end
```

Algorithm 4.3: ApplyTopDownBFS()

Input: Current time step t, frontier set F containing n burning nodes at t, tabu list *tabu* containing burnt and protected nodes.
Output: Distance-indexed list of sets of nodes *dist*, indegree list *indeg*, maximum time step $t_{\max}$.

```
1  dist[t] ← F ;                          // dist[t][0...n-1] ← v,  ∀v ∈ F
2  while dist[t] ≠ ∅ do
3  |   foreach v in dist[t] do
4  |   |   foreach v' neighbor of v do
5  |   |   |   if tabu[v'] = false then
6  |   |   |   |   dist[t+1] ← dist[t+1] ∪ v' ;
7  |   |   |   |   tabu[v'] ← true ;
8  |   |   |   end
9  |   |   |   if v' ∈ dist[t+1] then
10 |   |   |   |   indeg[v'] ← indeg[v'] + 1 ;
11 |   |   |   end
12 |   |   end
13 |   end
14 |   t ← t + 1 ;
15 end
16 t_max ← t − 1 ;
```

the algorithm traverses through the graph until time step $t + 1$. During each time step the algorithm calculates the theoretically free firefighters γ_{i-1} for $i - 1$ (line 3). After that, the algorithm inspects the adjacent nodes v' from each node v at time step i (lines 4 -13). If the distance of v' is $i - 1$, the value of the local heuristic $\eta[v']$ is incremented by the heuristic value of v. Since it is difficult to prevent nodes with high indegrees from catching fire, the algorithm weights the heuristic information additionally by adding a

Algorithm 4.4: ApplyBottomUpBFS()

Input: Distance-indexed list of sets of nodes *dist*, indegree list *indeg*, current time step t, maximum time step $t_{\max}$, tabu list *tabu* containing burnt and protected nodes.

Output: Candidate set $\mathcal{C}$, heuristic set η.

```
1  i ← t_max ;
2  while i > t + 1 do
3  |  γ_{i−1} ←calculate available firefighters for time step i − 1 ;
4  |  foreach v in dist[i] do
5  |  |  foreach v′ neighbor of v do
6  |  |  |  if v′ ∈ dist[i − 1] then
7  |  |  |  |  η[v′] ← η[v′] + η[v] + 1/indeg[v] ;
8  |  |  |  |  if indeg[v] ≤ γ_{i−1} and tabu[v′] = false then
9  |  |  |  |  |  C ← C ∪ v′ ;
10 |  |  |  |  |  tabu[v′] ← true ;
11 |  |  |  |  end
12 |  |  |  end
13 |  |  end
14 |  end
15 |  i ← i − 1 ;
16 end
```

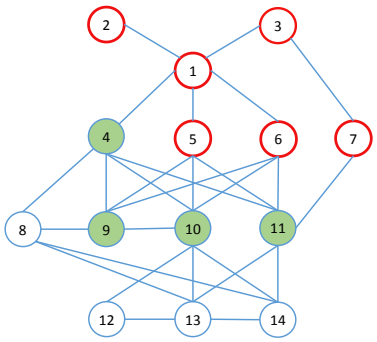
weight inversely proportional to the indegree of v (line 7). Consequently, nodes with neighbours of high indegree are less likely to be selected for protection. Whether an adjacent node v' is added to $\mathcal{C}$ is determined by the indegree of v (line 8). If the indegree of $v \leq \gamma_{i-1}$, v' is added to $\mathcal{C}$. This strategy excludes nodes that have no potential to prevent adjacent nodes from catching fire or postponing the fire spread.

Figure 4.1 illustrates this approach on the same graph that was presented in the introduction (see Fig. 1.1). The heuristic information and the candidate set for each time step are given in Figure 4.1a. These values are generated based on the assumption that the vertices within the candidate list with the highest heuristic are protected. We can see that at the time step 0 the candidate list comprises five candidates and thus the size of $\mathcal{C}$ is reduced by approximately 54%. Based on the heuristic information, nodes 4 und 10 are protected. At time step 1 the selection possibility is reduced to only two nodes. After these two nodes are protected, the fire is contained. The resulting graph is presented in Figure 4.1b.

This approach offers a certain degree of flexibility, as it is possible to apply a combination of various strategies. An example of a hybrid strategy would be after the construction phase to alternate between the described heuristic and an alternative complementary strategy such as prioritizing nodes that would catch fire at the next iteration.

	$t=0$		$t=1$		$t=2$	
Node	*Heuristic*	$v \in \mathcal{C}$	*Heuristic*	$v \in \mathcal{C}$	*Heuristic*	$v \in \mathcal{C}$
1	0.0	*no*	0.0	*no*	0	*no*
2	1.00	*no*	0.0	*no*	0	*no*
3	3.00	*yes*	0.0	*no*	0	*no*
4	**17.00**	***yes***	0.0	*no*	0	*no*
5	12.33	*no*	0.0	*no*	0	*no*
6	12.33	*no*	0.0	*no*	0	*no*
7	1.00	*no*	1.0	*no*	0	*no*
8	3.67	*yes*	1.0	*no*	1	*no*
9	1.00	*no*	**3.0**	***yes***	0	*no*
10	**5.67**	***yes***	0.0	*no*	0	*no*
11	3.67	*yes*	**5.0**	***yes***	0	*no*
12	1.00	*no*	1.0	*no*	1	*no*
13	1.00	*no*	1.0	*no*	1	*no*
14	1.00	*no*	1.0	*no*	1	*no*

(a) Metrics and candidates for each time step.



(b) Result at $t = 2$

Figure 4.1: Example of a graph with 14 vertices and two firefighters.

Algorithm 4.5: PlaceFirefighters()

Input: Candidate set $\mathcal{C}$, heuristic set η, Solution $\mathcal{P}_{\text{cr}} = (p_1 \dots p_n)$, set of untouched vertices V^{unt}, current time step t, number of firefighters D, determinism rate q_0.

Output: Solution $\mathcal{P}_{cr}$.

$n \leftarrow D$;
if $n \geq |C|$ **then**
  $\forall v \in C\ (p_v \leftarrow 1;\ n \leftarrow n-1)$;
end
while $n > 0$ *and* $|V^{unt}| > 0$ **do**
  if $C = \emptyset$ **then**
    $C \leftarrow V^{unt}$;
  end
  $r \leftarrow \mathcal{U}(0,1)$; `// apply uniform random function`
  if $r < q_0$ **then**
    $v \leftarrow \arg\max_{v \in C} (\eta_v \times \tau_{t,v})$;
  else
    $v \leftarrow$ select proportional to $\frac{(\eta_v \times \tau_{t,v})}{\sum_{v \in C}(\eta_v \times \tau_{t,v})}$ from all $v \in C$;
  end
  $p_v \leftarrow 1$; `// protect node v`
  $n \leftarrow n - 1$;
end

Once the candidate set $\mathcal{C}$ has been calculated, the node selection phase takes place in which at most D vertices are selected from $\mathcal{C}$ (see Algorithm 4.5). In case D is

greater than or equal to the size of $\mathcal{C}$, all nodes of the candidate set are protected (lines 2 - 4). For each remaining firefighter, the algorithm first checks whether $\mathcal{C}$ is empty and, if so, further vertices are chosen from the remaining set of untouched vertices $V^{unt} = V \setminus (V^{prot} \cup V^{burnt})$. In the next step a random number $r \in [0, 1]$ is generated. For $r \leq q_0$, where parameter $q_0 \in [0, 1]$ represents the determinism rate, the vertex with the largest product of the heuristic information and pheromone trail is deterministically chosen to be protected. In the other case, when $r > q_0$, the next vertex is chosen by applying the fitness proportionate selection function in line 13.

Following the node selection phase, Algorithm 4.2 continues spreading the fire to all untouched adjacent nodes after which t is incremented. These three steps of calculating the candidate set, selecting the nodes and spreading fire are iterated until the fire is contained.

4.2.3 Variable Neighborhood Search

As already mentioned in the introduction, the population-based algorithms have an advantage in their capability to explore wide regions of the solution space, however, with the drawback of a weak exploitation capability. On the other hand, the advantage of single-solution based algorithms is their superior exploitation capability within a limited search space. Due to such features, these two types of metaheuristics are frequently combined with the aim of enhancing the solution quality.

Algorithm 4.6 presents GVNS combined with VND with a best improvement strategy, which is a simplified version of the work of Hu et al. [HWR15]. The proposed version introduces an adaptive shaking but excludes the performance boosting *incremental evaluation scheme*. The Algorithm starts with an adaptive shaking (line 4) where l protected nodes change their status in a given solution $\mathcal{P}_{\text{cr}}$. In order to improve the accuracy of this phase a set of protected nodes $T \subseteq \mathcal{P}_{\text{cr}}$, the removal of which does not decrease the result of a given solution by more than one, is defined. This process is described in Algorithm 4.7 at line 2. For the purpose of escaping from local optima the whole set $\mathcal{P}_{\text{cr}}$ needs to be taken into account as well. The relation between these two sets is controlled by parameter z, where $z \leq |T|$ and $z \leq l$ (lines 5-11).

For the VND phase (lines 7-18) the set of neighborhood structures $\mathcal{N}_k$ for $k = 1, \ldots, k_{max}$ is defined as follows. For a given solution $\mathcal{P}'$, the neighborhood $\mathcal{N}_k(\mathcal{P}')$ is defined as a set $\mathcal{W}_k$, which comprises the vertices with k unprotected adjacent vertices. For each $w \in \mathcal{W}$ a VND procedure, which consists of three steps is applied. The first step protects all adjacent vertices after which the evaluation procedure is applied. During the traversal in the evaluation, $\mathcal{P}'$ is implicitly transformed to the encoding $\mathcal{S}'$ defined in Eq. 4.2 by saving each node's protection time (for a more detailed description of the evaluation process see [HWR15]). The subsequent refine procedure improves the solution either by aiming to protect further vertices until no improvement is achieved or applying a more costly local search. The described phases are applied iteratively until a termination criterion, which in this context is the last examined neighborhood, is met.

Algorithm 4.6: General Variable Neighborhood Search (based on [HWR15])

Input: solution $\mathcal{P} = (p_1 \dots p_n)$, parameter z.
Output: solution $\mathcal{S} = (s_1 \dots s_n)$

while *termination criteria are not met* **do**
  $l \leftarrow 0$;
  while $l \leq l_{max}$ **do**
    $\mathcal{P}' \leftarrow Shake(\mathcal{P}, l, z)$;
    $k \leftarrow 1$;
    $\mathcal{P}_{best_k} \leftarrow \mathcal{P}'$;
    while $k \leq k_{max}$ **do**
      $\mathcal{N}_k \leftarrow$ a set of vertices with k unprotected adjacent vertices ;
      foreach v *in* $\mathcal{N}_k$ **do**
        $\mathcal{P}'' \leftarrow \mathcal{P}'$;
        $\mathcal{P}''_{v'} \leftarrow 1, \quad \forall v'$ neighbor of v ;
        Evaluate($\mathcal{P}''$) ;
        Refine($\mathcal{P}''$) ;
        if $\mathcal{P}''$ *is better than* $\mathcal{P}_{best_k}$ **then** $\mathcal{P}_{best_k} \leftarrow \mathcal{P}''$;
      end
      if $\mathcal{P}_{best_k}$ *is better than* $\mathcal{P}'$ **then** $\mathcal{P}' \leftarrow \mathcal{P}_{best_k}$; $k \leftarrow 1$;
      else $k \leftarrow k + 1$;
    end
    if $\mathcal{P}'$ *is better than* $\mathcal{P}$ **then** $\mathcal{P} \leftarrow \mathcal{P}'$; $l \leftarrow 0$;
    else $l \leftarrow l + 1$;
  end
end

4.2.4 Pheromone Update

The pheromone model $\mathcal{T}$, where $\forall \tau_{t,v} \in \mathcal{T}$ corresponds to a time step $1 \leq t \leq t_{\max} \leq |V|$ and a node v, is applied in this algorithm. To update the pheromene trails, we apply the update rules from HCF (see [BD04, BBGM+14]):

$$\tau_{t,v} := \tau_{t,v} + p \cdot (\xi_{t,v} - \tau_{t,v}), \tag{4.3}$$

where

$$\xi_{t,v} := \kappa_{\text{ib}} \cdot \Delta(S_{ib}, t, v) + \kappa_{\text{rb}} \cdot \Delta(S_{rb}, t, v) + \kappa_{\text{bs}} \cdot \Delta(S_{bs}, t, v), \tag{4.4}$$

where

$$\Delta(S, t, v) := \begin{cases} 1 & \text{if } v \text{ is protected at time step } t \text{ in solution } S \\ 0 & \text{otherwise.} \end{cases} \tag{4.5}$$

The parameter p refers to the evaporation rate whereas the values κ_{ib}, κ_{rb}, and κ_{bs} present the weighting factor of the iteration-best solution $\mathcal{S}_{\text{ib}}$, the restart-best solution

Algorithm 4.7: Shake()

Input: Solution $\mathcal{P} = (p_1 \dots p_n)$, size k, parameter z.
Output: Protected set $\mathcal{P}$.

```
1  if z > 0 then
2  |  T ← {p ∈ P | L(p) = protected ∧ ∀p' neighbor of
   |    p : L(p') ≠ burnt ∨ L(p') = protected} ;
3  end
4  while k > 0 do
5  |  if z > 0 ∧ |T| > 0 then
6  |  |  v ← take node v from T randomly ;
7  |  |  p_v ← 0 ;                          // drop protection of node v
8  |  |  z ← z − 1 ;
9  |  else
10 |  |  drop protection from P randomly ;
11 |  end
12 |  k ← k − 1 ;
13 end
```

$\mathcal{S}_{\text{rb}}$, and the best-so-far solution $\mathcal{S}_{\text{bs}}$ respectively. The weights κ_{ib}, κ_{rb}, and κ_{bs} of the solutions $\mathcal{S}_{\text{ib}}$, $\mathcal{S}_{\text{rb}}$, and $\mathcal{S}_{\text{bs}}$ change depending on the convergence factor cf in combination with the Boolean variable *bs_update*. These factors are defined in Table 4.1.

	bs_update = 0				*bs_update* = 1
	$cf < 0.4$	$cf \in [0.4, 0.6[$	$cf \in [0.6, 0.8[$	$cf > 0.8$	
κ_{ib}	2/3	1/3	0	0	0
κ_{rb}	1/3	2/3	1	0	0
κ_{bs}	0	0	0	0	1

Table 4.1: Shows the weighting factors determined by the convergence factor cf and the variable *bs_update*. The content of this table is taken from [BBGM+14].

With the aim of preventing the algorithm from search stagnation, each pheromone trail $\tau_{t,v} \in \mathcal{T}$ is retained between $\tau_{\max} = 0.99$ and $\tau_{\min} = 0.01$ after the pheromone update phase has completed.

4.2.5 Convergence Factor Computation

The computation of the convergence factor cf is determined by the pheromone trails [BD04, BBGM+14]:

$$cf = 2 \cdot \left(\frac{\sum_{t=0}^{t_{\max}} \sum_{v \in V} \max\left(\tau_{\max} - \tau_{t,v}, \tau_{t,v} - \tau_{\min}\right)}{t_{\max} \cdot \left(\tau_{\max} - \tau_{\min}\right)} - 0.5 \right) \tag{4.6}$$

This factor indicates the degree of convergence of the pheromone values. When all pheromone trails denote the initial pheromone value that is 0.5, *cf* results in 0. In the opposite case, when all pheromone trails are set to $\tau_{\max}$ or $\tau_{\min}$, *cf* results in 1. The value of *cf* lies between 0 and 1 for all other scenarios.

