# Randomized Computation Offloading Algorithms for Mobile Cloud Computing

# RANDOMIZED COMPUTATION OFFLOADING ALGORITHMS FOR MOBILE CLOUD COMPUTING

BY

HALEH SHAHZAD, MSc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Doctor of Philosophy (2019) McMaster University

(Electrical & Computer Engineering) Hamilton, Ontario, Canada

| | |
|---|---|
| TITLE: | Randomized Computation Offloading Algorithms for Mobile Cloud Computing |
| AUTHOR: | Haleh Shahzad<br>MSc., (Electrical Engineering)<br>AmirKabir University of Technology, Tehran, Iran |
| SUPERVISOR: | Dr. T. H. Szymanski |
| NUMBER OF PAGES: | xv, 133 |

*To my love,* ***Majid****;*

*and To my loving parents,* ***Ahmad Reza*** *and* ***Mah Monir***

# Lay Abstract

Many applications envisioned for mobile devices require intensive computing power for application execution. Executing these computation intensive tasks on the mobile device leads to large delays and requires substantial energy from the limited battery storage on the device. Mobile Cloud Computing provides data processing and storage in powerful and centralized computing platforms located in the cloud. Having access to these cloud resources helps the mobile devices to run the application faster and with less battery usage. Computation offloading is a technique by which some of the tasks in a mobile application can be offloaded for execution on a remote server, so that mobile energy use can be reduced. There are different parameters that can affect the procedure of offloading and it is the responsibility of the mobile device to decide which tasks from the application should be executed locally and which ones should be executed remotely in order to reduce the energy consumption on the mobile device and reduce the time delay of the computation. The algorithms that are designed to make these decisions regarding the task execution location are called offloading algorithms. The goal of this thesis is to design efficient offloading algorithms for mobile cloud computing.

# Abstract

*Computation offloading* occurs when a mobile application arranges for tasks to be executed remotely, rather than running them locally on the device itself. This mechanism can be used to reduce mobile device energy consumption, resulting in improved battery lifetime. In this thesis, new algorithms that generate offloading decisions for mobile device applications are presented. *Randomization* is the main technique that is explored, where the algorithms iteratively improve an offloading decision vector by generating random bit strings that represent the task offload decisions. If fragments of these bit strings improve the performance, they are incorporated into the decision vector in a process similar to genetic optimization. In the experiments reported in this thesis, the proposed algorithm typically find good to excellent quality solutions, with low computational overhead compared to existing algorithms. Furthermore, the algorithms scale gracefully as the problem size grows, maintaining high computational efficiency and good solution quality.

# Acknowledgements

I would like to express my gratitude to my primary supervisor, Dr. Ted H. Szymanski, for his support of my PhD research. I would also like to thank the members of my supervisory committee, Dr. Dongmei Zhao and Dr. Terence D. Todd, for providing constructive feedback and assistance throughout my research and the writing of my thesis. I am especially grateful to Dr. Todd for being such a supportive advisor and a wonderful mentor, and for giving me the confidence and encouragement to finish my PhD. Your ethics, professionalism, kindness and your passion for your work have greatly influenced me, and I hope that I can use them as my inspiration in my next steps in life. Special thanks are also due to Dr. Timothy Davidson who has taken over as my supervisor due to the illness of my primary supervisor. I'm very grateful of your insight, critiques and support. Your vision, feedback and encouragement have pushed this thesis far and are greatly appreciated.

I would also like to express my appreciation to Cheryl Gies, the graduate administrative assistant in the Department of Electrical and Computer Engineering, for her continuous support.

I would, of course, like to thank my lovely parents, Ahmad Reza and Mah Monir, for their unwavering belief in my abilities, their endless support and encouragement through the highs and the lows of these years. Thank you for always believing in me

and pushing me to try harder. To my wonderful brother, Hamed, who has always been there for me with his continuous support and love.

Last, but certainly not least, my very special thanks to my loving husband, Majid, who has been always a constant source of support and encouragement during the challenges of my PhD journey. My dearest Majid, you were always there with me through the toughest moments, giving advice, support and love. Thanks for putting up with me through the hard times. Your sacrifices, your soothing words and your big heart helped me face all the obstacles and continue with my work. Having you by my side made me a stronger person and has propelled me forward.

# Notation and Abbreviations

| | |
|---|---|
| IoT | Internet of Things |
| MEC | Mobile Edge Computing |
| MCC | Mobile Cloud Computing |
| AR | Augmented Reality |
| ML | Machine Learning |
| SDP | Semidefinite Programming |
| SDR | Semidefinite Relaxation |
| LC | Local-Cloud |
| QCQP | Quadratically Constrained Quadratic Program |
| CAP | Computing Access Point |
| VM | Virtual Machine |
| BFS | Brute-Force Search |
| DVFS | Dynamic Voltage and Frequency Scaling |
| DVS | Dynamic Voltage Scaling |
| BPSO | Binary Particle Swarm Optimization |
| DP | Dynamic Programming |
| NE | Nash Equilibrium |
| BS | Base Station |
| GA | Genetic Algorithm |

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Computation Offloading

The vision of *Mobile Cloud/Edge Computing* is to provide real time access to the computation resources and storage that is available in the Internet cloud (Mao *et al.*, 2017b). Computation offloading is a technique by which some of the tasks in a mobile application can be offloaded for execution on a remote server, so that mobile energy use can be reduced (Li *et al.*, 2001; Rong and Pedram, 2003). In order to do this, *Offloading Algorithms* are needed to determine the place of execution for each task. The main goals of the offloading algorithm are to: (a) minimize the overall energy consumption of a mobile application, and (b) meet the execution time constraints of the application's computational tasks. As predicted by Meskar *et al.* (2017), in the near future, over 50 percent of mobile data traffic will be used for cloud based application support. For this reason, offloading algorithms are needed to efficiently access this functionality. Unfortunately, the problem of partitioning a set of tasks into two sets of locally-executed and remotely-executed tasks is, in general, NP-hard

(Gu *et al.*, 2004).

The ability to offload a task from a mobile device to a remote server will depend upon several external conditions including the instantaneous bandwidth and latency in both the wireless access and cloud networks. As a result, there is an emphasis in the literature on algorithms that typically produce good solutions in a reasonable time frame (see Section 2.1).

## 1.2 Mobile Edge/Fog Computing

One of the main issues in mobile cloud computing is the delay associated with accessing cloud resources. The dual goals of minimizing energy use and execution time therefore favour offloading tasks to nearby *Edge* or *Fog* servers, rather than to more distant *Cloud* servers. This has led to the emerging concepts of Mobile Edge Computing (MEC) and Fog Computing. Figure 1.1 shows the basic network architecture with fog, edge and cloud servers. Edge/Fog computing will provide powerful computation and storage resources at the edge of future networks. These edge/fog servers are beneficial for delay sensitive and real-time interactive applications, since they are physically close to the user, typically within the same city and within tens of kilometers, and sometimes significantly closer.

There are many applications where MEC can be utilized to enhance performance, such as augmented reality, video acceleration, dynamic content delivery and connected vehicles (Tang and He, 2018). The presence of edge-servers allows a mobile device to avoid the traffic congestion of accessing remote cloud-servers through the Internet (Al-Fuqaha *et al.*, 2015). Considering *Augmented Reality* applications as an example, it combines a person's view of a real world object and the augmented content display.

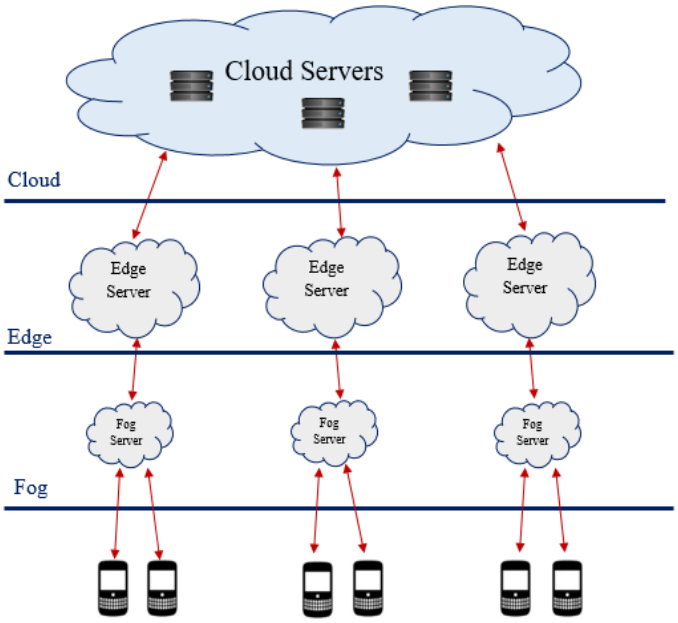


Figure 1.1: Principal Architecture with the Presence of Fog, Edge and Cloud Servers

Consider a visitor in an art gallery using an augmented reality application to display the supplementary content of paintings when those paintings are recognized based on images provided by the camera of the mobile device. By utilizing the MEC servers, the latency of the recognition process and the synthesis of the display of the supplemental content may be dramatically reduced. That will significantly improve the visitor's experience (Tang and He, 2018).

MEC typically refers to the base stations (BSs) as hosting the available offload servers, while fog computing is a generalized form of mobile edge computing. In fog computing, the type of servers is more broad, ranging from smart phones to set-top boxes (Mao *et al.*, 2017b). Tong *et al.* (2016) viewed the fog servers as geo-distributed desktops or workstations that directly receive workload from mobile devices via wireless links. These fog servers are connected to the edge and cloud servers through the Internet backbone, and work cooperatively in a hierarchical manner. If the workloads

that are received in any layer exceed its computational capacity, the excess can be offloaded to the next layer (Tong *et al.*, 2016).

## 1.3 Offloading for Mobile Applications

Several different classes of applications can benefit from computational offloading, such as complex computer games, navigation systems, video surveillance systems, and so on. Face recognition and augmented reality (AR) applications are two other popular examples. Face recognition, as shown in Figure 1.2, typically consists of five main tasks including image acquisition, face detection, pre-processing, feature extraction, and classification. The image acquisition task needs to be executed at the mobile device for supporting the user interface, while the other tasks could be offloaded for cloud processing. These often require complex computations such as signal processing and machine learning (ML) algorithms.

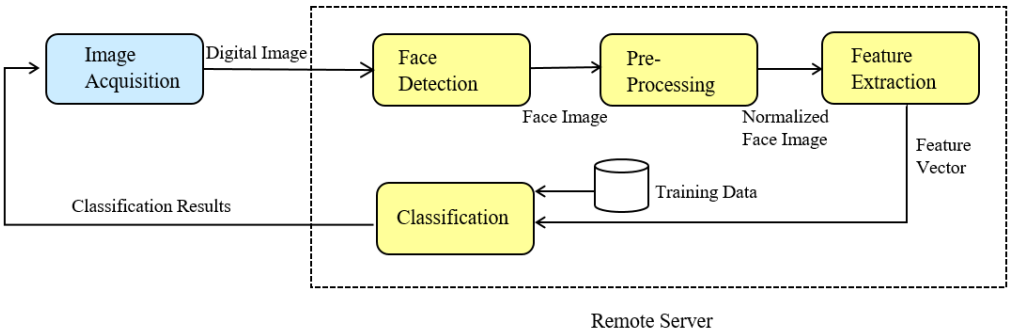


Figure 1.2: Main Tasks in a Face Recognition Application. Original Figure by Mao *et al.* (2017a), Copyright © 2017, IEEE

AR applications are able to combine data with physical reality and typically include five critical components as shown in Figure 1.3. These are called, the video source (which obtains raw video frames from the mobile camera), a tracker (which

tracks the position of the user), a mapper (which builds a model of the environment), an object recognizer (which identifies known objects in the environment), and a renderer (which prepares the processed frame for display)(Mao *et al.*, 2017b). Among these components, the video source and renderer should be executed locally, while the most computation-intensive components, i.e., the tracker, mapper and object recognizer, can be offloaded for remote execution (Mao *et al.*, 2017b).

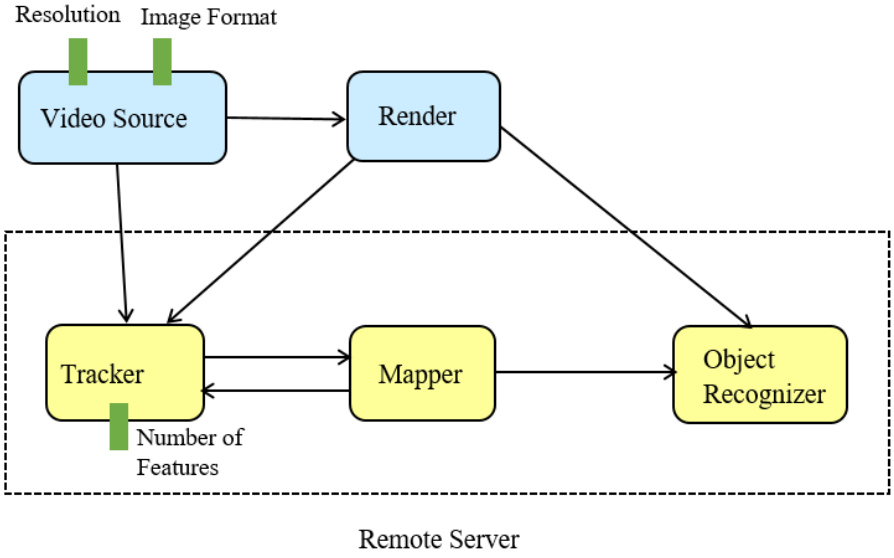


Figure 1.3: Main Tasks in an Augmented Reality Application. Original Figure by Mao *et al.* (2017a), Copyright © 2017, IEEE

Computer chess is one of the most popular games in the world. A chessboard has $8 \times 8 = 64$ positions, and each player controls 16 pieces at the beginning of the game. According to Kumar and Lu (2010), chess is Markovian. Each piece may be in one of the 64 possible locations, and needs 6 bits to represent the location. (Some pieces have restrictions on their moves, e.g., a bishop can move to only half of the board, and has only 32 possible locations). To represent the current state of a chess game, 6 bits/piece $\times$ 32 pieces = 192 bits, i.e., 24 bytes are sufficient. The state of a chess

game can therefore be summarized in 24 bytes, which is smaller than the payload of a typical wireless packet. Since the amount of computation in a chess game can be extremely large and the amount of data that must be exchanged between the mobile device and the remote server is very small, chess is an example where computational offloading is beneficial for most wireless networks.

It is clear that applications have a wide variety of offloadable tasks with various workloads and data input sizes. The optimal offloading decisions will vary depending on the mentioned factors combined with 1) the availability of the network fog, edge and cloud servers, and 2) the associated network bandwidth. The goal of this thesis is to quickly find good approximate solutions to this type of problem.

## 1.4 Thesis Outline

Chapter 2 reviews the literature in the area of computation offloading algorithms and Chapter 3 provides details of the application models. The DPH (Dynamic Programming with Hamming Distance Termination) and DPR (Dynamic Programming with Randomization) algorithms are the primary instances of the proposed family of randomized algorithms for offloading decision making and are given in Chapter 4. Two somewhat different algorithms, ROA-V1 (Randomized Offloading Algorithm) and ROA-V2 algorithms are introduced in Chapter 5. The Context Aware ROA algorithm is then presented in Chapter 6. The thesis is concluded in Chapter 7.

## 1.5 Original Contributions

Various Randomized Offloading Algorithms are proposed, which can quickly find good quality solutions for the computational offloading problems considered herein. Simulation results are presented and used to characterize their performance. The proposed algorithms are compared with other approaches, including the optimal solution of the offloading optimization problem, which can be found using a Brute-Force Search (BFS).

An algorithm called "Dynamic Programming with a Hamming Distance termination criterion" (DPH) is presented in Chapter 4. The principles of this algorithm were presented at the IEEE Canadian Conference on ECE in 2016 (Shahzad and Szymanski, 2016a). An algorithm called "Dynamic Programming with biased Randomized" (DPR) is also presented in Chapter 4. The principles of this algorithm were presented at the IEEE International Conference on Cloud Computing in 2016 (Shahzad and Szymanski, 2016b). The ROA-V1 algorithm considered in Sections 5.4-5.7 was first described in an internal manuscript entitled "Randomized Offloading Algorithms for Green Mobile Cloud and Mobile Edge Computing" (Shahzad and Szymanski, 2017), and was later incorporated into a larger manuscript (Shahzad and Szymanski, 2018). The ROA-V2 algorithm considered in Section 5.8 was first described in an internal manuscript entitled "An Improved Randomized Offloading Algorithm (v2) for evaluating mobile cloud, fog and edge computing systems"(Szymanski and Shahzad, 2018) and was later incorporated into Shahzad and Szymanski (2018). These papers represent joint work with my supervisor Prof. T.H. Szymanski.

Chapter 6 explores an algorithm that incorporates task graph information into

the process of generating biased random vectors and determining the number of substrings the candidate vector will be partitioned into. This chapter will be published in a future work.

It is observed that the use of randomization is a relatively simple, and yet powerful, technique that typically yields better solutions than prior methods, including dynamic programming, randomized rounding, and genetic optimization. As an offloading algorithm that will be embedded in the mobile device, the computational complexity of the algorithm is critical, since adding too much overhead leads to excessive battery consumption. Moreover, in terms of the user experience, the offloading algorithm itself should not be time consuming. Therefore, it is important to have fast and efficient offloading algorithms to help improve the mobile user experience. The proposed offloading algorithms directly address these criteria.

# Chapter 2

# Prior Work

This chapter presents an overview of previous work related to computation offloading solutions. First, we start with a discussion of offloading solutions for single user systems, which will be the focus of this thesis. Then, for context, we review multiple user systems. In both cases we discuss the objective and complexity of the proposed solutions. The goal of the majority of offloading algorithms is to minimize energy consumption at the mobile device ($E$) while satisfying an application execution time constraint ($T$), or, to find a trade-off between $E$ and $T$.

## 2.1 Single-User Case

In the single-user case, there is one mobile user with an application that contains a number of tasks that are candidates for offloading. The following review of the literature for the single-user case is a refined and significantly expanded version of that in Shahzad and Szymanski (2018).

The MAUI system (Cuervo *et al.*, 2010) enables energy-aware offloading of mobile

tasks to the cloud within the Microsoft .NET programming framework. MAUI first inspects the application, then it formulates a binary integer linear program to determine the actions to be taken during computation offloading. The objective of MAUI is to maximize the difference between energy savings and the energy cost of data transfer to execute a task remotely. The savings are essentially the energy cost if the task had been executed locally. Computation offloading systems called CloneCloud (Chun *et al.*, 2011) and ThinkAir (Kosta *et al.*, 2012) provided further improvements and included automatic partitioning, migration, and remote execution. ThinkAir focuses on the elasticity and scalability of the cloud and enhances the power of mobile cloud computing by parallelizing task execution using multiple virtual machine (VM) images.

Many computation offloading studies have chosen to focus on the static channel case, i.e., where the data rates of the network are taken to be fixed. Niu *et al.* (2013) proposed a method to partition tasks into local and remote execution sets in a dynamic fashion that adapts to changes in bandwidth. For example, more tasks tend to be selected for local execution when the network bandwidth temporarily decreases. The problem formulations used in many of these prior systems are based on finding approximate solutions to NP-Hard binary integer programming problems, i.e., the binary partitioning of tasks into local and remotely executed sets.

Huang *et al.* (2012) presented a dynamic offloading algorithm based on Lyapunov optimization. Their approach creates a relationship between the optimal solution to the offloading problem and an approximate solution. The objective is to find a good approximate solution. This scheme has a large execution time, as it needs many iterations to find a final solution. Yang *et al.* (2013) proposed an adaptive

partitioning scheme using a genetic algorithm to find good solutions to the NP-hard partition problem.

Toma and Chen (2013) introduced an approach to determine offloading decisions that uses dynamic programming to build a three-dimensional table. Their proposed algorithm executes with *pseudo-polynomial* time complexity. A pseudo-polynomial time algorithm is where the algorithm running time is polynomial in the numeric value of the input, but is exponential in the encoded input length. Their work did not consider mobile device energy use, which is an important consideration for battery powered devices.

Muñoz *et al.* (2013) provided a general framework that allows the joint allocation of radio and computational resources resulting in an optimized trade-off between energy consumption and latency.

Chen *et al.* (2015) considered a system that finds computation offloading decisions using *Semidefinite Relaxation* (SDR) followed by randomized rounding. They considered a single mobile device with multiple independent tasks that can access either: 1) a nearby *Computing Access Point* (CAP) that has limited computation power, or, 2) a remote cloud server with significant computational power. The algorithm starts by solving a semidefinite relaxation of the binary integer program. A customized Gaussian randomized rounding technique is then used to convert the floating point values to binary offloading decisions. The best of these solutions is then used as the final decision. Their results were compared with those from a *Brute-Force Search*, i.e., where all decision vectors are enumerated to find the true optimum. Their results indicate that the SDR algorithm can find solutions within 1% of the optimal energy and delay results, but the time complexity of the algorithm was not discussed.

Solving the underlying semidefinite programming (SDP) problem and implementing relaxation steps is computationally expensive, in general.

Kamoun *et al.* (2015) considered mobile edge computing and proposed an online and pre-calculated offline strategy in order to minimize the power consumption of the device while satisfying a predefined delay constraint. The online strategy used a Lagrangian relaxation approach while the offline strategy used pre-calculated parameters such as the arrival rate and the channel condition. The offline strategy is able to minimize the signaling overhead that leads to performance improvement of up to 50% for low and medium loads. The authors then investigated more offline solutions (Labidi *et al.*, 2015a), by using dynamic programming tools leading to deterministic and randomized offline strategies.

Cao *et al.* (2015) proposed an algorithm based on combinatorial optimization for an application with $N$ offloadable components. The basic idea in this approach is to divide the problem into several subproblems, and solve the subproblems first. Then the solution of the original problem is determined from the solutions of these subproblems.

A partial offloading scheme with a trade-off between the energy consumption and latency is considered by Munoz *et al.* (2015). An algorithm is presented by Lin *et al.* (2015), which starts from a minimum application completion time scheduling solution and subsequently performs energy reduction by migrating tasks ($N$ tasks) among the local cores ($k$ heterogeneous cores) and the cloud and by applying the DVFS (dynamic voltage and frequency scaling) technique.

A heuristic algorithm using Binary Particle Swarm Optimization (BPSO) is proposed by Deng *et al.* (2016).

Liu *et al.* (2016) provided a one dimensional search algorithm where the computation tasks are scheduled based on the queuing state of the task buffer, the execution state of the local processing unit, as well as the state of the transmission unit.

A low-complexity Lyapunov Optimization based Dynamic Computation Offloading (LODCO) algorithm is proposed by Mao *et al.* (2016a), which jointly decides the offloading decision, the CPU-cycle frequencies for mobile execution, and the transmit power for computation offloading.

Wang *et al.* (2016b) used the dynamic voltage scaling (DVS) technique to minimize energy consumption while satisfying a time constraint. Using this technique, the mobile device can adaptively adjust its computational speed to reduce energy consumption or shorten the computing time.

Kao *et al.* (2017) used a dynamic programming algorithm to minimize delay. They identified a subset of problem instances where the application task graphs can be described as serial trees.

The ROA algorithms proposed in Chapters 5 and 6 of this thesis are built upon some preliminary results reported by Shahzad and Szymanski (2016a) and Shahzad and Szymanski (2016b). An offloading algorithm called *Dynamic Programming with Hamming Distance Termination* (DPH) was presented by Shahzad and Szymanski (2016a). This algorithm exploits the fact that the dynamic programming technique splits a complex optimization problem into many smaller problems. These smaller problems can be solved once and their solutions can be stored in a lookup table. The DPH algorithm combines a 2-dimensional DP lookup table with randomization and

a Hamming distance termination criterion to find good offloading solutions. An extended version of the DPH algorithm called *Dynamic Programming with Randomization* (DPR) was presented by Shahzad and Szymanski (2016b). The DPR algorithm incorporates *Dynamic Programming* with "biased randomization". It generates random bit strings with a biased probability of generating 0s, which represent decisions to offload tasks, and uses these biased bit strings to update the 2-dimensional DP table. The proposed ROA algorithms presented in Chapter 5 are simpler and more powerful than DPH and DPR. They remove the 2-dimensional DP lookup table, simplify the rules for improving the decision vector, and more closely resemble *Genetic Optimization*.

The prior work for the single user case that is discussed in this section is summarized in Table 2.1.

Table 2.1: Comparison of Offloading Methods for a Single User

| **Reference** | **Objective** | **Proposed Solution** | **E/T reduction w.r.t. All-Local execution** | **Complexity** |
|---|---|---|---|---|
| MAUI (Cuervo *et al.*, 2010) | Maximize $E$ Satisfy $T$ | Algorithm based on 0-1 integer linear programming | Up to 45% reduction in $E$ | Not Stated |
| CloneCloud (Chun *et al.*, 2011) | Minimize $E$ Satisfy $T$ | Algorithm based on 0-1 integer linear programming | Up to 95% reduction in $E$ | Not Stated |
| ThinkAir (Kosta *et al.*, 2012) | Minimize $E$ and $T$ | Algorithm based on parallelizing task execution using multiple virtual machines (VM) | Not Stated | Not Stated |
| (Huang *et al.*, 2012) | Minimize $E$ Satisfy $T$ | Algorithm based on Lyapunov optimization | About 50% reduction in $E$ | Not Stated |

| | | | | |
|---|---|---|---|---|
| CLOUDNET (Zhang *et al.*, 2012) | Minimize $E$ Satisfy $T$ | Linear time searching algorithm with $V$ remoteable tasks and $R$ invoking relations | About 50% reduction in $E$ | $O(V+R)$ |
| (Yang *et al.*, 2013) | Minimize $T$ | Genetic algorithm using crossover and mutations | Not Stated | Not Stated |
| (Toma and Chen, 2013) | Minimize $E$ Satisfy $T$ | Algorithm based on dynamic programming with a 3D table for $n$ tasks | Not Stated | $O(n \log n + nT^2)$ |
| (Muñoz *et al.*, 2013) | Tradeoff between $E$ and $T$ | joint allocation of communication and computational resources | Not Stated | Not Stated |
| LC/LAC (Chen *et al.*, 2015) | Minimize a function of $(E,T)$ | Algorithm Based on semidifinite programming with integer relaxation | Not Stated | Not Stated |
| (Kamoun *et al.*, 2015) | Minimize $E$ Satisfy $T$ | Online learning based strategy, Offline precalculated strategy | Up to 78% reduction of $E$ | Not Stated |
| (Labidi *et al.*, 2015a) | Minimize $E$ Satisfy $T$ | Deterministic and randomized Offline strategies | Up to 78% reduction of $E$ | Not Stated |
| (Cao *et al.*, 2015) | Minimize $E$ Satisfy $T$ | Algorithm based on the combinatorial optimization method for $N$ tasks | Up to 47% reduction of $E$ | $O(N)$ |
| (Munoz *et al.*, 2015) | Tradeoff between $E$ and $T$ | Iterative algorithm finding the optimal value of the number of bits sent on the uplink | Up to 97% reduction of $E$ | Not Stated |
| (Lin *et al.*, 2015) | Minimize $E$ Satisfy $T$ | Algorithm based on dynamic voltage and frequency scaling with $N$ tasks and $k$ cores | Up to 19.6% reduction of $E$ | $O(N^3.k)$ |

| | | | | |
|---|---|---|---|---|
| (Deng *et al.*, 2016) | Minimize $E$ Satisfy $T$ | Binary particle swarm optimization with $G$ iterations, $K$ particles of $N$ dimension | Up to 25% reduction in $E$ | $O(G.K.N^2)$ |
| DPH (Shahzad and Szymanski, 2016a) | Minimize $E$ Satisfy $T$ | Heuristic algorithm based on dynamic programming and randomization | Up to 40% reduction in $E$ | Not Stated |
| DPR (Shahzad and Szymanski, 2016b) | Minimize $E$ Satisfy $T$ | Heuristic algorithm based on dynamic programming and biased randomization | Up to 50% reduction in $E$ | Not Stated |
| (Liu *et al.*, 2016) | Minimize $T$ | One dimentional search algorithm finding the optimal offloading policy | Up to 80% reduction of $T$ | Not Stated |
| LODCO (Mao *et al.*, 2016a) | Minimize $T$ | Lyapunov optimization-based dynamic computation offloading | Up to 64% reduction of $T$ | Not Stated |
| EPCO (Wang *et al.*, 2016b) | Minimize $E$ Satisfy $T$ | Using dynamic voltage scaling to achieve maximum allowed latency | Not Stated | Not Stated |
| (Kao *et al.*, 2017) | Minimize $T$, Satisfy resource Constraints | Using dynamic programming with $d_{in}$ maximum graph task indegree, $N$ tasks, $M$ devices, $L$ is the length of the longest path, $R$ is the dynamic range | Not Stated | $O\left(\frac{d_{in}NM^2L}{\varepsilon \log_2 R}\right)$ |

## 2.2 Multi-User Case

We have previously considered existing methods that deal with offloading for a single user. In this section we review the case where multiple users wish to access the remote cloud servers.

