

Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2017



Project Title: **Optimal Control and Scheduling of Computing Systems**

Student: **Anand Kasture**

CID: **00832896**

Course: **EEE4**

Project Supervisor: **Dr E.C. Kerrigan**

Second Marker: **Prof T. Parisini**

Abstract

The primary objective of this project is to reinforce the natural applicability of robust mathematical optimization methods to real-time multiprocessor task scheduling problems. The notion that optimal control theory can play a central role in the development of high performance, energy efficient computing systems is a consequence of increasingly promising results in literature today. This work aims to demonstrate the practical scalability of a custom written interior-point method (IPM) in solving a specific optimal control formulation of the task scheduling problem. In particular, we focus on the computational complexity associated with solving the Karush-Kuhn-Tucker (KKT) based system of equations during each IPM iteration, as this often represents the largest computational bottleneck in practice.

Our main contribution lies in establishing a worst case complexity result of $\mathcal{O}(N^2n)$, where N is the number of discrete time steps and n is the number of tasks, associated with solving the aforementioned system of equations. This result is achieved by carrying out a particular sequence of block eliminations on the left-hand side Jacobian matrix in order to drive down problem dimensionality first. Hereafter, we exploit the inherent sparsity structures of the linear algebraic objects involved in order to explicitly solve the reduced system of equations with the conjugate gradient method.

Acknowledgements

I would like to extend my heartfelt gratitude to Dr Eric Kerrigan, for his invaluable guidance and continuous support throughout the year. I have thoroughly enjoyed working closely with him on this incredibly challenging and interesting area of research.

I would also like to thank all my wonderful friends at Imperial College London for the unforgettable memories we have forged over the last four years.

Last but not least, I would like to thank my parents for their unconditional love. No words can express how grateful I am for all the opportunities they have provided me with. A special shout out also goes to my amazing little sister.

Contents

Abstract	i
Acknowledgements	iii
List of Tables	vii
List of Figures	ix
1 Introduction	1
1.1 Energy-Efficient Computing	1
1.2 Optimal Control of Computing Systems	1
1.3 Aims and Objectives	2
1.4 Thesis Outline	3
2 Background	4
2.1 Overview of Literature	4
2.2 Interior-Point Methods	7
2.2.1 History	7
2.2.2 Linear Programming	8
2.2.3 Primal-Dual Methods	9
2.2.4 Mehrotra's Predictor-Corrector Method	11
3 General Model	13
3.1 Feedback Scheduling Architecture	13
3.2 Workload Partitioning	14
3.2.1 Notation	14
3.2.2 Problem Definition	15
3.3 A Note on Task Ordering	17
4 The Workload-Partitioning Fixed-Deadline Problem (WP-FDP)	18
4.1 Introduction	18
4.2 Problem Definition	18
4.3 Solving WP-FDP with LINPROG	21
4.3.1 Conversion to Standard Form	22
4.3.2 Simulation Results and Analysis	25
4.4 The Complete KKT System of Equations	31
4.5 Solving WP-FDP with Custom IPM	33
4.5.1 Implementation Details	33
4.5.2 Simulation Results and Analysis	35
5 The Workload-Partitioning Different-Deadline Problem (WP-DDP)	37
5.1 Introduction	37
5.2 Problem Definition	37
5.3 The Complete KKT System of Equations	39
5.4 The Reduced KKT System of Equations	40
5.4.1 Application of the Sherman-Morrison Formula	46
5.4.2 Application of the Conjugate Gradient Method	50

5.5	Solving WP-DDP with Custom IPM	52
5.5.1	Implementation Details	52
5.5.2	Simulation Results and Analysis	55
6	Conclusions and Future Work	67
6.1	Summary of Thesis Achievements	67
6.2	Future Work	67
	Appendix A Software Package	69
A.1	DDP-AK7213-V5-CGS	69
A.2	DDP-AK7213-V4-CGS	77
A.3	Auxiliary Functions	83
	Bibliography	84

List of Tables

3.1	Problem Size Parameters	16
3.2	Additional Problem Parameters	16
3.3	Decision Variables	17
4.1	Simulation Results [FDP-AK7213-V1]	36