4.3 Parallel Implementation

This section presents the parallelization of the previously described sequential approach.

With the aim to utilize the GPU efficiently, we apply a hybrid mapping technique that combines task parallelism with data parallelism. More specifically, a predefined number of ants, where each is statically assigned to a thread, perform each ACO and VNS consecutively using a data parallelism strategy that switches to dynamic parallelism at runtime. A clearer overview of the described approach is provided in Figure 4.2.

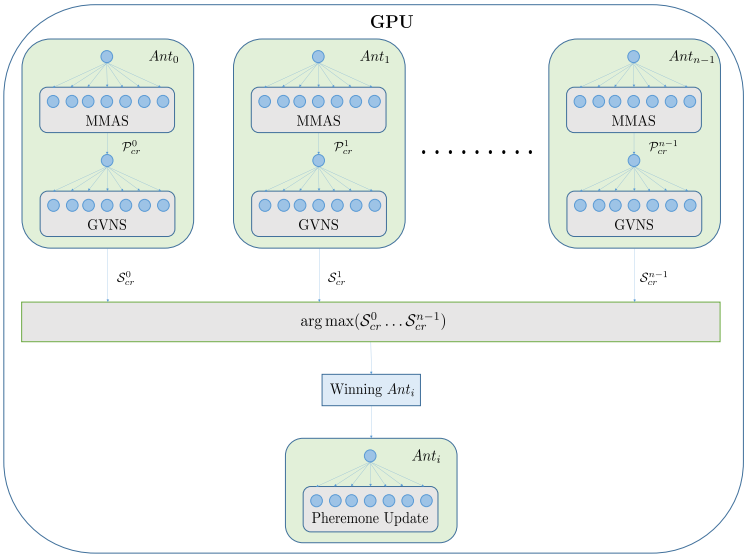
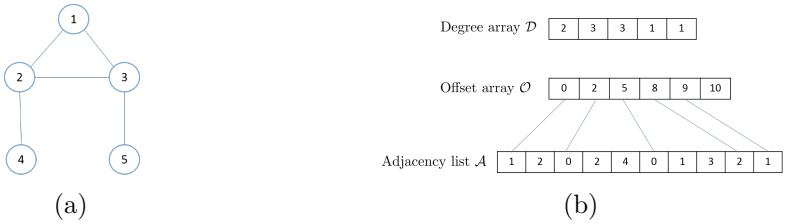


Figure 4.2: An overview of the parallel hybrid ACO implementation.

Various studies [BB17a, BB17b] have shown that none of the mapping techniques developed so far achieve optimal performance, as each comes with a trade-off between workload balance and computational overhead resulting from dynamic parallelism. Nevertheless, based on the literature in the field and tentative tests, we assume that the chosen approach might have the most promising trade-off.

4.3.1 Graph Representation and Data Structure

Instead of ordinary adjacency matrices, a Compressed Sparse Row (CSR) data structure is used for the graph representation due to its compact structure enabling efficient application. The CSR data structure consists of two arrays, an offset array $\mathcal{O}$ and a concatenated adjacency list $\mathcal{A}$. The offset array contains elements that refer to the start position of neighboring nodes in the adjacency list. In addition to that, a degree array $\mathcal{D}$ containing the degree of each node is saved in order to make the workload calculations more efficient. Figure 4.3 shows an example of a small graph represented in CSR.



(a) (b)

Figure 4.3: An example of the CSR data structure.

Furthermore, for each ant a_i bitmask representations for the candidate list $\mathcal{C}^i$ and tabu list $\mathcal{M}^i$ are used, s.t. the n-th bit indicates if vertex v is present in the respective set. For the analysis of the graph topology, for each ant an array $\mathcal{L}^i$ is required, which saves the time step of each node that catches fire in the simulation process. Additionally, for the breadth first search, each ant uses two buffers $(\mathcal{Q}^i_{\text{in}}, \mathcal{Q}^i_{\text{offset}})$ and $\mathcal{Q}^i_{\text{out}}$ to store the nodes of the current and the subsequent time step, where $(\mathcal{Q}^i_{\text{in}}, \mathcal{Q}^i_{\text{offset}})$ is represented in CSR format.

4.3.2 Parallel Hybrid ACO

A representation of the basic structure of the parallel hybrid approach is given in Algorithm 4.8. Similarly to the sequential counterpart, this algorithm saves three solutions $\mathcal{S}_{\text{ib}}$, $\mathcal{S}_{\text{rb}}$ and $\mathcal{S}_{\text{bs}}$. Instead of using a loop in the construction phase, this step is now performed in parallel by the ants. After that, a random selection of the best solution is performed (line 4). Henceforth, the ant that generated the selected solution is responsible for the remaining steps of an iteration. It updates the three solutions in lines 6-8. Then the pheromone update method is applied in a data-parallel manner. The number of threads necessary for this process amounts to the number of nodes $|V|$ multiplied by the highest time step $t_{\max}$ reached up to that point by any ant. The convergence factor is computed using the same strategy and the same number of threads. The described procedure is iterated until predefined termination criteria are fulfilled.

4.3.3 Parallel ACO Solution Construction

Equivalently to the sequential approach, the parallel approach comprises the same three phases. First candidate set $\mathcal{C}^i$ along with heuristic information η^i is generated after which

Algorithm 4.8: Parallel hybrid ACO for the firefighter problem on the GPU

Input: An undirected graph $G = (V, E)$, a set B_{init}, number of ants m, number of firefighters D.
Output: The best found solution $\mathcal{S}_{\text{bs}}$.

```
1  while termination criterion is not met do
2      P^i_cr ← ConstructACOSolutionGPU(ant_i) ;
3      S^i_cr ← ConstructVNDSolutionGPU(ant_i, P^i_cr) ;
4      index ← arg max(S^0_cr ... S^m_cr) ;
5      if ant_i = index then
6          S_ib ← S^i_cr ;
7          if f(S_ib) > f(S_rb) then S_rb ← S_ib ;
8          if f(S_ib) > f(S_bs) then S_bs ← S_ib ;
9          size ← |V| · t_max ;
10         ApplyPheromoneUpdateGPU«size»(cf , bs_update, T , S_ib, S_rb, S_bs);
11         cf ← ComputeConvergenceFactorGPU«size»(T );
12         if cf > 0.99 then
13             if bs_update = true then
14                 τ(t, j) ←0.5, ∀(t, j) ∈ T ;
15                 S_rb ← ∅;
16                 bs_update ← false;
17             else
18                 bs_update ← true;
19             end
20         end
21     end
22 end
23 return S_bs;
```

the node selection takes place and the fire is spread to all untouched adjacent nodes. These three steps are controlled by each ant a_i in parallel and iterated until the fire is contained.

Algorithm 4.9 illustrates the parallel approach where $\mathcal{C}_t^i$ and η_t^i are generated. In lines 5-12 the top-down BFS is applied. For each time step t, two procedures are performed: the calculation of an exclusive prefix sum and a frontier expansion. In order to enable an efficient mapping between threads and edges for the frontier expansion an exclusive prefix sum is calculated over the degrees of all elements v in frontier queue $\mathcal{Q}_{\text{in}}^i$ at t. A detailed description of this process is given in Algorithm 4.10, which is based on the chained approach of Liu and Aluru [LA16] and Yan et al. [YLZ13]. The algorithm starts with an intra-warp scan (see Algorithm 4.11) where each warp w_i scans its subset. Subsequently (lines 8-20) an intra-block scan is performed. In this step the last thread of each warp saves its sum in a shared memory *shrd*, which is used for communication within a CUDA

Algorithm 4.9: GenerateCandidateSetGPU()

Input: Current time step t, frontier set F containing n burning nodes at t.
Output: Candidate set $\mathcal{C}^i$, heuristic η^i.

```
1  t_cur = t ;
2  i ← Q^i_offset[t] ;                          // read starting index i of Q^i_in for t
3  Q^i_in[i ... i + (n − 1)] ← v,  ∀v ∈ F ;     // copy F to Q^i_in starting at index i
4  burningNodes ← n ;
5  while burningNodes ≠ 0 do
6      t ← t + 1 ;                                          // increment time step t
7      Q^i_offset[t] ← burningNodes + i ;        // save starting index of Q^i_in for t
8      sum ←ApplyPrefixSumGPU«burningNodes»(&Q^i_in[i], &Q^i_out, t) ;
9      i ← i + burningNodes ;
10     globalCounter[ant_i] ← 0 ;
11     burningNodes ← ExpandFrontierGPU«sum»(&Q^i_out, &Q^i_in[i]) ;
12 end
13 while t > t_cur + 1 do
14     i ← Q^i_offset[t − 1] ;
15     burningNodes ← Q^i_offset[t] − i ;                    // calculate workload
16     t ← t − 1;
17     γ_t ← calculate available firefighters for time step t;
18     sum ← ApplyPrefixSumGPU«burningNodes»(&Q^i_in[i], &Q^i_out, t) ;
19     ExpandFrontierGPU«sum»(&Q^i_out, γ_t, C^i, η^i) ;
20 end
```

block (lines 8-10). Then an additional warp scan is performed on the shared memory (lines 12-16). To finish the intra-block scan each warp except the first one ($w_i > 0$) has to add the sum of the previous warps (lines 18-20). In the next step an inter-block communication takes place. Starting with the first block b_0, the sum of each block b_i is sequentially passed to the adjacent block b_{i+1} after which the sum of these two is calculated and passed further on (lines 22-31). Finally, to complete the scan operation, a last intra-block scan is performed in line 32. During the whole described procedure the time step (distance) of each node along with its offset is implicitly saved.

The main aim of the frontier expansion is to create the frontier for the next level. Algorithm 4.12, which is taken from the atomic based approach of Bernaschi et al. [BBM16] presents the steps of the frontier expansion procedure. By applying a binary search in the previously scanned result and taking the old offset each thread is mapped to a corresponding index of the adjacency list (lines 1-4). In line 5 the indegree of node v is incremented. Afterwards, an investigation of whether a node v has already been visited by another thread or not is conducted. If not, it is added into the next frontier queue.

The bottom-up BFS applied in lines 13 - 20 in Algorithm 4.9 follows the same strategy as sequential Algorithm 4.2, however, using the parallel techniques from the top-down

Algorithm 4.10: ApplyPrefixSumGPU() (based on [LA16, YLZ13])

Input: Buffer $\mathcal{Q}^i_{\text{in}} \subseteq V$ containing n elements, current time step t.
Output: Buffer $\mathcal{Q}^i_{\text{out}} = \{0, x_0, x_0 + x_1, \ldots, \ldots x_{n-3} + x_{n-2} + x_{n-1}\}$, offset list *offset*

```
1  l_i ← threadId mod 32 ;                          // laneId = threadId % warp size
2  w_i ← threadId/32 ;                              // warpId = threadId / warp size
3  v ← Q^i_in[threadId] ;                                          // get node v
4  L^i[v] ← t ;                                    // save distance of node v
5  offset[threadId] ← O[v] ;
6  d_v ← D[v] ;                                                 // get degree of node
7  d_v ← WarpScan(d_v, l_i) ;                       // apply intra-warp scan
8  if l_i = 31 or threadId = (n − 1) then
9  |  shrd[w_i] = d_v ;                           // save reduction to shared memory
10 end
11 ___syncthreads() ;                                          // barrier synchronization
12 if w_i = 0 then
13 |  tmp ← shrd[l_i] ;
14 |  tmp ← WarpScan(tmp, l_i) ;                  // block scan in first warp
15 |  shrd[l_i] ← tmp ;
16 end
17 ___syncthreads() ;
18 if w_i > 0 then
19 |  d_v ← d_v + shrd[w_i − 1] ;                                // finish intra-block scan
20 end
21 ___syncthreads() ;
22 if threadId = blockSize − 1 or threadId = (n − 1) then
23 |  tmp ← 0 ;                                  // start of inter-block communication
24 |  do
25 |  |  tmp ← read(Indicator[ant_i]) ;
26 |  while tmp < blockId;
27 |  sum_block ← sum_global[ant_i] ;
28 |  sum_global[ant_i] ← sum_block + d_v ;       // add reduction of current block
29 |  _threadfence() ;                                  // wait for write completion
30 |  atomicInc(Indicator[ant_i]) ;                              // increment indicator
31 end
32 Q^i_out[threadId + 1] ← d_v + sum_block ;                    // last intra-block scan
```

approach presented above. Similarly to the sequential approach, the parallel variant starts from the latest time step $t_{\max}$ and traverses through the graph until time step $t + 1$, during which the candidate list and the heuristic information are computed.

In the node protection phase we combine various parallel techniques from previous parallel

Algorithm 4.11: WarpScan()

Input: value x, lane id l_i
Output: Scanned set $\mathcal{X} = \{x_0, x_0 + x_1, \ldots, \ldots x_{29} + x_{30} + x_{31}\}$

```
1 #pragma unroll
2 foreach i = 1; i < 32; i ≪ 1 do
3 |   tmp = ___shfl_up(v, i) ;
4 |   if l_i ≥ i then  x ← x + tmp ;
5 end
6 return x ;
```

Algorithm 4.12: ExpandFrontierGPU() (based on [BBM16])

Input: Scanned $\mathcal{Q}^i_{\text{out}} = \{0, x_0, x_0 + x_1, \ldots, \ldots x_{n-3} + x_{n-2} + x_{n-1}\}$ containing n elements, offset array *offset*.
Output: Buffer $\mathcal{Q}^i_{\text{in}}$

```
1  index ← BinarySearch(n) ;
2  localOffset ← threadId − Q^i_out[index] ;
3  index ← offset[index] + localOffset ;
4  v ← A[index] ;
5  if L^i[v] = ∞ then atomicInc(indeg[v]);
6  m_i ← 1 ≪ (v  mod 32);
7  if tabu[v/32] & m_i then return ;
8  m_o ← atomicBitwiseOr(&tabu[v/32], m_i) ;
9  if !(m_i & m_o) then
10 |   index ← atomicInc(globalCounter[ant_i]) ;
11 |   Q^i_in[index] = v ;
12 end
```

ACO implementations for the TSP, which are described in the following papers [CGN+13, DS13, Ski16]. To compute the node selection we follow the Double-Spin Roulette (DS-Roulette) strategy from Dawson et al. [DS13] decomposing the candidate list $\mathcal{C}^i_t$ into subsets c^i_t with size s. Depending on its size, C^i_t can be computed either within one block — when $|C^i_t| \leq s$ — or distributed among several blocks. In our implementation we define a maximum block size of 512 threads, i.e., 16 warps per block, where each block performs the calculations warpwise. The described warp parallelism approach enables a significant decrease in execution time due to the fact that all threads within a warp execute without divergence throughout the whole phase. The subset size s, which is dynamically defined depending on a given situation, controls how many elements are computed in one block. For instance, a block with 512 threads would need 10 iterations for the computation of a subset with the size $s = 5120$.