Barbarossa *et al.* (2013) considered a single cloud and proposed a method to jointly optimize the communication and computation resources to minimize the power consumption at the mobile, while satisfying an average application latency constraint. Sardellitti *et al.* (2014b) proposed a distributed iterative algorithm based on Successive Convex Approximation (SCA) techniques converging to a local optimal solution of the original non-convex problem. A local optimal solution is also called a local minimum solution, i.e., the best solution within a small region of possible solutions. The work of Sardellitti *et al.* (2014b) was extended by Sardellitti *et al.* (2014a) to multiple clouds. The goal is the optimal assignment of each mobile user to a cloud server while satisfying time constraints. The results show that with increasing numbers of clouds, the energy consumption of the user can be decreased.

A trade-off between the energy consumption and the execution delay for multiple users is considered by Muñoz *et al.* (2014). They adjust the data rate for the mobile user to achieve a target energy savings while minimizing delay.

Zhao *et al.* (2015) considered the joint optimization of radio and computational resources and the problem is formulated as a nonlinear constraint problem. The users are assumed to be able to determine whether to partition the application and how many parts should be offloaded to the MEC. The proposed algorithm is based on the delay constraint and the application type of each mobile user.

Offline and online dynamic programming approaches are investigated by Labidi

*et al.* (2015b). Deterministic algorithms to find the optimal scheduling offloading policy were proposed.

A trade-off between the energy consumption at the mobile device and the execution delay in a multi-channel environment is proposed by Chen *et al.* (2016b). The authors proposed a distributed computation offloading algorithm based on game theory achieving a Nash equilibrium. Up to 40% reduction of energy is reported for 50 users.

Chen *et al.* (2016a) consider multiple users, one computing access point (CAP) and one cloud server. The goal is to minimize the overall cost of energy, computation and maximum delay. They proposed an algorithm based on semidefinite relaxation and randomization. System cost is decreased by up to 45% and the overall complexity of the proposed algorithm is $O(N^6)$ per iteration, where $N$ is the number of users. This complexity may be too high for a large number of users.

You and Huang (2016) proposed a TDMA based system with a threshold based structure where time is divided into time slots. During each slot, the users may offload a part of their data to the MEC according to their priorities. The priority of each user is defined based on its channel gain and local computing energy consumption. As a result, higher priorities are given to users that are not able to meet the application latency constraints.

A trade-off between the energy consumption and the execution delay is considered by Mao *et al.* (2016b). An online algorithm based on Lyapunov optimization is proposed. At each time slot, the optimal frequencies of the local CPUs are obtained in closed form, while the optimal transmit power and bandwidth allocation for computation offloading are determined with the Gauss-Seidel method. The proposed

algorithm is able to control the power consumption and the execution delay depending on the selected priority. A 90% reduction in the energy consumption is shown in the paper.

You *et al.* (2017) investigated resource allocation based on time-division multiple access (TDMA) and orthogonal frequency-division multiple access (OFDMA) for multiple users.

A search-adjust algorithm based on genetic optimization with limited cloud resources is proposed by Liu *et al.* (2017). The offloading requests submitted by all users over a specific period of time are considered and the best possible offloading solution is obtained based on the computing power and bandwidth of each user, as well as available cloud resources.

Prior work for the multi-user case that is reviewed in this section is summarized in Table 2.2.

Table 2.2: Comparison of existing offloading methods for multiple users

| **Reference** | **Objective** | **Proposed Solution** | **E/T reduction vs. All-Local execution** | **Complexity** |
|---|---|---|---|---|
| (Barbarossa *et al.*, 2013) | Minimize $E$ Satisfy $T$ | Joint allocation of communication and computation resources | Not Stated | Not Stated |
| (Sardellitti *et al.*, 2014b) | Minimize $E$ Satisfy $T$ | Iterative algorithm based on successive approximation techniques (SCA) | Not Stated | Not Stated |
| (Sardellitti *et al.*, 2014a) | Minimize $E$ Satisfy $T$ | Iterative algorithm based on successive approximation techniques (SCA) | Not Stated | Not Stated |

| | | | | |
|---|---|---|---|---|
| (Muñoz *et al.*, 2014) | Tradeoff between $E$ and $T$ | Joint allocation of communication and computational resources | Up to 90% reduction of $E$ | Not Stated |
| (Zhao *et al.*, 2015) | Minimize $E$ Satisfy $T$ | Application type and delay based resource allocation scheme with $N$ users | Up to 40% reduction of $E$ | $O(N)$ |
| (Labidi *et al.*, 2015b) | Maximize $E$ Satisfy $T$ | Deterministic offline and online strategies based on dynamic programming | Not Stated | Not Stated |
| (Yang *et al.*, 2015) | Maximize $T$ | SearchAdjust algorithm with limited cloud resources | Not Stated | Not Stated |
| (Zhang *et al.*, 2016) | Minimize $E$ Satisfy $T$ | A three stage scheme based on TYPE classification and priority assignment with $N$ users, $I$ iterations and $K$ channels | Up to 18% reduction in $E$ | $O(\max(I^2 + N, IK + N))$ |
| (Chen *et al.*, 2016b) | Tradeoff between $E$ and $T$ | A game theoretic approach as a distributed offloading algorithm | Up to 40% reduction in $E$ | Not Stated |
| (Chen *et al.*, 2016a) | Minimize $E$ Satisfy $T$ | Algorithm based on the semidefinite relaxation and randomization | Up to 45% reduction of total cost | $O(N^6)$ |
| (You and Huang, 2016) | Minimize $E$ Satisfy $T$ | Optimal resource allocation policy with threshold based structure for TDMA | Not Stated | Not Stated |
| (Mao *et al.*, 2016b) | Tradeoff between $E$ and $T$ | Lyapunov optimization based dynamic computation offloading | Up to 90% reduction of $E$ | Not Stated |
| (You *et al.*, 2017) | Minimize $E$ Satisfy $T$ | Optimal resource allocation policy with threshold based structure for TDMA and FDMA | Not Stated | $O(K + N)$ |

| | | | | |
|---|---|---|---|---|
| (Liu *et al.*, 2017) | Joint optimization of $E$ and $T$ | A search and adjust based algorithm using genetic optimization | Up to 50 % reduction of $E$ | Not Stated |
| (Meskar *et al.*, 2017) | Minimize $E$ Satisfy $T$ | modeled as a competitive game and each of $N$ users minimizes its own energy consumption | about 50% reduction of $E$ | $O(N^2)$ |

## 2.3 Summary

This chapter has provided a review of research related to computational offloading, with emphasis on the objective, complexity and the efficiency of the solutions in terms of reduction in energy consumption and execution time. Various approaches that have been proposed in recent years for both single and multiple user systems were investigated and summarized. The focus of the majority of the offloading algorithms is to minimize the energy consumption at the mobile device ($E$) while satisfying an application execution time constraint ($T$) or to find a trade-off between these two objectives.

# Chapter 3

# Application Models

In this chapter, different types of mobile device application models are first discussed. The chapter concludes by considering the types of task graphs that are used in our performance results.

## 3.1 Different Types of Application Models

There are a number of issues that are critical in modeling computation tasks, including latency, bandwidth utilization, context awareness, generality, and scalability. The computation-task models that are most often used in the computational offloading literature are: 1) Binary Offloading, and, 2) Partial Offloading (Mao *et al.*, 2017a).

In binary offloading, the task (application) cannot be partitioned into smaller parts and has to be executed as a whole, either locally at the mobile device or offloaded to the remote server. However, many mobile applications are composed of multiple components (tasks) and each task can be executed either on the mobile device or remote server, making it possible to implement partial computation offloading (Mao

*et al.*, 2017a). In this thesis, the partial model is considered.

There are two types of applications in terms of offloadable tasks. In the first, the application can be divided into $S'$ offloadable tasks. Since each task may differ in the amount of data and required computation, it is essential to decide which tasks should be offloaded to the remote server. There is the possibility that this type of application is fully offloaded if no parts are processed locally. The second type of application is always composed of some non-offloadable part(s) that cannot be offloaded and some that can (Mach and Becvar, 2017).

The simplest task model for partial offloading is the *parallel model*, where the task inputs are independent so the tasks can be executed by different entities, e.g., parallel execution at the mobile device and remote servers. However, the dependency between different tasks in many applications cannot be ignored as it significantly affects execution and computation offloading (Mao *et al.*, 2017a). In this case, the outputs of some tasks are the inputs to others. In addition, because of software or hardware constraints, some tasks can only be executed locally. This model is more complicated than the parallel model, and considers the inter-dependency among different computation tasks. This model is called the *task graph model* (Mao *et al.*, 2017a) and we use the term *general model* to reference it. Both these models are explained in more detail in this chapter.

### 3.1.1 Parallel Model

This model is considered in many references such as (Toma and Chen, 2013; Zhang *et al.*, 2017; Wang *et al.*, 2016a, 2017; Chen *et al.*, 2015). In the parallel model, there are total of $S$ tasks with one entry and one exit task that must be executed locally.

The other $S'$ tasks can be executed in parallel either in the mobile device or remote server since there is no dependency between these tasks as shown in Figure 3.1. The $D_{(i,j)}$ parameters on the edges of the task graph in Figure 3.1 denote the transfer data size between tasks $i$ and $j$. This model can be considered as a special case of the general model.

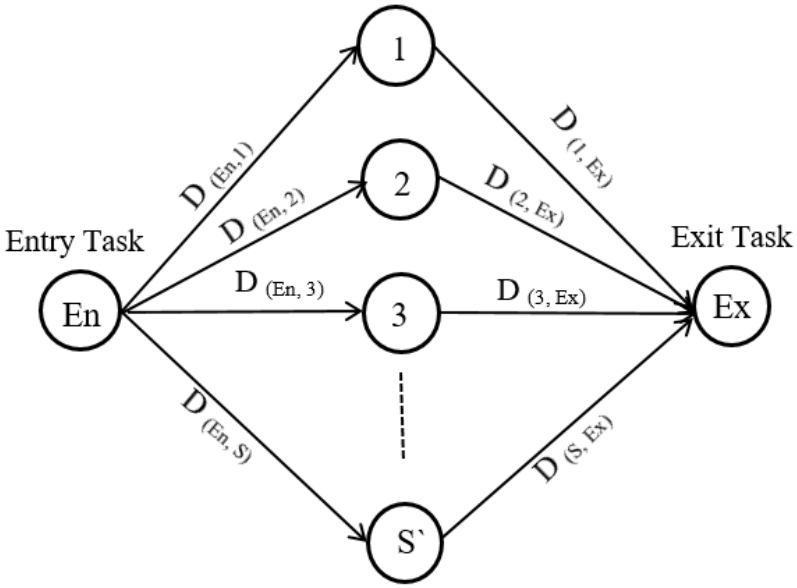


Figure 3.1: Application Model for Parallel Tasks, Original Figure by Mao *et al.* (2017a), Copyright © 2017, IEEE

### 3.1.2 General Model

This model is widely used in many papers to model application execution (Zhang *et al.*, 2012; Deng *et al.*, 2016; Tian *et al.*, 2005; Kao *et al.*, 2017; Yang *et al.*, 2013). An application is represented by a directed task graph $G = (V, EM)$, where $V$ and $EM$ are the sets of vertices and edges that give the inter-dependency among different application computation functions and routines. The graph is typically a directed

acyclic graph (DAG), which is a finite directed graph with no directed cycles (Mao *et al.*, 2017a).

Each node $i \in V$ represents a task (different procedure) and a directed edge $e = (i, j) \in EM$ represents the calling relationship, such that task (node) $i$ must complete its execution before task (node) $j$ starts. The weight of each edge shows the amount of data that needs to be transferred between the caller and the callee. There are a total of $S$ tasks (nodes) in the task graph (application). Normally, $S$ is smaller than 100, e.g., in computer vision applications, $S$ is in the range of $10 \sim 30$ (Ra *et al.*, 2011). Given a task graph, the task without any parent is called the entry task, and the task without any child is called the exit task. In the task graph of an application, there may exist multiple entry tasks and multiple exit tasks (Lin *et al.*, 2015). The transmission cost (here we consider energy consumption as the cost) is dependent on the amount of data that needs to be transmitted and the network conditions.

There are two kinds of task graphs: 1) unidirectional, and 2) bidirectional. In the unidirectional task graph as shown in Figure 3.2.a, computation starts at the entry task 1 and terminates at the exit task 5. When a task $i$ finishes execution it will forward its data over its outgoing edges (if any) and terminates. In the bidirectional graph shown in Figure 3.2.b, computation starts and terminates at the entry task 1. Each outgoing edge $(i, j)$ of a task $i$ leads to a dependent task $j$ which must be called, which will return control and data back to task $i$. A task $i$ may perform some execution, it will then call all its dependent tasks (i.e., functions) by forwarding data over its outgoing edges. It will then wait for each dependent function to return control and its data. Task $i$ then returns control and its data to its parent task and terminates (Shahzad and Szymanski, 2018).

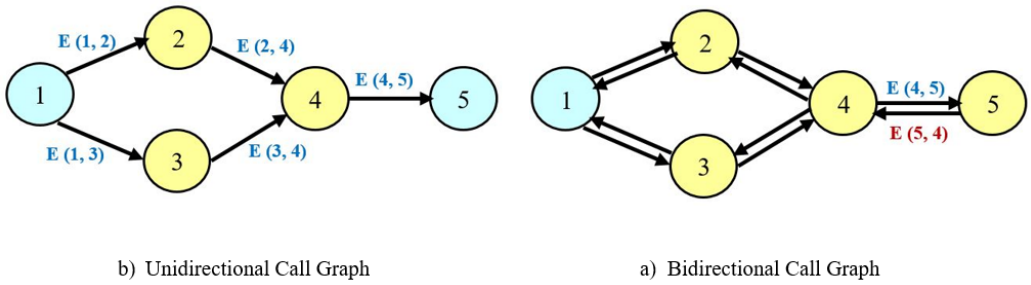


Figure 3.2: Two Different Types of Task Graphs

There are three typical task dependency models (i.e., task components such as functions or routines): a) sequential, b) parallel, and c) general dependency (Mao *et al.*, 2017a), as shown in Figure 3.3. As mentioned earlier, normally, tasks that directly handle user interaction, access local I/O devices or access specific information on the mobile device, e.g., collecting the I/O data and displaying the computation results on the screen, camera, or acquiring position, must be locally processed by the mobile user. For mobile initiated applications, the first and the last tasks are normally required to be executed locally. Thus, node 1 and node $S$ in Figure 3.3 are components that must be executed locally. The $D_{(i,j)}$ parameters on the edges of the task graph in Figure 3.3 denote the transfer data size between tasks $i$ and $j$. As explained earlier, parallel dependency is a special case of the general model that we call the *parallel model* since tasks 2 to $S-1$ are independent of each other and can be executed locally or offloaded to the remote server.

When a task is offloaded, the task's input data is sent to the remote server if this task and the preceding task are not executed in the same place. If a task is offloaded and the preceding task is executed locally, specific data necessary for executing the task on the server will be transferred. The time taken to transfer a data input of the task between a mobile device and the cloud or edge server through a wireless network

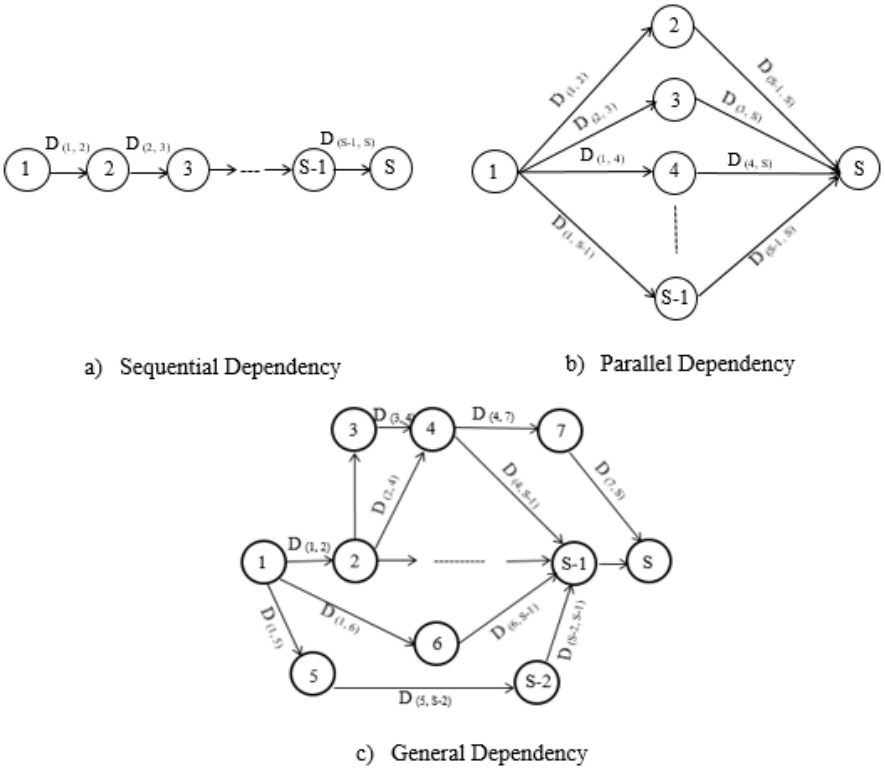


Figure 3.3: Typical Topologies of Task Graphs, Original Figure by Mao *et al.* (2017a), Copyright © 2017, IEEE

is an important consideration, since each application may have a time constraint by which all its tasks must be completed.

## 3.2 Examples of Task Graphs

In this section, some of the application task graphs from different references are considered. Some of these task graphs will be used in the simulations in the thesis. Chen *et al.* (2015) provided the range of their task graph parameters and this information was used to generate inputs for some of our experiments, as shown in Figure 3.4. There is a total of $S = 12$ tasks where $S' = 10$ of them are offloadable. The tasks shown in blue are entry and exit tasks that are not offloadable. The yellow nodes show the offloadable tasks. The edge weights in the task graph show the transfer data in Megabytes. The computation workload of each offloadable task in Giga CPU cycles is also provided.

Yang *et al.* (2013) provides the flow graph of a real world application, called QR-code recognition. This application is modeled as a set of functional components and a set of streaming data from one component to another. The QR-code recognition consists of three phases: image capturing, image pre-processing and QR code decoding. The three phases include 9 functional components and 10 edges as shown in Figure 3.5. Considering a $640 \times 480$ (300 KBytes) input image, the size of data that is transferred between the components are labeled on the edges in Figure 3.5 (Yang *et al.*, 2013).

Another application task graph is derived from (Deng *et al.*, 2016) and is shown in Figure 3.6. There are three examples of this task graph with different computational workload and exchange data sizes that are shown in Figures 3.6, parts a,b and c.

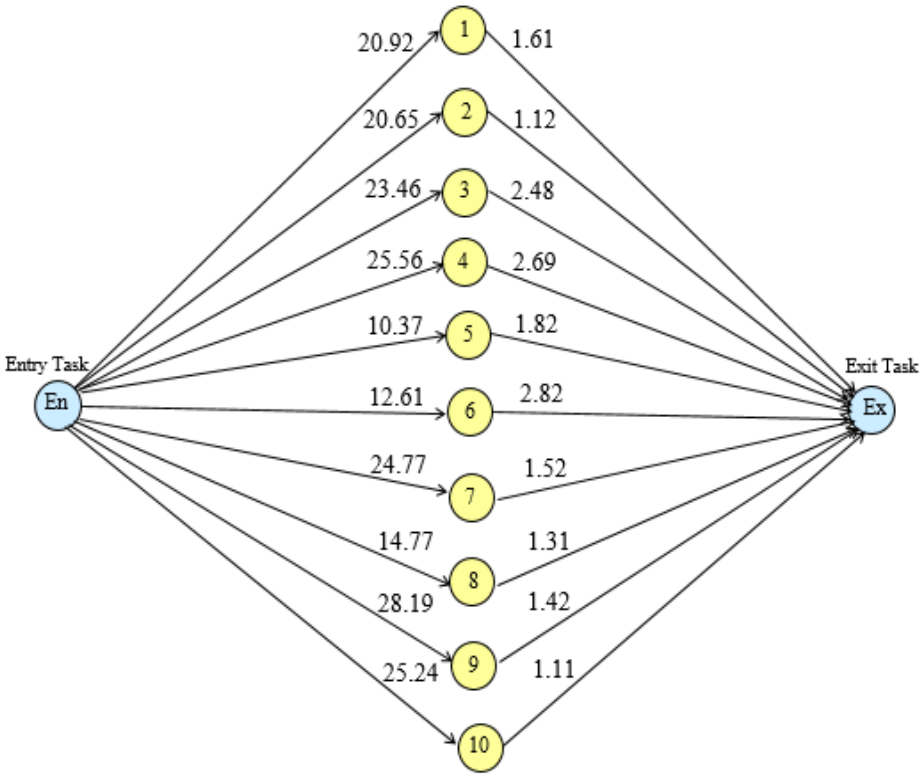


Figure 3.4: An Example of a Parallel Task Graph Based on the Parameters of Chen *et al.* (2015) with $S' = 10$.
Workload of Offloadable Tasks 1 to 10 = [39.75, 39.23, 44.56, 48.57, 19.70, 23.97, 47.07, 28.07, 53.57, 47.95] Gcycles. Transfer Data Sizes in Mbytes.

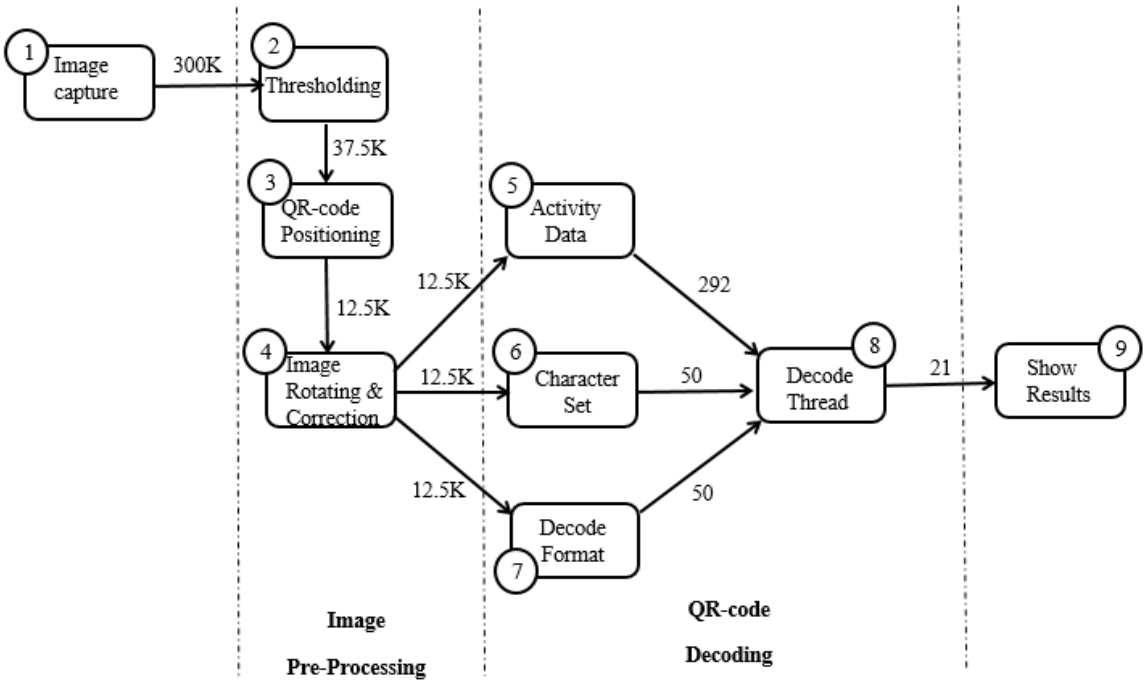


Figure 3.5: QR Application Task Graph by Yang *et al.* (2013), Workload of Tasks 2 to 8 = [42.24, 68.64, 58.08, 26.4, 21.12, 15.84, 147.84] Mcycles, Copyright © 2013, IEEE

Transfer data sizes in Kbits are labeled on the edges of the task graphs.

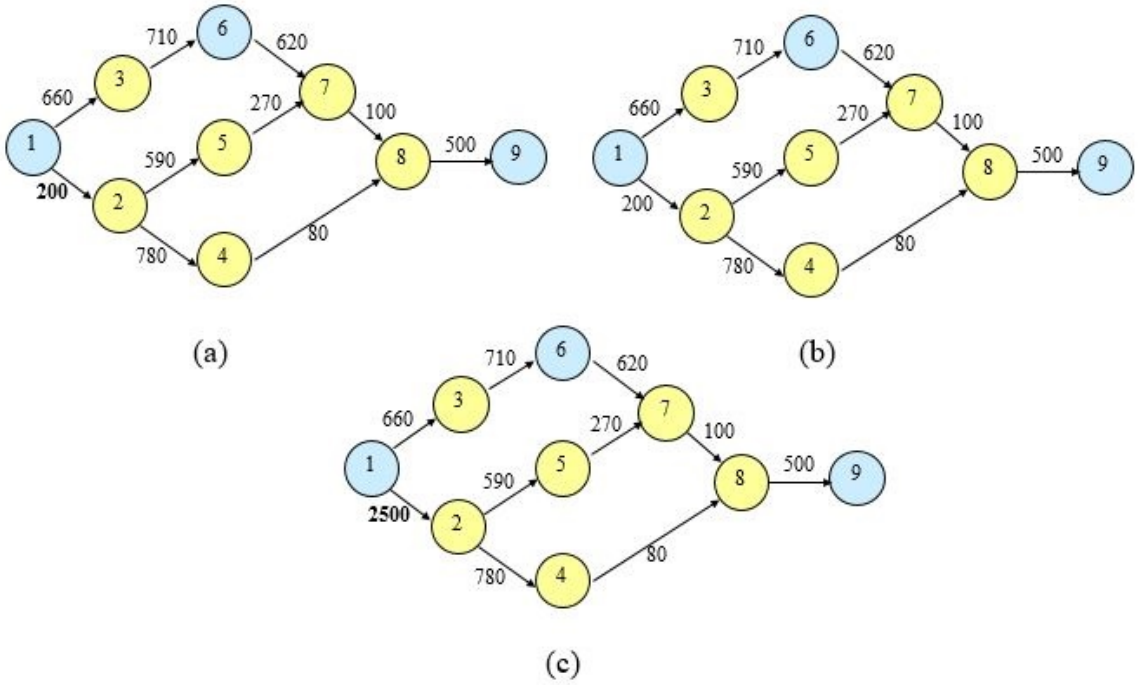


Figure 3.6: Application Task Graph by Deng *et al.* (2016). Workload (a) = [30, 25, 16, 32, 15, 37, 18, **3**, 10] Mcycles, Workload (b)= [30, 25, 16, 32, 15, 37, 18, **20**, 10] Mcycles, Workload (c)= [30, 25, 16, 32, 15, 37, 18, 3, 10] Mcycles and Transfer Data in Kbits, , Copyright © 2016, IEEE

Figure 3.7 shows the task graph of the face recognition application considered by Zhang *et al.* (2012). The input object to be recognized is a 42 KB (398 × 545) JPEG image, and the searching space is a set of 32 images with the same size. The graph has 23 tasks in total, 19 offloadable tasks, and 36 edges. The blue nodes are the unoffloadable tasks that must be executed locally and the yellow nodes are the offloadable tasks. The weights of the edges show the amount of data in kilobytes that needs to be transfered if the place of execution of the caller and callee tasks is not the same (the blue number is the input data size to the next task in the direction of the

arrow on the edge and the red number is the size of the returned information, which moves in the opposite direction on the edge).

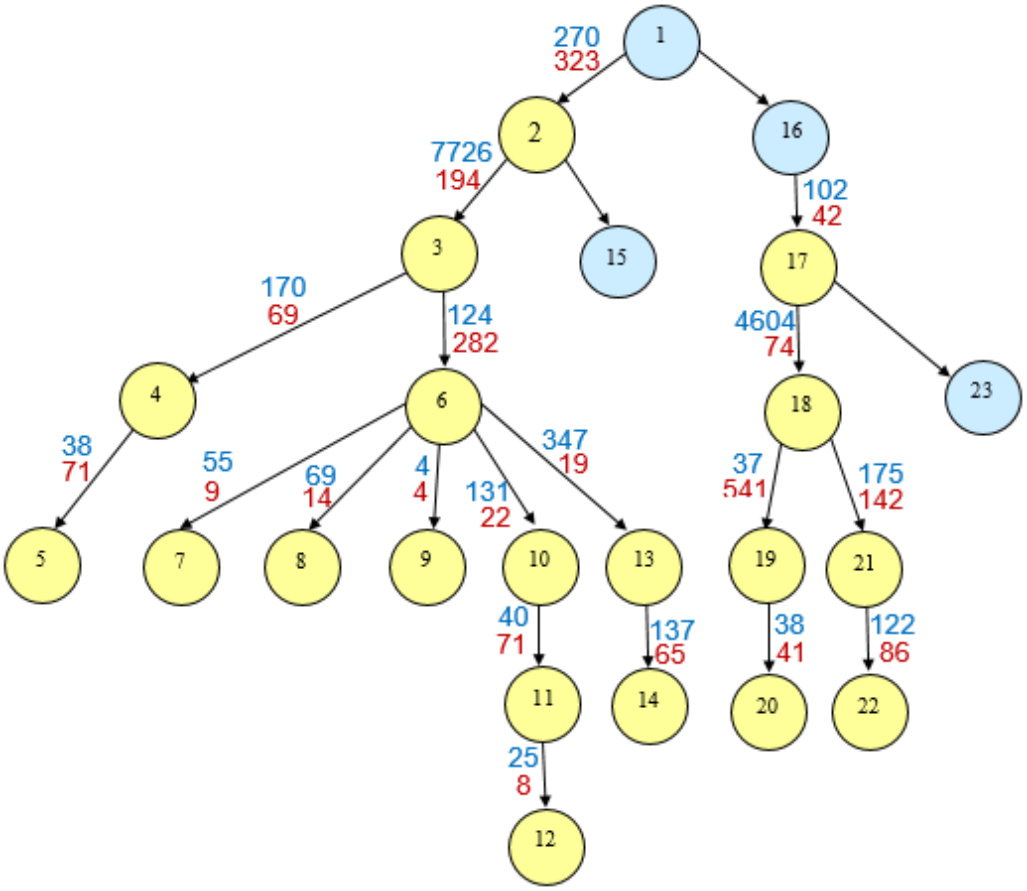


Figure 3.7: Task Graph with $S = 23$ by Zhang *et al.* (2012), Task CPU Cycles = [0, 13.22, 51.02, 0.97, 0.53, 141.5, 0.09, 4.94, 0.01, 11.64, 0.46, 3.01, 2.48, 8.9, 0, 0, 6.65, 21.83, 10.23,3.06, 3.91, 0.82, 0] Mega Cycles and Transfer Data in Kbytes, Copyright © 2012, IEEE

Another example of the task graph that is used in this thesis is that of Kao *et al.* (2017) with a total of 15 tasks and 18 edges. This task graph is shown in Figure 3.8 and transfer data sizes are shown on the edges.

Another graph by Tian *et al.* (2005) is shown in Figure 3.9. The computation

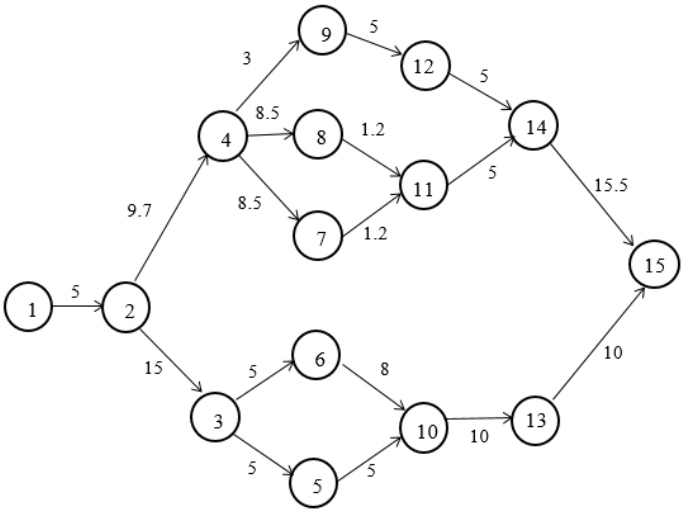


Figure 3.8: Task Graph with $S = 15$ from Reference (Kao *et al.*, 2017), Task CPU Cycles = [10.5, 3, 1.2, 3, 2,10 , 5.5 , 5.5, 10, 3.3, 5, 3, 5, 10, 1] Mega cycles and Transfer Data in Kbytes, Copyright © 2017, IEEE

load and data input of the tasks (weights of edges) are uniformly distributed over [300 Kcycles ± 10 %] and [800 bits ± 10 %], respectively.

The task graph of Tang *et al.* (2018) which they stated is based on a real world application with a total of $S = 20$ tasks and 30 edges is shown in Figure 3.10. The workload of each task and the amount of data transmission between pairs of tasks are generated randomly. Edges of the task graph are labeled with the transmission data in Kbytes.

### 3.2.1 Generation of Pseudo-random Graphs

Szymanski (2018) described 300 pseudo-random task graphs for evaluating mobile cloud and edge computing systems. It also introduced a method for generating pseudo-random task graphs from a given seed graph. In this section, the method

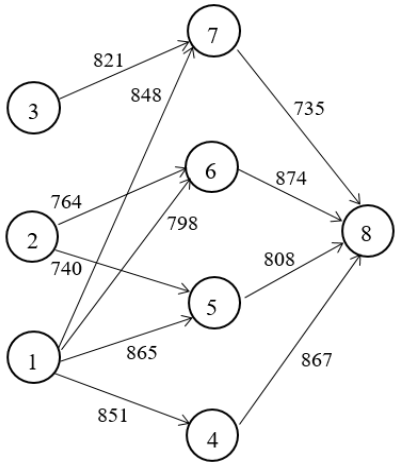


Figure 3.9: Task Graph with $S = 8$ by Tian *et al.* (2005), Transfer Data in Bits, Task CPU Cycles = [278, 295, 325, 318, 328,310, 316, 280] Kilo Cycles, Copyright © 2005, IEEE

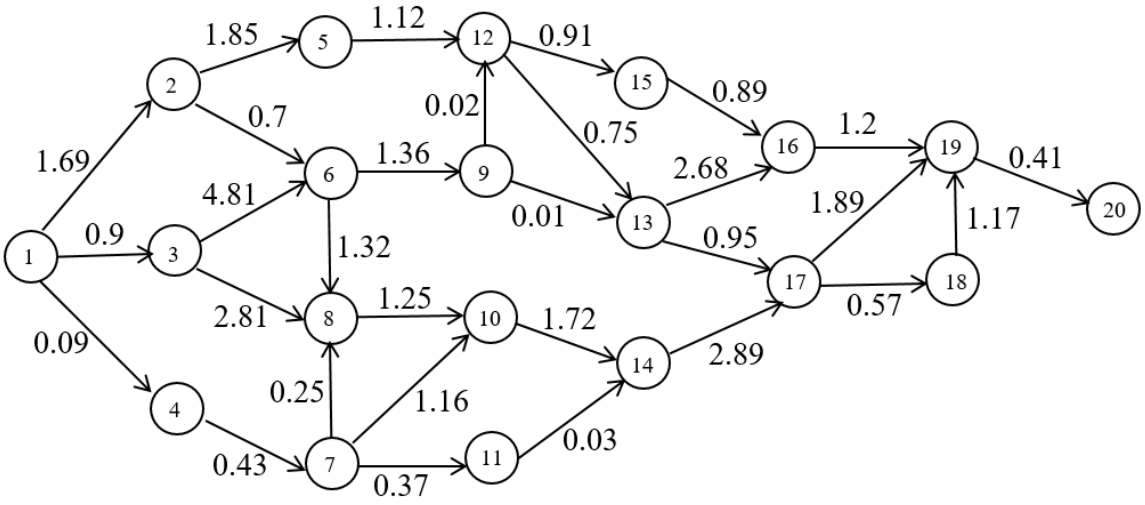


Figure 3.10: Task Graph with $S = 20$ by Tang *et al.* (2018), Task CPU Cycles = [4.09, 3.29, 8.54, 0.82, 4.94, 5.43 , 1.85, 3.72, 0.09, 4.05, 0.19, 2.11, 6.94, 4.81, 1.43, 2.5, 7.18, 1.55, 2.42, 4.95] Mega Cycles and , Transfer Data in Kbytes, Copyright © 2018, IEEE

of generating pseudo-random task graphs is summarized.[1]

"In the seed graph, as in [Figure 3.6], each of the $S = 9$ tasks has a computational workload expressed in Megacycles, and each of the 10 edges $(i, j)$ has an amount of data to be transferred expressed in bits. Each pseudo-random task graph is specified with 19 numbers, which includes the $S = 9$ task complexities and 10 edge weights, plus a binary vector of length $S$ which specifies which tasks must execute locally. To generate each pseudo-random task graph, each task complexity and edge weight of the seed graph shown in [Figure 3.6 a] was multiplied by a [uniformly distributed] random coefficient $C \in [0.1, \cdots, 10.0]$, thereby providing a range of variability of 2 decades. In addition, nodes 1 and 9 must execute locally, and one other node was selected randomly from the remaining nodes to execute locally".

"The same method of generating task graphs as described for the task graph in [Figure 3.6] was used for the task graph in [Figure 3.7]. Each pseudo-random task graph can be specified with 59 numbers, comprising 23 task complexities and 36 edge weights. The set of 100 task graphs used is available at the IEEE DataPort website. The task graph in [Figure 3.7] did not specify 4 task complexities (for tasks 1, 15, 16, 23) and 6 edge weights for 3 bidirectional edges (2,15), (1,16), and (17,23). To generate 99 pseudo-random task graphs, these 4 task complexities and 6 edge weights in the seed graph were assigned values as follows. The 23 task complexities are [10, 13.22, 51.02, 0.97, 0.53, 141.5, 0.09,4.94, 0.01, 11.64, 0.46, 3.01, 2.48, 8.9, 5, 10, 6.65, 21.83, 10.23, 3.06, 3.91, 0.82, 5] Megacycles per task respectively. The 6 missing edge weights for edges (1,16), (2,15) and (17,23) are [(250,300), (100,20), (100,40)] Kilobytes per edge respectively". Executable MATLAB code for the task graphs explained in this section is available at IEEE DataPort (Szymanski, 2018).

[1]The quoted text in this section is taken from Szymanski (2018)

## 3.3 Summary

In this chapter, background information related to the application models was introduced and reviewed. Different types of models and their characteristics from an application viewpoint were presented and discussed. Examples of task graphs were shown that are used to represent various applications. These include the examples that are used in our simulation experiments and referenced later in the thesis.

# Chapter 4

# The DPH and DPR Algorithms

In this chapter, two dynamic programming algorithms called DPH, "Dynamic Programming with Hamming Distance Termination," and DPR, "Dynamic Programming with Randomization," are described. (See also Shahzad and Szymanski, 2016a,b, respectively.) Dynamic programming is an optimization approach that transforms a complex problem into a sequence of simpler problems that are solved in an interactive iterative manner. The proposed DPH and DPR algorithms introduce randomization. In particular, we periodically generate random bit strings of 0s and 1s and utilize their sub-strings when they improve the solution, in a process similar to genetic optimization. We also fill a dynamic programming table in a creative way so as to avoid the extra computation for common sub-strings. It is shown that the algorithms can find good solutions after a reasonable number of iterations.

DPH uses a Hamming distance termination criterion that is used to obtain a final decision quickly. This criterion is met when a given fraction of tasks are designated for offloading. The final solution depends on the wireless network transmission rate and the computational power of the cloud servers. The DPR algorithm also uses a

Hamming distance search criterion with a preference to offload tasks. DPR will offload as many tasks as possible to the cloud server when the network transmission rate is high, thereby reducing the total task execution time and mobile energy consumption.

Our numerical results will show that the DPH and DPR algorithms can find good quality solutions with low computational overhead for parallel task graphs. In particular, our comparisons between the results for the DPH and DPR algorithms, the optimal results from Brute-Force search, and the results from the semidefinite relaxation method of Chen *et al.* (2015) will show that our proposed algorithms can find good solutions for parallel task graphs, and that those solutions can be obtained in short computation times.

A single user with a parallel task model, as explained in Section 3.1.1, is used to evaluate the two proposed algorithms. In this case, we assume that there is one mobile user with an application that contains $S'$ tasks, one cloud server, and a wireless network that is used for offloading, as illustrated in Figure 4.1. It is assumed that the channel transmission capacity is variable since wireless channel quality and network congestion will affect the network transmission capacity. The mobile device needs to decide whether each task should be processed locally or offloaded, according to the current wireless network conditions. The time taken to transfer a task between a mobile device and the remote server through a wireless link is an important issue since there may be a total execution time constraint for all tasks of an application. More details of the problem formulation used in this chapter are described in the following section.

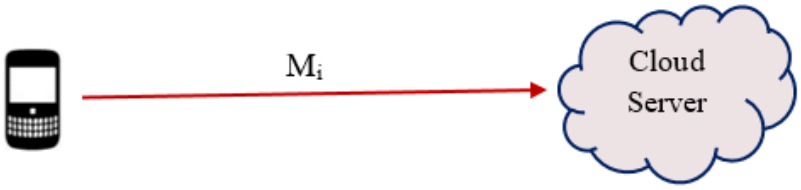


Figure 4.1: Our System Model

## 4.1 Problem Formulations for the Parallel Model

The parallel computing model has been employed by many authors, including Chen *et al.* (2015) and Zhang *et al.* (2017). Consider an application consisting of some unoffloadable (i.e., local) tasks and $S'$ offloadable tasks. Normally, unoffloadable tasks include those that directly handle user interaction, access local I/O devices, or access specific information on the mobile device. These local tasks must be processed by the mobile device.

For each task $i$, let $M_i \in \{0, 1\}$ be an execution indicator variable, with $M_i = 1$ if task $i$ is executed at the mobile device and 0 otherwise. If it is executed locally, the energy consumption is $El_i$. The term $Et_i$ denotes the mobile device energy needed to transmit the input of task $i$ to the remote server and to transmit the task output from the remote server to the mobile device. The associated input and output data sizes are defined as $DI_i$ and $DO_i$, respectively. $Tc_i$ is the time needed to execute task $i$ at the remote server, during which the mobile device waits in an idle mode and $Tt_i$ is the required upload and download transmission time of the input and output data for task $i$ between the mobile device and the cloud. The Variable $Tl_i$ is the local execution time to process task $i$. $Cc_i$ denotes the cloud processing cost of task $i$ and $EC_i$ is defined as the cost to offload task $i$ to the cloud. These parameters are summarized in Table 4.1.

Table 4.1: Description of the Model Parameters

| **Symbols** | **Meaning** |
|---|---|
| $M_i$ | Decision variable for task $i$ |
| $C_i$ | Computation load of task $i$ (CPU cycles) |
| $R$ | Transmission rate (bps) |
| $El_i/Tl_i$ | Energy / Time to execute task $i$ locally |
| $Et_i/Tt_i$ | Energy / Time to transfer input and output data for task $i$ to or from the cloud server |
| $EC_i/TC_i$ | Processing cost/delay for task $i$ offloaded to the cloud |
| $Tc_i$ | Cloud processing time for task $i$ |
| $Cc_i$ | Cost for having the cloud process task $i$ |
| $DI_i/DO_i$ | Size of task $i$ input / output data to be transferred if task $i$ is executed remotely |

It is clear that the transmission energy used to upload each task will depend on the channel transmission quality. Therefore, changes in wireless path loss should affect the offloading decisions. For example, if the transmission time of the input for each task is equal to the size of the task input divided by the channel transmission rate, then any transmission rate variation should affect the final offload decisions. Similar to Chen *et al.* (2015), we assume the same upload and download mobile energy cost for a given transmission data size.

We denote $M = [M_1, M_2, ..., M_{S'}]$ as the vector of binary offloading decisions for our set of tasks. $EC_i$, $TL$ and $TC$ are defined as the weighted processing costs of offloading task $i$ to the cloud servers, processing delay at the mobile user and worst case processing delay at the cloud server (in the sense of Cuervo *et al.*, 2010), respectively. The optimization problem that we want to solve is defined as follows.

$$\begin{aligned} \min_{\{M_i\}} \quad & \sum_{i=1}^{S'} El_i\, M_i + EC_i(1 - M_i) + \zeta \max\{TL(\{M_i\}), TC(\{1 - M_i\})\} \\ \text{s.t.:} \quad & M_i \in \{0, 1\}, \quad \forall i \in \{1, 2, \ldots, S'\} \end{aligned} \tag{4.1}$$

In Eq. (4.1), we compute the total cost to the mobile user in the way that was proposed by Chen *et al.* (2015), i.e., the weighted sum of the local computation energy consumption, the energy to offload the tasks to the remote server, and to process them and the corresponding worst case transmission and processing delays. The factor $\zeta$ is a weighting factor that can be adjusted to change the emphasis on execution delay and energy consumption. The terms $EC_i$, $TL$ and $TC$ are defined as

follows with $\beta$ being the relative weight of the cloud energy cost, $Cc_i$.

$$EC_i = Et_i + \beta Cc_i \tag{4.2}$$

$$TL(\{M_i\}) = \sum_{i=1}^{S'} M_i \, Tl_i \tag{4.3}$$

$$TC(\{1 - M_i\}) = \sum_{i=1}^{S'} (1 - M_i)(Tt_i + Tc_i) \tag{4.4}$$

In our architecture we assume that the mobile device can offload a task only to the single remote server (a cloud or edge server); as illustrated in Figure 4.1. The transmission time, $Tt_i$ of each task between the mobile device and cloud is equal to the size of each task divided by the transmission rate ($R$) which is measured in megabits per second. The transmission energy consumption of the mobile device for both upload and download is set to the $1.42 \times 10^{-7}$ J/bit, as in Chen *et al.* (2015). With $DI_i$ and $DO_i$ representing the input and output data sizes of task $i$ (expressed in megabits), the transmission time per task (in seconds), and the transmission energy per task (in micro Jules) are calculated as follows.

$$Tt_i = DI_i/R + DO_i/R$$

$$Et_i = 0.142 \; DI_i + 0.142 \; DO_i$$

The number of combinations of binary values $M_i$ to search for the optimal solution grows exponentially with the number of tasks. Our goal is to determine which tasks should be offloaded to the remote server so that the total cost is minimized.

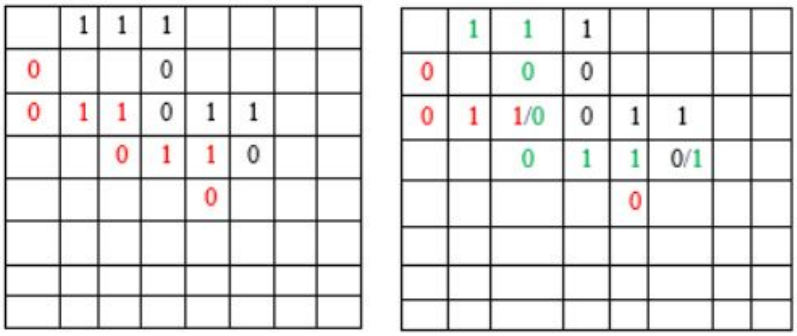


Figure 4.2: Table Filling Examples

## 4.2 The DPH Algorithm

This section describes the "Dynamic Programming with Hamming Distance Termination" (DPH) algorithm (see also Shahzad and Szymanski, 2016a). In this scheme, we use an $S' \times S'$ table to store the bit strings that show which tasks should be offloaded, where $S'$ is the number of offloadable tasks in the parallel task graph of interest. For the first step, a random bit string of length $S'$ is generated that determines an initial solution. This string is assigned to the table such that 1s are assigned to the next horizontal cell, and 0s are assigned to the next vertical cell. If the first bit of the stream is 1, the starting cell for task 1 is (1, 2) otherwise the starting cell for task 1 is (2, 1). This approach will avoid extra computations for common bit strings.

A 2D 8×8 table is shown in Figure 4.2. To clarify, assume that $S' = 8$ and the first random strings are 11100110 (black numbers) or 00110110 (red numbers), i.e., 2 examples are given. Assume that the second random bit stream in each case is 11000111. The starting cell of the second stream is (1, 2) since the first bit is 1. By following the aforementioned rules to fill the table, the resulting green stream is shown in Figure 4.2. The FillTable function that is used in the DPH algorithm represents the rules of table filling as explained above.

### 4.2.1 Detailed Description of the DPH Algorithm

In the DPH algorithm, random bit strings of 0s and 1s are periodically generated and their sub-strings are utilized when they improve the solution. The focus of the DPH algorithm is to convert the offloading problem to a set of sub-problems using a table. The table will also help to determine the location of sub-strings. The pseudocode of the DPH algorithm is shown in Algorithm 4.1.

In line 1, a random binary vector of size $S'$ is generated as the initial best solution, $V_{best}$. The TableFill function is called in line 2 with $V_{best}$ as its input and it returns the indices of cells that are filled with $V_{best}$ as the initial best path, i.e., $P_{best}(i)$ gives the cell coordinates for task $i$. In the first step of the TableFill in Algorithm 4.2, the first bit of the solution, $M$, is checked to define the starting cell to fill the table. Then, each bit in $M$ is assigned to the correct position in the table. We keep track of the filled cells of the table and store the indices in the $CellIndex$ matrix.

In line 3, the energy consumption of each cell in the initial best path, $P_{best}$, is calculated. In line 4, the main algorithm loop, with a total number of iterations, $Itr$, is initialized. In total, $Itr$ random bit strings will be generated so that the initial solution can be improved.

In the first step of the main loop in line 5, a random binary vector of length $S'$ is generated, which determines the new solution, $V_{new}$. This stream is assigned to the table as explained previously. The TableFill function in Algorithm 4.2 is called in line 6 with $V_{new}$ as its input and it returns the new path, $P_{new}$, which contains the indices of the filled cells in the table using $V_{new}$. Variable $LCT$ in line 7 keeps track of the last common task between the new and best paths. This common task is associated with a cell in the table which is the intersection of the best path and the new path.

**Algorithm 4.1:** DPH Algorithm Pseudocode

$V_{best}$ = random binary vector of size $S'$ // get initial (best) binary vector
$P_{best}$ = TableFill($V_{best}$) // get associated (best) table path
// set energy along best path
$ES(P_{best}(t)) = El(t)V_{best}(t) + EC(t)(1 - V_{best}(t)) \quad \forall t \in \{1, 2, \ldots, S'\}$
**for** $k = 1$ **to** $Itr$ **do**
  $V_{new}$ = random binary vector of size $S'$ // get new binary vector
  $P_{new}$ = TableFill($V_{new}$) // get associated new path
  LCT = 1 // set last common task between best/new paths
  **for** $T = 1$ **to** $S'$ **do**
    **if** $P_{best}(T) == P_{new}(T)$ **then**
      CTP = $(LCT, LCT + 1, \cdots, T)$ // set common task path
      // find the best CTP segment
      **if**
       $\sum_{t \in CTP} El(t)V_{new}(t) + EC(t)(1 - V_{new}(t)) < \sum_{t \in CTP} ES(P_{best}(t))$
       **then**
        // update the best path
        $P_{best}(t) = P_{new}(t)$, $V_{best}(t) = V_{new}(t)$,
        $ES(P_{best}(t)) = El(t)V_{new}(t) + EC(t)(1 - V_{new}(t)) \quad \forall t \in CTP$
      **else**
        // update the new path
        $P_{new}(t) = P_{best}(t)$, $V_{new}(t) = V_{best}(t) \quad \forall t \in CTP$
      **end**
      LCT = $T$ // record the current common task, for future use
    **else**
      $ES(P_{new}(T)) = El(T)V_{new}(T) + EC(T)(1 - V_{new}(T))$
    **end**
  **end**
  $E_{best} = \sum_{c \in P_{best}} ES(c)$
  $E_{total} = E_{best}$
  calculate $TL$ and $TC$ using $M = V_{best}$ and Eq. (4.3) and Eq. (4.4)
  calculate objective function in Eq. (4.1) as total cost using $E_{total}$, $TL$ and $TC$
  $N_o = S' - \sum_{t=1}^{S'} M(t)$ // number of task offloads in solution $M$
  **if** $N_o \geq HDC$ **then**
    return $E_{total}, V_{best}$
  **end**
**end**

**Algorithm 4.2:** TableFill Function Pseudocode in Algorithm 4.1

```
1  Function TableFill(M):
       // M is a binary task vector
       // CellIndex is a vector that maps tasks to table cell coordinates
2      if M(1) == 1 then
3          CellIndex(1) = (1, 2)
4      else
5          CellIndex(1) = (2, 1)
6      for i = 2 to S' do
7          if M(i) == 1 then
8              CellIndex(i) = CellIndex(i − 1) + (1, 0)
9          else
10             CellIndex(i) = CellIndex(i − 1) + (0, 1)
11     end
12 return CellIndex
```

In line 8, an inner loop is initiated which iterates through all the tasks from 1 to $S'$. If the path of the new solution has some common cells with the path of the best solution in the table in line 9, the common task path, $CTP$, is set to the tasks between the last common task, $LCT$, and the current common task $T$ in line 10. This common path with the corresponding sub-strings of $V_{best}$ and $V_{new}$ will be investigated.

In line 11, if the total energy of the selected sub-string in the new solution is less than the total energy of the selected sub-string in the best solution, we update the best path, i.e., the best solution so far and the energies with the new ones for this sub-string in lines 12-13. Otherwise, in line 15, we do the opposite. Line 17 will record the current common task for setting the common task path in subsequent iterations.

For the cells that are not common between the new and best path, the energy consumption of that cell will be calculated in line 19. Once the inner loop is finished, we compute the best total energy consumption, $E_{best}$, using the best path we have so far, $P_{best}$, and the energy consumption of the cells in the best path. Total local execution

time, $TL$, and total time when tasks are executed remotely, $TC$, are calculated in line 24 using Eq. (4.3), Eq. (4.4) and the best solution, $V_{best}$. In line 25, we calculate the objective function based on Eq. (4.1). Then the number of tasks that are offloaded is found in line 26. In line 27, we terminate and accept a solution that has a Hamming distance larger than a given threshold, referred to as the Hamming Distance Criterion (HDC), compared to an all 1's stream. The all 1's stream denotes the case where all tasks are executed locally. This heuristic termination criterion encourages offloading and yields good results in our experiments.

The computation complexity of the DPH algorithm is determined by the evaluation of the combination of the energy and time of the sub-strings based on equation (4.1), the population size of the random bit strings, number of tasks and the number of iterations in the evaluation process.

**Lines 1-3:** Initializing step and generating the first random binary vector have time complexity of $O(S')$. In line 2, TableFill function also has a complexity of $O(S')$ to assign the initial binary vector to the table.

**Lines 4-30:** Loop $k$ has a complexity of $O(Itr)$ and it contains the TableFill function, which leads to a complexity of $O(S' \cdot Itr)$. Evaluation of the objective equation over the loops ($k$ and $T$) has time complexity of $O(Itr \cdot S'^2)$ where $Itr$ is the total iterations of the outer loop, $S'$ is the total iterations of the inner loop and $S'^2$ is the time complexity of the inner loop with calculations in lines 10-16 to evaluate the energy equation and sub-string incorporation in the current solution vector.