According to Cecilia et al. [CGN+13] expensive repetitive calculations can be avoided

by precalculating the fitness values $\mathcal{Z} = \{\eta_v \times \tau_{t,v} \mid \forall v \in \mathcal{C}_t^i\}$, since neither the heuristic knowledge nor the pheromone trail change during a protection phase at time step t.

An illustration of the parallel single node selection for the case $r > q_0$ is given in Figure 4.4. For each free firefighter at time step t the computation steps of each warp are the following. First, each thread of a warp reads its corresponding element from the fitness values $\mathcal{Z}$ with coalesced access and performs a multiplication of $\mathcal{Z}_v$ by the associated tabu value. The result $\mathcal{X}$ of the multiplication amounts either to the fitness value or 0, based on the status of the tabu list. Second, each warp performs an intra-warp reduction on $\mathcal{X}$ and stores the results in shared memory. These steps are repeated until the block is finished computing its predefined subsets.

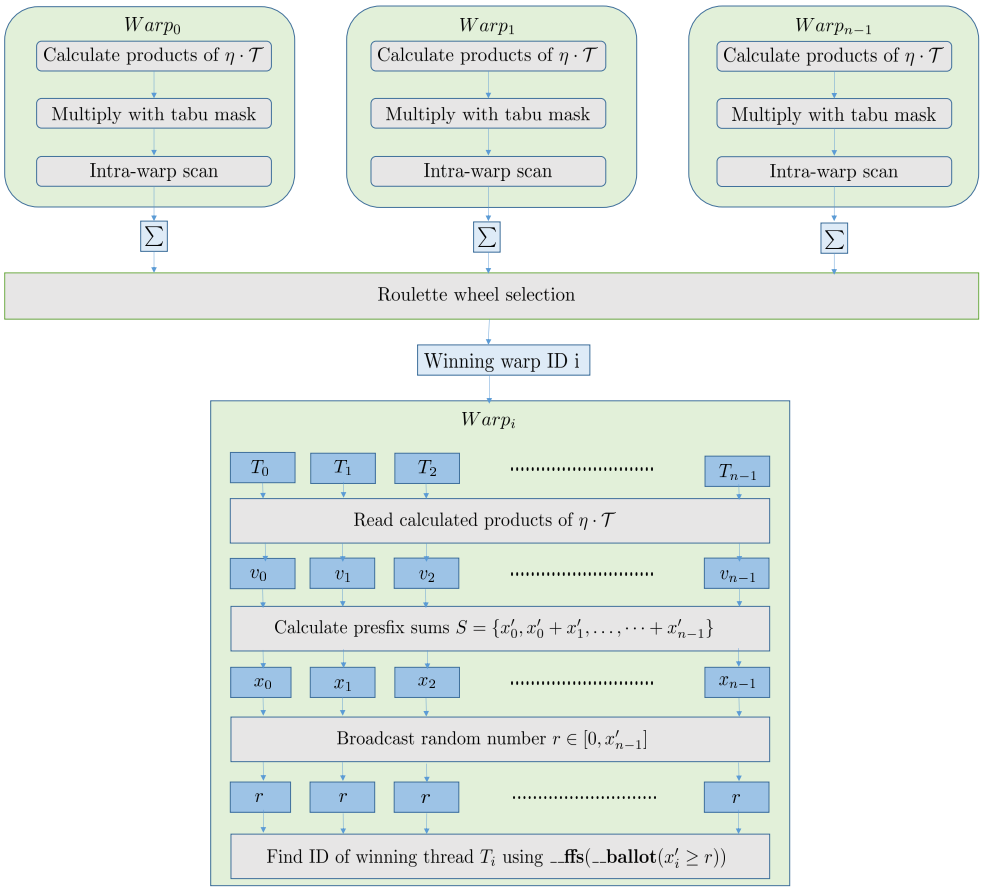


Figure 4.4: An overview of the parallel ACO node protection phase (based on [CGN+13, DS13, Ski16]).

Third, a warp responsible for selecting the next node to be protected needs to be chosen via a first roulette wheel selection. There are two possible situations: If the number of blocks equals to one, then a roulette wheel selection takes place within a single block using local

memory. Otherwise, each block copies values from shared to global memory on which the selection is performed on a separate kernel. In the last step, a final roulette wheel selection is performed after a specific warp has been selected. The fitness values $\mathcal{Z}$ are either read from shared memory or reread from global memory and multiplied by the associated tabu value resulting in $\mathcal{X}'$. Then, a scan $S = \{x'_0, x'_0 + x'_1, \ldots, \cdots + x'_{n-1}\}$ on $\mathcal{X}'$ is perfromed after which a random value r is generated using the XORWOW pseudo-random number generator from the cuRAND library [NVI17c] by a single thread. This thread broadcasts $r \cdot x'_{n-1}$ to all threads of this warp. The last step entails finding the winning thread T_i using the warp functions _ffs(_ballot($x'_i \geq r$)), where _ballot(*predicate*) returns an N bit integer whose n-th bit is set if the predicate of the n-th thread evaluates to true while _ffs(*int*) finds the first set bit in an N bit integer.

For the case $r \leq q_0$ the same strategy is applied. The only difference is that the roulette wheel selection is replaced by reductions that are applied to find the maximum value $e_{\max}$. Then the set with the best values can be computed with _ballot($x'_i = e_{max}$). Out of this set the node to be protected is chosen randomly in the final step.

4.3.4 Parallel Variable Neighborhood Search

In order to optimize the performance of the VNS a closer inspection of its various components was conducted with the aim of establishing which of them have parallelization potential. The procedures that we decided to parallelize are the following:

- shake solution,
- create neighborhood structure,
- protect nodes,
- evaluate solution,
- improve solution.

Based on preliminary tests we decided to perform parallelization starting with a workload size of 16 elements. This limit applies to the shake and node protection procedures. All the other mentioned procedures can be parallelized without any restrictions. Unlike the sequential algorithm, for performance reasons we left out the adaptive shaking in the parallel approach. Instead, the strategy of Hu et al. [HWR15] was chosen in which the protection status of l protected nodes is randomly changed. To create the neighborhood structure the parallel radix sort of Satish et al. [SHG09] is used. We adapted it slightly by exchanging the prefix sum procedure. Node protection is a straightforward task. In case less than 16 nodes have to be protected, this task is done sequentially. Otherwise, the process is parallelized with the same number of threads. The major component of the evaluation function is the breadth-first search, which was already introduced in the section about creating the candidate list. During the traversal through the graph the

solution is repaired in case it is infeasible and implicitly transformed into the target encoding. After the evaluation has been performed, the program calculates for each time step, beginning with the first one, whether there are any remaining free fire fighters γ. If so, γ nodes that are reached at that time step are protected non-deterministically. This procedure is iterated until the last time step is reached. The basic structure of the complete algorithm is illustrated in Figure 4.5.

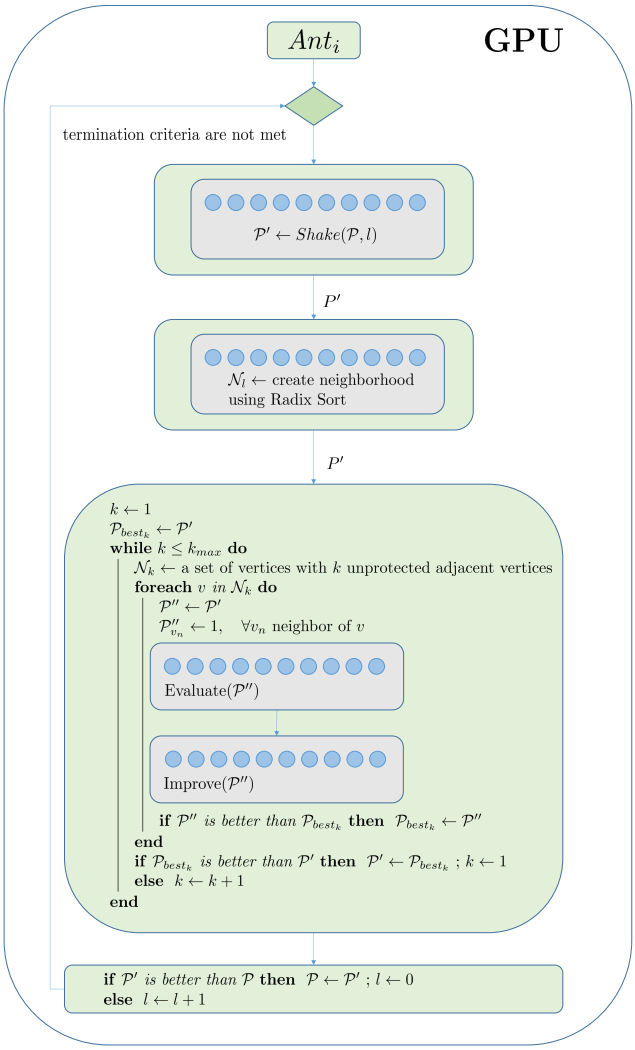


Figure 4.5: An overview of the parallel VNS implementation.

4.3.5 Parallel Pheromone Update

The parallelization of the pheromone update procedure is a straightforward process and is presented in Algorithm 4.13. In the first step, each thread is mapped to an element

of the pheromone model. To determine the node in the pheromone matrix a modulo operation is performed in line 1. To determine the time step in the pheromone model it is necessary to perform a division in which the thread ID is divided by the number of nodes (line 2). Based on the weighting factors κ_{ib}, κ_{rb}, and κ_{bs}, along with the three solutions $\mathcal{S}_{\text{ib}}$, $\mathcal{S}_{\text{rb}}$, and $\mathcal{S}_{\text{bs}}$ a value representing the influence is calculated, which is then used for the update (lines 3-6). In line 7 the pheromone trail is updated according to the formula of HCF. In the last step we check whether the pheromone value lies in the range between τ_{min} and τ_{max} (lines 8-13).

Algorithm 4.13: ApplyPheromoneUpdateGPU()

Input: Solutions $\mathcal{S}_{\text{ib}}$, $\mathcal{S}_{\text{rb}}$, and $\mathcal{S}_{\text{bs}}$, weighting factors κ_{ib}, κ_{rb}, and κ_{bs}, size n.
Output: Updated pheremones $\mathcal{T}$.

1 $v = \mathit{threadId} \mod n$;
2 $t = \mathit{threadId}/n$;
3 $\mathit{value} \leftarrow \varnothing$;
4 **if** $s^v_{ib} = t$ **then** $\mathit{value} \leftarrow \mathit{value} + \kappa_{\text{ib}}$;
5 **if** $s^v_{rb} = t$ **then** $\mathit{value} \leftarrow \mathit{value} + \kappa_{\text{rb}}$;
6 **if** $s^v_{bs} = t$ **then** $\mathit{value} \leftarrow \mathit{value} + \kappa_{\text{bs}}$;
7 $\tau_{t,v} \leftarrow \tau_{t,v} + p \cdot (\mathit{value} - \tau_{t,v})$;
8 **if** $\tau_{t,v} > \tau_{max}$ **then**
9 $\quad \tau_{t,v} \leftarrow \tau_{\text{max}}$
10 **end**
11 **if** $\tau_{t,v} < \tau_{min}$ **then**
12 $\quad \tau_{t,v} \leftarrow \tau_{\text{min}}$
13 **end**

4.3.6 Parallel Convergence Factor Computation

For this computation the same strategy as for the pheromone update stage, where each thread is mapped to a pheromone trail, is applied. First, the numerator of Equation 4.6 needs to be calculated, which can be done by the chained reduction method, similarly to the prefix sum algorithm. The initialized threads perform this operation collaboratively. After this phase the last thread contains the result of the operation in the numerator. This last thread is responsible for completing the calculation (lines 9-12 of Algorithm 4.14).

Algorithm 4.14: ComputeConvergenceFactorGPU()

Input: Solutions $\mathcal{S}_{\text{ib}}$, $\mathcal{S}_{\text{rb}}$, and $\mathcal{S}_{\text{bs}}$, weighting factors κ_{ib}, κ_{rb}, and κ_{bs}, size n.
Output: Updated convergence factor *cf*.

$v = \mathit{threadId} \mod n$;
$t = \mathit{threadId}/n$;
if $(\tau_{max} - \tau_{t,v}) > (\tau_{t,v} - \tau_{min})$ **then**
  $x \leftarrow \tau_{\max} - \tau_{t,v}$;
else
  $x \leftarrow \tau_{t,v} - \tau_{\min}$;
end
$x \leftarrow$ apply reduction using chained approach.
if $\mathit{threadId} = n - 1$ **then**
  $x \leftarrow \dfrac{x}{n \cdot (\tau_{\max} - \tau_{\min})}$;
  $cf \leftarrow (x - 0.5) \cdot 2$;
end

CHAPTER 5

Experimental Results

In this section, the results of the proposed implementations on various instances of the FFP are presented. For the purpose of comparability, we use the same test set-up as in [BBGM$^+$14, HWR15]. This includes test instances and their grouping, the parameters of the algorithms, number of iterations as well as the representation of the results. The experiments consist of several parts. First, the solution quality of ACO, VNS and the hybrid approach of the sequential implementation are investigated. The solution quality of these tests is compared to the results of the mentioned previous works. Then the same tests are performed for their respective parallel implementations comparing them both to previous work and our sequential algorithms. We furthermore perform a computational performance test of the parallel implementation comparing it to our sequential implementation.

5.1 Test Instances

For the proposed algorithms the benchmark set of Blum et al. [BBGM$^+$14] was used. The set comprises four different subsets containing 30 instances of Erdős–Rényi [1] graphs per subset. The graph size of the subsets is 50, 100, 500, and 1000 nodes. For each graph size category, graphs with three different edge densities are considered. The density of edges defines the probability of the existence of an edge between two vertices in a graph. For each edge density and vertex count combination there are 10 graphs. Moreover each instance was tested for firefighters $D \in \{1, \ldots, 10\}$, which means that altogether 1200 tests were performed for each algorithm. For a more detailed description see Section 6.4.

[1] Is a random graph $G(n, p_e)$, where n is the number of vertices and $p_e \in (0, 1)$ the probability of each edge being placed independently.

	CPU	GPU	CPU
Manufacturer	Intel	NVIDIA	Intel
Model	Core 2 Duo E6600	GeForce GTX 960	Core i5-4300U
Operating system	Windows 7	Windows 7	Windows 7
Platform	.Net Framework 4.5	CUDA 7	.Net Framework 4.5
Codename	Conroe	Maxwell	Haswell ULT
Cores	2	1024	2 (4 HT)
Clock frequency	2.4 GHz	1.165 GHz	1.90 - 2.90 GHz
L1 Cache size	32 KB + 32 KB	384 KB (48KB for each SM)	64 KB + 64 KB
L2 Cache size	4096 KB	2048 KB	512KB
L3 Cache size			3072 KB
DRAM memory	4 GB DDR2	4 GB GDDR5	12 GB DDR3

Table 5.1: Hardware platforms used for the experiments.