The time complexity of the DPH algorithm is the worst time complexity among all the steps that are explained. Therefore, the time complexity of DPH algorithm is $O(Itr \cdot S'^2)$.

### 4.2.2 A Comparator Algorithm

The system that we have considered (see Figure 4.1 and Figure 4.3.a) is a limiting case of the system considered by Chen *et al.* (2015), which is illustrated in Figure 4.3.b. The system model presented by Chen *et al.* (2015) consists of one mobile user, one computing access point (CAP), and one remote cloud server, as shown in Figure 4.3.b. The CAP can either process the tasks or offload (some of) them to the cloud server. The decision of the mobile user to whether to process its task locally or remotely is denoted by $x_i$. When task $i$ is offloaded to the CAP, the decision whether the task should be processed by the CAP or further offloaded to the cloud is denoted by $y_i$. The goal of Chen *et al.* (2015) is to optimize the offloading decision of the user to minimize the overall cost of energy, computation, and delay. It is shown that the problem can be formulated as a non-convex quadratically constrained quadratic program (QCQP), which is NP-hard in general. Chen *et al.* (2015) used semidefinite relaxation (SDR) and a randomization mapping method to generate good solutions to the QCQP problem.

The relative weights of the processing cost for task $i$ being offloaded to the CAP or cloud are denoted by $\alpha$ and $\beta$, respectively; see Figure 4.3.b. If $\alpha$, which is the weight of processing at the CAP, becomes large, no task will be processed there and the system considered by Chen *et al.* (2015) implicitly reduces to our model that considers a single cloud server, and is illustrated in Figure 4.3.a. In that case, the models in Figure 4.3 can be made equivalent by equating the transmission times and transmission energies in the natural way.

A reason for choosing this comparator is that Chen *et al.* (2015) developed a sophisticated SDR scheme for providing high quality offloading decisions. Hence

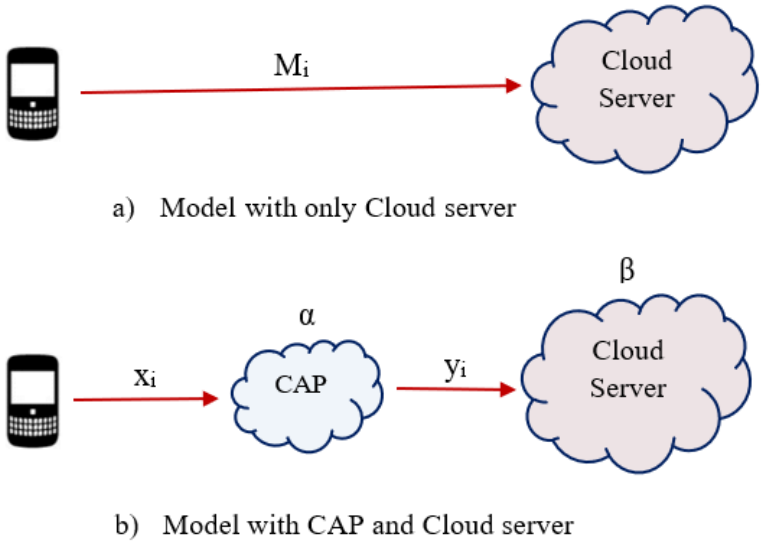


Figure 4.3: Comparison of Our Model and the Model of Chen *et al.* (2015)

their algorithm chosen as a comparator.

### 4.2.3 Simulation Results

The DPH algorithm was programmed using MATLAB. We adopt the mobile device characteristics from Chen *et al.* (2015), which are based on the Nokia N900, and set the number of offloadable tasks to $S' = 10$. The task graph model used has the topology illustrated in Figure 3.4. This parallel model can also be considered as a set of parallel applications that are executed in the mobile device (note that in this case there is no entry or exit task).

As explained in Section 4.2.2, Chen *et al.* (2015) proposed a semidefinite relaxation approach with randomization mapping to solve the optimization problem in Eq. (4.1) but with both CAP and cloud. Optimizing the offloading decision for independent tasks with one CAP and one remote cloud server, so as to minimize a weighted sum of

the costs of energy, computation, and delay, is the focus of that work. This algorithm was programmed in MATLAB, using the CVX tool. The SDP problem is solved by using the SeDuMi software. In our implementation, the relative cost of computation at the CAP, denoted by $\alpha$ is made so large so that computation at the CAP is avoided.

The local computation time is $4.75 \times 10^{-7}$ s/bit and the local processing energy consumption is $3.25 \times 10^{-7}$ J/bit, when the x264 CBR encode application (with 1900 cycles/byte used by Miettinen and Nurminen, 2010) is considered as task $i$, as used by Chen *et al.* (2015). The energy consumption for transmitting and receiving mobile user data are both $1.42 \times 10^{-7}$ J/bit. The input and output data sizes of each task are chosen randomly from uniformly distributions on 10 Mbyte to 30 Mbyte and on 1 Mbyte to 3 Mbyte, respectively, as they were by Chen *et al.* (2015). Based on these parameters, the local energy consumption for each task, $El_i$, and the local execution time for each task, $Tl_i$, are computed using the following equations, where $DI_i$ is the input data size of task $i$ (expressed in megabits).

$$El_i = 0.325 \ DI_i$$

$$Tl_i = 0.475 \ DI_i$$

When tasks are offloaded to the cloud, Chen *et al.* (2015) used a peak transmission rate of 15 Mbps. The time that is taken to execute a task in the cloud depends on the CPU rate of the cloud server and is therefore given by Eq. (4.5), where $10^{10}$ cycles/s is the CPU rate of the cloud server:

$$Tc_i = \frac{C_i}{10^{10}} \tag{4.5}$$

Chen *et al.* (2015) also defined a single metric to evaluate the performance, called the "total cost". The total cost of the mobile user was defined as the weighted sum of the total energy consumption (measured in joules) and the corresponding worst case time to offload and process all tasks (measured in seconds) as in Eq. (4.1).

As outlined in Section 4.2.2, the transmission time ($TC$) in Eq. (4.4) is set equal to (i) the time taken to transfer the task to the intermediate device at a given rate specified by Chen *et al.* (2015), plus, (ii) the time to transfer the task to the cloud at the network transmission rate. To be compatible with Chen *et al.* (2015), we assume that all tasks are executed in the cloud. Specifically, we set $\beta = 5 \times 10^{-7}$ J/bit, $\alpha$ is large ($\alpha = 1 \times 10^{-6}$ J/bit) and we set the value of the costs $Cc_i$ and $Ca_i$ to be the same as that of the input data size $DI_i$, as in Chen *et al.* (2015).

Table 4.2 shows a comparison of the total cost comparison between the optimal results from Brute-Force search (BFS), DPH, the LC (Local-Cloud) algorithm of Chen *et al.* (2015), and the All-Local and All-Remote methods. The weight $\zeta$ in Eq. (4.1) is assigned units of J/s and hence the total cost is expressed in Joules. BFS is obtained by generating all possible solutions and finding the optimum (i.e., that which achieves the minimum cost). For the chosen graph, results of the DPH algorithm are about 9% larger than the optimal results, but they require much less computation compared to the algorithm of Chen *et al.* (2015). The latter algorithm uses semidefinite programming followed by 100 randomization/relaxation trials. It is clear that using the All-Remote method, where all the tasks are executed in the remote server, yields a larger total cost due to the high cost of transferring data and processing at the cloud servers.

Table 4.2: Comparison of the Total Cost Based on Eq. (4.1) for Different Algorithms

| | **All-Local Energy** | **All-Remote Energy** | **Optimal Energy** | **LC** | **DPH** |
|---|---|---|---|---|---|
| **Total Cost (Joules)** | 1322.1 | 1265.2 | 1128.1 | 1133.4 | 1236.4 |

Figure 4.4 presents the total cost for 6 cases: (1) All-Local execution, (2) All-Remote, (3) DPH, (4) the LC algorithm, (5) Brute Force Search and (6) a lower bound on minimum cost that is obtained from the SDR (semidefinite relaxation) objective, as the wireless transmission rate changes from 4 to 20 Mbps. The total cost of the DPH solution is computed using Eq. (4.1), using the same weights $\beta$ and $\zeta$ as in Chen *et al.* (2015). Figure 4.4 shows that for the chosen graph, the total cost for DPH is better than the All-Remote case, for this choice of parameters. For low transmission rates, All-Local execution consumes less energy than all-offloaded, since the cost of transmitting tasks to the cloud is relatively high.

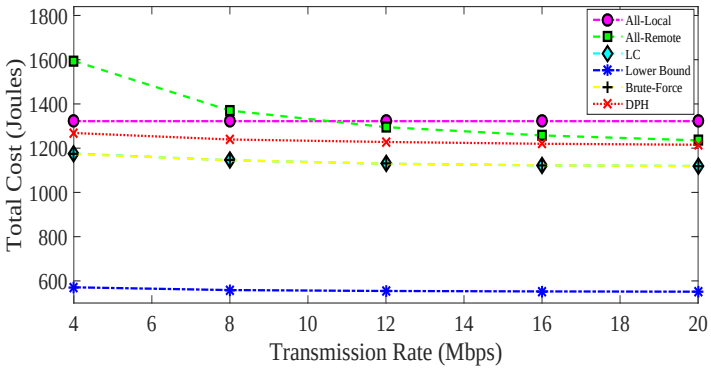


Figure 4.4: The Total Cost of Different Methods versus Transmission Rate $R$ (Mbps). The results for the LC and Brute-Force methods overlap at the scale of this plot

Figure 4.5 plots the total cost versus the weight $\beta$ of the cloud processing cost. As $\beta$ increases, the benefits of offloading decrease, and the tasks are more likely to

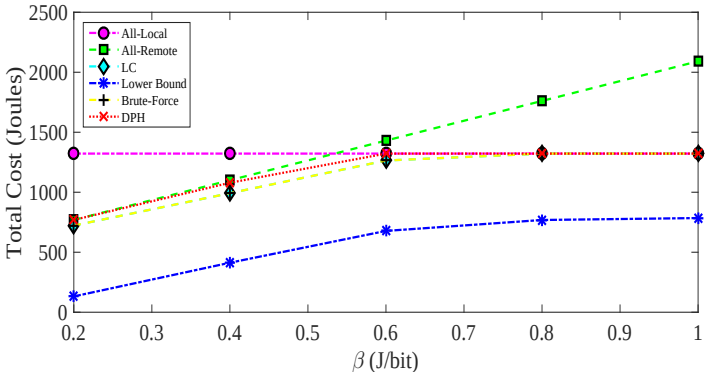


Figure 4.5: The Total Cost of Different Methods versus $\beta$ (J/bit)

be executed at the mobile device. For that reason, DPH, LC and Brute-force results will converge to the All-Local method for large $\beta$.

Figure 4.6 compares the performance of different methods versus the weight $\zeta$ of the processing delays. By increasing the effect of the delay (increasing the weight $\zeta$), the total delay becomes more dominant, i.e., we place more emphasis on delay rather than energy consumption. As a result, the total cost will also increase in all algorithms.

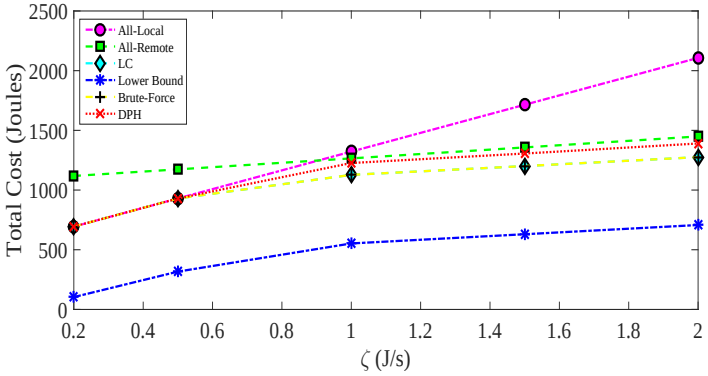


Figure 4.6: The Total Cost of Different Methods versus $\zeta$ (J/s)

Table 4.3 shows the execution time of the DPH algorithm and LC algorithm (Chen

*et al.*, 2015) in seconds as well as the total cost of these methods when the number of offloadable tasks ranges from 10 to 30. In terms of the user experience, the execution time of the offloading algorithm itself should not be long, as it is a component of the overall delay. For the chosen graphs, the execution time of LC algorithm that is implemented in MATLAB ranges from 25 to 14 times that of DPH. However, there is a trade off between the execution time of the algorithms and the total cost of the obtained solution, with the total cost of DPH being between 6 and 16 percent larger than that of LC.

Table 4.3: Execution Time and Total Cost of DPH and LC for Different Numbers of Tasks

| $S'$ | **10** | **12** | **15** | **18** | **20** | **25** | **30** |
|---|---|---|---|---|---|---|---|
| **Execution Time of DPH (s)** | 0.1183 | 0.1256 | 0.1321 | 0.1424 | 0.1512 | 0.1913 | 0.2537 |
| **Execution Time of LC (s)** | 3.022 | 3.082 | 3.261 | 3.065 | 3.327 | 3.371 | 3.701 |
| **Percentage of Time Decrease** | 96.08 | 95.9 | 95.9 | 95.35 | 95.45 | 94.3 | 93.14 |
| **Total Cost of LC (Joules)** | 1128.1 | 1507.5 | 1510.9 | 2142 | 2214.8 | 2795.8 | 2897.4 |
| **Total Cost of DPH (Joules)** | 1236.4 | 1630.6 | 1760.8 | 2280.3 | 2410.4 | 2998.1 | 3356.3 |
| **Percentage of Cost Increase** | 9.6 | 8.16 | 16.5 | 6.45 | 8.83 | 7.2 | 15.8 |

As explained previously, Chen *et al.* (2015) formulated the offloading problem with both CAP and cloud as a non-convex quadratically constrained quadratic program (QCQP), which is NP-hard in general. To solve this problem, they proposed an efficient algorithm based on semidefinite relaxation (SDR). The problem is first relaxed into a semidefinite programming (SDP) problem, then randomization is used to generate candidate binary solutions. The method then selects the candidate with

the lowest cost. The SDP problem can be solved in polynomial time using the convex optimization toolbox, CVX tool in MATLAB and standard SDP software, such as SeDuMi. Given a solution accuracy $\epsilon > 0$, the SDR problem can be solved with a worst case complexity of $O(\max\{m,n\}^4 n^{1/2} log(1/\epsilon))$, where $n = 2S' + 2$, $m = 2S'$ and $S'$ is the number of offloadable tasks (Luo *et al.*, 2010).

## 4.3 The DPR Algorithm

In this section, we extend DPH and describe an algorithm called "Dynamic Programming with Randomization" (DPR); (see also Shahzad and Szymanski, 2016b). Unlike DPH, the DPR algorithm iteratively improves an offloading decision vector by generating random bit strings with a biased probability of generating 0s, which represents a decision to offload a task. If fragments of these bit strings improve the decision vector, they are incorporated into the decision vector, in a process that is similar to genetic optimization. We also fill the DP table in a creative way to avoid duplicating the computations for the common bits in the random bit strings (as in the DPH algorithm). Our results show that the DPR algorithm finds good solutions quickly. It uses a Hamming distance termination criterion, with a preference to offload as many tasks as possible to the cloud server when network conditions are good. The DPR algorithm favors the most likely and beneficial outcomes first, by exploiting biased randomization.

We experimented with several different biases when generating the random bit strings. In the first experiment, the 0s and 1s in the random bit strings have an equal probability of 0.5. In a second experiment, the probability of generating 0s is greater than generating 1s, regardless of the network conditions. In the last experiment, the

offloading probability (i.e., the probability of generating a 0 in random bit strings) was set to be proportional to the network transmission rate. This approach introduces an offloading preference or bias according to the network conditions, which improves the convergence time of the DPR algorithm.

Our simulation results and, and comparisons with the optimal results from Brute-Force search and the results of the LC algorithm (Chen *et al.*, 2015), show that the DPR algorithm can find good solutions to minimize energy use for the parallel model with lower computation time than the LC algorithm. The DPR algorithm is computationally efficient and can scale to larger problems for parallel task graphs with a lower complexity compared to other Dynamic Programming algorithms.

### 4.3.1 Detailed Description of the DPR Algorithm

The main distinction between DPH and DPR is that DPR exploits the use of biased randomization instead of pure randomization when solutions are generated. The pseudocode of the DPR algorithm is shown in Algorithm 4.3. DPR is similar to the pseudocode of DPH in Algorithm 4.1 except for lines 1 and 5. In lines 1 and 5, biased binary random vectors are generated, i.e., $V_{best}$ and $V_{new}$, that determine which task should be offloaded. This bias reflects a preference to offload tasks. Let $P_{OFF}$ be the probability of generating a zero in a random bit string, indicating that the task will be executed in the cloud. This probability can be adjusted to reflect the current wireless network conditions, which will result in a quicker convergence to a good solution.

**Algorithm 4.3:** DPR Algorithm Pseudocode

$V_{best}$ = biased random binary vector of size $S'$ with probability of $P_{OFF}$ // get initial (best) binary vector
$P_{best}$ = TableFill($V_{best}$) // get associated (best) table path
// set energy along best path
$ES(P_{best}(t)) = El(t)V_{best}(t) + EC(t)(1 - V_{best}(t)) \quad \forall t \in \{1, 2, \ldots, S'\}$
**for** $k = 1$ **to** $Itr$ **do**
  $V_{new}$ = biased random binary vector of size $S'$ with probability of $P_{OFF}$// get new binary vector
  $P_{new}$ = TableFill($V_{new}$) // get associated new path
  LCT = 1 // set last common task between best/new paths
  **for** $T = 1$ **to** $S'$ **do**
    **if** $P_{best}(T) == P_{new}(T)$ **then**
      CTP = $(LCT, LCT + 1, \cdots, T)$ // set common task path
      // find the best CTP segment
      **if** $\sum_{t \in CTP} El(t)V_{new}(t) + EC(t)(1 - V_{new}(t)) < \sum_{t \in CTP} ES(P_{best}(t))$ **then**
        // update the best path
        $P_{best}(t) = P_{new}(t)$, $V_{best}(t) = V_{new}(t)$,
        $ES(P_{best}(t)) = El(t)V_{new}(t) + EC(t)(1 - V_{new}(t)) \quad \forall t \in CTP$
      **else**
        // update the new path
        $P_{new}(t) = P_{best}(t)$, $V_{new}(t) = V_{best}(t) \quad \forall t \in CTP$
      **end**
      LCT = $T$ // record the current common task, for future use
    **else**
      $ES(P_{new}(T)) = El(T)V_{new}(T) + EC(T)(1 - V_{new}(T))$
    **end**
  **end**
  $E_{best} = \sum_{c \in P_{best}} ES(c)$
  $E_{total} = E_{best}$
  calculate $TL$ and $TC$ using $M = V_{best}$ and Eq. (4.3) and Eq. (4.4)
  calculate objective function in Eq. (4.1) as total cost using $E_{total}$, $TL$ and $TC$
  $N_o = S' - \sum_{t=1}^{S'} M(t)$ // number of task offloads in solution $M$
  **if** $N_o \geq HDC$ **then**
    return $E_{total}, V_{best}$
  **end**
**end**

### 4.3.2 Simulation Results

The DPR algorithm was programmed using MATLAB. As in Section 4.2.3, we adopt the mobile device characteristics used by Chen *et al.* (2015), which are based on the Nokia N900 smart phone, and set the number of tasks to $S' = 10$ offloadable tasks. The topology of the task graph model that is used is shown in Figure 3.4. The local computation time is set to $4.75 \times 10^{-7}$ s/bit and the local processing energy consumption is set to $3.25 \times 10^{-7}$ J/bit. The x264 CBR encode application (1900 cycles/byte) is considered as task $i$ in our simulations, as in Chen *et al.* (2015).

The energy consumption for transmitting and receiving data for the mobile device are both set to $1.42 \times 10^{-7}$ J/bit. As in Section 4.2.3, the input ($DI$) and output ($DO$) data sizes of each task are chosen randomly, from uniformly distributions on 10 Mbyte to 30 Mbyte and on 1 Mbyte to 3 Mbyte, respectively, as in Chen *et al.* (2015). When tasks are offloaded to the cloud, the peak transmission rate is 15 Mbps and the transmission time of each task is equal to the size of each task divided by the current transmission rate. The CPU execution rate in the cloud is set to $10^{10}$ cycle/s.

We tried three different configurations for determining the bias probability $P_{OFF}$ when generating random bit streams. First, we set $P_{OFF} = 0.5$. In this case, the DPR algorithm reduces to the DPH algorithm in Section 4.2 (see also Shahzad and Szymanski, 2016a). Second, we used a fixed bias probability of $P_{OFF} = 0.8$, regardless of the network transmission rates and present several simulation results. In the last configuration, we allow the bias probability $P_{OFF}$ to be proportional to the network transmission rate as follows,

$$P_{OFF} = R/\theta \tag{4.6}$$

where $(1/\theta)$ is a proportionality constant ($\theta$ was set to 21 in the following experiments). This formula is one example of how $P_{OFF}$ can be related to the network transmission rate ($R$).

As mentioned in Section 4.2.3, the total cost of the mobile user is defined as the weighted sum of the total energy consumption, the costs to offload and process all tasks, and the corresponding worst-case transmission and processing delays. The formulation is presented in Eq. (4.1).

Similar to Section 4.2.3, we assume $\alpha = 1 \times 10^{-6}$ J/bit, $\beta = 5 \times 10^{-7}$ J/bit and we set the value of the cost $C_{c_i}$ to be the same as that of the input data size $DI_i$. Table 4.4 shows a comparison of total cost for All-Local execution, All-Remote execution, the LC (Local-Cloud) algorithm (Chen *et al.*, 2015), the optimal results from Brute-Force search, and the results from our DPR algorithm. Table 4.4 shows that for the chosen task graph the DPR algorithm finds good solutions as compared to the optimal results (about 1% higher than the optimal result).

Table 4.4: Comparison of the Total Cost of Based on Eq. (4.1) for Different Algorithms

| | **All-Local** | **All-Remote** | **Optimal** | **LC** | **DPR** |
|---|---|---|---|---|---|
| **Total Cost (Joules)** | 1322.1 | 1265.2 | 1128.1 | 1133.4 | 1145.3 |

Figure 4.7 presents the total cost of the DPR algorithm compared to All-Remote and All-Local, the LC algorithm (Chen *et al.*, 2015), the lower bound on minimum cost, which is the result of solving the SDR problem, and the optimal result of Brute-Force search, as the transmission rate is varied between 4 and 20 Mbps. The total cost is computed using Eq. (4.1). Figure 4.7 illustrates that the cost of the All-Remote

execution model decreases when the transmission rate increases. The LC algorithm (Chen *et al.*, 2015) is able to find a near optimal result for the selected range of transmission rates. For this experiment, we used a fixed bias probability without considering the transmission rate. When the network conditions are poor, the DPR result is not as good, since it tries to offload as many tasks as possible and therefore the final decision for low network transmission rates will be affected by the large cost of offloading, as shown in Figure 4.7. As the network transmission rate increases, DPR will be able to find a better result since offloading becomes more favorable. Indeed, for the chosen task graph the result of the highest transmission rate is within 1% of the optimal value from Brute-Force search.

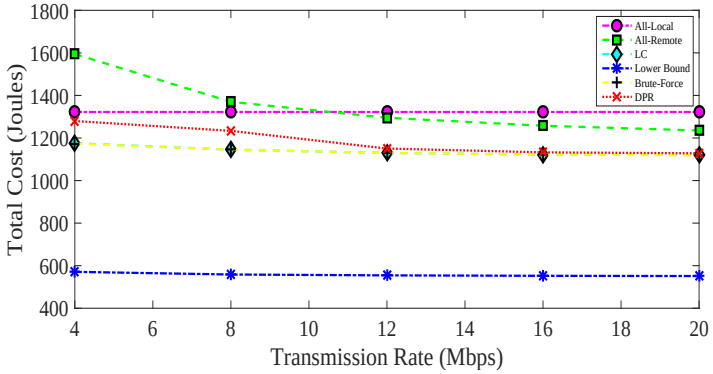


Figure 4.7: Total Cost vs Transmission Rate $R$ (Mbps) for Different Methods when $P_{OFF} = 0.8$

Figure 4.8 presents the total cost of the different algorithms versus the network transmission rate, similar to Figure 4.7, but with a bias probability, $P_{OFF}$, that is proportional to the transmission rate. Figure 4.8 shows that with this new strategy, in poor network conditions, DPR is able to find a better result compared to the fixed bias probability, since the bias towards the offloading is smaller. As a result, the initial solution has more degrees of freedom and less tendency toward offloading,

which leads to a better final solution.

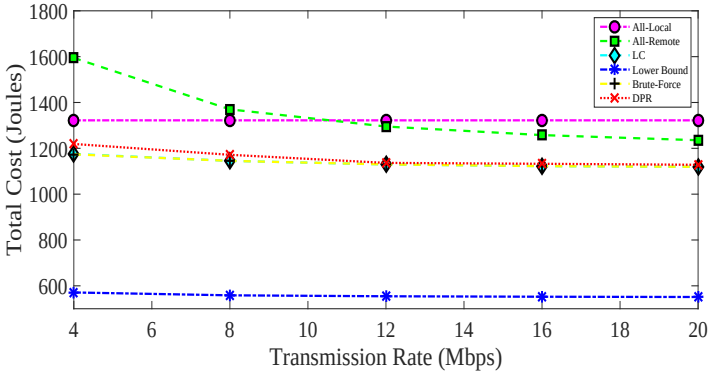


Figure 4.8: Total Cost vs Transmission Rate $R$ (Mbps) for Different Methods when $P_{OFF} = R/\theta$

Figure 4.9 shows a comparison of total cost between different methods when the weight of cloud processing cost, $\beta$, is changed between $0.2 \times 10^{-6}$ and $1 \times 10^{-6}$. Figure 4.9 illustrates that as $\beta$ increases, the best solution will converge to the All-Local solution due to the increased cost of offloading. This cost may include the service provider cost, price or any other related cost of using cloud services.

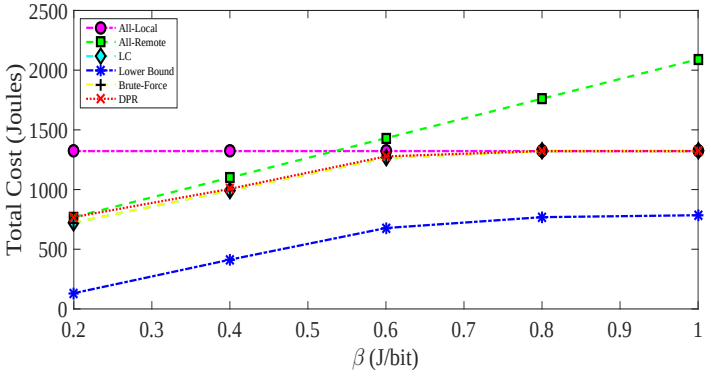


Figure 4.9: Total Cost vs $\beta$ (J/bit) for Different Methods.

Figure 4.10 shows the comparison of different methods when the weighting of processing delay ($\zeta$) ia varied between 0.2 and 2 J/s. This changes the emphasis on

delay rather than energy consumption. As a result, the total cost will also increase in all methods.

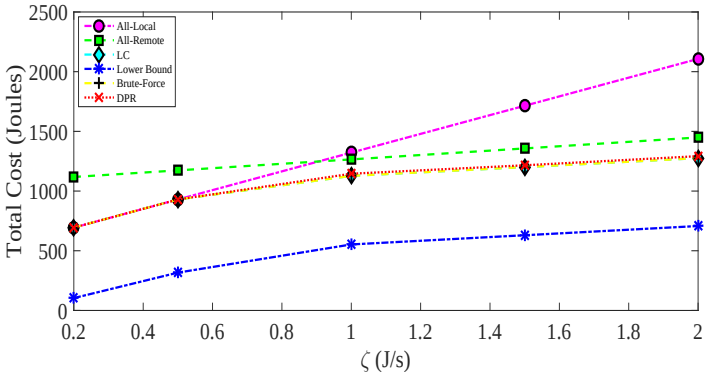


Figure 4.10: Total Cost vs $\zeta$ (J/s) for Different Methods.

Table 4.5 shows the execution time of the DPR algorithm with $P_{OFF} = 0.5$ and 0.8 when the number of tasks in the parallel task graph in Figure 3.4 ranges from 12 to 25. The input and output data sizes are chosen randomly from the same distributions used in the earlier experiments. The download and upload transmission rates are 8 Mbps and 6 Mbps, respectively. As can be seen in Table 4.3, the LC algorithm has an execution time that is more than 3 seconds for the case of 12 tasks, which is about 40 times the execution time of the DPR algorithm. The results in Table 4.5 are consistent with the fact that the execution time of the DPR algorithm is polynomial in the number of tasks (rather than growing exponentially as many other algorithms exhibit). Doubling the number of tasks from 12 to 24 only increases the execution time by 20%. Also, by adjusting the bias probability to 0.8 when generating random bit streams, the algorithm converges more quickly. It is clear that for the chosen transmission rates, having a higher probability to offload improves the execution time since with these network rates, offloading is beneficial.

Table 4.5: Execution Time of DPR for Different Numbers of Tasks

| $S'$ | **12** | **14** | **18** | **20** | **25** |
|---|---|---|---|---|---|
| **Execution Time of DPR (ms) for $P_{OFF} = 0.5$** | 72.9 | 74.2 | 78.3 | 80 | 81.3 |
| **Execution Time of DPR (ms) for $P_{OFF} = 0.8$** | 66.3 | 71.9 | 72.5 | 74.5 | 77.8 |

Table 4.6 shows the execution time of the DPR algorithm for upload transmission rates of 2, 4, 6, 10 and 15 Mbps, download transmission rate of 4 Mbps, for the three different bias probabilities, when the number of offloadable tasks is $S' = 12$. It is clear that for higher transmission rates, offloading to the cloud is beneficial, so that increasing the bias probability $P_{OFF}$ to 0.8 results in improved performance. Conversely, when the transmission rate is low, the bias probability $P_{OFF}$ should be decreased to reflect the lower benefits of offloading. A single value for the bias probability is not effective for all possible network transmission rates. When Eq. (4.6) is used to adjust the bias probability $P_{OFF}$ to be proportional to the network transmission rate, the DPR algorithm can find a good quality solution quickly over a wide range of network conditions. In this experiment we chose $\theta = 21$, so with transmission rates of 2, 4, 6, 10, 15 Mbps, the adapted values of $P_{OFF}$ are approximately 0.1, 0.2, 0.28, 0.48, 0.7, respectively. In particular, when $R = 2$ Mbps, Eq. (4.6) generates $P_{OFF}$= 0.1 and this leads to much faster convergence than the pre-set value $P_{OFF}$= 0.8. When $R$ = 15 Mbps, Eq. (4.6) generates $P_{OFF}$= 0.7 and this results in a similar convergence time to the case of the pre-set value $P_{OFF}$= 0.8.

Table 4.6: Execution Time of DPR versus Upload Transmission Rate, for Three Different Bias Probabilities $P_{OFF}$ in the DPR Algorithm

| | $R = 2$ | $R = 4$ | $R = 6$ | $R = 10$ | $R = 15$ |
|---|---|---|---|---|---|
| **Execution Time of DPR (ms) for $P_{OFF} = 0.5$** | 75.1 | 73.5 | 72.9 | 71.6 | 70.1 |
| **Execution Time of DPR (ms) for $P_{OFF} = 0.8$** | 85 | 84 | 83.2 | 67.7 | 68.7 |
| **Execution Time of DPR (ms) for $P_{OFF}$ using Eq.** (4.6) | 73.8 | 73 | 71 | 69.7 | 68.7 |

## 4.4 Summary

In this chapter, two dynamic programming algorithms called DPH, "Dynamic Programming with Hamming Distance Termination" and DPR, "Dynamic Programming with Randomization" were introduced. Random bit strings of 0s and 1s are periodically generated and sub-strings that improve the solution are incorporated into the solution, in a method similar to genetic optimization. When random binary vectors are generated, DPH uses pure randomization while DPR exploits biased randomization. Our numerical results show that for the chosen (parallel) task graphs, this approach can find good solutions with low computation times.

# Chapter 5

# Randomized Offloading Algorithms (ROA-V1 and ROA-V2)

In this chapter, different versions of a computation offloading algorithm called the *Randomized Offloading Algorithm* (ROA) are described (see also Shahzad and Szymanski, 2017; Szymanski and Shahzad, 2018; Shahzad and Szymanski, 2018). As in Chapter 4, we are given $S$ computational tasks. In the case of parallel task graphs, we will explicitly formulate the offloading problems in terms of the $S'$ tasks that are offloadable. For the case of general task graphs, we will simplify the notation by considering a generic decision vector of length $S$. The decision vector is improved iteratively, and randomization is used to generate a *candidate vector* bit-string in each iteration. If fragments of the candidate vector improve the decision vector, they are incorporated into the decision vector, in a process similar to genetic optimization. Our ROA algorithms also use user-controlled parameters to determine how many iterations are used, and how much computational effort is used when incorporating candidate vectors into the current decision vector.

The proposed ROA algorithms presented in this chapter are simpler and more powerful than DPH and DPR that were presented in Chapter 4. They remove the 2-dimensional DP lookup table, simplify the rules for improving the decision vector, and more closely resemble *Genetic Optimization*. For convenience, the parameters used to formulate these models are provided in Table 5.1. Note that as in Chen *et al.* (2015), we assume that the per bit energy for both uploading and downloading is the same.

Table 5.1: Description of the Model Parameters

| **Symbols** | **Meaning** |
|---|---|
| Both Parallel and General Model | |
| $El_i/Tl_i$ | Energy / Time taken to execute task $i$ locally |
| $EI_i/TI_i$ | Energy / Time consumed in mobile device when task $i$ is executing remotely |
| Parallel Model | |
| $Et_i/Tt_i$ | Energy / Time taken to transfer input data of task $i$ to the remote server and back |
| $T_{NET}$ | 2-way delay to access the remote server |
| General Model | |
| $Et_{ij}/Tt_{ij}$ | Energy / Time taken to transfer input data of task $j$ from task $i$ from/to the remote server |
| $T_{NET}$ | 1-way delay to access the remote server |

## 5.1 Problem Formulations for the Parallel Model

Two different problem formulations are considered in this section. The first is similar to that in Chapter 4 (and that in Chen *et al.*, 2015), in which the objective is a linear combination of the energy consumption and the time delay, and the constraints are simply the binary constraints on the decisions. The second formulation uses the energy consumption as the objective function and includes constraints on the execution time.

In both formulations, we are given $S$ computational tasks with $S'$ independent offloadable tasks that can be executed either locally or remotely.

### 5.1.1 Parallel Model with Energy and Time as the Objective

As in Chapter 4, we define $M = [M_1, M_2, \ldots, M_{S'}]$ as the vector of binary offloading decisions, where $M_i \in \{0, 1\}\ \forall i$ and $M_i = 0$ indicates that the task is to be offloaded. The optimization problem that we want to solve is given as follows.

$$\begin{aligned} \min_{\{M_i\}} \quad & \sum_{i=1}^{S'} El_i\, M_i + EC_i(1 - M_i) + \zeta \max\{TL(\{M_i\}), TC(\{1 - M_i\})\} \\ \text{s.t.:} \quad & M_i \in \{0, 1\},\ \forall i \in \{1, 2, \ldots, S'\} \end{aligned} \tag{5.1}$$

The objective in Eq. (5.1) gives the total mobile user cost, as in Chapter 4 (and Chen *et al.*, 2015). It is the weighted sum of the total energy consumption, the costs to offload and process tasks in the remote server and the corresponding worst case transmission and processing delays. The parameter $\zeta$ is the weighting of the total delay, and can be adjusted to change the emphasis on delay and energy consumption

(Chen *et al.*, 2015). Similarly, $EC_i$, $TL$ and $TC$, which are defined below, are, respectively, the weighted processing cost of offloading to the cloud servers with $\beta$ being the relative weight, the processing delay at mobile user, and the worst case processing delay at cloud servers. The terms are defined as follows,

$$EC_i = Et_i + \beta Cc_i \tag{5.2}$$

$$TL(\{M_i\}) = \sum_{i=1}^{S'} M_i \, Tl_i \tag{5.3}$$

$$TC(\{1 - M_i\}) = \sum_{i=1}^{S'} (1 - M_i)(Tt_i + Tc_i) \tag{5.4}$$

Here, as in Chapter 4, $Tc_i$ denotes cloud processing time for task $i$, $Cc_i$ is the cost for letting the cloud process task $i$, and $Tt_i$ gives the transmission time for task $i$ between the mobile device and cloud.

### 5.1.2 Parallel Model with an Energy Objective

The energy consumption and its corresponding worst-case execution time (in the sense of Cuervo *et al.*, 2010) are defined as follows:

$$E = \sum_{i=1}^{S'} M_i El_i + (1 - M_i)(EI_i + Et_i) \tag{5.5}$$

$$T = \max\Big(\sum_{i=1}^{S'} M_i Tl_i \, , \sum_{i=1}^{S'} (1 - M_i)(TI_i + Tt_i + T_{NET})\Big) \tag{5.6}$$

The term $T$ denotes the total execution time of the application (which consists of all the tasks). This must satisfy the following condition, where $T_C$ is the execution time

requirement of the application,

$$T \leqslant T_C \tag{5.7}$$

The optimization problem that we want to solve with respect to Eq. (5.5) and Eq. (5.6) is as follows,

$$\begin{aligned} \min_{\{M_i\}} \quad & E \\ \text{s.t.} : \quad & T \leqslant T_C \\ & M_i \in \{0,1\},\ \forall i \in \{1,2,\ldots,S'\} \end{aligned} \tag{5.8}$$

The number of combinations of binary values $M_i$ to search for the optimal solution grows exponentially with the number of offloadable tasks. Our goal is to determine which tasks should be offloaded to the remote server so that energy is minimized. Determining an optimal decision vector $M$, that minimizes energy use and meets the execution time constraint is NP-Hard.

## 5.2 Problem Formulations for the General Model

In this section, a single mobile user with a general application model is considered. This model was discussed in Section 3.1.2 and the model parameters in this section are provided in Table 5.1. In the general case, we are given $S$ computational tasks. The set of offloadable tasks varies depending on the application task graph so they are not considered as a separate set of tasks, as before. Instead, the optimization problem is formulated considering the total number of tasks $S$.

The formulation used in this section is provided by many authors, including

Cuervo *et al.* (2010) and Deng *et al.* (2016). The total energy used in an offloading solution is defined as follows.

$$E = \sum_{i=1}^{S} M_i El_i + (1 - M_i) EI_i + \sum_{(i,j) \in EM} |M_i - M_j| Et_{ij} \tag{5.9}$$

The total elapsed time to execute all the tasks that constitute the application is, in the worst-case (Cuervo *et al.*, 2010),

$$T = \sum_{i=1}^{S} M_i Tl_i + (1 - M_i) TI_i + \sum_{\substack{(i,j) \\ \in EM}} |M_i - M_j| (Tt_{ij} + T_{NET}) \tag{5.10}$$

As in Cuervo *et al.* (2010), the worst-case time consumption is considered in order to reduce the complexity of the the solution algorithm and its implementation. The total execution time $T$ must satisfy the following, where $T_C$ is the execution time constraint.

$$T \leq T_C \tag{5.11}$$

The final problem to solve is as follows,

$$\begin{aligned} \min_{\{M_i\}} \quad & E \\ \text{s.t.:} \quad & T \leqslant T_C \\ & M_i \in \{0, 1\}, \ \forall i \in \{1, 2, \ldots, S\} \end{aligned} \tag{5.12}$$

Determining an optimal decision vector $M$ that minimizes the energy consumption and meets the execution time constraint is NP-Hard.

## 5.3 Motivation Based on Genetic Optimization

A Genetic Algorithm (GA) is a population based optimization method consisting of a random initial population of "chromosomes". Pseudocode of the steps in a GA is presented in Algorithm 5.1 (see Kalra and Singh, 2015). Each chromosome denotes a possible solution. In our application, that is a candidate decision vector. An initial population of the chromosomes is generated in line 1. A fitness function is defined to check the quality of the solutions (chromosomes). In our case, the fitness function is simply the objective of the optimization problem. Based on the fitness function evaluation (line 2), chromosomes are selected (line 3) for the crossover (line 4) and mutation (line 5) operations. As a result, offspring for the new population will be generated. The quality of offspring is further evaluated by the fitness function (line 6) and the population is updated with better offspring (line 7). This process is repeated to obtain a sufficient number of offspring (lines 8-10).

In the literature, different variations of each step in the GA are explored, i.e., they define their own methods to do the initialization, selection, crossover, mutation, fitness and replacement steps. A summary of some of these papers is provided in this section. Our proposed algorithms are also inspired by GA but with different procedures that are explained in Sections 5.4 and 5.8.

**Algorithm 5.1:** Pseudocode of the Genetic Algorithm

**Result:** Best Chromosome
1 **Initialization:** Generate initial population of the chromosomes
2 **Fitness:** Evaluate each chromosome based on the fitness function
3 **Selection:** Select chromosomes to generate the new population
4 **Crossover:** Perform the crossover on the selected chromosomes from step 3
5 **Mutation:** Perform the mutation operation on the chromosomes
6 **Fitness:** Evaluate the new chromosomes (offspring) based on the fitness function
7 **Replacement:** Update the population by substituting bad chromosomes with better offsprings
8 **while** *stopping condition is not met* **do**
9 | Repeat Steps 3-7
10 **end**

Tout *et al.* (2017) used a multi-objective genetic algorithm to find components that should be offloaded and those to be executed locally for trading off computing capacity, memory and battery usage. The algorithm starts with a population of $N$ randomly generated individuals, each with a size $H$. In the selection process, the fittest $b$ individuals in the population are selected using a tournament selection method, which involves randomly chosen chromosomes from the population in several rounds. The winner in each round, which has the best fitness, is then selected for crossover. The crossover operator is based on the differential evolution of individuals that optimizes offloading. Taking two parent individuals, genes that produce better fitness when compared to their parents are used to form the offspring. Standard bit flip mutation is used in the algorithm. The evaluation process continues until any of the stopping criteria is met, where either the fitness of the best individual in successive populations did not improve or the defined number of iterations is reached.

Cheng *et al.* (2015) also proposed an algorithm based on the principles of the genetic algorithms. In the first step, a set of feasible solutions as an initial population,

is generated. Order crossover is used as the crossover operator where given any two parent chromosomes, a crossover point is randomly chosen. Then the left segment of the offspring is taken from the first parent and the right fragment of the offspring is taken from the remaining parts of the first parent, but in the order of the other parent. The mutation operation is implemented by swapping two randomly selected tasks and to guarantee the feasibility, every chromosome is checked after mutation to abandon infeasible ones. This process is repeated until the desired size of the populations is obtained. The complexity of the algorithm is not reported by Cheng *et al.* (2015). However it is claimed that this algorithm outperforms traditional GA with a much lower execution time.

Qiu *et al.* (2015) used a genetic-based algorithm to assign tasks to cores in a chip multiprocessor system. A randomly generated population is used in this algorithm. In the selection process, fitness functions of all chromosomes are evaluated and chromosomes are sorted in descending order of fitness. A rank-based roulette wheel selection method is used to select chromosomes. In the crossover procedure, $R$ pairs of chromosomes are selected randomly by using the rank-based roulette wheel method. A crossover point is selected randomly, the upper part of a string and the lower part of another string are kept unchanged while the remaining parts are re-ordered based on the task order in the upper part of the first string and lower part of the other, respectively. A bit is randomly selected for mutation and it is changed to another randomly selected value. The algorithm iterates until the total generation reaches a predefined number or an improvement is not available.

Tseng *et al.* (2018) proposed a multi-objective genetic algorithm (GA) to predict

the resource utilization and energy consumption in cloud data centers. Two chromosomes are randomly selected and a two-point crossover approach with randomly selected points is utilized. If the fitness value of the offspring is better than the parents, the offspring goes back to the new population. Otherwise, the offspring goes through the mutation process and will be checked afterwards. A chromosome is able to mutate itself in the mutation operation, so that the likelihood of being trapped in a local solution is reduced. A chromosome with the smallest fitness value is selected as the final result.

Yang *et al.* (2013) used computation partitioning of a data stream application between the mobile and cloud to achieve maximum speed/throughput. In order to solve the partitioning problem, a genetic based algorithm is proposed. The initial population is randomly generated with the roulette wheel selection method. Two randomly selected chromosomes then go through the crossover process. At a randomly chosen point, two selected chromosomes are divided into two parts. New offspring take one part from the first chromosome and another part from the second. Mutation takes an individual and randomly changes one or multiple values. The algorithm terminates when the number of generations has reached a certain upper bound. It is shown that this algorithm takes about 117 generations to reach 90% throughput for a graph size of 30.

## 5.4 The ROA-V1 Algorithm

In this section, ROA-V1 is described. This description is a slightly modified version of that in Shahzad and Szymanski (2017). This version of the algorithm uses a user-defined parameter, $\rho$, which determines how many sub-strings the candidate vector

will be partitioned into and is evaluated for the parallel and general task models.

The pseudocode of the *Randomized Offloading Algorithm* (ROA-V1) is shown in Algorithm 5.2. In the initialization steps (lines 1 and 2), the parameters $T_C$, the network transmission rate and $\rho$ are defined. The value $T_C$ is the time constraint for the total execution time of the application and $\rho$ denotes a threshold that controls the computational cost. The ROA-V1 algorithm first generates an initial decision vector $M$, also called the current solution, by generating a random bit-string of 0s and 1s of length $S'$ (in line 3), where $S'$ denotes the number of tasks in the application. This random bit-string represents a first decision vector of binary values, where each task may be offloaded (denoted with a 0) or may be executed locally (denoted with a 1). It then attempts to improve this solution iteratively. In each iteration $i > 1$, it generates another random bit-sting of length $S'$ in line 5, called $V_{new}$, similar to the "candidate solutions" in genetic optimization. The string $V_{new}$ will be partitioned into several sub-strings, using a user-defined parameter $\rho$ defined below. ROA-V1 then evaluates whether each sub-string of $V_{new}$ should be incorporated into the current solution $M$. A sub-string in $V_{new}$ is incorporated into $M$ by replacing the existing sub-string if it improves the current solution. When generating the candidate vector $V_{new}$, biased randomization can also be used. For example, if the network conditions are good then tasks are more likely to be offloaded, and the probability of generating 0s in the random bit-stream can be increased accordingly (Shahzad and Szymanski, 2016b).

The parameter $\rho$ is a user-supplied probability between 0 and 1, which determines how many sub-strings the candidate vector will be partitioned into, and the expected length of each sub-string. The ROA-V1 algorithm determines the bit positions where the candidate vector $V_{new}$ and the current solution vector $M$ differ, by xor-ing the two

**Algorithm 5.2:** Randomized Offloading Algorithm (ROA-V1) Pseudocode

Initialize time constraint ($T_C$) and Transmission Rate
Initialize $\rho$ as the computation threshold
$M$ = random binary vector of size $S'$ // initial binary solution vector
**for** $i = 1$ **to** $Itr$ **do**
  $V_{new}$ = random binary vector of size $S'$ // new binary solution vector
  $I_{M,V} = ( i \mid M(i) \neq V_{new}(i) \ \ \forall i = 1, 2, \ldots, S')$ // ordered indices where $M$ and $V_{new}$ do not agree
  $l = 1$ // $l$ is lower task fragment index
  **for** $u \in I_{M,V}$ **do**
    // $u$ is upper task fragment index
    **if** uniform$(0, 1) \leq \rho$ **then**
      // test to incorporate this $V_{new}(l : u)$ fragment into $M$
      $E^{new} = \sum_{i=l}^{u} ES(i, V_{new}(i))$ // task fragment energy using $V_{new}$
      $E^{M} = \sum_{i=l}^{u} ES(i, M(i))$ // task fragment energy using $M$
      $T_{total} = \sum_{i=1}^{l-1} TS(i, M(i)) + \sum_{i=l}^{u} TS(i, V_{new}(i)) + \sum_{i=u+1}^{S'} TS(i, M(i))$
      **if** $E^{new} < E^{M} \wedge T_{total} < T_C$ **then**
        // substring $l{:}u$ of $V_{new}$ incorporated into $M$
        $M(i) = V_{new}(i) \quad \forall i \in \{l, l+1, \ldots, u\}$
      **end**
    **end**
    $l = u + 1$
  **end**
**end**
Compute $E_{total}$ and $T_{total}$ given $M$
Return $E_{total}$, $T_{total}$ and decision vector $M$

vectors and recording the ordered indices in $I_{M,V}$ in line 6. For every index in $I_{M,V}$, a random number between 0 and 1 is generated in line 9. If the random number is less than or equal to the parameter $\rho$ (in line 9), then this bit-position defines the end of one sub-string in $V_{new}$ and the beginning of a new sub-string in $V_{new}$. ROA-V1 then performs computations (in lines 10 to 14) to determine whether or not to incorporate the recently-terminated sub-string in $V_{new}$ into the current solution $M$. In line 6, the expected number of different bits in $M$ and $V_{new}$ is $S'/2$, since the candidate vector $V_{new}$ is randomly generated. The expected number of times a sub-string will be terminated is therefore $(S'/2) \cdot \rho$. The expected length of a sub-string is therefore $S'$ divided by the expected number of sub-strings, which is $\approx 2/\rho$.

If $\rho = 1$, the candidate vector is partitioned into $\approx S'/2$ sub-strings with an expected length of 2 bits. Each sub-string in $V_{new}$ differs from the corresponding sub-string in $M$ by at least 1 bit. If $\rho = 0.5$, the candidate vector is partitioned into $\approx S'/4$ sub-strings with an expected length of 4 bits. Each sub-string in $V_{new}$ differs from the corresponding sub-string in $M$ by at least 2 bits on average. For smaller $\rho$, the candidate vector $V_{new}$ will be partitioned into a smaller number of longer sub-strings, which will minimize the amount of work the ROA-V1 algorithm performs in each iteration. The inner loop from lines 8 to 18 will determine and process the sub-strings in the candidate vector $V_{new}$ (Each sub-string starts and ends at indices $l$ and $u$). The expected energy and time of each sub-string, given the candidate vector $V_{new}$, are calculated in lines 10 and 12 (by using the energy consumption of each task and summing over bits $l$ to $u$). The algorithm then compares the total energy of each sub-string, in both vectors $V_{new}$ and $M$, in line 13. The best sub-string is incorporated into the current solution vector $M$, provided that the time-constraint is

not violated. The solution vector will be returned at the end of the algorithm.

## 5.5 Time Complexity of ROA-V1

The time complexity of the ROA-V1 algorithm is determined by evaluation of the sub-strings based on either Eq. (5.1) or Eq. (5.5) and Eq. (5.6), the population size of the random bit strings, number of tasks in the task graph and the number of iterations in the evaluation process.

**Lines 1-7:** Initializing step, main loop ($i$) and generating the first random population have time complexity of $O(S')$.

**Lines 8-15:** Loop $u$ has a complexity of $O(S')$. Lines 10, 11 and 12 have the complexity of $O(S')$ for the parallel task graph. Lines 13 and 14 have the complexity of $O(1)$. The time complexity of lines 8-15 is $O(S'^2)$.

Evaluation of the Energy and Time equations over the loop ($Itr$) has time complexity of $O(Itr \cdot S'^2)$ where $Itr$ is the total iterations of the outer loop and $S'^2$ is the time complexity of the loop in lines 8-15 that evaluates the energy and time equations and sub-string incorporation in the current solution vector.

The time complexity of the ROA-V1 algorithm is the worst time complexity among all the steps that are explained. Therefore, the time complexity of ROA-V1 is $O(Itr \cdot S'^2)$.

## 5.6 ROA-V1 Simulation Results for the Parallel Model

The topology of the task graph model that is used to test this version of the algorithm is the parallel topology that was introduced in Section 5.1. We consider three different sets of simulation parameters. The first set is used for comparisons with the work of Chen *et al.* (2015). The second is used to explore the effects of tasks with variable amounts of computational work. The third is used to explore the effects of nearby edge servers versus remote cloud servers. These three sets are summarized in Table 5.2 (The material in this section is a refined version of that in Shahzad and Szymanski, 2017).

In our simulations, a mobile device is characterized by several parameters, namely, $f_L$, $P_L$, $P_I$, $P_T$, $C$, $DI$, $DO$, where $f_L$ denotes the local computation rate expressed in CPU clock cycles per second, and the mobile processor can issue a fixed number of machine instructions per clock cycle, typically between 1 and 4. The term $P_L$ denotes the local power consumption of the mobile device (in watts), $P_I$ is the idle mode power consumption (also in watts), and $P_T$ is the power consumption during transmission (also in watts). The variable $C_i$ denotes the required CPU cycles to process each bit (or megabit) of data in task $i$, and reflects the computation workload of the task. $DI$ denotes the size of the input data per task (in bytes), and $DO$, the size of the output data per task. In the experiments, $DI$ and $DO$ are chosen randomly from uniform distributions on the intervals given in Table 5.2. In all the simulation results, the label "All-Local" shows the results when all the tasks are executed locally, and the label "All-Remote" shows the results of executing all the tasks remotely on the

cloud or edge server. The cloud and edge servers are characterized by two parameters: $f_C$ and $f_E$, where $f_C$ denotes the cloud server computation rate in CPU cycles per second and $f_E$ denotes the edge server computation rate. We assume the edge and cloud servers have comparable computational power and set $f_C = f_E$.

Table 5.2: Simulation Parameters

| **Parameters** | **Definition** | **Model Comparison** | **Variable Work Per Task** | **Faster Cloud Servers** |
|---|---|---|---|---|
| $f_L$ | Local computation power | 500 Mcycle/sec | 500 Mcycle/sec | 1 Gcycle/sec |
| $f_C$ | Cloud server computation power | 10 Gcycle/sec | 10 Gcycle/sec | 100 Gcycle/sec |
| $P_L$ | Local power consumption | 0.7 w | 0.7 w | 8mw |
| $P_I$ | Idle mode power consumption | 30mw | 30mw | 0.2mw |
| $P_T$ | Transmission power consumption | 1.1 w | 1.1 w | 0.5 w |
| $C$ | Computation workload of the task | 1900 CPU cycles/byte | 70-2200 CPU cycles/byte | 2.5-25 Giga CPU cycles |
| $DI$ | Input data size | 10-30 Mbyte | 10-30 Mbyte | 0.5-10 Mbit |
| $DO$ | Output data size | 1-3 Mbyte | 1-3 Mbyte | 0.5-2 Mbit |
| $S'$ | Number of offloadable tasks in an application | 10 | 12 | 20 |

### 5.6.1 Algorithm Comparison

In this section, we use the same parameters as Chen *et al.* (2015) to evaluate the ROA-V1 algorithm and compare its results with those obtained when the algorithm of Chen *et al.* (2015) is applied to the case of an expensive CAP; see Sections 4.2.3, 4.3.2 and the discussion below. The problem formulation is the "total cost minimization" formulation presented in Section 5.1.1. As discussed earlier in the thesis Chen *et al.* (2015) used *Semidefinite Relaxation* followed by randomized rounding to find

offloading task decisions. For this study, their algorithm was implemented in MATLAB. The results from the proposed algorithm were also compared with the optimum solution, which is obtained through a Brute-Force Search (for small problems). BFS is a technique which consists of testing all possible solutions, and recording the best obtained (i.e., that which achieves the minimum cost). Chen *et al.* (2015) showed that SDP followed by 100 relaxation iterations can yield solutions that are within 1% of the optimal solution.

As in our earlier experiments, we use the mobile device parameters of Chen *et al.* (2015), which are based on the Nokia N900/500MHz smartphone, and are shown in Table 5.2. We set the number of tasks to $S' = 10$, as used by Chen *et al.* (2015). In this subsection, we consider only cloud servers. Using the parameters of Chen *et al.* (2015), which are shown in Table 5.2, the local computation time for a task can be seen to be 0.475 seconds per megabit, and the local energy use for a task is equal to 0.325 watts per megabits per second.