5.2 Hardware

The solution quality tests were conducted on the same machine with the aim of creating the same test conditions. In order to create comparable results to those achieved by Blum et al. [BBGM+14] a similar hardware configuration was used. Before every test session all the background processes were shut down ensuring that the machine is in idle modus. Two different platforms were used for the implementations. The parallel algorithms were implemented using CUDA 7. The results were obtained on a NVIDIA GeForce GTX 960 with 1024 cores (8 Maxwell Streaming Multiprocessors containing 128 processing cores each), 1.165 GHz and GDDR5 memory. The sequential implementations were tested using C# in Microsoft's .Net Framework 4.5 Framework. For the tests we used an Intel Core 2 Duo E6600 with 2.4 GHz and 4 GB DDR2 memory. To establish fairness of the test conditions for the speed-up tests we used another, more up to date hardware configuration for the sequential algorithms. More precisely, the specifications are the following: Intel Core i5-4300U with 1.9 GHz and 12 GB DDR3 memory. It should be mentioned that C# is known to be slower than the *C* language and that due to this difference the speed-up results may vary if they were to be implemented in *C*. The detailed specifications are given in Table 5.1.

5.3 Parameter Settings

For the sequential ACO implementation we used the same parameter settings as Blum et al. [BBGM+14], namely, we defined the number of ants (n_a) to be 10 per iteration, and the evaporation rate (p) was 0.1. All tests were performed under the assumption that the fire starts always at vertex 0. For the number of firefighters $D \in \{1, 2, 3\}$ the value of parameter q_0 is set to 0.5. For $D \in \{4, 5, 6\}$, q_0 is set to 0.7, and for the rest of the values of D, q_0 is set to 0.9. The threshold parameter B, which defines whether VNS is applied after an ACO solution is generated, is set to 0; that is, VNS is always applied after ACO computation. For the GPU implementation we performed tests with

a complete warp i.e. 32 ants in order to maximize the utilization of the GPU.

5.4 Solution Quality

Each of the six algorithms were applied once to each of the 120 graphs of the benchmark set with the chosen computation time limit of $n/2$ seconds, where n is the number of vertices. Tables 5.2 to 5.13 show the results according to the number of vertices and the edge density. A comparison of the previous four approaches, the VNS taken from Hu et al. [HWR15], the CPLEX, ACO and Hybrid ACO (HyACO) from Blum et al. [BBGM$^+$14], and our six implementations (ACO-SEQ, VNS-SEQ, HyACO-SEQ, ACO-GPU, VNS-GPU, HyACO-GPU) is made. Each line of the tables shows a different number of firefighters D while each cell represents the average number of unburnt vertices over 10 Erdős–Rényi graphs. The values in bold typeface show the best results. A summary of the test results is provided in the last two lines of the tables. Here the sum of the average solution values (Σ) and the best values reached (in %) is given.

For the sake of a clearer overview and in order to make an easier comparison the summarized results from each table are illustrated for a selected number of algorithms in an accompanying graph (Figures 5.1 to 5.6). The horizontal axis represents the test instances and correspond each to one table. For example, the value 100_ep0.1, where 100 stands for the number of nodes and _ep0.1 is the edge probability, belongs to Table 5.7.

As can be gathered from Tables 5.2 to 5.13, there are several improvements that can be observed. Firstly, comparing the ACO with ACO-SEQ, it can be seen that the former shows a decrease in quality as the size of the instances grows. Contrary to that, the variant with dynamic candidate lists outperforms its comparative counterpart as well as keeps a relatively stable solution quality regardless of the size of instances (see Figure 5.1).The average improvement achieved by this approach is 10.56% over all instances. Interestingly, the VNS implemented by Hu et al. performs still better than any of the sequential ACO variants. Considered in isolation, the results suggest that the VNS metaheuristic is better suited for the FFP than ACO. In addition to that, the proposed VNS-SEQ variant still achieves better results than Blum's ACO despite not having an *incremental evaluation scheme* implemented.

Figure 5.2 compares the three best sequential algorithms. Several interesting observations can be noted. Comparing the VNS with the proposed HyACO-SEQ, we can clearly see that the hybrid variant achieves better performance for each instance set, particularly for dense graphs. The average solution quality improvement is 0.3%. When we compare HyACO-SEQ with the HyACO implemented by Blum et al., HyACO-SEQ shows an overall better performance. However, the results for the instance set with 500 nodes and edge probability 0.025 (500_ep0.025) are counterintuitive, as it is the only instance set where HyACO outperforms both the VNS and HyACO-SEQ. The reason for this can be seen in Table 5.10: We can see that the performances of HyACO for $D = 6$ and $D = 7$ stand out as the only two values that are better than all the other ones. The performance difference between HyACo and HyACO-SEQ for this instance set is 1.75%.

To sum up, 118 times out of 120 test runs our proposed hybrid approach shows better solution quality than HyACO, with an average solution quality improvement of 0.47%.

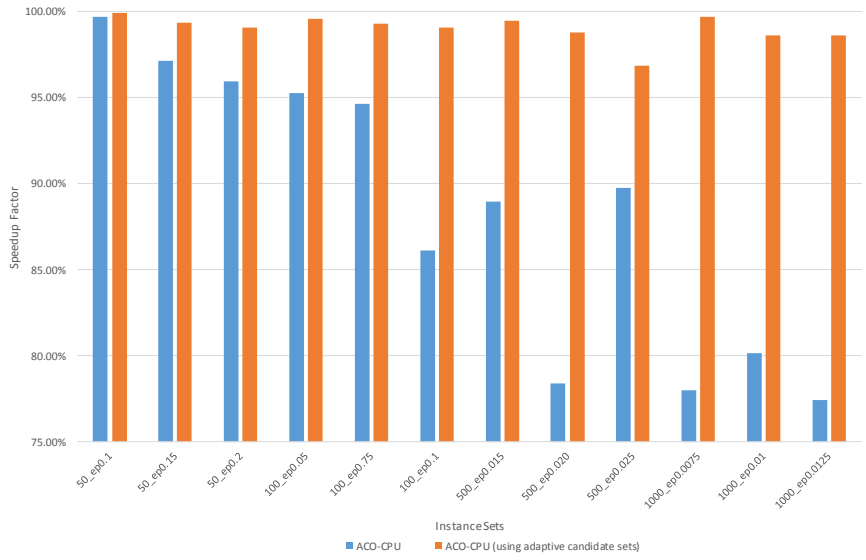


Figure 5.1: A comparative overview of the different solution qualities of ACO by Blum et al. and the proposed sequential ACO with dynamic candidate sets for all tested instances.

A comparison of the parallel VNS (VNS-GPU) with the sequential variant developed by Hu et al. (VNS) is given in Figure 5.3. It can be derived from this figure that the parallel approach achieves a better performance for all instance sets above 100. Hu et al. observed that VNS achieves better performances on sparse graphs and with larger instances. Interestingly, we observed the same behavior for the parallel variant. The results shown in the figure display the same characteristics starting at instance set 100.

In Figure 5.4 a comparison of ACO-SEQ and ACO-GPU is presented. From this data, we can see that the solution quality is steadily better for node sizes starting at 500. The best results are achieved for the largest instance set with the highest density. With smaller instance sets the algorithm shows inconsistencies for dense graphs.

The HyACO-GPU is compared with the VNS by Hu et al. and the HyACO by Blum et al. a summary of which is shown in Figure 5.5. The graph shows that the parallel variant achieves by far the best results (100%) for all instances except the already discussed results from Table 5.10. The difference for this instance set is reduced to 0.97% in comparison to the sequential results (see also Fig. 5.2).

Figure 5.6 shows an overview of the proposed three parallel algorithms. Analogously to the sequential results, the VNS algorithm outperforms the ACO. Moreover, we can derive that there is a gain in the solution quality with the hybrid variant implemented

on the GPU proving that hybridization yields remarkably better performances on both platforms (GPU and CPU).

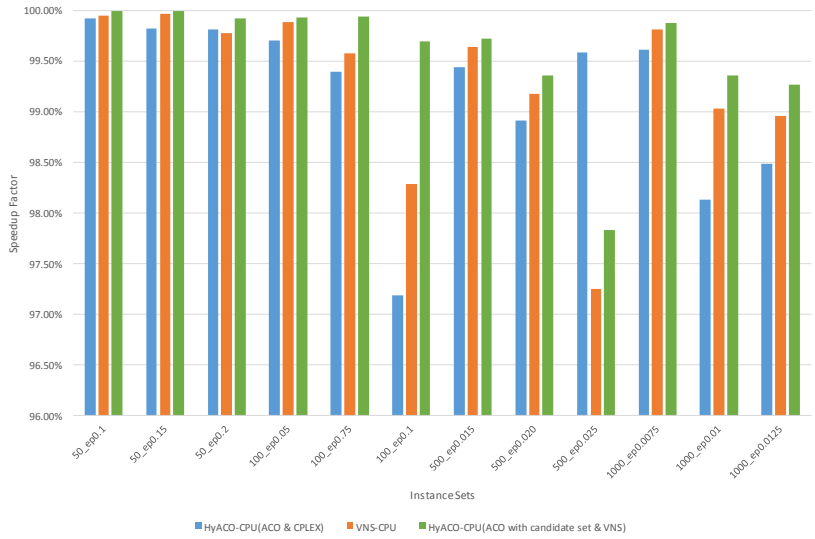


Figure 5.2: A comparative overview of the different solution qualities of HyACO by Blum et al., VNS by Hu et al. and the proposed sequential HyACO for all tested instances.

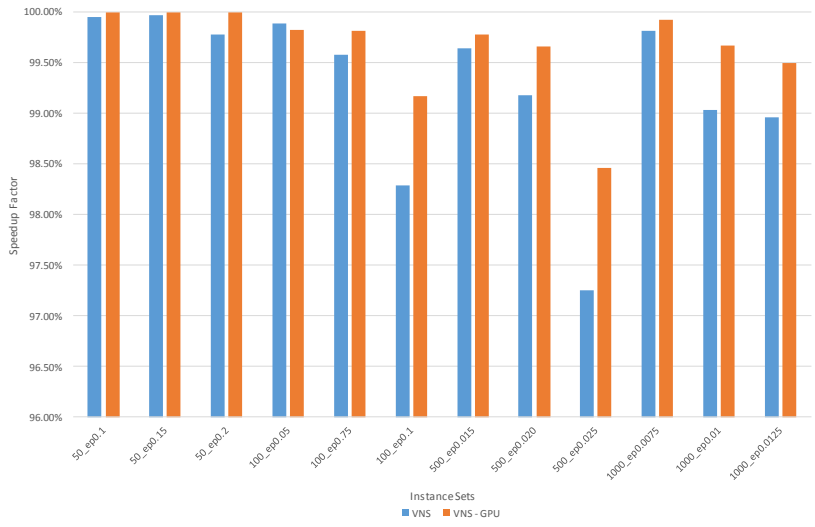


Figure 5.3: A comparative overview of the different solution qualities of VNS by Hu et al. and the proposed parallel VNS for all tested instances.

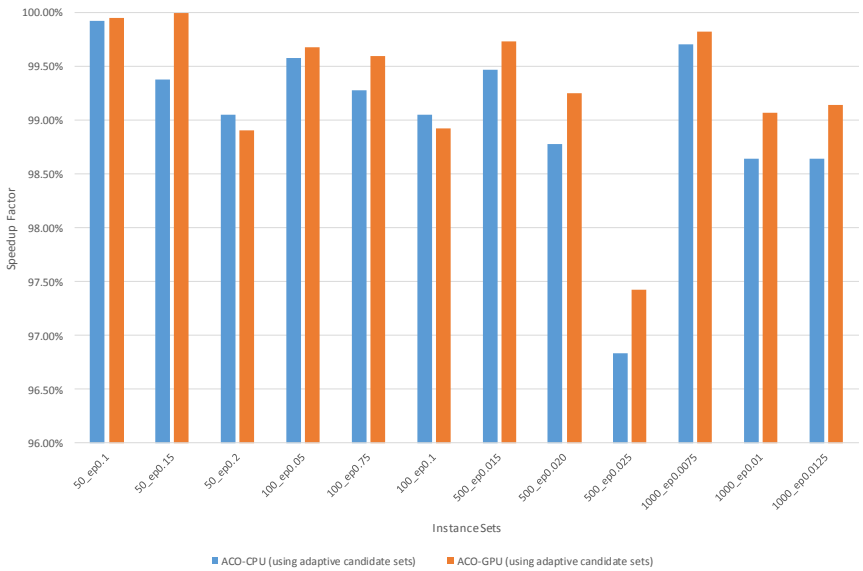


Figure 5.4: A comparative overview of the different solution qualities of the proposed sequential and parallel ACO algorithms for all tested instances.

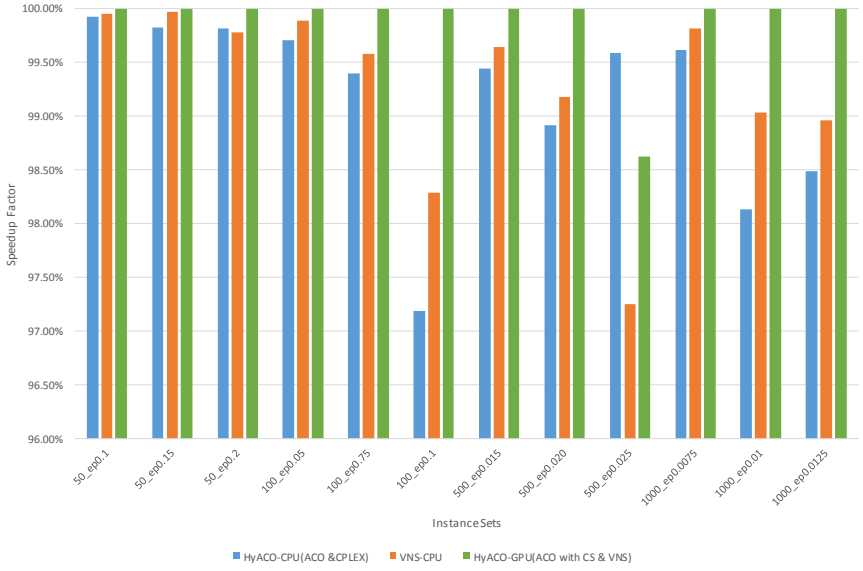


Figure 5.5: A comparative overview of the different solution qualities of HyACO by Blum et al., VNS by Hu et al. and the proposed parallel HyACO for all tested instances.

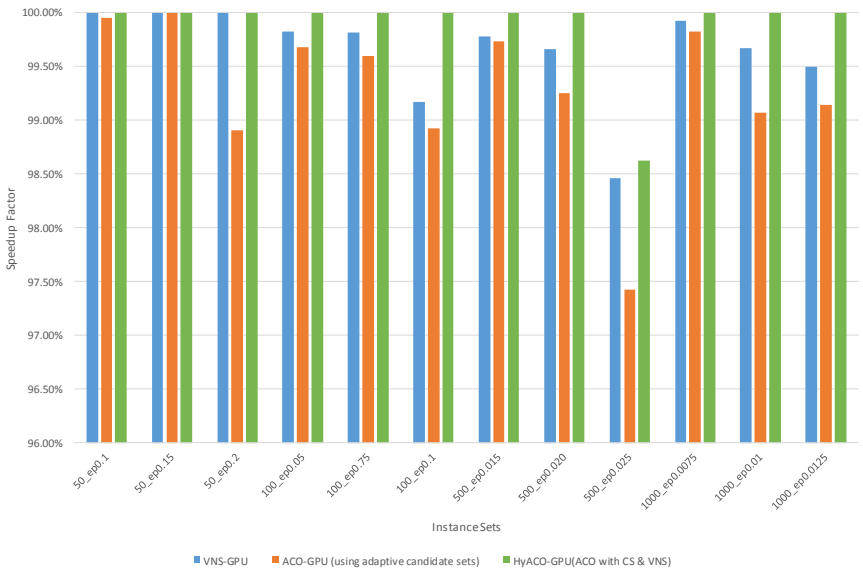


Figure 5.6: A comparative overview of the different solution qualities of the proposed parallel algorithms for all tested instances.