As stated in Section 4.3, Chen *et al.* (2015) introduced a two stage offloading model with a CAP and a cloud and a new parameter, $\alpha$, to control the cost of processing in the CAP and the cloud. When $\alpha$ is large, then no tasks will be processed at the CAP, and all tasks will be sent to the cloud for processing; which results in a system model analogous to ours. Chen *et al.* (2015) presented results for the LC case (Local-Cloud), where all processing is performed in the cloud. To compare our results with their results, we assume that all tasks are executed in the cloud (specifically, we set $\beta = 5 \times 10^{-7}$ J/bit, $\alpha$ to be large and we set the value of the cost $Cc_i$ to be the same as that of the input data size $DI_i$). Table 5.3 presents the total cost of the LC case (Chen *et al.*, 2015) and the optimal results from the Brute-Force Search. The results for

the ROA-V1 algorithm are also shown as are the results for two other cases, when all the tasks are executed either locally (All-Local) or remotely (All-Remote). Table 5.3 shows that the ROA-V1 algorithm finds nearly identical solutions when compared to the method of *Semidefinite Relaxation* followed by 100 iterations of randomization (Chen *et al.*, 2015). For the chosen task graph, the ROA-V1 algorithm yields nearly-optimal solutions that are significantly better than the All-Remote and All-Local decision vectors, as shown in Table 5.3.

Table 5.3: Comparison of the Total Cost Based on Eq. (5.1) for Different Algorithms

| | **All-Local** | **All-Remote** | **Optimal** | **LC** | **ROA-V1** |
|---|---|---|---|---|---|
| **Total Cost (Joules)** | 1322.1 | 1265.2 | 1128.1 | 1133.4 | 1138.24 |

### 5.6.2 Variable Work per Task

In this section, we explore the effects of varying the work required per task. The problem formulation is the delay-constrained formulation presented in Section 5.1.2. We compare the results of the ROA-V1 algorithm and the Brute-Force Search results. Each algorithm was implemented in MATLAB. A parallel task graph similar to that in Figure 3.4 is used, but with $S' = 12$ offloadable tasks. The workload of the task graph is [18.3837, 7.0845, 21.3102, 20.3503, 61.3687, 8.1341, 52.5786, 10.6591, 61.6240, 42.8682, 25.1096, 36.0973] Giga CPU cycles. The input and output data sizes of the tasks are [23.53, 27.51, 12.49, 16.57, 29.77, 27.62, 26.63, 19.68, 29.65, 28.11, 23.99, 19.42] Mbytes and [1.55, 1.77, 1.18, 1.01, 1.35, 1.54, 2.87, 1.72, 1.58, 2.87, 2.57, 2.62] Mbytes, respectively.

The comparison between the total energy consumption of the mobile application using these methods is summarized in Table 5.4. For the ROA-V1 algorithm, we provide the results for several values of the parameter $\rho$. In this table, the column "$Itr$" indicates the number of iterations. The column "Mean Energy" indicates the average energy used, averaged over the results of 10 runs of the ROA-V1 algorithm. The value "$\Delta$" equals the difference between the mean ROA-V1 energy result, and the optimal energy result obtained from the Brute-Force Search method.

When $\rho = 1$ in ROA-V1, the algorithm will perform more work when evaluating a candidate vector in each iteration. According to Table 5.4, setting $\rho = 1$ gives the best performance for a fixed number of iterations (i.e., for one row of the table), but it also uses the most computation since the ROA-V1 algorithm will perform more work per iteration. We have observed that the execution times of the ROA-V1 algorithm for $K$ iterations with $\rho = 1$, and for $2K$ iterations when $\rho = 0.5$, are comparable.

The results in Table 5.4 indicate that for the chosen task graph using a smaller $\rho$ and more iterations is preferable. This case allows for more candidate sub-strings to be generated and evaluated, which, in general, improves the ROA-V1 algorithm performance. Hence, from the computational time perspective, it is beneficial to select a smaller $\rho$ and allow for more iterations. From the results in Table 5.4, it is clear that ROA-V1 finds excellent results within 50 iterations for the chosen (parallel) task graph.

### 5.6.3 Faster Cloud Servers

In this section, the simulation parameters are changed to consider more complex applications, and to consider more powerful cloud servers. The parameters in this

Table 5.4: Comparison of Total Energy Consumption (in Joules) for the variable work per task parameters.

| ***Itr*** | | **ROA** $\rho = 0.25$ | **ROA** $\rho = 0.5$ | **ROA** $\rho = 0.75$ | **ROA** $\rho = 1$ | **Optimal Energy** |
|---|---|---|---|---|---|---|
| **10** | **Mean Energy** | 313.25 | 307.17 | 305.36 | 300.95 | |
| | **Δ%** | 7.30 % | 5.20 % | 4.60 % | 3.10 % | 291.84 |
| **20** | **Mean Energy** | 297.04 | 292.81 | 292.18 | 291.84 | |
| | **Δ%** | 1.70 % | 0.33 % | 0.115 | 0% | 291.84 |
| **50** | **Mean Energy** | 291.84 | 291.84 | 291.84 | 291.84 | |
| | **Δ%** | 0 % | 0 % | 0 % | 0 % | 291.84 |
| **100** | **Mean Energy** | 292.25 | 291.84 | 291.84 | 291.84 | |
| | **Δ%** | 0 % | 0 % | 0 % | 0 % | 291.84 |

section are summarized in the last column of Table 5.2. In particular, the cloud server become much more powerful than the mobile device, and the work per task also becomes much larger. The problem formulation is the delay-constrained formulation presented in Section 5.1.2.

The local computational performance in the mobile device is taken to be $f_L =$ 1 Gigacycles per second, typical of current devices (as in Song *et al.*, 2014). The corresponding remote server performance is $f_C = 100$ Gcycles per second, reflecting very powerful remote servers (as in Meskar *et al.*, 2017). We assume that many cloud servers are always available to execute the offloaded tasks, so the queuing delay that the user's tasks spend waiting for a cloud server is zero.

For the mobile device, the local power consumption, $P_L$, the idle mode power

consumption, $P_I$, and the transmission power consumption, $P_T$, are set to 8 mW, 0.2 mW and 0.5 W, respectively (as in Song *et al.*, 2014). With these parameters, a mobile device uses significantly more power to execute a task locally, compared to transmitting the task to the cloud and waiting in idle mode.

The $C_i$ parameter for each task $i$ is now a large number chosen randomly between 2.5 to 25 Gcycles, to reflect complex tasks. The task graph that is used in this section is similar to Figure 3.4 with $S' = 20$ and workload of [21.44, 21.24, 8.26, 16.30, 15.60, 14.66, 22.07, 8.45, 9.65, 5.18, 23.64, 17.02, 13.28, 16.88, 14.75, 17.06, 14.73, 18.72, 14.25, 24.85] Giga CPU cycles.

For example, as described in Section 1.3, a chess game task may have a very small size, but it can express a very large amount of computational work. The input and output data sizes are [ 6.25, 1.85, 4.5, 7.39, 8.49, 5.92, 9.6, 9.28, 4.05, 6.58, 0.928, 3.8, 4.15, 0.706, 8.11, 8.22, 6.36, 5.54, 2.86, 2.66] Mbits and [ 1.08, 0.538, 0.776, 1.06, 1.6, 0.765, 0.898, 0.836, 0.631, 0.771, 1.58, 1.49, 1.44, 1.87, 1.62, 1.07, 1.36, 0.913, 1.18, 1.71] Mbits, respectively. In our simulations, we calculate the energy consumption of the offloading decisions made by each algorithm as the data rate when accessing the network ranges from 15 to 60 Mbps. Based on the model of Kumar and Lu (2010), the local energy consumption and local execution time are given by following:

$$El_i = P_L \; C_i / f_L \tag{5.13}$$

$$Tl_i = C_i / f_L \tag{5.14}$$

As explained in "www.cloudtestfiles.net", one can ping and traceroute cloud servers around the world to see the expected round trip access delays for specific servers. Alternatively, the website "www.internettrafficreport.com" reports the average round

trip Internet delays in each continent, in real-time. In North America, the average delay is 27 ms, in Europe the average delay is 67 ms, and in Asia the average delay is 129 ms. We also assume a delay of 90 ms to access a distant cloud server, and a delay of 10 milliseconds to access a nearby edge server located in the same city.

In order to consider the security issue when offloading tasks to remote servers, we assume an extra energy consumption in the mobile device equal to 10% of the offloading energy consumption, to encrypt a task before it is sent to the cloud. The total transmission time includes the transmission time to offload task $i$ ($Tt_i$) to the remote server plus the network delay ($T_{NET}$). The transmission energy used in the mobile device to offload a task depends only on $Tt_i$, the time taken to transmit the task.

When the tasks are offloaded to the cloud, the mobile device waits for the cloud server to complete the execution of the task, and meanwhile the mobile device switches to the idle mode. We assume that the cloud server starts executing a task without any delay, so we have $TI_i = C_i/f_C$.

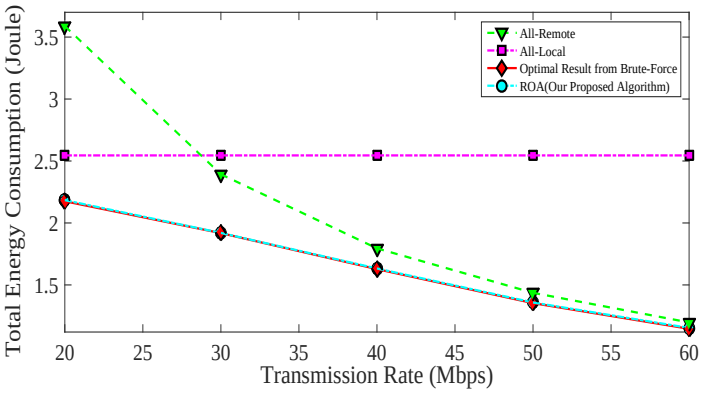


Figure 5.1: Comparison of Total Energy Consumption of the Application in the All-Remote, All-Local, Brute-Force and ROA-V1 Methods for a Cloud Server

Figure 5.1 shows the total energy consumption of the mobile device based on

Eq. (5.5) versus the network transmission rate, assuming a distant cloud server with an access delay of 90 milliseconds. We plot the results for several different algorithms, All-Remote, All-Local, the Brute-Force Search optimal result, and the results of the ROA-V1 algorithm. In the ROA-V1 algorithm, we allow for 25 iterations with $\rho = 0.5$, and we report the average energy used, averaged over 20 trial runs. For each trial run, the same parameters (i.e., input data size, output data size) are used, but each run generates different candidate vectors and hence may yield a different solution. The results of the ROA-V1 algorithm and the Brute-Force Search algorithm overlap on Figure 5.1, and are indistinguishable. Hence, the ROA-V1 algorithm yields excellent energy results for the chosen task graph, comparable to the Brute-Force Search optimum. Table 5.5 shows the exact numbers for the total energy consumption of the ROA-V1 algorithm, versus the optimal results. The average run time of the ROA-V1 algorithm is about 39 ms.

The ROA-V1 algorithm executes very quickly, and can find good offloading decisions with low computation effort for the chosen task graph. Considering that parallel task graph, if we increase the number of iterations to 100, the ROA-V1 algorithm can match the optimal result from Brute-Force-Search more than 99% of the time.

Table 5.5: Comparison of Total Energy Consumption (in Joules) in the Optimal and ROA-V1 Methods.

| **Method / Rate (in Mbps)** | **20** | **30** | **40** | **50** | **60** |
|---|---|---|---|---|---|
| **Total Energy Consumption from Brute-Force** | 2.1752 | 1.9181 | 1.6276 | 1.3532 | 1.1461 |
| **Total Energy Consumption from ROA-V1** | 2.18431 | 1.9198 | 1.63277 | 1.3585 | 1.15296 |

## 5.7 ROA-V1 Simulation Results for the General Model

In this section, we explore the performance of the ROA-V1 algorithm on a class of general task graphs. The class of task graphs that is considered is that shown in Figure 3.6 with $S = 9$ tasks. One hundred variations of that seed graph were generated as explained in Section 3.2.1. Since we are exploring the general task graph in this section, the total number of tasks, $S$, in an application is considered rather than the set of offloadable tasks $S'$. The system model and problem formulation were explained in Section 5.2.

The results for ROA-V1 when the number of iterations, $Itr$, ranges from 20 to 100 with different selections of the $\rho$ parameter are provided in Table 5.6. It is clear from Table 5.6 that, although the task graph is not very large, ROA-V1 was only able to find results within $\Delta = 2.9\%$ compared to the optimal Brute-Force search. Due to the reduced effectiveness of ROA-V1 for the general model, we explored new modifications to improve its performance. As a result, ROA-V2 is proposed in the next section.

## 5.8 The ROA-V2 Algorithm

In this section, the ROA-V2 algorithm is described. ROA-V2 is designed to achieve better performance on general task graphs and to be more computationally efficient than ROA-V1. Some of the ideas that underlie ROA-V2, an early version of the pseudocode and some related results were stated in Szymanski and Shahzad (2018) and were included in Shahzad and Szymanski (2018). In this section we provide an

Table 5.6: Total energy consumption for ROA-V1 (in Joules) for the general model with the task graph in Figure 3.6

| ***Itr*** | | **ROA** $\boldsymbol{\rho = 0.5}$ | **ROA** $\boldsymbol{\rho = 0.75}$ | **Optimal Energy** |
|---|---|---|---|---|
| **20** | **Mean Energy** | 0.1986 | 0.1978 | |
| | **Δ%** | 6.8 % | 6.4% | 0.1859 |
| **50** | **Mean Energy** | 0.1972 | 0.1966 | |
| | **Δ%** | 6 % | 5.75 % | 0.1859 |
| **100** | **Mean Energy** | 0.1944 | 0.1912 | |
| | **Δ%** | 4.57 % | 2.9 % | 0.1859 |

enriched description of ROA-V2, a refined version of the pseudocode and the results of other numerical experiments. Furthermore, in Sections 5.8.3 and 5.8.4 we describe preliminary versions of ROA-V2 (denoted ROA-V2.1 and ROA-V2.2, respectively) that provide insights into the rationale behind ROA-V2.

The basic ideas behind ROA-V2 are as follows: The algorithm increases the benefits of randomization by using two nested loops so that the inner loop can restart from a newly generated initial solution. In this case, instead of improving on one initial solution throughout the iterations, in each execution of the outer loop, we explore a new random initial solution and therefore a higher number of random binary solutions are investigated. The initial solutions will be improved through inner loop iterations to find the best final result.

Algorithm 5.3 provides a pseudocode description for ROA-V2, that highlights the outer and inner execution loops. The outer loop generates $P$ initial decision vectors (lines 3-29) and the inner loop tries to iteratively improve the initial decision vector using a process based on the ROA-V1 algorithm (lines 5-26). An initial decision vector, $M$, is randomly generated for each element $p$ in $P$, which is a bit-string of

length $S$ (in line 4). In each iteration of the inner loop, a new candidate vector, $c$, of length $S$, is obtained (lines 8-13). Vector $c$ is either a random binary vector, or is obtained by a mutation step, as in conventional genetic optimization. In the mutation case, a prior decision vector, $v_2$, from $p \in P$, is used with a random vector $v_1$, to create the new $c$ vector (lines 8-11). A random permutation of length $S$ is generated in each mutation step. Set $I$ specifies the bit-locations where bits are copied from the random vector $v_1$ to $c$. For the rest of the bit-locations, bits are copied from the prior decision vector $v_2$ to $c$. This operation is performed in line 11 by the DoubleMerge function.

The inner loop in Algorithm 5.3 iteratively improves one initial decision vector from the set $P$ during each pass. For every bit location where $M$ and $c$ disagree (recorded in $CP$), a sub-string will be made from the last uncommon bit ($l$) to the next ($u$). The energy consumption of the sub-strings in $c$ and $M$ is calculated in lines 18-19. In line 20, if the total execution time using the new sub-string does not violate the time constraint ($T_C$) and the energy consumption of the new sub-string is better, the sub-string will be incorporated into the final solution in line 22.

The matrix $PD$ is used to store the $P$ optimized decision vectors (line 27). Once all the $P$ decision vectors have been updated, the minimum energy vector $M$ is selected and returned (lines 30-31).

The mutation step, which is often used in conventional genetic optimization, is included to see if using previously optimized decision vectors offer any practical improvements. In our experiments, we have found that for the task graphs that we have tested, the mutation step gives no practical benefit. Based on these observations, this step can be removed without affecting the quality of the final solution.

**Algorithm 5.3:** ROA-V2 Algorithm Pseudocode

Initialize time constraint ($T_C$) and Transmission Rate
Initialize $\gamma$ as the incorporating threshold
**for** $p = 1$ **to** $P$ **do**
    $M$ = random binary vector of length $S$
    **repeat** $R'$ **times**
        $\gamma_r$ = uniform(0, 1) // random number between 0 and 1
        // $P1$ controls number of iterations we want to complete before incorporating the previous solutions
        **if** $p > P1 \wedge \gamma_r < \gamma$ **then**
            $v_1$ = random binary vector of length $S$
            // $PD$ is a matrix of binary vector solutions (initially empty)
            // $PD$ is a $P \times S$ matrix of prior solutions from inner loop
            $v_2 = PD(\text{uniformInt}(1, p), :)$ // randomly selected (row) vector from $PD$
            $I$ = set of randomly selected indices of $v_1$
            // DoubleMerge takes $|I|$ bits from $v_1$ and remaining bits from $v_2$
            $c = \text{DoubleMerge}(v_1, v_2, I)$
        **else**
            $c$ = random binary vector of length $S$
        **end**
        $CP = (\, i \mid M(i) \neq c(i), \forall i = \{1, 2, \cdots, S\})$ // ordered set of indices where $M$ and $c$ disagree
        $l = 1$
        **for** $u \in CP$ **do**
            // $u$ is upper task fragment index
            $E^c = \sum_{i=l}^{u} ES(i, c(i))$ // compute total task energy using $c$
            $E^M = \sum_{i=l}^{u} ES(i, M(i))$, // compute total task energy using $M$
            // $TS_i$ is the time consumption of task $i$ using the decision bit for task $i$
            $T_{total} = \sum_{i=1}^{l-1} TS(i, M(i)) + \sum_{i=u+1}^{S} TS(i, M(i)) + \sum_{i=l}^{u} TS(i, c(i))$
            **if** $E^c < E^M \wedge T_{total} < T_C$ **then**
                $M(i) = c(i)\ \forall i \in \{l, l+1, \ldots, u\}$ // $M$ adopts this fragment of $c$
            **end**
            $l = u + 1$
        **end**
    **end**
    $PD(p, :) = M$
    Compute $E_{total}$ and $T_{total}$ given $M$, using Eq. (5.9) and Eq. (5.10)
**end**
Find the vector $M$ in the population(1 to $P$) that minimizes $E_{total}$
Return $E_{total}$, $T_{total}$ and decision vector $M$

### 5.8.1 ROA-V2 Simulation Results for the General Model

This section considers ROA-V2 algorithm performance for the delay-constrained energy minimization problem, which was introduced in Section 5.2. MATLAB was used to implement the algorithm, and its performance was assessed using three general task graph classes, from Section 3.1.2. Table 5.7 summarizes the model parameters that were used in the experiments. They specify various mobile device characteristics such as processing, computation and idle mode power levels, and remote server parameters such as computation power.

Table 5.7: Simulation Parameters

| **Parameters** | **Definition** | **Model 1** | **Model 2** | **Model 3** |
|---|---|---|---|---|
| | Topology | Figure 3.6 | Figure 3.7 | Figure 3.7 |
| $S$ | Number of tasks per application | 9 | 23 | 23 |
| $f_L$ | Local computation power | 500 Gcycle/sec | 1 Gcycle/sec | 1 Gcycle/sec |
| $f_C$ | Cloud server computation power | N/A | 50 Gcycle/sec | N/A |
| $f_E$ | Edge server computation power | 5 Gcycle/sec | N/A | 50 Gcycle/sec |
| $P_L$ | Local power consumption | 0.5W | 1W | 1W |
| $P_I$ | Idle mode power consumption | 1 mW | 100 mW | 100 mW |
| $PT_s$ | Sending power consumption | 50 mW | 200 mW | 200 mW |
| $PT_r$ | Receiving power consumption | 20 mW | 200 mW | 200 mW |
| $R_s$ | Wireless Sending bit rate | 2 Mbps | 5 Mbps | 5 Mbps |
| $PT_r$ | Wireless Receiving bit rate | 2 Mbps | 5 Mbps | 5 Mbps |
| $C_i$ | Computation complexity of task $i$ | 0.3 - 370 Mcycle | 0.009 - 1415 Mcycle | 0.009 - 1415 Mcycle |
| $T_{NET}$ | Expected 1-way network delay | 0 | 20 ms | 5 ms |

**Model 1 ($S = 9$)**

This model evaluates the system performance using the three task graphs (shown in Figure 3.6) that were used in Deng *et al.* (2016). Reference Deng *et al.* (2016) considered a system where edge servers were placed in a femto-cellular network, of the type proposed in the TROPIC EU project (http://www.ict-tropic.eu).

In this section, it is assumed that the edge servers can be accessed with zero latency (i.e., $T_{NET} = 0$). The task graphs in Figure 3.6 use the same topology, with $S = 9$ tasks, $S' = 6$ offloadable tasks, and 10 edges. However, the graphs have differing edge weights that represent different input and output data sizes. The system parameters for Model 1, which are listed in Table 5.7, are identical to those used by Deng *et al.* (2016). A comparison of the results of the algorithm of Deng *et al.* (2016) and our algorithm is presented in Table 5.8 for the 3 task graphs in Figure 3.6 and the results are presented as "Graph 1", "Graph 2" and "Graph 3" in the Table 5.8.

In order to conduct a richer experiment, 100 pseudo-random task graphs were used for the Model 1 topology from Figure 3.6. The three task graphs shown in Figure 3.6, which were presented in Deng *et al.* (2016), are part of this set. The method described in Section 3.2.1 is used to generate the remaining pseudo-random task graphs from the initial "seed graph" shown in Figure 3.6. The average energy performance over the generated task graphs is included in Table 5.8 as the row labelled "100 Graphs".

In Table 5.8, the total number of iterations in Algorithm 5.3, i.e., $P \times R'$, is represented with the symbol $Itr$. In this experiment, we explored the effect of changing $Itr$ in the range between 100 and 400 (i.e., $Itr = \{100, 200, 300, 400\}$). The number of iterations for the inner loop is, $R' = 10$, so that the number of outer loop iterations correspond to $P = \{10, 20, 30, 40\}$. The column labeled "Optimal BFS Energy"

Table 5.8: Comparison of Total Energy Consumption (in millijoules) for ROA-V2 and the algorithm of Deng *et al.* (2016) for Model 1

| | **Optimal BFS Energy** | **Results of Deng *et al.* (2016)** | | **ROA *Itr* = 100** | **ROA *Itr* = 200** | **ROA *Itr* = 300** | **ROA *Itr* = 400** |
|---|---|---|---|---|---|---|---|
| **Graph 1** | 118.3 | 118.3 | **Mean Energy** | 118.3 | 118.3 | 118.3 | 118.3 |
| | | | **Δ%** | 0 % | 0 % | 0 % | 0 % |
| **Graph 2** | 118.5 | 118.5 | **Mean Energy** | 118.5 | 118.5 | 118.5 | 118.5 |
| | | | **Δ%** | 0 % | 0 % | 0 % | 0 % |
| **Graph 3** | 176.2 | 176.2 | **Mean Energy** | 176.2 | 176.2 | 176.2 | 176.2 |
| | | | **Δ%** | 0 % | 0 % | 0 % | 0 % |
| **100 Graphs** | 185.9 | N/A | **Mean Energy** | 185.9 | 185.9 | 185.9 | 185.9 |
| | | | **Δ%** | 0 % | 0 % | 0 % | 0 % |

shows the minimum BFS energy. The columns labeled "ROA" show the minimum energy of the ROA-V2 algorithm for different numbers of iterations.

The symbol "Δ" gives the difference between the mean ROA-V2 energy and the optimal BFS energy, expressed as a percentage of the latter. For task graphs 1-3, ROA-V2 finds the optimal BFS energy solution for total iterations of $Itr = 100$.

All the results of ROA-V2 in Table 5.8 are averaged over five runs of the algorithm. It can be seen that ROA-V2 finds excellent quality solutions within one hundred iterations. In order to investigate the energy result tends as $Itr$ is increased, we conducted another experiment when the total number of iterations changes between 10 and 100. The results of this experiment over the set of 100 pseudo-random task graphs are presented in Table 5.9. ROA-V2 finds an energy value that is within 0.48% of the optimal result from BFS search, when $Itr = 50$.

Table 5.9: Comparison of Total Energy Consumption (in millijoules) for ROA-V2 and the algorithm of Deng *et al.* (2016)

| | **Optimal BFS Energy** | **Results of Deng *et al.* (2016)** | | **ROA *Itr* = 10** | **ROA *Itr* = 20** | **ROA *Itr* = 50** | **ROA *Itr* = 100** |
|---|---|---|---|---|---|---|---|
| **100 Graphs** | 185.9 | N/A | **Mean Energy** | 200 | 192 | 186.8 | 185.9 |
| | | | Δ% | 7.5 % | 3.17 % | 0.48 % | 0 % |

### Model 2 - Distant Cloud Servers ($S$=23)

Model 2 is used to investigate the performance of ROA-V2 using the larger and more complex task graph given in Zhang *et al.* (2012), as shown in Figure 3.7. The parameters were changed to consider more powerful distant cloud servers and newer mobile devices. These are summarized in Table 5.7. In this case the cloud server is set to be 50 times more powerful than the mobile device. Also, the mobile device power consumption is higher and the work per task is larger. The average 2-way delays (response time) in each continent is reported in the web-site "www.internettrafficreport.com". For North America, Europe and Asia, the average 2-way delays are roughly 21 ms, 69 ms, and 51 ms, respectively. An expected 1-way network delay of 20 msec is used for the cloud server in Model 2.

In Figure 3.7, non-offloadable tasks are shown as the blue nodes, and indicate those that must be executed locally on the mobile device. Similarly, offloadable tasks are shown as the yellow nodes. The edge weightings give the amount of data (in kilobytes) that must be transferred.

The performance of the ROA-V2 algorithm was evaluated using 100 pseudo-random task graphs that were generated from the Model 2 seed graph, which is

shown in Figure 3.7. This was done using the technique described in Section 3.2.1. In Table 5.10, the rows labelled "Graph 1", "Graph 2" and "Graph 3" correspond to three randomly selected task graphs among the aforementioned 100 pseudo-randomly generated task graphs.

The ROA-V2 and BFS energy results for Model 2 are shown in Table 5.10. This table can be interpreted in a manner similar to that of Table 5.8 for Model 1. The mean per task graph energy for All-local execution is 732.91 mJ, and is 483.06 mJ for BFS, over the set of 100 task graphs. Therefore, when averaged over these 100 task graphs, computation offloading provides an energy savings of about 34% compared to All-local execution. For these task graphs, columns $Itr = \{100, 200, 300\}$ show ROA-V2 energies of 547.5, 510.06, and 497.04 mJ, respectively. ROA-V2 gives corresponding $\Delta$ values of 13.3%, 5.5% and 2.8%. These results were obtained by averaging over 5 runs.

**Model 3 - Proximate Edge Servers ($S = 23$)**

The effects of more powerful edge servers are considered in this section. It is assumed that these servers are accessible with a 1-way delay of 5 milliseconds, rather than the 20 ms that was used in Model 2. The same set of 100 task graphs, including the same three randomly selected task graphs as in Model 2, are used in the experiments. The model parameters are given in Table 5.7.

Table 5.11 shows the average energy results for the 100 pseudo-randomly chosen task graphs. Over the entire set, the mean per task graph energy for All-Local execution is 732.91 mJ and the mean BFS energy is 477.1 mJ. The BFS values are slightly lower than those in Model 2 since the edge servers are accessible with the

Table 5.10: Comparison of Total Energy Consumption (in millijoules) for ROA-V2 under Model 2

| | **All-Local Energy** | **Optimal BFS Energy** | | **ROA** $Itr = 100$ | **ROA** $Itr = 200$ | **ROA** $Itr = 300$ |
|---|---|---|---|---|---|---|
| $R' = 10$ | | | | | | |
| **Graph 1** | 888.7 | 481.7 | **Mean Energy** | 537.16 | 516.37 | 484.02 |
| | | | $\Delta\%$ | 11.5 % | 7.1 % | 0.48 % |
| **Graph 2** | 449.4 | 244.7 | **Mean Energy** | 276.7 | 264.74 | 250.8 |
| | | | $\Delta\%$ | 13 % | 8.1 % | 2.4 % |
| **Graph 3** | 774.8 | 589.3 | **Mean Energy** | 624.5 | 613 | 590.65 |
| | | | $\Delta\%$ | 5.9 % | 4 % | 0.2 % |
| **10 Graphs** | 794.1 | 553.23 | **Mean Energy** | 608.6 | 576.7 | 567.49 |
| | | | $\Delta\%$ | 10 % | 4.24 % | 2.5 % |
| **100 Graphs** | 732.9 | 483.06 | **Mean Energy** | 547.5 | 510.06 | 497.04 |
| | | | $\Delta\%$ | 13.3 % | 5.5 % | 2.8 % |

lower (5 ms) delay.

The Table 5.11 columns given by $Itr = \{100, 200, 300\}$ correspond to population sizes of $P = \{10, 20, 30\}$, where $R' = 10$ iterations per population member were used. The ROA-V2 algorithm gives energy values of 522, 503.3 and 489.3mJ, respectively and with corresponding $\Delta$ values of 9.6%, 5.72% and 2.55%. These are based on 5 run averages. Table 5.11 shows that for the chosen set of task graphs ROA-V2 is able to find good quality solutions that are within about 2.5% of the optimal BFS energy solutions using only a few hundred iterations.

Table 5.11: Comparison of Total Energy Consumption (in millijoules) for ROA-V2 using Model 3

| | **All-Local Energy** | **Optimal BFS Energy** | | **ROA *Itr* = 100** | **ROA *Itr* = 200** | **ROA *Itr* = 300** |
|---|---|---|---|---|---|---|
| | | | $R' = 10$ | | | |
| **Graph 1** | 888.7 | 475.75 | **Mean Energy** | 516.9 | 511.6 | 503.5 |
| | | | **$\Delta$%** | 8.64 % | 7.53 % | 5.83 % |
| **Graph 2** | 449.4 | 235.7 | **Mean Energy** | 235.7 | 235.7 | 235.7 |
| | | | **$\Delta$%** | 0 % | 0 % | 0 % |
| **Graph 3** | 774.8 | 583.3 | **Mean Energy** | 612.8 | 610.5 | 603.25 |
| | | | **$\Delta$%** | 5.05 % | 4.66 % | 3.4 % |
| **100 Graphs** | 732.9 | 477.1 | **Mean Energy** | 522 | 503.3 | 489.3 |
| | | | **$\Delta$%** | 9.6 % | 5.72 % | 2.55 % |

### 5.8.2 Time Complexity of ROA-V2

The time complexity of the ROA-V2 algorithm is determined by the energy and time evaluation of the sub-strings using equations (5.9) and (5.10), the bit string population size, the number of task graph tasks and the number of iterations used.

**Lines 1-5:** In the initializing step, there are two nested loops ($p$ and $R'$) and generating the first random population has a time complexity of $O(S)$.

**Lines 6-13 (Mutation Step):** In this case $w_1$ bits from vector $v_1$ and $w_2$ bits from vector $v_2$ are merged. Since these two vectors are being merged, the time complexity is therefore $O(w_1 + w_2)$, which is $O(S)$.

**Lines 17-25:** Loop $u$ has a time complexity of $O(S)$ and lines 18, 19 and 20 are $O(S + E)$, where $E$ is the number of task graph relations. Lines 21 and 22 have a time complexity of $O(1)$ and that for lines 13-22 is O($S^2$). The evaluation of energy and time expressions for two nested loops, i.e., $p$ and $R'$, has a time complexity of $O(P \cdot R' \cdot S^2)$ where $P$ is the number of outer loop iterations, $R'$ is the total inner loop iterations and $S^2$ is the time complexity of the lines 17-25 loop to evaluate the energy and time equations and sub-string incorporation for the current solution vector.

**Lines 27-31:** The time complexity of line 27 is $O(1)$ needed to save the results and in line 30, the minimum of $P$ elements is selected. The latter time complexity is therefore $O(P)$. Returning the final results has a complexity of $O(1)$.

The overall time complexity for the ROA-V2 algorithm is the worst time among all of the above steps. Therefore, the time complexity of ROA-V2 is given by $O(P \cdot R' \cdot S^2)$.

#### Time Complexity Comparison with Genetic Algorithm

Based on the work of Tseng *et al.* (2018), the time complexity of GA is defined by the selection process, fitness function evaluation, crossover and mutation operations alongside generation number and population size, that is $O(Iteration \times Population\ Size \times O(Fitness) \times (O(Selection) + O(Crossover) + O(Mutation)))$.

The time complexity of the genetic-based offloading algorithm presented in Tout *et al.* (2017) is $O(\lambda NH)$ for independent tasks where $\lambda$ is the number of generations, $N$ is the population size and $H$ is the individual size. For general task graphs, this time complexity is $O(\lambda NH^2)$. Comparing the result for ROA-V2 for a general task graph, ROA-V2 has an $O(N)$ factor lower time complexity.

### 5.8.3 Preliminary Version of ROA-V2: ROA-V2.1 Algorithm

In this and the following section, preliminary versions of the ROA-V2 algorithm are described. These were provided in Section 5.8. An algorithm called ROA-V2.1 is described. In this algorithm, similar to ROA-V1, $\rho$ is used to determine the number of sub-string partitions of the candidate vector. Compared to ROA-V1, instead of using one loop for generating the random vectors, two nested loops are used (lines 3, 5 in ROA-V2), which is the main idea behind ROA-V2 (Szymanski and Shahzad, 2018). This strategy helps to restart the initial random vectors rather than focusing on improving one initial random vector using multiple iterations.

#### Simulation Results

The performance of the ROA-V2.1 algorithm is investigated for a class of general task graphs using the delay-constrained energy minimization problem that was presented

in Section 5.2. MATLAB was used to program the ROA-V2.1 algorithm. Table 5.12 gives the set of parameters used to explore the effects of dependency between the tasks.

Table 5.12: Simulation Parameters

| **Parameters** | **Definition** | **(Deng *et al.*, 2016)** |
|---|---|---|
| $f_L$ | Local computation rate | 500 Mcycle/sec |
| $f_C$ | Cloud server computation rate | 5 Gcycle/sec |
| $P_L$ | Local power consumption | 0.5 w |
| $P_I$ | Idle mode power consumption | 1 mw |
| $PT_s$ | Sending power consumption | 50 mw |
| $PT_r$ | Receiving power consumption | 20 mw |
| $C_i$ | Computation complexity of the task $i$ | 0.3-370 Mcycle |
| $T_{NET}$ | 1-way network delay | 0 |
| $S$ | Number of tasks in an application | 9 |

Results obtained from different offloading algorithms were compared, including ROA-V2.1 and Brute-Force Search. As before, the algorithms were implemented in MATLAB. In the first experiment, the task graph was that taken from Deng *et al.* (2016) and is shown in Figure 3.6. Three examples with different computational workload and exchange data sizes were provided in Deng *et al.* (2016), as shown in Figure 3.6.

Table 5.13 summarizes the comparison of total energy consumption using these methods. For ROA-V2.1, results are provided for several values of the $\rho$ parameter. As before, the parameter "$Itr$" gives the number of iterations and "$Mean$" indicates

Table 5.13: Comparison of Total Energy Consumption (in Joules) for $\rho = 0.5$, 0.75, 1 based on the task graph topology in Figure 3.6

| **$\rho = 0.5$** | | | | | | |
|---|---|---|---|---|---|---|
| | | **Optimal Energy** | **ROA-V2.1 $Itr = 10$** | **ROA-V2.1 $Itr = 20$** | **ROA-V2.1 $Itr = 50$** | **ROA-V2.1 $Itr = 100$** |
| **Graph 1** | **Mean(E)** | | 0.1259 | 0.1183 | 0.1183 | 0.1183 |
| | **Δ%** | 0.1183 | 6.4 % | 0 % | 0 % | 0 % |
| **Graph 2** | **Mean(E)** | | 0.1261 | 0.1185 | 0.1185 | 0.1185 |
| | **Δ%** | 0.1185 | 6.41% | 0 % | 0 | 0% |
| **Graph 3** | **Mean(E)** | | 0.1727 | 0.1726 | 0.1726 | 0.1726 |
| | **Δ%** | 0.1726 | 0.05 % | 0 % | 0 % | 0 % |
| **100 Graphs** | **Mean(E)** | | 0.2 | 0.1946 | 0.1878 | 0.1860 |
| | **Δ%** | 0.1859 | 7.5 % | 4.67 % | 1.02 % | 0.05 % |
| **$\rho = 0.75$** | | | | | | |
| | | **Optimal Energy** | **ROA-V2.1 $Itr = 10$** | **ROA-V2.1 $Itr = 20$** | **ROA-V2.1 $Itr = 50$** | **ROA-V2.1 $Itr = 100$** |
| **Graph 1** | **Mean(E)** | | 0.1185 | 0.1183 | 0.1183 | 0.1183 |
| | **Δ%** | 0.1183 | 0.1 % | 0 % | 0 % | 0 % |
| **Graph 2** | **Mean(E)** | | 0.1185 | 0.1185 | 0.1185 | 0.1185 |
| | **Δ%** | 0.1185 | 0 % | 0 % | 0 | 0% |
| **Graph 3** | **Mean(E)** | | 0.1726 | 0.1726 | 0.1726 | 0.1726 |
| | **Δ%** | 0.1726 | 0 % | 0 % | 0 % | 0 % |
| **100 Graphs** | **Mean(E)** | | 0.2 | 0.1927 | 0.1874 | 0.1860 |
| | **Δ%** | 0.1859 | 7.5 % | 3.6 % | 0.8 % | 0.05 % |
| **$\rho = 1$** | | | | | | |
| | | **Optimal Energy** | **ROA-V1.2 $Itr = 10$** | **ROA-V2.1 $Itr = 20$** | **ROA-V2.1 $Itr = 50$** | **ROA-V2.1 $Itr = 100$** |
| **100 Graphs** | **Mean(E)** | | 0.2 | 0.1920 | 0.1868 | 0.1859 |
| | **Δ%** | 0.1859 | 7.5 % | 3.17 % | 0.48 % | 0 % |

the average energy used over 10 algorithm runs. The value "$\Delta$" gives the percentage ratio between the $Mean$ and the optimal result obtained from Brute-Force Search. The simulation parameters that were used are identical to Deng *et al.* (2016), and Graph 1, Graph 2 and Graph 3 are the same as those from the same reference. The row marked *100 Graphs* gives the results averaged over the 100 generated task graphs using he same seed graph but with randomly selected data sizes, computation workload and non-offloadable tasks. This is explained in detail in Section 3.2.1.

In ROA-V2.1 with $\rho = 1$, the algorithm will perform more work when evaluating a candidate vector in each iteration. As can be seen in Table 5.13, setting $\rho$ to this value results in the best performance for a fixed number of iterations, but it also uses the most computation.

The results shown in Table 5.13 suggest that using a smaller value of $\rho$ and higher numbers of iterations is preferable. This allows for more candidate solutions to be evaluated, which in general improves the ROA-V2.1 algorithm performance. Therefore, from a computational time perspective, it is better to use a smaller value of $\rho$ and allow for more iterations.

A larger task graph is considered in the second experiment. The task graph in Figure 3.7 with $S = 23$ was considered in Model 2. Table 5.14 shows the results of 10 pseudo-random task graphs from the seed graph in Figure 3.7. It is clear that even with 500 total iterations ($Itr = 500$), $\Delta = 5.22\%$ is the best result that is obtained. For this reason, another preliminary version was investigated, which is presented as the ROA-V2.2 algorithm.

Table 5.14: Comparison of Total Energy Consumption (in Joules) for $\rho = 0.75$ based on the task graph in Figure 3.7

| $\boldsymbol{\rho = 0.75}$ | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | **Optimal Energy** | **ROA-V2.1** $\boldsymbol{Itr = 100}$ | **ROA-V2.1** $\boldsymbol{Itr = 200}$ | **ROA-V2.1** $\boldsymbol{Itr = 300}$ | **ROA-V2.1** $\boldsymbol{Itr = 400}$ | **ROA-V2.1** $\boldsymbol{Itr = 500}$ |
| **10 Graphs** | **Mean(E)** | | 640.55 | 622.11 | 611.43 | 588.23 | 582.11 |
| | $\Delta\%$ | 553.23 | 15.7 % | 12.4 % | 10.5 % | 6.3 % | 5.22 % |

### 5.8.4 Preliminary Version of ROA-V2: ROA-V2.2 Algorithm

The main difference between ROA-V2.2 and the two previous versions (ROA-V1 and ROA-V2.1) is that the use of $\rho$ as a threshold is eliminated. Instead, ROA-V2.2 selects one of the results from the inner loop (line 9 in ROA-V2), which is the main idea behind ROA-V2 (Szymanski and Shahzad, 2018). An XOR operation is performed on this and the new candidate vector in order to determine the sub-string cut points, i.e., cut points are at the positions of having different bits between a previous result and a newly generated vector. This method helps to reduce the algorithm randomness and is better at incorporating information from previous iteration results.

#### Simulation Results

The performance of ROA-V2.2 is investigated using a class of general task graphs for the delay-constrained energy minimization problem presented in Section 5.2. The proposed algorithm was programmed using MATLAB. Table 5.12 shows the parameters used in this section to explore the effects of data exchange between tasks. The results in Table 5.15 indicated that when $Itr = 20$, ROA-V2.2 finds the optimal results for the three graph versions of Figure 3.6. Using the 100 randomly generated

graphs from the seed graph in Figure 3.6, ROA-V2.2 provides better results compared to ROA-V2.1 for $\rho = 1$ as shown in Table 5.13 in $Itr = [10, 20, 50]$. When $Itr = 100$, both ROA-V2.1 and ROA-V2.2 find the optimal results.

Table 5.15: Comparison of Total Energy Consumption (in Joules) for ROA-V2.2 based on the task graph in Figure 3.6

| $R' = 10$ | | | | | | |
|---|---|---|---|---|---|---|
| | | **Optimal Energy** | **ROA-V2.2 $Itr = 10$** | **ROA-V2.2 $Itr = 20$** | **ROA-V2.2 $Itr = 50$** | **ROA-V2.2 $Itr = 100$** |
| **Graph 1** | **Mean Energy** | 0.1183 | 0.1185 | 0.1183 | 0.1183 | 0.1183 |
| | **$\Delta$%** | | 0.1 % | 0 % | 0 % | 0 % |
| **Graph 2** | **Mean Energy** | 0.1185 | 0.1185 | 0.1185 | 0.1185 | 0.1185 |
| | **$\Delta$%** | | 0 % | 0 % | 0 | 0% |
| **Graph 3** | **Mean Energy** | 0.1726 | 0.1726 | 0.1726 | 0.1726 | 0.1726 |
| | **$\Delta$%** | | 0 % | 0 % | 0 % | 0 % |
| **100 Graphs** | **Mean Energy** | 0.1859 | 0.1918 | 0.1896 | 0.1866 | 0.1859 |
| | **$\Delta$%** | | 3.17 % | 1.99 % | 0.37 % | 0 % |

## 5.9 Summary

In this chapter, different versions of an offloading algorithm called the *Randomized Offloading Algorithm* were proposed to address the computational offloading optimization problem. ROA finds a decision vector of length $S$, where each bit denotes a decision to execute the task locally or remotely. The decision vector is improved iteratively. The results of different versions of ROA for both parallel and general models were presented and they show that ROA can find good quality solutions for classes of both models.

# Chapter 6

# Context-Aware Randomized Offloading Algorithm (CA-ROA)

In this chapter, a new version of the ROA algorithm is proposed, which is called the Context-Aware Randomized Offloading Algorithm (CA-ROA). This algorithm is also based on randomization and tries to incorporate better sub-strings from the candidate vector into the decision vector. The distinction between this algorithm and other versions of ROA (ROA-V1 and ROA-V2) is that it uses task graph information to both generate biased random vectors and determine the number of sub-strings the candidate vector will be partitioned into.

## 6.1 Problem Formulations for General Model

The model parameters that are used in the problem formulation are summarized in Table 6.1. As before, for task $i$, let $M_i \in \{0, 1\}$ be an execution indicator variable, i.e., $M_i = 1$ if task $i$ is executed at the mobile device and 0 otherwise. If it is executed

locally, the energy consumption is $El_i$ and the execution time is $Tl_i$. Similarly, if the task is executed in the remote server and the mobile device is in an idle mode for $TI_i$ seconds, the energy consumption of the device is $EI_i$. $Et_{ij}$ is the energy needed to transmit the necessary data for task $j$ from task $i$ to the remote server or from the remote server to the mobile device and $Tt_{ij}$ is the corresponding time. Network delay is given by $T_{NET}$.

Table 6.1: Description of the Model Parameters

| **Symbols** | **Meaning** |
|---|---|
| $El_i/Tl_i$ | Energy / Time taken to execute task $i$ locally |
| $EI_i/TI_i$ | Energy / Time consumed in mobile device when task $i$ is executing remotely |
| $Et_{ij}/Tt_{ij}$ | Energy / Time taken to transfer input data of task $j$ from task $i$ from/to the remote server |
| $T_{NET}$ | 1-way delay to access the remote server |

The optimization problem that we want to solve is to minimize the energy consumption of the mobile device while constraining the total execution time. The problem can be formulated as

$$\begin{aligned} \min_{\{M_i\}} \quad & E \\ \text{s.t.}: \quad & T \leqslant T_C \\ & M_i \in \{0,1\},\ \forall i \in \{1,2,\ldots,S\} \end{aligned} \tag{6.1}$$

The total energy consumption of the mobile device that we want to minimize is defined as follows, where $EM$ is the edge matrix that contains all the interdependencies

between tasks in the application task graph.

$$E = \sum_{i=1}^{S} M_i El_i + (1 - M_i) EI_i + \sum_{(i,j) \in EM} |M_i - M_j| Et_{ij} \tag{6.2}$$

In the worst case, the total elapsed time to execute all the tasks in an offloading solution is defined as follows (Cuervo *et al.*, 2010).

$$T = \sum_{i=1}^{S} M_i Tl_i + (1 - M_i) TI_i + \sum_{\substack{(i,j) \\ \in EM}} |M_i - M_j| (Tt_{ij} + T_{NET}) \tag{6.3}$$

As outlined by Cuervo *et al.* (2010), the worst case time is used to keep the complexity of the problem down. In general, determining an optimal decision vector $M$, for $i = 1, 2, \ldots, S$ which minimizes energy use and meets the execution time constraint is NP-Hard.

## 6.2 The CA-ROA Algorithm

The main idea of the CA-ROA algorithm is to start from a good solution and then improve it through iteration. Having a good starting point helps to achieve a better final solution and a faster convergence. The pseudocode of the *Context Aware Randomized Offloading Algorithm* (CA-ROA) is shown in Algorithm 6.1. In this version, biased randomization is used to generate the initial random vectors (line 4). We consider the ratio of computation load, $C$, to the input data size, $DI$, for each task ($F$ vector in line 1). In order to be a good candidate for offloading, more computation load and less input data size for a task is desired, therefore a larger $F_i$ will make task $i$ more favourable for offloading. Calculating the ratio of these two parameters

gives a good indication of the tasks that are more suitable for offloading. In line 2, the elements of $F$ are sorted in descending order to have the more suitable offloading candidates at the top.

In line 3, the main loop of the algorithm with a total of $P$ iterations is initiated. In each iteration, a biased random binary vector is generated as the first solution vector, $M$. Details of the *BiasedRandomization* function are provided in Algorithm 6.2. Since there is a dependency between the tasks based on the application task graph, the necessity of transferring the input data for each task depends on the execution location of the intended task and its parent tasks. $F$ does not consider this location and although it provides useful information that can affect the offloading decision, it is not sufficient.

In the inner loop (lines 5-28), in each iteration a new candidate vector $V_{new}$ is generated. In line 6, $\gamma_r$, which is a random number between 0 and 1, is generated. The candidate vector $V_{new}$ is either generated randomly as a vector of length $S$ as in line 14 or additional steps are used to determine the $V_{new}$ vector (lines 8-12) when $\gamma_r$ satisfies the threshold. In line 7, if the randomly generated $\gamma_r$ is less than a threshold that we define ($\gamma$ is the probability of incorporating results of the prior iterations to the next one), the steps in lines 8-11 will be performed. Two prior decision vectors $(v_2, v_3)$ from the results of previous iterations are randomly selected (lines 9-10), along with a randomly generated vector $v_1$ (line 8), to form a new vector $V_{new}$ (line 12). In line 11, all the indices of identical bits between $v_2$ and $v_3$ are found and stored in $I_{2,3}$. $V_{new}$ is obtained by merging $v_1$ and $v_2$ with respect to $I_{2,3}$ (line 13).

To the point that using $F$ solely is insufficient, highly correlated tasks in terms of input data size are considered, i.e., tasks that need a large amount of input data

from their precedents. As the function $NeighbourUpdate$ (line 16) is explained in Algorithm 6.3, we select the same execution location for these correlated tasks to avoid the cost of transferring input data. Line 17 is related to determining the cut points, $CP$, for making sub-strings. $k$ indices from the lower $\gamma_c$ of the sorted $v_F$ vector are randomly selected. We have already considered large $F_i$ as the desired tasks for offloading (line 4 of the Algorithm 6.4). Now we are trying to select the cut points from the rest of $F$ which have not yet been considered. The energy consumption of the sub-strings are obtained (lines 20-22) and compared in line 23. If the selected sub-string improves the energy and it is not causing a violation of the time constraint, then it is incorporated into the $M$ vector (line 24). The $P$ optimized decision vectors are stored in a matrix $PD$. After all $P$ decision vectors have been iteratively optimized, the final vector $M$ which minimizes energy use is selected from the population and returned (lines 32-33)

Pseudocode for the BiasedRandomization function in line 4 of Algorithm 6.1 is provided in Algorithm 6.2. In line 2, the top $W$ indices of the $v_F$ vector are selected. These are the tasks that are favorable for offloading, so in line 6, for the selected $W$ indices, 0 is generated with a probability of $BF$. In line 11, for the rest of the tasks, a random bit with a probability of 0.5 is generated (pure randomization). The solution vector $M$ is returned as a result.

The NeighbourUpdate function is called in line 16 of Algorithm 6.1 and details are provided in Algorithm 6.3. In line 3, the total input and output data that needs to be transferred between tasks $i$ and $j$ are calculated in $DataMatrix$. Then in line 5, we select the top components of the $DataMatrix$ that are greater than $DF$ ($DF$ is the threshold) and we store the task numbers $i$ and $j$ corresponding to the selected

**Algorithm 6.1:** CA-ROA Algorithm Pseudocode

// $I = \{1, 2, \ldots, S\}$
// $S_L$ is the set of all tasks that must be executed locally
// $C_i$ is the computation (execution) load for task $i$.
// $DI_i$ is the (wireless) upload/download input data transfer size of task $i$.
// $PD$ is a matrix of binary vector solutions (initially empty)
$F_i = C_i/DI_i \ \forall i \in I$ // set the job offload priority for each task.
$v_F = \text{sort}(\{F_i\})$ // vector of job indices in decreasing order of $F_i$
**for** $p = 1$ **to** $P$ **do**
  $M = \text{BiasedRandomization}(v_F)$ // $M$ is a $1 \times S$ biased solution vector
  **repeat** $R'$ **times**
    $\gamma_r = \text{uniform}(0, 1)$ // random number between 0 and 1
    // make sure we have completed at least 2 iterations
    **if** $p \notin \{1, 2\} \wedge \gamma_r < \gamma$ **then**
      $v_1 =$ random binary vector of length $S$
      // $PD$ is a $P \times S$ matrix of prior solutions from inner loop
      $v_2 = PD(\text{uniformInt}(1, p), :)$ // randomly selected (row) vector from $PD$
      $v_3 = PD(\text{uniformInt}(1, p), :)$ // another randomly selected vector from $PD$
      $I_{2,3} = \{\, i \mid v_2(i) == v_3(i)\}$ // set of all indices where $v_2$ and $v_3$ agree
      // TripleMerge takes $|I_{2,3}|$ common bits from $v_2, v_3$ and remaining bits from $v_1$
      $V_{new} = \text{TripleMerge}(v_1, v_2, I_{2,3})$
    **else**
      $V_{new} =$ random binary vector of length $S$
    **end**
    $c = \text{NeighbourUpdate}(V_{new}, I_{2,3}, S_L)$
    $CP =$ ordered set of $k$ tasks randomly selected from lower $\gamma_c$ of $v_F$ // randomly select $k$ cut points to compare energy over
    $l = 1$ // $l$ is lower task fragment index
    **for** $u \in CP$ **do**
      // $u$ is upper task fragment index
      $E^c = \sum_{i=l}^{u} ES(i, c(i))$ // compute total task energy using $c$
      $E^M = \sum_{i=l}^{u} ES(i, M(i)),$ // compute total task energy using $M$
      // $TS_i$ is the time consumption of task $i$ using the decision bit for task $i$
      $T_{total} = \sum_{i=1}^{l-1} TS(i, M(i)) + \sum_{i=u+1}^{S} TS(i, M(i)) + \sum_{i=l}^{u} TS(i, c(i))$
      **if** $E^c < E^M \wedge T_{total} < T_C$ **then**
        $M(i) = c(i) \ \forall i \in \{l, l+1, \ldots, u\}$ // $M$ adopts this fragment of $c$
      **end**
      $l = u + 1$
    **end**
  **end**
  $PD(p, :) = M$
  Compute $E_{total}$ and $T_{total}$ given $M$, using Eq. (6.2) and Eq. (6.3)
**end**
Find the vector $M$ in the population(1 to $P$) that minimizes $\text{E}_{total}$
Return $E_{total}$, $T_{total}$ and decision vector $M$

**Algorithm 6.2:** BiasedRandomization Pseudocode of Algorithm 6.1

```
1  Function BiasedRandomization(v_F):
       // Given an offload task priority vector v_F, return a biased binary
       solution vector.
       // BF is the Bias Factor (set to 0.9)
2      v_W = v_F(1 : W)  // slice off the top W tasks in v_F
3      M = 1  // binary offload solution vector (initially All-Local execution)
4      for t ∈ {1, 2, ..., S} do
5          if t ∈ v_W then
6              if uniform(0, 1) > BF then
7                  M(t) = 0  // offload
8              else
9                  M(t) = 1  // no offload
10             end
11         else if uniform(0, 1) > 0.5 then
12             M(t) = 0  // offload
13         else
14             M(t) = 1  // no offload
15         end
16     end
   // M is a biased random bit stream of 0s and 1s of length S
17 return M
```

components in the $I_{max}$ vector. Then in lines 7-15, for any pair of tasks in the $I_{max}$ vector, we will select the same decision variables. If the task must be executed locally, the decision variable for both tasks in the pair of $i$ and $j$ will be 1 (local execution). In lines 9-13, we check to see if one of the tasks is among the ones where its decision bit came from the previous solutions and it is recorded in $I_{2,3}$ in line 11 of Algorithm 6.1. In lines 9 and 11, if either task $i$ or task $j$ is in the set $I_{2,3}$, we keep the decision bit for that task and select the same decision bit for the other.