	Edge propability $p_e = 0.1$									
D	CPLEX	ACO	HyACO	VNS	ACO-SEQ	VNS-SEQ	HyACO-SEQ	VNS-GPU	ACO-GPU	HyACO-GPU
1	**7.4**	**7.4**	**7.4**	7.3	7.3	7.1	**7.4**	**7.4**	**7.4**	**7.4**
2	**26.6**	26.4	26.5	**26.6**	26.4	26.5	**26.6**	**26.6**	**26.6**	**26.6**
3	**41.8**	40.9	41.6	41.7	**41.8**	40.4	**41.8**	**41.8**	41.6	**41.8**
4	**47.9**	47.8	**47.9**	**47.9**	**47.9**	**47.9**	**47.9**	**47.9**	**47.9**	**47.9**
5	**48.5**	**48.5**	**48.5**	**48.5**	**48.5**	**48.5**	**48.5**	**48.5**	**48.5**	**48.5**
6	**48.8**	**48.8**	**48.8**	**48.8**	**48.8**	**48.8**	**48.8**	**48.8**	**48.8**	**48.8**
7	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**
8	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**
9	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**
10	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**
$\sum$	**417.0**	415.8	416.7	416.8	416.7	415.2	**417.0**	**417.0**	416.8	**417.0**
%	**100.00**%	99.71%	99.93%	99.95%	99.93%	99.57%	**100.00**%	**100.00**%	99.95%	**100.00**%

Table 5.2: Results for graphs with 50 vertices and edge propability $p_e = 0.1$.

	Edge propability $p_e = 0.15$									
D	CPLEX	ACO	HyACO	VNS	ACO-SEQ	VNS-SEQ	HyACO-SEQ	VNS-GPU	ACO-GPU	HyACO-GPU
1	**4.5**	**4.5**	**4.5**	**4.5**	4.4	4.3	**4.5**	**4.5**	**4.5**	**4.5**
2	**9.7**	**9.7**	**9.7**	**9.7**	9.6	9.4	**9.7**	**9.7**	**9.7**	**9.7**
3	**18.8**	16.5	18.5	18.7	18.0	16.3	**18.8**	**18.8**	**18.8**	**18.8**
4	**31.2**	30.5	30.9	**31.2**	31.0	30.7	**31.2**	**31.2**	**31.2**	**31.2**
5	**39.1**	36.1	**39.1**	**39.1**	38.4	38.1	**39.1**	**39.1**	**39.1**	**39.1**
6	**43.7**	42.7	**43.7**	**43.7**	43.6	**43.7**	**43.7**	**43.7**	**43.7**	**43.7**
7	**46.3**	45.4	**46.3**	**46.3**	46.2	46.2	**46.3**	**46.3**	**46.3**	**46.3**
8	**48.1**	46.8	**48.1**	**48.1**	**48.1**	**48.1**	**48.1**	**48.1**	**48.1**	**48.1**
9	**48.6**	48.2	**48.6**	**48.6**	**48.6**	**48.6**	**48.6**	**48.6**	**48.6**	**48.6**
10	**48.8**	**48.8**	**48.8**	**48.8**	**48.8**	**48.8**	**48.8**	**48.8**	**48.8**	**48.8**
$\sum$	**338.8**	329.2	338.2	338.7	336.7	334.2	**338**	**338**	338	**338**
%	**100.00**%	97.17%	99.82%	99.97%	99.38%	98.64%	**100.00**%	**100.00**%	**100.00**%	**100.00**%

Table 5.3: Results for graphs with 50 vertices and edge propability $p_e = 0.15$.

	Edge propability $p_e = 0.2$									
D	CPLEX	ACO	HyACO	VNS	ACO-SEQ	VNS-SEQ	HyACO-SEQ	VNS-GPU	ACO-GPU	HyACO-GPU
1	**3.1**	**3.1**	**3.1**	**3.1**	**3.1**	**3.1**	**3.1**	**3.1**	**3.1**	**3.1**
2	**7.2**	**7.2**	**7.2**	**7.2**	7.1	7.1	7.1	**7.2**	7.1	**7.2**
3	**11.2**	11.1	**11.2**	**11.2**	11.1	11.1	**11.2**	**11.2**	**11.2**	**11.2**
4	**17.5**	16.0	17.2	17.2	16.9	16.9	**17.4**	**17.5**	17.1	**17.5**
5	**27.7**	26.1	27.6	27.6	27.1	27.5	**27.7**	**27.7**	27.0	**27.7**
6	**33.0**	31.4	**33.0**	32.9	32.6	32.8	**33.0**	**33.0**	32.5	**33.0**
7	**37.5**	35.7	**37.5**	**37.5**	37.0	37.4	**37.5**	**37.5**	36.7	**37.5**
8	**42.7**	40.3	42.6	42.6	42.4	42.5	**42.7**	**42.7**	42.3	**42.7**
9	**46.1**	44.4	**46.1**	**46.1**	**46.1**	46.0	**46.1**	**46.1**	46.0	**46.1**
10	**47.5**	47.1	**47.5**	**47.5**	**47.5**	**47.5**	**47.5**	**47.5**	**47.5**	**47.5**
$\sum$	**273.5**	262.4	273.0	272.9	270.9	271.9	273.3	**273.5**	270.5	**273.5**
%	**100.00**%	95.94%	99.82%	99.78%	99.05%	99.42%	99.93%	**100.00**%	98.90%	**100.00**%

Table 5.4: Results for graphs with 50 vertices and edge propability $p_e = 0.2$.

	Edge propability $p_e = 0.05$									
D	CPLEX	ACO	HyACO	VNS	ACO-SEQ	VNS-SEQ	HyACO-SEQ	VNS-GPU	ACO-GPU	HyACO-GPU
1	**9.2**	9.1	**9.2**	9.1	8.9	8.4	9.0	9.1	**9.2**	**9.2**
2	26.9	25.7	27.6	27.7	26.9	25.3	**27.8**	27.7	27.2	**27.8**
3	62.8	54.6	62.7	63.6	62.4	60.8	**63.9**	63.4	62.9	**63.9**
4	85.3	66.3	85.5	86.0	85.8	79.0	86.1	85.7	98.0	**86.4**
5	**97.3**	92.3	**97.3**	**97.3**	**97.3**	93.3	**97.3**	**97.3**	**97.3**	**97.3**
6	**98.5**	98.3	**98.5**	**98.5**	**98.5**	**98.5**	**98.5**	**98.5**	**98.5**	**98.5**
7	**98.8**	**98.8**	**98.8**	**98.8**	**98.8**	**98.8**	**98.8**	**98.8**	**98.8**	**98.8**
8	**98.9**	**98.9**	**98.9**	**98.9**	**98.9**	**98.9**	**98.9**	**98.9**	**98.9**	**98.9**
9	**99.9**	**99.9**	**99.9**	**99.9**	**99.9**	**99.9**	**99.9**	**99.9**	**99.9**	**99.9**
10	**99.9**	**99.9**	**99.9**	**99.9**	**99.9**	**99.9**	**99.9**	**99.9**	**99.9**	**99.9**
$\sum$	775.7	742.0	776.5	777.9	775.5	761.0	778.3	777.4	776.6	**778.8**
%	99.60%	95.27%	99.70%	99.88%	99.58%	97.71%	99.94%	99.82%	99.72%	**100.00**%

Table 5.5: Results for graphs with 100 vertices and edge propability $p_e = 0.05$.

	Edge propability $p_e = 0.075$									
D	CPLEX	ACO	HyACO	VNS	ACO-SEQ	VNS-SEQ	HyACO-SEQ	VNS-GPU	ACO-GPU	HyACO-GPU
1	**5.4**	**5.4**	**5.4**	**5.4**	5.3	5.2	**5.4**	**5.4**	**5.4**	**5.4**
2	11.3	11.2	11.3	11.2	11.2	10.7	**11.5**	11.3	**11.5**	**11.5**
3	41.5	41.0	41.6	41.5	41.2	40.9	**41.6**	**41.6**	41.3	**41.6**
4	53.7	52.4	53.3	53.8	53.4	52.6	54.1	54.1	54.1	**54.5**
5	65.7	63.5	65.9	66.5	65.9	65.2	67.0	67.0	66.5	**67.3**
6	87.5	75.1	87.3	87.6	87.0	85.5	**88.5**	88.2	87.3	**88.5**
7	98.1	87.9	98.1	98.1	**98.1**	98.0	**98.1**	**98.1**	**98.1**	**98.1**
8	**98.6**	93.5	**98.6**	**98.6**	**98.6**	**98.6**	**98.6**	**98.6**	**98.6**	**98.6**
9	**99.8**	**99.8**	**99.8**	**99.8**	**99.8**	**99.8**	**99.8**	**99.8**	**99.8**	**99.8**
10	**99.0**	**99.0**	**99.0**	**99.0**	**99.0**	**99.0**	**99.0**	**99.0**	**99.0**	**99.0**
$\sum$	659.6	627.8	659.3	660.5	658.5	654.5	662.9	662.1	660.6	**663.3**
%	99.44%	94.63%	99.40%	99.58%	99.28%	98.67%	99.94%	99.82%	99.59%	**100.00%**

Table 5.6: Results for graphs with 100 vertices and edge propability $p_e = 0.075$.

	Edge propability $p_e = 0.1$									
D	CPLEX	ACO	HyACO	VNS	ACO-SEQ	VNS-SEQ	HyACO-SEQ	VNS-GPU	ACO-GPU	HyACO-GPU
1	**3.9**	**3.9**	**3.9**	**3.9**	**3.9**	**3.9**	**3.9**	**3.9**	**3.9**	**3.9**
2	8.5	8.3	8.7	**8.7**	8.6	8.3	**8.7**	**8.7**	**8.7**	**8.7**
3	21.4	21.0	21.3	21.4	21.4	21.2	21.5	21.6	**21.7**	**21.7**
4	25.5	24.5	25.5	25.7	25.7	25.0	25.9	**26.1**	25.6	**26.1**
5	30.2	29.1	29.5	30.1	30.2	39.8	30.4	30.3	30.2	**30.7**
6	41.8	33.9	41.0	42.3	42.5	35.2	42.6	42.8	42.4	**43**
7	58.7	46.4	56.3	56.9	58.8	53.7	59.3	57.3	57.9	**59.5**
8	74.8	62.0	74.0	74.8	75.4	69.6	**76.1**	75.4	75.3	**76.1**
9	89.2	77.3	88.0	89.2	90.0	83.6	90.7	90.7	90.2	**90.8**
10	94.7	85.9	94.4	94.6	94.6	94.3	**94.9**	94.8	94.6	**94.9**
$\sum$	448.7	329.3	442.6	447.6	451.1	424.6	454.0	451.6	450.5	**455.4**
%	98.53%	72.31%	97.19%	98.29%	99.06%	93.27%	99.69%	99.17%	98.92%	**100.00%**

Table 5.7: Results for graphs with 100 vertices and edge propability $p_e = 0.1$.

	Edge propability $p_e = 0.015$									
D	CPLEX	ACO	HyACO	VNS	ACO-SEQ	VNS-SEQ	HyACO-SEQ	VNS-GPU	ACO-GPU	HyACO-GPU
1	7.6	7.5	7.8	7.6	7.4	6.9	7.4	7.8	**7.9**	**7.9**
2	5.6	13.0	13.6	14.6	13.9	13.7	14.8	**15.3**	14.8	**15.3**
3	3.1	18.8	21.3	22.3	20.5	20.7	22.1	23.1	22.3	**23.6**
4	150.2	119.9	168.3	170.3	169.0	169.0	268.3	268.5	268.1	**171.4**
5	250.5	218.9	265.9	267.5	266.9	267.4	268.3	268.5	268.1	**269.4**
6	349.1	268.8	363.0	362.9	363.1	362.6	363.7	363.3	363.7	**364.8**
7	448.9	407.6	453.5	453.7	453.6	453.4	453.7	453.9	453.8	**452.2**
8	449.1	453.9	455.0	454.7	454.7	454.3	455.0	455.2	454.7	**455.4**
9	449.1	454.7	456.6	456.9	456.6	456.0	457.4	456.1	457.3	**457.8**
10	**498.8**	455.5	**498.8**	**498.8**	**498.8**	**498.8**	**498.8**	**498.8**	**498.8**	**498.8**
$\sum$	2612.0	2418.6	2703.8	2709.3	1704.5	2702.8	2711.3	2712.8	2711.6	**2718.9**
%	96.07%	88.96%	99.45%	99.65%	99.45%	99.41%	99.72%	99.78%	99.73%	**100.00%**

Table 5.8: Results for graphs with 500 vertices and edge propability $p_e = 0.015$

	Edge propability $p_e = 0.02$									
D	CPLEX	ACO	HyACO	VNS	ACO-SEQ	VNS-SEQ	HyACO-SEQ	VNS-GPU	ACO-GPU	HyACO-GPU
1	5.3	5.2	5.6	**5.7**	5.5	5.3	5.6	**5.7**	**5.7**	**5.7**
2	10.6	10.4	11.2	11.3	11.0	10.4	11.4	11.5	11.6	**11.9**
3	60.3	63.1	63.6	65.0	64.2	63.6	64.7	65.5	65.3	**65.7**
4	69.5	67.6	70.4	70.0	68.8	68.9	69.8	70.7	69.8	**70.9**
5	45.6	72.7	74.6	75.1	74.1	73.8	75.4	76.5	75.1	**76.8**
6	102.4	123.8	126.5	126.1	125.3	125.4	126.6	127.4	126.5	**128.1**
7	102.7	128.1	130.1	133.1	133.1	131.8	133.6	134.2	132.7	**136.0**
8	299.0	135.8	315.8	316.1	316.1	316.2	317.1	317.4	316.4	**318.1**
9	349.0	317.5	363.8	364.1	363.7	363.6	364.6	364.7	364.2	**365.4**
10	409.2	321.1	410.2	409.3	409.2	409.4	409.9	409.9	409.8	**410.3**
$\sum$	1462.6	1245.3	1571,6	1575.8	1569.5	1568.4	1578.7	1583.5	1577.0	**1588.9**
%	92.05%	78.37%	98.91%	99.18%	98.78%	98.71%	99.36%	99.66%	99.25%	**100.00%**

Table 5.9: Results for graphs with 500 vertices and edge propability $p_e = 0.02$

	Edge propability $p_e = 0.025$									
D	CPLEX	ACO	HyACO	VNS	ACO-SEQ	VNS-SEQ	HyACO-SEQ	VNS-GPU	ACO-GPU	HyACO-GPU
1	4.2	4.4	4.3	**4.5**	4.4	4.3	4.3	4.4	**4.5**	**4.5**
2	9.1	8.5	9	**9.3**	9.1	9.2	9.1	**9.3**	**9.3**	**9.3**
3	12.8	12.7	13.8	13.7	13.6	13.6	13.7	**14.5**	14.1	14.3
4	17.7	16.9	18.6	18.1	17.9	18.1	18.7	19.2	18.5	**19.3**
5	6.5	21.6	22.4	23.2	22.7	23.3	23.6	**24.5**	23.5	24.4
6	25.6	26.3	**33.8**	28.7	28.4	29.0	29.0	30.2	29.3	30.3
7	78.1	77.6	**93.0**	80.1	79.7	80.2	80.7	81.6	79.8	81.9
8	154.2	127.6	173.5	175.5	174.9	175.8	176.3	177.2	175.1	**175.5**
9	232.2	221.8	225.4	223.8	223.0	224.1	224.7	224.9	223.8	**226.1**
10	221.5	225.1	229.6	227.3	227.0	227.9	228.9	228.4	227.7	**229.9**
$\sum$	753.0	742.4	**823.5**	804.2	800.7	805.5	809.0	814.2	805.6	815.5
%	91.06%	89.78%	**99.59%**	97.25%	96.83%	97.80%	97.84%	98.46%	97.42%	98.62%

Table 5.10: Results for graphs with 500 vertices and edge propability $p_e = 0.025$.

	Edge propability $p_e = 0.0075$									
D	CPLEX	ACO	HyACO	VNS	ACO-SEQ	VNS-SEQ	HyACO-SEQ	VNS-GPU	ACO-GPU	HyACO-GPU
1	105.2	107.4	107.8	108.0	107.3	107.2	107.8	108.1	108.2	**108.3**
2	107.9	112.7	115.0	114.6	113.4	113.6	115.1	**116.0**	114.7	**116.0**
3	101.7	118.0	118.0	122.1	120.6	120.0	122.5	122.8	121.9	**123.8**
4	399.4	318.3	415.7	417.9	416.5	416.1	418.4	418.8	417.6	**419.4**
5	399.6	419.0	421.2	422.8	421.9	421.2	423.8	424.1	422.8	**425.3**
6	598.8	423.6	614.2	617.1	616.7	616.2	618.8	618.5	617.8	**619.6**
7	898.1	523.5	902.5	903.3	903.0	902.8	903.6	903.8	903.4	**904.0**
8	**998.2**	528.6	**998.2**	**998.2**	**998.2**	**997.8**	997.8	**998.2**	**998.2**	**998.2**
9	**998.9**	905.6	**998.9**	**998.9**	**998.9**	**998.9**	**998.9**	**998.9**	**998.9**	**998.9**
10	**999.0**	**999.0**	**999.0**	**999.0**	**999.0**	**999.0**	**999.0**	**999.0**	**999.0**	**999.0**
$\sum$	5606.8	4455.7	5690.5	5701.9	5695.5	5692.8	5705.7	5708.2	5702.5	**5712.5**
%	98.15%	77.99%	99.61%	99.81%	99.70%	99.66%	99.88%	99.92%	99.82%	**100.00%**

Table 5.11: Results for graphs with 1000 vertices and edge propability $p_e = 0.0075$.

	Edge propability $p_e = 0.01$									
D	CPLEX	ACO	HyACO	VNS	ACO-SEQ	VNS-SEQ	HyACO-SEQ	VNS-GPU	ACO-GPU	HyACO-GPU
1	4.9	5.7	6.0	6.3	6.1	6.1	6.2	**6.7**	6.6	6.6
2	4.4	10.9	10.6	12.1	11.5	11.6	11.9	13.0	12.6	**13.1**
3	13.7	15.9	17.0	17.9	16.9	17.0	17.9	19.0	18.2	**19.3**
4	14.8	21.4	24.1	24.1	23.1	22.6	24.1	25.2	24.3	**25.9**
5	104.3	27.1	123.6	126.6	125.2	125.4	127.0	128.4	126.6	**128.6**
6	201.5	129.1	226.0	228.2	227.5	227.3	228.7	230.3	228.5	**230.6**
7	299.7	525.5	325.2	328.2	326.7	327.0	329.2	330.1	327.3	**330.2**
8	399.5	329.7	424.4	426.5	425.4	426.3	427.8	428.3	426.1	**429.6**
9	399.6	427.9	427.7	431.8	430.8	431.4	433.6	433.1	431.7	**435.1**
10	499.5	432.6	528.4	530.7	530.8	531.0	533.1	531.9	531.3	**534.1**
$\sum$	1941.8	1725.8	2113.0	2132.4	2124.0	2125.7	2139.5	2146.0	2133.2	**2153.1**
%	90.18%	80.15%	98.13%	99.03%	98.64%	98.72%	99.36%	99.67%	99.07%	**99.995**%

Table 5.12: Results for graphs with 1000 vertices and edge propability $p_e = 0.01$.

	Edge propability $p_e = 0.0125$									
D	CPLEX	ACO	HyACO	VNS	ACO-SEQ	VNS-SEQ	HyACO-SEQ	VNS-GPU	ACO-GPU	HyACO-GPU
1	4.0	4.9	4.7	5.3	5.0	4.8	4.9	5.1	**5.4**	**5.4**
2	8.0	9.4	10.1	10.2	10.1	9.9	10.0	10.5	10.5	**10.6**
3	4.6	14.3	13.9	15.4	14.9	15.0	15.5	16.2	16.2	**16.6**
4	99.9	116.6	116.5	117.8	116.8	117.5	118.0	118.8	118.3	**119.2**
5	103.3	120.8	120.9	122.8	121.3	122.6	123.2	123.8	123.6	**124.5**
6	99.9	125.5	126.2	128.7	127.3	128.3	129.3	129.4	129.3	**130.7**
7	199.8	226.7	226.6	228.8	228.2	228.8	229.8	230.1	228.7	**230.9**
8	218.7	231.1	234.8	233.9	233.5	233.4	235.1	235.3	233.8	**236.2**
9	301.0	236.2	332.1	332.6	332.6	333.1	334.4	334.7	333.1	**338.9**
10	602.2	335.0	620.5	619.6	619.5	526.5	620.5	620.9	619.5	**621.1**
$\sum$	1641.1	1420.5	1806.3	1815.1	1809.2	1719.9	1820.7	1824.8	1818.4	**1834.1**
%	89.49%	77.45%	98.48%	98.96%	98.64%	93.77%	99.27%	99.49%	99.14%	**100.00**%

Table 5.13: Results for graphs with 1000 vertices and edge propability $p_e = 0.0125$.

5.5 Computational Performance Evaluation

For the evaluation of the computational performance of the proposed implementations we follow the guidelines established by Barr and Hickman [BGK+95] for computing the speed-up.

In the first test, based on the largest three instances, the proposed sequential ACO algorithm was tested against the parallel counterpart in order to investigate whether the number of ants has an impact on the speed-up. The test was limited to ten iterations with three different numbers of ants. We, moreover, set $q_0 = 1$,i.e., the node selection is performed deterministically in order to achieve similar runs on both platforms. However, due to different data type accuracies between the platforms, there is no guarantee that the algorithms would always select the same nodes. Tables 5.14 to 5.16 contain the mean speed-ups and execution times for the subset with 1000 vertices using 8, 16 and 32 ants. The test results offer no unambiguous conclusion; they suggest that for sparse graphs there might be a higher speed-up when fewer ants are employed. In other words, the higher the graph density, the more similar the speed-up results. However, it remains unclear to which degree the number of ants has an influence on the speed-up. Another, more evident, finding is that the overall speed-up rises with heightened graph density (see Fig. 5.7).

In a further test, the speed-up of the parallel implementations was examined on various instance sizes so as to measure the scalability. Tables 5.17 to 5.19 present mean speed-ups and execution times executed on instance sets for each instance size with highest density and applied with 32 ants. The speed-up at small instance sizes is rather low. For instance sizes 50 and 100 all three algorithms achieve a comparable speed-up factor, which amounts to 2-3 and 4-6, respectively. The speed-up rises significantly at instance size 500, at which ACO achieves 40.63x, VNS 30.72x and the hybrid algorithm 42.33x. This behaviour of the speed-up is very much in accordance with the solution quality results, as the best results are achieved with the more complex instances. In Figure 5.8 it can also be observed that VNS does not achieve such a rise in speed-up as it is 43.66% lower than the ACO speed-up for the largest instances. Yet, the hybrid algorithm has almost the same speed-up as the ACO, which leads to the conclusion that the hybridization is a meritable approach causing no performance loss in a parallelized scenario.

	Edge propability $p_e = 0.0075$			Edge propability $p_e = 0.001$			Edge propability $p_e = 0.0125$		
D	ACO-GPU	ACO-SEQ	Speedup	ACO-GPUU	ACO-SEQ	Speedup	ACO-GPU	ACO-SEQ	Speedup
1	43.78	3309.42	75.59	38.61	3344.89	86.65	33.55	3469.78	103.43
2	45.27	3423.84	75.62	39.00	3498.30	89.71	34.50	3610.90	104.67
3	50.13	3688.25	73.57	41.50	3769.66	90.83	37.59	3856.03	102.59
4	42.38	3427.43	80.88	45.09	3934.77	87.26	35.48	3907.20	110.13
5	41.94	3073.17	73.27	45.72	4208.00	92.03	38.26	4047.29	105.79
6	37.91	3347.13	88.29	44.43	4010.14	90.27	42.69	4280.43	100.26
7	30.75	3114.15	101.27	39.14	3696.64	94.45	38.69	4323.33	111.75
8	22.55	2671.52	118.49	37.46	3803.63	101.54	43.10	4749.21	110.18
9	13.27	2200.66	165.80	39.29	4295.08	109.33	41.87	4889.02	116.76
10	11.75	2167.26	184.44	39.86	3955.12	99.24	34.31	4597.84	134.01
$\bar{x}$	33.97	3042.28	103.72	41.01	3851.62	94.13	38.00	4173.10	109.96

Table 5.14: Mean speed-up and execution time results (in ms) for graphs with 1000 vertices for graph densities 0.75%, 1% and 1.25% executed with 8 ants and 10 iterations.

	Edge propability $p_e = 0.0075$			Edge propability $p_e = 0.001$			Edge propability $p_e = 0.0125$		
D	ACO-GPU	ACO-SEQ	Speedup	ACO-GPU	ACO-SEQ	Speedup	ACO-GPU	ACO-SEQ	Speedup
1	70.40	4725.32	67.12	62.67	4790.27	76.44	53.38	5351.02	100.25
2	73.25	4943.91	67.49	63.12	5231.48	82.88	55.71	5505.12	98.82
3	81.33	5125.63	63.02	66.93	5810.60	86.82	60.79	6125.88	100.78
4	68.07	4841.49	71.12	72.70	6357.76	87.46	57.36	6042.86	105.35
5	67.59	4730.58	69.99	73.83	7743.37	104.88	63.92	6444.73	100.83
6	60.99	4696.27	77.01	71.27	7182.45	100.77	69.22	6843.50	98.87
7	46.90	4103.62	87.50	63.34	5991.03	94.58	62.85	7506.29	119.42
8	36.34	3787.26	104.21	60.42	5776.22	95.60	69.52	7555.10	108.68
9	21.05	2663.32	126.52	63.48	5560.45	87.60	67.83	7954.74	117.27
10	19.21	2487.15	129.46	64.69	6014.24	92.97	55.83	7311.90	130.97
$\bar{x}$	54.51	4210.46	86.34	66.24	6045.79	91.00	61.64	6664.11	108.12

Table 5.15: Mean speed-up and execution time results (in ms) for graphs with 1000 vertices for graph densities 0.75%, 1% and 1.25% executed with 16 ants and 10 iterations.

	Edge propability $p_e = 0.0075$			Edge propability $p_e = 0.001$			Edge propability $p_e = 0.0125$		
D	ACO-GPU	ACO-SEQ	Speedup	ACO-GPU	ACO-SEQ	Speedup	ACO-GPU	ACO-SEQ	Speedup
1	116.68	7798.46	66.83	102.38	7868.41	76.85	89.72	8761.15	97.65
2	121.13	8031.05	66.30	105.15	8812.41	83.81	93.50	9507.90	101.69
3	134.69	8678.05	64.43	111.37	9765.79	87.69	101.07	10391.46	102.82
4	112.25	8004.93	71.31	120.80	10895.87	90.20	95.90	9996.61	104.23
5	111.11	7381.95	66.44	122.11	11702.29	95.83	103.10	11209.89	108.73
6	100.60	7857.95	78.11	117.69	10863.82	92.31	114.25	12503.67	109.44
7	77.40	6466.68	83.55	104.07	9414.72	90.47	103.82	12804.48	123.34
8	59.18	5785.95	97.76	99.34	9075.47	91.36	114.72	13626.07	118.77
9	34.65	3881.32	112.03	103.85	9193.73	88.53	111.86	15274.88	136.55
10	30.88	3271.41	105.94	105.45	9556.11	90.62	91.12	12885.78	141.42
$\bar{x}$	89.86	6715.78	81.27	109.22	9714.86	88.77	101.91	11696.19	114.46

Table 5.16: Mean speed-up and execution time results (in ms) for graphs with 1000 vertices for graph densities 0.75%, 1% and 1.25% executed with 32 ants and 10 iterations.

	Instances 50_ep0.2			Instances 100_ep0.1			Instances 500_ep0.025			Instances 1000_ep0.0125		
D	ACO-GPU	ACO-SEQ	Speedup	ACO-GPU	ACO-SEQ	Speedup	ACO-GPU	ACO-SEQ	Speedup	ACO-GPU	ACO-SEQ	Speedup
1	49.43	145.84	2.95	56.86	254.91	4.48	75.17	2890.98	38.46	89.72	8761.15	97.65
2	52.74	147.57	2.80	60.28	354.22	5.88	80.03	3106.09	38.81	93.50	9507.90	101.69
3	55.53	127.89	2.30	59.53	341.57	5.74	84.94	3405.75	40.09	101.07	10391.46	102.82
4	65.47	127.65	1.95	64.52	323.80	5.02	87.85	3706.28	42.19	95.90	9996.61	104.23
5	60.55	118.85	1.96	71.38	297.47	4.17	94.24	3975.08	42.18	103.10	11209.89	108.73
6	55.59	103.55	1.86	78.49	375.75	4.79	102.22	4396.38	43.01	114.25	12503.67	109.44
7	52.93	110.64	2.09	80.22	369.41	4.60	100.55	3803.47	37.83	103.82	12804.48	123.34
8	52.96	118.74	2.24	67.75	293.84	4.34	85.80	3547.15	41.34	114.72	13626.07	118.77
9	44.00	104.94	2.39	53.19	204.85	3.85	74.36	2945.58	39.61	111.86	15274.88	136.55
10	33.02	83.83	2.54	34.00	155.07	4.56	77.59	3317.67	42.76	91.12	12885.78	141.42
$\bar{x}$	52.22	118.95	2.31	62.62	297.09	4.74	86.28	3509.44	40.63	101.91	11696.19	114.46

Table 5.17: ACO mean speed-up and execution time results (in ms) for graphs with 50, 100, 500 and 1000 vertices, graph densities 20%, 10%, 2.5% and 1.25% executed with 32 ants.