The time complexity of the CA-ROA algorithm is determined by the energy and time evaluation of the sub-strings based on equations (6.2) and (6.3), the population

**Algorithm 6.3:** NeighbourUpdate Pseudocode in line 16 of Algorithm 6.1

```
1  Function NeighbourUpdate(c, I_{2,3}, S_L):
       // group offload decisions for tasks that intercommunicate
       // EM is the edge matrix of the task graph that contains all the
       inter-dependencies between tasks
2      for (i, j) ∈ EM do
           // There is an edge between i and j in the task graph
           // DI is the transfer input data between task i and task j
           // DO is the transfer output data between task i and task j
3          DataMatrix(i, j) = DI(i, j) + DO(i, j)
4      end
5      I_max = { (i, j) | DataMatrix(i, j) ≥ DF }  // set of indices where
       DataMatrix(i, j) ≥ DF
6      for (i, j) ∈ I_max do
           // select the same decision bit for tasks i and j to avoid the cost of
           transferring data
7          if j or i ∈ S_L then
8              c(i) = c(j) = 1  // tasks that must be executed locally
9          else if i ∈ I_{2,3} then
10             c(j) = c(i)
11         else if j ∈ I_{2,3} then
12             c(i) = c(j)
13         else
14             c(i) = c(j)
15         end
16     end
17 return c
```

size of the random bit strings, number of tasks in the task graph and the number of iterations in the evaluation process.

**Lines 1- 5:** Initializing step, two nested loops ($p$ and $R'$) and generating the first random population have time complexity of $O(S)$. The sorting instruction in line 2 has the complexity of $O(S\log(S))$. The MATLAB sort function is a Quick Sort.

**Lines 7-15 (Mutation Step):** In this step, we are trying to merge $w_1$ bits from vector $v_1$ and $w_2$ bits from vector $I_{2,3}$, therefore we are merging two vectors with size of $w_1$ and $w_2$, respectively. The time complexity of this step is $O(w_1 + w_2)$ which is $O(S)$.

**Line 16 (NeighbourUpdate function):** In this function, we first go through the edge matrix ($EM$) which has the complexity of $O(S + E)$ where $E$ shows the number of relations in the task graph. The for loop in lines 6-16 of this function has the complexity of $O(S)$ at worst.

**Lines 19-28:** Loop $u$ has a complexity of $O(S)$. Lines 18, 19 and 20 have the complexity of $O(S + E)$ where $E$ denotes the number of relations in the task graph. Lines 21 and 22 have the complexity of $O(1)$. The time complexity of lines 13-22 is $\mathrm{O}(S^2)$. Evaluation of the energy and time equations over the 2 nested loops ($p$ and $R'$) has time complexity of $O(P \cdot R' \cdot S^2)$ where $P$ is the total iterations of the outer loop, $R'$ is the total iterations of the inner loop and $S^2$ is the time complexity of the loop in lines 17-25 to evaluate the energy and time equations and sub-string incorporation in the current solution vector.

**Lines 29-33:** Complexity of line 29 is $O(1)$ to store the results. In line 32, we are trying to find the minimum of $P$ elements so the complexity is $O(P)$. Returning the results has the order of $O(1)$.

The time complexity of the CA-ROA algorithm is the worst time complexity among all the steps that are explained. Therefore, the computation complexity of CA-ROA is $O(P \cdot R' \cdot S^2)$.

## 6.3 Simulation Results for CA-ROA

In this section we examine the performance of the CA-ROA algorithm for the delay-constrained energy minimization problem that was presented in Section 6.1. The simulations are performed in MATLAB and we evaluated the CA-ROA algorithm over three sets of parameters that are provided in Table 6.2 as Model 1, Model 2 and Model 3. The CA-ROA algorithm is tested with $\gamma_c = \frac{2}{3}$ to select the cut points, $BF = 0.9$ as the bias factor, and $DF$ is selected in a way that $I_{max}$ contains 15 indices.

The first set of parameters (Model 1) is similar to Model 1 in Table 5.7 in Chapter 5 when $T_{NET} = 0$. The task graphs are from Deng *et al.* (2016), as shown in Figure 3.6. The second set of parameters in this section (Model 2) is similar to that in Table 5.7, Model 2 in Chapter 5 with the task graph from Zhang *et al.* (2012) as shown in Figure 3.7 with $T_{NET} = 20$ ms. In Model 2, the CA-ROA algorithm is tested with 10 and 100 pseudo-random task graphs, based upon the seed graph shown in Figure 3.7. The process of generating pseudo-random task graphs from the seed graph in Figure 3.7 is explained in Section 3.2.1. The third set of parameters (Model 3) is based on the Samsung N9002 device (Liu *et al.*, 2017). The task graph in Figure 3.10 is considered as the application task graph with $S = 20$.

Table 6.2: Simulation Parameters (Szymanski and Shahzad, 2018)

| **Parameters** | **Definition** | **Model 1** | **Model 2** | **Model 3** |
|---|---|---|---|---|
| | Topology | Figure 3.6 | Figure 3.7 | Figure 3.10 |
| $S$ | Number of tasks per application | 9 | 23 | 20 |
| $f_L$ | Local computation power | 500 Gcycle/sec | 1 Gcycle/sec | 1.5 Gcycle/sec |
| $f_C$ | Cloud server computation power | 5 Gcycle/sec | 50 Gcycle/sec | 50 Gcycle/sec |
| $P_L$ | Local power consumption | 0.5w | 1w | 1.3w |
| $P_I$ | Idle mode power consumption | 50 mw | 50 mw | 5.92 mw |
| $PT_s$ | Sending power consumption | 50 mw | 50 mw | 0.5 w |
| $PT_r$ | Receiving power consumption | 20 mw | 50 mw | 0.5 w |
| $R_s$ | Wireless Sending bit rate | 2 Mbps | 5 Mbps | 5 Mbps |
| $PT_r$ | Wireless Receiving bit rate | 2Mbps | 5 Mbps | 5 Mbps |
| $C_i$ | Computation workload of the task $i$ | 0.3 - 370 Mcycle | 0.009 - 1415 Mcycle | 0.9 - 85.5 Mcycle |
| $T_{NET}$ | Expected 1-way network delay | 0 | 20 ms | 10 ms |

### 6.3.1 Simulation Results of CA-ROA for Model 1

Model 1 evaluates our algorithm using the three task graphs that were originally presented by Deng *et al.* (2016), as shown in Figure 3.6. The task graph has $S = 9$ tasks, with $S' = 6$ offloadable tasks and 10 edges. The comparison of the results of Deng *et al.* (2016) and our algorithm as well as results from Brute-Force search, with system parameters identical to those of Deng *et al.* (2016), are presented in Table 6.3. In Table 6.3, the symbol $Itr$ denotes the total number of iterations in Algorithm 6.1, which is $P \cdot R'$. The rows labeled "Graph 1", "Graph 2" and "Graph 3" present the energy results for the three task graphs shown in Figure 3.6. The three CA-ROA columns correspond to the total number of iterations $Itr = \{100, 200, 300\}$. The sub-row "$\Delta$" displays the difference between the mean CA-ROA energy (Mean Energy), and the optimal BFS energy, expressed as the percent of the optimal BFS energy. It is clear from Table 6.3 that for the chosen task graphs and parameter settings, the CA-ROA finds the optimal results in 100 iterations.

Table 6.3: Comparison between the Total Energy Consumption (in mJ) for ROA-V2 and the algorithm of Deng *et al.* (2016)

| | **Optimal BFS Energy** | **Results from Deng *et al.* (2016)** | | **CA-ROA *Itr* = 100** | **CA-ROA *Itr* = 200** | **CA-ROA *Itr* = 300** |
|---|---|---|---|---|---|---|
| **Graph 1** | 118.3 | 118.3 | **Mean Energy** | 118.3 | 118.3 | 118.3 |
| | | | **$\Delta$%** | 0 % | 0 % | 0 % |
| **Graph 2** | 118.5 | 118.5 | **Mean Energy** | 118.5 | 118.5 | 118.5 |
| | | | **$\Delta$%** | 0 % | 0 % | 0 % |
| **Graph 3** | 176.2 | 176.2 | **Mean Energy** | 176.2 | 176.2 | 176.2 |
| | | | **$\Delta$%** | 0 % | 0 % | 0 % |

### 6.3.2 Simulation Results of CA-ROA for Model 2

The average results of the 10 and 100 variations of the task graph in Figure 3.7 for the second set of parameters (Model 2) are shown in Table 6.4. Table 6.4 shows the results for $\gamma = 0.25$, $R' = 10$. The symbol "Mean Energy" presents the mean energy results over the set of 10 and 100 pseudo-random task graphs that are averaged over 5 runs of the algorithm. "$\Delta$" shows the difference between the mean CA-ROA energy, and the optimal BFS energy, expressed as a percent of the optimal BFS energy. The $Itr$ parameter in Table 6.4 denotes the total number of iterations in the CA-ROA flowchart, i.e., $P \cdot R'$.

The row labeled "1 Graph" presents the energy results for the task graph shown in Figure 3.7. The rows "10 Graphs" and "100 Graphs" denote the results of the 10 and 100 pseudo-random task graphs from the seed task graph in Figure 3.7. The column labeled "Optimal Energy" shows the minimum BFS energy. The columns labeled "CA-ROA" show the energy used for the CA-ROA algorithm for various numbers of iterations, $Itr$. In Table 6.4, the 3 CA-ROA columns correspond to the total number of iterations $Itr = \{100, 200, 300\}$. The number of iterations per population member $R' = 10$, so that the population sizes for these columns correspond to $P = \{10, 20, 30\}$.

It is clear from the last column of Table 6.4 that the CA-ROA algorithm can find results that are within 1% of the optimal results by using $Itr = 300$ iterations in total for the 10 and 100 variations of the seed task graph.

### 6.3.3 Simulation Results of CA-ROA for Model 3

The results of CA-ROA from the task graph provided in Figure 3.10 but with 10 times the computation complexity and edge data sizes of the original graph (as listed

Table 6.4: Comparison of the Total Energy Consumption (in Joules) based on the graph of Zhang *et al.* (2012) for $\gamma = 0.25$, $R' = 10$

| $\boldsymbol{\gamma = 0.25, R' = 10}$ | | | | | |
|---|---|---|---|---|---|
| | | **Optimal Energy** | **CA-ROA** $\boldsymbol{Itr = 100}$ | **CA-ROA** $\boldsymbol{Itr = 200}$ | **CA-ROA** $\boldsymbol{Itr = 300}$ |
| **1 Graph** | **Mean Energy** | 240.73 | 244.4 | 244.4 | 243.7 |
| | **Δ%** | | 1.5 % | 1.5 % | 1.2 % |
| **10 Graphs** | **Mean Energy** | 553.23 | 575.4 | 565.1 | 559.2 |
| | **Δ%** | | 4 % | 2.14 % | 1 % |
| **100 Graphs** | **Mean Energy** | 483.06 | 507.19 | 495.3 | 488.8 |
| | **Δ%** | | 4.9 % | 2.5 % | 1.1 % |

in Table 6.2) are summarized in Table 6.5. The column labeled "All-Local Energy" shows the energy use when all tasks execute locally (on the mobile device). Table 6.5 can be interpreted in the same manner as Table 6.4 for Model 2. For the task graph in Figure 3.10, the mean energy for All-Local execution is 614.5 mJ. For the chosen task graph and parameter setting, the CA-ROA results are within 1% of the optimal result for $Itr = 300$ iterations which is about 227.1 mJ. CA-ROA provides about 63% energy savings compare to the All-Local method.

## 6.4 Summary

In this chapter, Context-Aware Randomized Offloading, i.e., Algorithm CA-ROA was presented. CA-ROA uses task graph information to both generate biased random vectors and determine the number of sub-strings the candidate vector will be partitioned

Table 6.5: Comparison between the Total Energy Consumption (in mJ) based on the graph of Tang *et al.* (2018)

| $\gamma = 0.25, R' = 10$ | | | | | | |
|---|---|---|---|---|---|---|
| | | **All-Local Energy** | **Optimal Energy** | **CA-ROA** $Itr = 100$ | **CA-ROA** $Itr = 200$ | **CA-ROA** $Itr = 300$ |
| **1 Graph** | **Mean Energy** | 614.5 | 224.3 | 228.2 | 228.2 | 227.1 |
| | **Δ%** | | | 1.7 % | 1.7 % | 1.1 % |

into. Performance results for the CA-ROA algorithm for the general model show that for the considered task graphs it finds near optimal results within 1% of the optimal results from Brute-Force search.

# Chapter 7

# Conclusions

In the future, many resource-constrained mobile devices will be able to reduce their energy consumption by utilizing computation offloading. To exploit this, offloading algorithms are needed that decide which tasks should be offloaded to an edge or cloud server, so that energy consumption can be minimized while satisfying the application's execution time constraints. Since the offloading algorithms must be embedded in the mobile devices, it is vital that they should not consume significant battery energy themselves, while performing the computations needed.

In this thesis, a series of efficient Randomized Offloading Algorithms were proposed to solve this optimization problem. The main feature of these algorithms is that they exploit randomization techniques to iteratively improve an offloading decision vector. In each iteration, a randomly-generated bit-string called the candidate vector is generated. Sub-strings of the candidate vector are incorporated into a decision vector if they improve the solution quality, in a process similar to Genetic Optimization. Other optimization techniques such as dynamic programming and some user-defined parameters are used alongside the proposed randomization technique.

Experimental results using parallel and general application task graphs show that the proposed algorithms can find nearly-optimal solutions quickly and effectively. The results of DPH and DPR in Chapter 4 and ROA-V1 in Chapter 5 for the parallel model show the good performance of the these algorithms. The energy results of the ROA-V2 and CA-ROA algorithms in Chapters 5 and 6 used more than 100 pseudo-random task graphs (general model). The results obtained are typically within 3% of the optimal energy results obtained through Brute-Force-Search and are often much closer.

As discussed in Table 2.2, a system in which multiple users offload some of their computation-intensive tasks to a remote server represents one possible generalization of the work proposed in this thesis. That generalization could include considering the case where each mobile user has multiple tasks to execute, with various levels of task dependency. The mobile users may share communication resources while offloading tasks to the cloud. The goal would be to jointly optimize the offloading decisions of all users as well as the allocation of communication resources and computational resources among the users to minimize the overall cost of energy consumption and delay for all users. One would need to jointly consider both the offloading decisions and the sharing of the communication and computation resources among the users as they compete to access the cloud through the wireless links.

The models and algorithms that were introduced in Chapters 4 and 5 for parallel task graphs can be extended to accommodate not only multiple users, but also the possibility of having different numbers of tasks per user. Another important extension would be to consider multiple remote server offloading, including both edge and cloud servers. The decision problem would be extended to include server selection for each

task.

Future extensions to the algorithms for general task graphs proposed in Chapters 5 and 6 could take into account multiple users with shared communication channels, shared computational resources, and task inter-dependencies. In traditional cloud computing systems, a central scheduler is used to solve for the decision variables for all the users, while in the distributed scheduling model, mobile users make their own decisions to execute their tasks remotely or locally. This is referred to as selfish decision making, where each user tries to minimize its own energy consumption. This problem could be coupled to the manner in which communication resources (such as channel time slots) are allocated during the computation offload process. Various game-theoretic formulations of this problem may provide significant insight into algorithm design and performance, and it appears that generalization of the algorithms in Chapters 5 and 6 may provide a good tradeoff between performance and computational cost. Another extension to this work would be to consider multiple remote servers, multiple available channels and multiple tasks per user.

# Bibliography

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., and Ayyash, M. (2015). Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials*, **17**(4), 2347–2376.

Barbarossa, S., Sardellitti, S., and Di Lorenzo, P. (2013). Joint allocation of computation and communication resources in multiuser mobile cloud computing. *In Proceedings of the 14th IEEE Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 26–30.

Cao, S., Tao, X., Hou, Y., and Cui, Q. (2015). An energy-optimal offloading algorithm of mobile computing based on hetnets. *In Proceedings of the IEEE International Conference on Connected Vehicles and Expo (ICCVE)*, pages 254–258.

Chen, M.-H., Liang, B., and Dong, M. (2015). A semidefinite relaxation approach to mobile cloud offloading with computing access point. *In Proceedings of the 16th IEEE International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 186–190.

Chen, M.-H., Liang, B., and Dong, M. (2016a). Joint offloading decision and resource

allocation for multi-user multi-task mobile cloud. *In Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1–6.

Chen, X., Jiao, L., Li, W., and Fu, X. (2016b). Efficient multi-user computation offloading for mobile-edge cloud computing. *IEEE/ACM Transactions on Networking*, **24**(5), 2795–2808.

Cheng, Z., Li, P., Wang, J., and Guo, S. (2015). Just-in-time code offloading for wearable computing. *IEEE Transactions on Emerging Topics in Computing*, **3**(1), 74–83.

Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A. (2011). Clonecloud: elastic execution between mobile device and cloud. *In Proceedings of the 6th Conference on Computer Systems*, pages 301–314.

Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. (2010). Maui: making smartphones last longer with code offload. *In Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, pages 49–62.

Deng, M., Tian, H., and Fan, B. (2016). Fine-granularity based application offloading policy in cloud-enhanced small cell networks. *In Proceedings of the IEEE International Conference on Communications Workshops (ICC)*, pages 638–643.

Gu, X., Nahrstedt, K., Messer, A., Greenberg, I., and Milojicic, D. (2004). Adaptive offloading for pervasive computing. *IEEE Pervasive Computing*, **3**(3), 66–73.

Huang, D., Wang, P., and Niyato, D. (2012). A dynamic offloading algorithm for

mobile computing. *IEEE Transactions on Wireless Communications*, **11**(6), 1991–1995.

Kalra, M. and Singh, S. (2015). A review of metaheuristic scheduling techniques in cloud computing. *Egyptian Informatics Journal*, **16**(3), 275–295.

Kamoun, M., Labidi, W., and Sarkiss, M. (2015). Joint resource allocation and offloading strategies in cloud enabled cellular networks. *In Proceedings of the IEEE Intrnational Conference on Communications (ICC)*, pages 5529–5534.

Kao, Y.-H., Krishnamachari, B., Ra, M.-R., and Bai, F. (2017). Hermes: Latency optimal task assignment for resource-constrained mobile computing. *IEEE Transactions on Mobile Computing*, **16**(11), 3056–3069.

Kosta, S., Aucinas, A., Hui, P., Mortier, R., and Zhang, X. (2012). Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. *In Proceedings of the IEEE INFOCOM*, pages 945–953.

Kumar, K. and Lu, Y.-H. (2010). *Cloud computing for mobile users: Can offloading computation save energy?*, volume 43. Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, 17 th Fl New York NY 10016-5997 United States.

Labidi, W., Sarkiss, M., and Kamoun, M. (2015a). Energy-optimal resource scheduling and computation offloading in small cell networks. *In Proceedings of the 22nd International Conference on Telecommunications (ICT)*, pages 313–318.

Labidi, W., Sarkiss, M., and Kamoun, M. (2015b). Joint multi-user resource scheduling and computation offloading in small cell networks. *In Proceedings of the IEEE*

*11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 794–801.

Li, Z., Wang, C., and Xu, R. (2001). Computation offloading to save energy on handheld devices: a partition scheme. *In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 238–246.

Lin, X., Wang, Y., Xie, Q., and Pedram, M. (2015). Task scheduling with dynamic voltage and frequency scaling for energy minimization in the mobile cloud computing environment. *IEEE Transactions on Services Computing*, **8**(2), 175–186.

Liu, J., Mao, Y., Zhang, J., and Letaief, K. B. (2016). Delay-optimal computation task scheduling for mobile-edge computing systems. *In Proceedings of the International Symposium on Information Theory (ISIT)*, pages 1451–1455.

Liu, W., Gong, W., Du, W., and Zou, C. (2017). Computation offloading strategy for multi user mobile data streaming applications. *In Proceedings of the IEEE 19th International Conference on Advanced Communication Technology (ICACT)*, pages 111–120.

Luo, Z., Ma, W., So, A. M., Ye, Y., and Zhang, S. (2010). Semidefinite relaxation of quadratic optimization problems. *IEEE Signal Processing Magazine*, **27**(3), 20–34.

Mach, P. and Becvar, Z. (2017). Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, **19**(3), 1628–1656.

Mao, Y., Zhang, J., and Letaief, K. B. (2016a). Dynamic computation offloading for

mobile-edge computing with energy harvesting devices. *IEEE Journal on Selected Areas in Communications*, **34**(12), 3590–3605.

Mao, Y., Zhang, J., Song, S., and Letaief, K. B. (2016b). Power-delay tradeoff in multi-user mobile-edge computing systems. *In Proceedings of the IEEE Global Communications Conference (GLOBECOM)*, pages 1–6.

Mao, Y., You, C., Zhang, J., Huang, K., and Letaief, K. B. (2017a). A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials*, **19**(4), 2322–2358.

Mao, Y., You, C., Zhang, J., Huang, K., and Letaief, K. B. (2017b). A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials*, **19**(4), 2322–2358.

Meskar, E., Todd, T. D., Zhao, D., and Karakostas, G. (2017). Energy aware offloading for competing users on a shared communication channel. *IEEE Transactions on Mobile Computing*, **16**(1), 87–96.

Miettinen, A. P. and Nurminen, J. K. (2010). Energy efficiency of mobile clients in cloud computing. *HotCloud*, **10**, 4–4.

Muñoz, O., Pascual-Iserte, A., and Vidal, J. (2013). Joint allocation of radio and computational resources in wireless application offloading. *In Proceedings of the IEEE Future Network and Mobile Summit (FutureNetworkSummit)*, pages 1–10.

Muñoz, O., Iserte, A. P., Vidal, J., and Molina, M. (2014). Energy-latency trade-off for multiuser wireless computation offloading. *In Proceedings of the IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, pages 29–33.

Munoz, O., Pascual-Iserte, A., and Vidal, J. (2015). Optimization of radio and computational resources for energy efficiency in latency-constrained application offloading. *IEEE Transactions on Vehicular Technology*, **64**(10), 4738–4755.

Niu, J., Song, W., Shu, L., and Atiquzzaman, M. (2013). Bandwidth-adaptive application partitioning for execution time and energy optimization. *In Proceedings of the IEEE International Conference on Communications (ICC)*, pages 3660–3665.

Qiu, M., Ming, Z., Li, J., Gai, K., and Zong, Z. (2015). Phase-change memory optimization for green cloud with genetic algorithm. *IEEE Transactions on Computers*, **64**(12), 3528–3540.

Ra, M.-R., Sheth, A., Mummert, L., Pillai, P., Wetherall, D., and Govindan, R. (2011). Odessa: enabling interactive perception applications on mobile devices. *In Proceedings of the 9th International Conference on Mobile systems, Applications, and Services*, pages 43–56.

Rong, P. and Pedram, M. (2003). Extending the lifetime of a network of battery-powered mobile devices by remote processing: a markovian decision-based approach. *In Proceedings of the 40th annual Design Automation Conference*, pages 906–911.

Sardellitti, S., Barbarossa, S., and Scutari, G. (2014a). Distributed mobile cloud computing: Joint optimization of radio and computational resources. *In Proceedings of the IEEE Globecom Workshops (GC Wkshps)*, pages 1505–1510.

Sardellitti, S., Scutari, G., and Barbarossa, S. (2014b). Joint optimization of radio and computational resources for multicell mobile cloud computing. *In Proceedings of*

*the IEEE 15th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 354–358.

Shahzad, H. and Szymanski, T. (2017). Randomized computational offloading for green mobile cloud and fog computing. Internal Memorandum.

Shahzad, H. and Szymanski, T. (2018). Randomized offloading algorithm for green mobile cloud and mobile edge computing. Submitted to *IEEE ACCESS*.

Shahzad, H. and Szymanski, T. H. (2016a). A dynamic programming offloading algorithm for mobile cloud computing. *In Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–5.

Shahzad, H. and Szymanski, T. H. (2016b). A dynamic programming offloading algorithm using biased randomization. *In Proceedings of the 9th IEEE International Conference on Cloud Computing (CLOUD)*, pages 960–965.

Song, J., Cui, Y., Li, M., Qiu, J., and Buyya, R. (2014). Energy-traffic tradeoff cooperative offloading for mobile cloud computing. *In Proceedings of the IEEE 22nd International Symposium of Quality of Service (IWQoS)*, pages 284–289.

Szymanski, T. and Shahzad, H. (2018). An improved randomized offloading algorithm (v2) for evaluating mobile cloud, fog and edge computing systems. Internal Memorandum.

Szymanski, T. H. (2018). 300 pseudo-random task graphs for evaluating mobile cloud, fog and edge computing systems. IEEE Dataport, http://dx.doi.org/10.21227/kak5-8n96, doi:10.21227/kak5-8n96.

Tang, C., Xiao, S., Wei, X., Hao, M., and Chen, W. (2018). Energy efficient and deadline satisfied task scheduling in mobile cloud computing. *In Proceedings of the IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 198–205.

Tang, L. and He, S. (2018). Multi-user computation offloading in mobile edge computing: A behavioral perspective. *IEEE Network*, **32**(1), 48–53.

Tian, Y., Ekici, E., and Ozguner, F. (2005). Energy-constrained task mapping and scheduling in wireless sensor networks. *In Proceedings of the IEEE International Conference on Mobile Adhoc and Sensor Systems Conference*, pages 8–pp.

Toma, A. and Chen, J.-J. (2013). Computation offloading for frame-based real-time tasks with resource reservation servers. *In Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 103–112.

Tong, L., Li, Y., and Gao, W. (2016). A hierarchical edge cloud architecture for mobile computing. *In Proceedings of the 35th IEEE International Conference on Computer Communications INFOCOM*, pages 1–9.

Tout, H., Talhi, C., Kara, N., and Mourad, A. (2017). Smart mobile computation offloading: Centralized selective and multi-objective approach. *Expert Systems with Applications*, **80**, 1–13.

Tseng, F.-H., Wang, X., Chou, L.-D., Chao, H.-C., and Leung, V. C. (2018). Dynamic resource prediction and allocation for cloud data center using the multiobjective genetic algorithm. *IEEE Systems Journal*, **12**(2), 1688–1699.

Wang, J., Peng, J., Wei, Y., Liu, D., and Fu, J. (2017). Adaptive application offloading decision and transmission scheduling for mobile cloud computing. *China Communications*, **14**(3), 169–181.

Wang, X., Chen, X., Wu, W., An, N., and Wang, L. (2016a). Cooperative application execution in mobile cloud computing: A stackelberg game approach. *IEEE Communications Letters*, **20**(5), 946–949.

Wang, Y., Sheng, M., Wang, X., Wang, L., and Li, J. (2016b). Mobile-edge computing: Partial computation offloading using dynamic voltage scaling. *IEEE Transactions on Communications*, **64**(10), 4268–4282.

Yang, L., Cao, J., Yuan, Y., Li, T., Han, A., and Chan, A. (2013). A framework for partitioning and execution of data stream applications in mobile cloud computing. *ACM SIGMETRICS Performance Evaluation Review*, **40**(4), 23–32.

Yang, L., Cao, J., Cheng, H., and Ji, Y. (2015). Multi-user computation partitioning for latency sensitive mobile cloud applications. *IEEE Transactions on Computers*, **64**(8), 2253–2266.

You, C. and Huang, K. (2016). Multiuser resource allocation for mobile-edge computation offloading. *In Proceedings of the IEEE Global Communications Conference (GLOBECOM)*, pages 1–6.

You, C., Huang, K., Chae, H., and Kim, B.-H. (2017). Energy-efficient resource allocation for mobile-edge computation offloading. *IEEE Transactions on Wireless Communications*, **16**(3), 1397–1411.

Zhang, K., Mao, Y., Leng, S., Zhao, Q., Li, L., Peng, X., Pan, L., Maharjan, S., and Zhang, Y. (2016). Energy-efficient offloading for mobile edge computing in 5g heterogeneous networks. *IEEE Access*, **4**, 5896–5907.

Zhang, L., Fu, D., Liu, J., Ngai, E. C.-H., and Zhu, W. (2017). On energy-efficient offloading in mobile cloud for real-time video applications. *IEEE Transactions on Circuits and Systems for Video Technology*, **27**(1), 170–181.

Zhang, Y., Liu, H., Jiao, L., and Fu, X. (2012). To offload or not to offload: an efficient code partition algorithm for mobile cloud computing. *In Proceedings of the IEEE 1st International Conference on Cloud Networking (CLOUDNET)*, pages 80–86.

Zhao, Y., Zhou, S., Zhao, T., and Niu, Z. (2015). Energy-efficient task offloading for multiuser mobile cloud computing. *In Proceedings of the IEEE/CIC International Conference on Communications in China (ICCC)*, pages 1–5.