	Instances 50_ep0.2			Instances 100_ep0.1			Instances 500_ep0.025			Instances 1000_ep0.0125		
D	VNS-GPU	VNS-SEQ	Speedup	VNS-GPU	VNS-SEQ	Speedup	VNS-GPU	VNS-SEQ	Speedup	ACO-GPU	VNS-SEQ	Speedup
1	47.94	245.57	5.12	111.51	1144.77	10.27	484.97	25845.35	53.29	1085.89	114931.57	105.84
2	60.59	255.20	4.21	179.46	1220.13	6.80	651.27	25949.82	39.84	1583.78	136659.60	86.29
3	88.08	286.71	3.26	188.12	1370.43	7.29	774.80	24909.16	32.15	2151.91	138250.48	64.25
4	135.40	349.68	2.58	223.83	1445.77	6.46	891.79	27239.63	30.54	1980.31	118069.26	59.62
5	142.31	231.07	1.62	255.41	1475.09	5.78	1066.27	28258.32	26.50	2121.64	118143.64	55.69
6	119.16	201.30	1.69	341.73	1673.61	4.90	1161.50	27276.11	23.48	2743.09	131880.58	48.08
7	117.68	218.50	1.86	365.18	953.93	2.61	1156.02	23625.15	20.44	2761.25	138732.68	50.24
8	136.34	230.60	1.69	435.75	1005.13	2.31	849.42	20802.69	24.49	2825.58	142142.07	50.31
9	124.41	210.84	1.69	377.98	824.64	2.18	660.60	17954.85	27.18	2540.89	188296.19	74.11
10	84.80	138.17	1.63	343.51	355.45	1.03	787.96	23094.62	29.31	3023.15	152439.33	50.42
$\bar{x}$	105.67	236.76	2.54	282.25	1146.90	4.96	848.46	24495.57	30.72	2281.75	137954.54	64.48

Table 5.18: VNS mean speed-up and execution time results (in ms) for graphs with 50, 100, 500 and 1000 vertices, graph densities 20%, 10%, 2.5% and 1.25% executed with 32 ants.

	Instances 50_ep0.2			Instances 100_ep0.1			Instances 500_ep0.025			Instances 1000_ep0.0125		
D	HyACO-GPU	HyACO-SEQ	Speedup	HyACO-GPU	HyACO-SEQ	Speedup	HyACO-GPU	HyACO-SEQ	Speedup	HyACO-GPU	HyACO-SEQ	Speedup
1	87.47	245.57	2.81	137.98	1021.99	7.41	539.78	24457.59	45.31	1217.33	118601.46	97.43
2	90.60	255.20	2.82	154.81	1072.84	6.93	559.10	24037.90	42.99	1234.38	129919.30	105.25
3	97.40	286.71	2.94	145.10	1079.48	7.44	590.81	24479.27	41.43	1300.00	126535.03	97.33
4	112.67	349.68	3.10	150.94	984.33	6.52	638.42	26383.03	41.33	1115.89	121950.80	109.29
5	107.90	231.07	2.14	162.25	1119.55	6.90	672.06	29320.29	43.63	1191.64	129935.66	109.04
6	100.25	201.30	2.01	174.86	1154.85	6.60	707.54	30045.61	42.46	1277.50	180573.05	141.35
7	97.38	218.50	2.24	196.46	1016.89	5.18	633.65	25485.65	40.22	1138.70	122623.41	107.69
8	90.56	230.60	2.55	176.68	807.62	4.57	555.28	20959.83	37.75	1136.91	129996.47	114.34
9	78.49	210.84	2.69	153.29	521.04	3.40	482.24	20088.11	41.66	1054.46	110065.42	104.38
10	68.59	138.17	2.01	116.81	332.85	2.85	526.41	25622.19	48.67	990.26	96213.14	97.16
$\bar{x}$	93.13	236.76	2.53	156.92	911.14	5.78	590.53	25087.95	42.55	1165.71	126641.37	108.33

Table 5.19: Hybrid ACO-VNS mean speed-up and execution time results (in ms) for graphs with 50, 100, 500 and 1000 vertices, graph densities 20%, 10%, 2.5% and 1.25% executed with 32 ants.

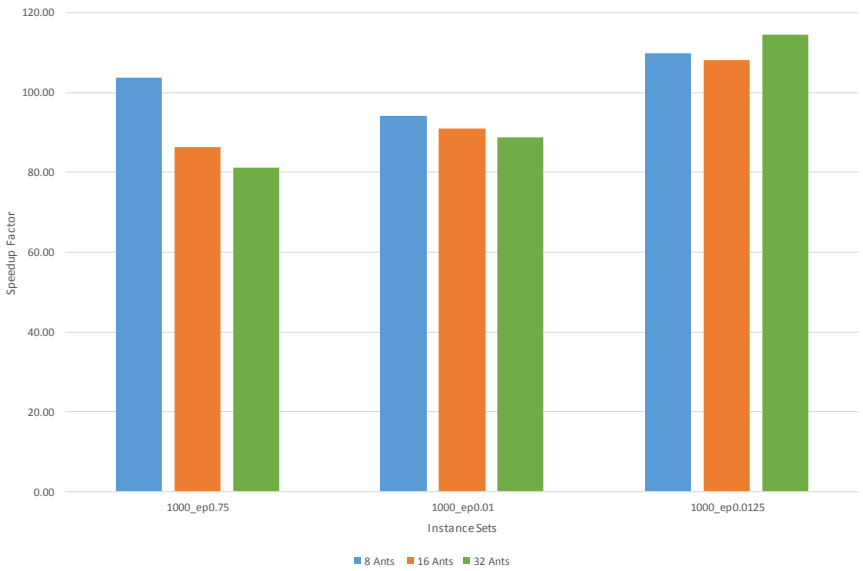


Figure 5.7: Comparison of the mean speed-up for instance sets with 1000 veritces, densities 0.75%, 1% and 1.25% using 8, 16 and 32 ants.

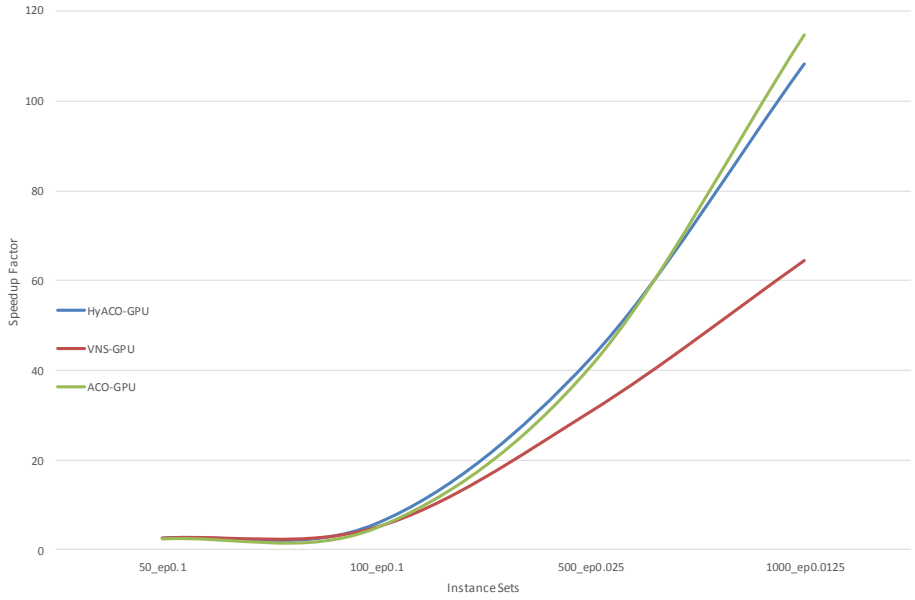


Figure 5.8: A comparison of parallel ACO, VNS and the hybrid approach for graphs with 50, 100, 500 and 1000 vertices, graph densities 20%, 10%, 2.5% and 1.25% executed with 32 ants.

CHAPTER 6

Conclusion

The purpose of this thesis was to implement, to the best of our knowledge, the first parallel hybrid metaheuristic for the Firefighter problem while combining a population-based with a single solution approach. We expected to achieve performance enhancement on various graphs along with higher solution quality. Another aim was to gain a better theoretical understanding of the field of applied heuristics in the context of the FFP while developing various strategies for the selection of vertices to be defended.

On the grounds of our conducted tests we were able to show that mechanisms with higher complexity, which take into account the changing states of the graph, tends to display better results than methods with static heuristics. More specifically, in this thesis we presented an ACO algorithm in which we consider the topology of the graph after each time step in order to generate a candidate list. This list contains only nodes that, if protected, in combination with other protected nodes, have the potential to protect neighboring nodes. In addition to that, we developed a dynamic heuristic that is created simultaneously with the generation of the candidate list. The average improvement of the solution quality achieved by this approach was 10.56% over all instances.

A further step was the development of an VNS combined with VND introducing an adaptive shaking function in order to improve the algorithm's accuracy within the node removal phase.

Tests revealed that the combination of two metaheuristics provides an improvement in the solution quality. Having combined the ACO and the VNS, we implemented a hybrid algorithm, which generated better results than each algorithm on their own. More precisely, in comparison with the original VNS the proposed hybrid approach showed better performance over all instances and achieved on average 0.3% better solution quality. When compared with the original hybrid approach the proposed hybrid computed on 118 out of 120 test runs better solutions while the average improvement lies at 0.47%.

Finally, we parallelized the three introduced algorithms, the ACO, the VNS and the hybrid, and were able to show that each single implementation outperforms its sequential counterpart in that it both yields faster and better results. The highest ACO speed-up was 141x whereas the average speed-up for the instance set with the largest graphs was 114x. The VNS achieved a speed-up of up to 106x and an average speed-up of 64x. The hybrid algorithm achieved a speed-up up to 114x and an average speed-up of 108x. Similarly to the hybridization of the sequential implementation, the hybridized parallel approach shows a remarkable improvement in the solution quality.

From the strategic point of view, we discovered that it is essential to consider the dynamics of the fire spread in order to make the right decisions and develop efficient strategies. From the metaheuristic point of view, we showed that a hybridization of parallel solutions enables significantly better results. In order to achieve a better evaluation of the proposed algorithms, further tests on higher instance sizes and different types of graphs would be necessary.

For future work, all the mentioned results hold promise for a further development of dynamic strategies that consider the changing states of a graph. Another conclusion that can be drawn is that parallelization offers advantages in the solution of highly complex problems such as the presented FFP and should therefore be researched to a greater extent.

Bibliography

[AF98] Mikhail J. Atallah and Susan Fox, editors. *Algorithms and Theory of Computation Handbook*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1998.

[AS17] Nikolaos Antoniadis and Angelo Sifaleras. A hybrid CPU-GPU parallelization scheme of variable neighborhood search for inventory optimization problems. *Electronic Notes in Discrete Mathematics*, 58:47 – 54, 2017. 4th International Conference on Variable Neighborhood Search.

[Bat68] Kenneth E. Batcher. Sorting Networks and Their Applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[BB15] Federico Busato and Nicola Bombieri. BFS-4K: An Efficient Implementation of BFS for Kepler GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):1826–1838, July 2015.

[BB17a] Federico Busato and Nicola Bombieri. A Dynamic Approach for Workload Partitioning on GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 28(6):1535–1549, June 2017.

[BB17b] Federico Busato and Nicola Bombieri. A performance, power, and energy efficiency analysis of load balancing techniques for GPUs. In *2017 12^{th} IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–8, June 2017.

[BBGM+14] Christian Blum, Maria J. Blesa, Carlos García-Martínez, Francisco J. Rodríguez, and Manuel Lozano. The firefighter problem: Application of hybrid ant colony optimization algorithms. In Christian Blum and Gabriela Ochoa, editors, *Evolutionary Computation in Combinatorial Optimisation*, pages 218–229, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[BBM16] Mauro Bisson, Massimo Bernaschi, and Enrico Mastrostefano. Parallel Distributed Breadth First Search on the Kepler Architecture. *IEEE Transactions on Parallel and Distributed Systems*, 27(7):2091–2102, July 2016.

[BCM+15] Massimo Bernaschi, Giancarlo Carbone, Enrico Mastrostefano, Mauro Bisson, and Massimiliano Fatica. Enhanced GPU-based Distributed Breadth First Search. In *Proceedings of the* 12^{th} *ACM International Conference on Computing Frontiers*, CF '15, pages 10:1–10:8, New York, NY, USA, 2015. ACM.

[BCR13] Cristina Bazgan, Morgan Chopin, and Bernard Ries. The firefighter problem with more than one firefighter on trees. *Discrete Applied Mathematics*, 161(7-8):899–908, 2013.

[BD04] Christian Blum and Marco Dorigo. The hyper-cube framework for ant colony optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(2):1161–1172, April 2004.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[BGK+95] Richard S. Barr, Bruce L. Golden, James P. Kelly, Mauricio G. C. Resende, and William R. Stewart. Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics*, 1(1):9–32, Sep 1995.

[BH12] Nathan Bell and Jared Hoberock. Chapter 26 - Thrust: A Productivity-Oriented Library for CUDA. In Wen-mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 359 – 371. Morgan Kaufmann, Boston, 2012.

[Ble90] Guy E. Blelloch. Prefix Sums and Their Applications. Technical report, School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213-3890, 1990.

[BOL+09] Hongtao Bai, Dantong OuYang, Ximing Li, Lili He, and Haihong Yu. MAX-MIN Ant System on GPU with CUDA. In *2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC)*, pages 801–804, Dec 2009.

[BPRR11] Christian Blum, Jakob Puchinger, Günther R. Raidl, and Andrea Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Appl. Soft Comput.*, 11(6):4135–4151, 2011.

[BRD01] Christian Blum, Andrea Roli, and Marco Dorigo. HC-ACO: The Hyper-Cube Framework for Ant Colony Optimization. In *IEEE Transactions on Systems, Man, and Cybernetics—Part B*, pages 399–403, 2001.

[CAP11] CAPS Enterprise, Cray Inc., NVIDIA, and the Portland Group. The OpenACC Application Programming Interface, 2011. V1.0.

[CC17] Janka Chlebíková and Morgan Chopin. The firefighter problem: Further steps in understanding its complexity. *Theor. Comput. Sci.*, 676:42–51, 2017.

[CDD+13] Vítor Santos Costa, Simone Dantas, Mitre Costa Dourado, Lucia Draque Penso, and Dieter Rautenbach. More fires and more fighters. *Discrete Applied Mathematics*, 161(16-17):2410–2419, 2013.

[CFvL11] Marek Cygan, Fedor V. Fomin, and Erik Jan van Leeuwen. Parameterized Complexity of Firefighting Revisited. In *Parameterized and Exact Computation - 6th International Symposium, IPEC 2011, Saarbrücken, Germany, September 6-8, 2011. Revised Selected Papers*, pages 13–26, 2011.

[CGN+13] José M. Cecilia, José M. GarcíA, Andy Nisbet, Martyn Amos, and Manuel UjaldóN. Enhancing Data Parallelism for Ant Colony Optimization on GPUs. *J. Parallel Distrib. Comput.*, 73(1):42–51, January 2013.

[con] Control Flow. https://www.princeton.edu/ achaney/tmve/wiki100k/docs/control_flow.html/. Accessed: 2017-8-16.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[CVY08] Leizhen Cai, Elad Verbin, and Lin Yang. Firefighting on Trees: (1-1/e)-Approximation, Fixed Parameter Tractability and a Subexponential Algorithm. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Algorithms and Computation, 19th International Symposium, ISAAC 2008, Gold Coast, Australia, December 15-17, 2008. Proceedings*, volume 5369 of *Lecture Notes in Computer Science*, pages 258–269. Springer, 2008.

[CW09] Leizhen Cai and Weifan Wang. The Surviving Rate of a Graph for the Firefighter Problem. *SIAM J. Discrete Math.*, 23(4):1814–1826, 2009.

[CWY09] Wei Chen, Yajun Wang, and Siyu Yang. Efficient Influence Maximization in Social Networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 199–208, New York, NY, USA, 2009. ACM.

[DBT00] Marco Dorigo, Eric Bonabeau, and Guy Theraulaz. Ant Algorithms and Stigmergy. *Future Gener. Comput. Syst.*, 16(9):851–871, June 2000.

[DDGK10] Audrey Delevacq, Pierre Delisle, Marc Gravel, and Michaël Krajecki. Parallel Ant Colony Optimization on Graphics Processing Units. In Hamid R. Arabnia, Steve C. Chiu, George A. Gravvanis, Minoru Ito, Kazuki Joe, Hiroaki Nishikawa, and Ashu M. G. Solo, editors, *PDPTA*, pages 196–202. CSREA Press, 2010.

[DG97] Marco Dorigo and Luca M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, Apr 1997.

[DGS+08] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. Fast Scan Algorithms on Graphics Processors. In *Proceedings of the* 22^{Nd} *Annual International Conference on Supercomputing*, ICS '08, pages 205–213, New York, NY, USA, 2008. ACM.

[DH07] Mike Develin and Stephen G. Hartke. Fire containment in grids of dimension three and higher. *Discrete Applied Mathematics*, 155(17):2257–2268, 2007.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[DMC96] Marco Dorigo, Vittorio Maniezzo, and Albert Colorni. Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(1):29–41, Feb 1996.

[DNM14] Mayank Daga, Mark Nutter, and Mitesh Meswani. Efficient breadth-first search on a heterogeneous processor. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 373–382, Oct 2014.

[DS04] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

[DS13] Laurence Dawson and Iain Stewart. Improving Ant Colony Optimization performance on the GPU using CUDA. In *2013 IEEE Congress on Evolutionary Computation*, pages 1901–1908, June 2013.

[Far12] Rob Farber. *CUDA Application Design and Development*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[Fay16] Damien Fay. *Predictive Partitioning for Efficient BFS Traversal in Social Networks*, pages 11–26. Springer International Publishing, Cham, 2016.

[FDB+14] Zhisong Fu, Harish K. Dasari, Bradley Bebee, Martin Berzins, and Bryan Thompson. Parallel Breadth First Search on GPU clusters. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 110–118, Oct 2014.

[FH13] Ohad N. Feldheim and Rani Hod. 3/2 firefighters are not enough. *Discrete Applied Mathematics*, 161(1-2):301–306, 2013.

[FKMR07] Stephen Finbow, Andrew King, Gary Macgillivray, and Romeo Rizzi. The Firefighter Problem for Graphs of Maximum Degree Three. *Discrete Math.*, 307(16):2094–2105, July 2007.

[Flo56] Merrill M. Flood. The Traveling-Salesman Problem. *Operations Research*, 4(1):61–75, 1956.

[FM09] Stephen Finbow and Gary MacGillivray. The Firefighter Problem: a survey of results, directions and questions. *Australasian J. Combinatorics*, 43:57–78, 2009.

[Fog03] Patricia Fogarty. Catching the fire on grids. Master's thesis, Department of Mathematics, University of Vermont, USA, 01 2003.

[GBRL15] Carlos García-Martínez, Christian Blum, Francisco J. Rodríguez, and Manuel Lozano. The firefighter problem: Empirical results on random graphs. *Computers & OR*, 60:55–66, 2015.

[GGN+08] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett H. Phillips, Yao Zhang, and Vasily Volkov. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28(4):13–27, 2008.

[GLMN17] Allan Grønlund, Kasper Green Larsen, Alexander Mathiasen, and Jesper Sindahl Nielsen. Fast Exact k-Means, k-Medians and Bregman Divergence Clustering in 1D. *CoRR*, abs/1701.07204, 2017.

[GP10] Michel Gendreau and Jean-Yves Potvin. *Handbook of Metaheuristics*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

[Har95] Bert Hartnell. Firefighter! An application of domination. In *20th Conference on Numerical Mathematics and Computing*, pages 218–229, 1995.

[Har04a] Stephen G. Hartke. Attempting to Narrow the Integrality Gap for the Firefighter Problem on Trees. In *Discrete Methods in Epidemiology, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, March 18-19, 2004*, pages 225–232, 2004.

[Har04b] Stephen G. Hartke. Attempting to Narrow the Integrality Gap for the Firefighter Problem on Trees. In James Abello and Graham Cormode, editors, *Discrete Methods in Epidemiology, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, March 18-19, 2004*, volume 70 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 225–232. DIMACS/AMS, 2004.

[HCZ16] Liang Hu, Xilong Che, and Si-Qing Zheng. A Closer Look at GPGPU. *ACM Comput. Surv.*, 48(4):60:1–60:20, March 2016.

[HGLS07] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12, Nov 2007.

[HKOO11] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. *SIGPLAN Not.*, 46(8):267–276, February 2011.

[HL00] Bert Hartnell and Qiyan Li. Firefighting on trees: How bad is the greedy algorithm? In *Proceedings of the Thirty-first Southeastern International Conference on Combinatorics, Graph Theory and Computing (Boca Raton, FL, 2000), Congr. Numer. 145 (2000), 187-192*, 2000.

[HM99] Pierre Hansen and Nenad Mladenović. *An Introduction to Variable Neighborhood Search*, pages 433–458. Springer US, Boston, MA, 1999.

[HM01] Pierre Hansen and Nenad Mladenovic. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449–467, 2001.

[HN07] Pawan Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[HWR15] Bin Hu, Andreas Windbichler, and Günther R. Raidl. A New Solution Representation for the Firefighter Problem. In *EvoCOP*, volume 9026 of *Lecture Notes in Computer Science*, pages 25–35. Springer, 2015.

[IKM11] Yutaka Iwaikawa, Naoyuki Kamiyama, and Tomomi Matsui. Improved Approximation Algorithms for Firefighter Problem on Trees. *IEICE Transactions*, 94-D(2):196–199, 2011.

[JC14] Ty McKercher John Cheng, Max Grossman. *Professional CUDA C Programming*. Wrox Press Ltd., Birmingham, UK, UK, 1st edition, 2014.

[Jon95] Terry Jones. One Operator, One Landscape. Technical report, Santa Fe Institute 1399 Hyde Park Road Santa Fe, New Mexico 87501 United States of America, 1995.

[KGBH16] Eric Klukovich, Mehmet Hadi Gunes, Lee Barford, and Frederick C. Harris. Accelerating BFS shortest paths calculations using CUDA for Internet topology measurements. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 66–73, July 2016.

[KGZY02] Vana Kalogeraki, Dimitrios Gunopulos, and D. Zeinalipour-Yazti. A Local Search Mechanism for Peer-to-peer Networks. In *Proceedings of the*

Eleventh International Conference on Information and Knowledge Management, CIKM '02, pages 300–307, New York, NY, USA, 2002. ACM.

[KH13] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2013.

[KM10] Andrew King and Gary Macgillivray. The Firefighter Problem for Cubic Graphs. *Discrete Math.*, 310(3):614–621, February 2010.

[LA16] Yongchao Liu and Srinivas Aluru. LightScan: Faster Scan Primitive on CUDA Compatible Manycore Processors. *CoRR*, abs/1604.04815, 2016.

[LH15] Hang Liu and H. Howie Huang. Enterprise: breadth-first graph traversal on GPUs. In *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2015.

[Lip17] Piotr Lipinski. Solving the Firefighter Problem with two elements using a multi-modal Estimation of Distribution Algorithm. In *2017 IEEE Congress on Evolutionary Computation, CEC 2017, Donostia, San Sebastián, Spain, June 5-8, 2017*, pages 2161–2168. IEEE, 2017.

[LMT10] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Neighborhood Structures for GPU-Based Local Search Algorithms. *Parallel Processing Letters*, 20(4):307–324, 2010.

[LMT13] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. GPU Computing for Parallel Local Search Metaheuristic Algorithms. *IEEE Transactions on Computers*, 62(1):173–185, Jan 2013.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[LW66] Eugene L. Lawler and D. E. Wood. Branch-and-Bound Methods: A Survey. *Oper. Res.*, 14(4):699–719, August 1966.

[LWH10] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An Effective GPU Implementation of Breadth-first Search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010. ACM.

[Mar17] Eduardo Martínez-Pedroza. A note on the relation between Hartnell's firefighter problem and growth of groups. *ArXiv e-prints*, January 2017.

[MC17] Xinxin Mei and Xiaowen Chu. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, Jan 2017.

[MG11] Duane Merrill and Andrew S. Grimshaw. High Performance and Scalable Radix Sorting: a Case Study of Implementing Dynamic Parallelism for GPU Computing. *Parallel Processing Letters*, 21(2):245–272, 2011.

[MGG15] Duane Merrill, Michael Garland, and Andrew Grimshaw. High-Performance and Scalable GPU Graph Traversal. *ACM Trans. Parallel Comput.*, 1(2):14:1–14:30, February 2015.

[MH97] Nenad Mladenović and Pierre Hansen. Variable Neighborhood Search. *Comput. Oper. Res.*, 24(11):1097–1100, November 1997.

[Mic14] Krzysztof Michalak. Auto-adaptation of Genetic Operators for Multi-objective Optimization in the Firefighter Problem. In Emilio Corchado, José A. Lozano, Héctor Quintián, and Hujun Yin, editors, *Intelligent Data Engineering and Automated Learning - IDEAL 2014 - 15th International Conference, Salamanca, Spain, September 10-12, 2014. Proceedings*, volume 8669 of *Lecture Notes in Computer Science*, pages 484–491. Springer, 2014.

[MK16] Krzysztof Michalak and Joshua D. Knowles. Simheuristics for the Multiobjective Nondeterministic Firefighter Problem in a Time-Constrained Setting. In Giovanni Squillero and Paolo Burelli, editors, *Applications of Evolutionary Computation - 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 - April 1, 2016, Proceedings, Part II*, volume 9598 of *Lecture Notes in Computer Science*, pages 248–265. Springer, 2016.

[MW03] Gary MacGillivray and Ping Wang. On the firefighter problem. *J. Combin. Math. Combin. Comput.*, 47:83–96, 2003.

[NMG17] NVIDIA Modern GPU library, 2017. [https://github.com/moderngpu/moderngpu; accessed 6-December-2017].

[NVI12] NVIDIA. Kepler GK110 Whitepaper, 2012.

[NVI16] NVIDIA. NVIDIA GeForce GTX 980, 2016. Whitepaper.

[NVI17a] NVIDIA. NVIDIA CUB library, 2017. [https://nvlabs.github.io/cub/; accessed 6-December-2017].

[NVI17b] NVIDIA Corporation. CUDA C best practices guide, 2017. Version 9.0.176.

[NVI17c] NVIDIA Corporation. cuRAND: CUDA Toolkit Documentation, 2017. Version 9.1.185.

[NVI17d] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2017. Version 9.0.176.

[NW01] Marc Najork and Janet L. Wiener. Breadth-first Crawling Yields High-quality Pages. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 114–118, New York, NY, USA, 2001. ACM.

[Phe08] Chuck Pheatt. Intel® Threading Building Blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008.

[PHM05] José A. Moreno Pérez, Pierre Hansen, and Nenad Mladenović. *Parallel Variable Neighborhood Search.* John Wiley & Sons, 1st edition, 09 2005.

[PNC11] Martín Pedemonte, Sergio Nesmachnow, and Héctor Cancela. A Survey on Parallel Ant Colony Optimization. *Appl. Soft Comput.*, 11(8):5181–5197, December 2011.

[Rei91] Gerhard Reinelt. TSPLIB - A Traveling Salesman Problem Library. *INFORMS Journal on Computing*, 3(4):376–384, 1991.

[RGAN16] L. Remis, M. J. Garzaran, R. Asenjo, and A. Navarro. Breadth-First Search on Heterogeneous Platforms: A Case of Study on Social Networks. In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 118–125, Oct 2016.

[SH00] Thomas Stützle and Holger H. Hoos. MAX-MIN Ant System. *Future Gener. Comput. Syst.*, 16(9):889–914, June 2000.

[SHG08] Shubhabrata Sengupta, Mark Harris, and Michael Garland. Efficient parallel scan algorithms for GPUs. Technical report, https://research.nvidia.com/sites/default/files/publications/nvr-2008-003.pdf, 2008.

[SHG09] Nadathur Satish, Mark J. Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In 23^{rd} *IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–10, 2009.

[SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan Primitives for GPU Computing. In *Proceedings of the* 22^{nd} *ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[SJ17] Elias Stehle and Hans-Arno Jacobsen. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 417–432, New York, NY, USA, 2017. ACM.

[SKC+10] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *SIGMOD Conference*, pages 351–362. ACM, 2010.

[Ski16] Rafał Skinderowicz. The GPU-based parallel Ant Colony System. *Journal of Parallel and Distributed Computing*, 98:48 – 60, 2016.

[Stü98] Thomas Stützle. *Parallelization strategies for Ant Colony Optimization*, pages 722–731. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[Tal09] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

[TGEF11] Yuri Torres, Arturo González-Escribano, and Diego R. Llanos Ferraris. Understanding the impact of CUDA tuning techniques for Fermi. In *HPCS*, pages 631–639, 2011.

[TIT18] TITAN - Advancing the Era of Accelerated Computing https://www.olcf.ornl.gov/titan/, 2018. Accessed: 2018-4-24.

[Ton15] Tony Scudiero. GPU Memory Bootcamp 2: Beyond Best Practices. http://on-demand.gputechconf.com/gtc/2015/presentation/S5376-Tony-Scudiero.pdf, 2015. Accessed: 2017-10-16.

[Top17] TOP500. The List. https://www.top500.org/list/2017/11/, 2017. Accessed: 2018-4-24.

[UIN12] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. An Efficient GPU Implementation of Ant Colony Optimization for the Traveling Salesman Problem. In *2012 Third International Conference on Networking and Computing*, pages 94–102, Dec 2012.

[Vol10] Vasily Volkov. Better Performance at Lower Occupancy. http://on-demand.gputechconf.com/gtc/2010/presentations/S12238-Better-Performance-at-Lower-Occupancy.pdf, 2010. Accessed: 2017-10-16.

[WDZ09] Jiening Wang, Jiankang Dong, and Chunfeng Zhang. Implementation of Ant Colony Algorithm Based on GPU. In *Sixth International Conference on Computer Graphics, Imaging and Visualization: New Advances and Trends, CGIV 2009, 11-14 August 2009, Tianjin, China*, pages 50–53, 2009.

[WFW10] Weifan Wang, Stephen Finbow, and Ping Wang. The surviving rate of an infected network. *Theor. Comput. Sci.*, 411(40-42):3651–3660, 2010.

[Whi09] "NVIDIA's Next Generation CUDA Compute Architecture: Fermi.", 2009. Whitepaper.

[WKP11] Craig M. Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi GF100 GPU Architecture. *IEEE Micro*, 31(2):50–59, March 2011.

[XcF10] Shucai Xiao and Wu chun Feng. Inter-block GPU communication via fast barrier synchronization. In *IPDPS*, pages 1–12, 2010.

[YLZ13] Shengen Yan, Guoping Long, and Yunquan Zhang. StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization. *SIGPLAN Not.*, 48(8):229–238, February 2013.