List of Figures

2.1	Fluid vs Practical Schedule (Cho et al. 2006)	6
3.1	Feedback Scheduling Architecture	13
4.1	Remaining Time-Evolution and System Workload Plots [FDP-LINPROG] .	26
4.2	Number of Iterations and Function Duration vs Number of Tasks [FDP-LINPROG]	27
4.3	Time per iteration vs Number of Tasks [FDP-LINPROG]	28
4.4	Number of Iterations and Function Duration vs Number of Steps [FDP-LINPROG]	29
4.5	Time per iteration vs Number of Steps [FDP-LINPROG]	29
4.6	Number of Iterations and Function Duration vs Number of Speed Levels [FDP-LINPROG]	30
4.7	Time per iteration vs Number of Speed Levels [FDP-LINPROG]	31
4.8	Sparsity pattern of Jacobian matrix for different problem sizes [FDP-AK7213-V1]	34
4.9	Function Profile Report [FDP-AK7213-V1]	34
5.1	Function Profile Report [DDP-AK7213-V2]	56
5.2	Sparsity pattern of the reduced Jacobian matrix for different problem sizes [DDP-AK7213-V2]	56
5.3	Function Profile Report [DDP-AK7213-V3]	57
5.4	Time per iteration vs Number of Tasks [DDP-AK7213-V3]	58
5.5	Time per iteration vs Number of Time Steps [DDP-AK7213-V3]	59
5.6	Time per iteration vs Number of Speed Levels [DDP-AK7213-V3]	60
5.7	Time per iteration vs Number of Tasks [DDP-AK7213-V4CGS]	61
5.8	Time per iteration vs Number of Time Steps [DDP-AK7213-V4CGS]	62
5.9	Time per iteration vs Number of Speed Levels [DDP-AK7213-V4CGS]	63
5.10	Time per iteration vs Number of Tasks [DDP-AK7213-V5CGS]	64
5.11	Time per iteration vs Number of Time Steps [DDP-AK7213-V5CGS]	65
5.12	Time per iteration vs Number of Speed Levels [DDP-AK7213-V5CGS]	66

Chapter 1

Introduction

1.1 Energy-Efficient Computing

The year 2015 marked fifty years since Gordon Moore, the co-founder of Intel, predicted that the number of transistors on every square inch of silicon would continue to double approximately every two years (Naffziger & Koomey 2015). Today, we live in an era where the significance of computing systems in our day-to-day lives cannot be overstated. This widespread presence has partly come about due to the enormous advancements in computational capacity that have been able to keep up with Moore's bold prediction.

The snowballing ubiquity of Information and Communications Technology (ICT) enabled solutions has also raised concerns regarding the sector's greenhouse gas footprint. For instance, Vereecken et al. (2010) predicts a rise in global ICT energy consumption to more than 14% by the year 2020. Estimates presented in Heddeghem et al. (2014) show that the combined electricity consumption of three main ICT categories (communication networks, personal computers and data centres) is growing at a rate of approximately 7% every year. This rate is higher than the annual growth in worldwide consumption. Note that the Global e-Sustainability Initiative (GeSI) SMARTer 2030 report actually predicts a decrease in ICT emissions as a proportion of global emissions by the year 2030 (GeSI 2014). Therefore, the continuous focus on energy-efficiency research across all ICT domains is set to be of crucial importance in driving a sustainable future.

1.2 Optimal Control of Computing Systems

A computing system can be usually classified into a hierarchy of discrete components, where each component is responsible for carrying out a particular function. These components constantly interact with each other in order to fulfil the overall requirements imposed upon the system by the user(s).

In recent years, we have witnessed an increasing level of interest in the fundamental re-design of the interactive computing framework from the optimal control community. In fact, the notion that modern control theory and numerical optimisation methods can

play a central role in the development of high performance, energy efficient computing systems is a direct consequence of advancements in computational capacity coupled with promising results in literature.

In the Kerrigan (2015) opinion paper, the author explicitly summarises the potential improvements that may be attained across various measures of system performance through the use of an optimisation based model predictive control (MPC) framework as opposed to the suboptimal ad-hoc methods that have been traditionally implemented in industry. More specifically, the author points out the natural applicability of optimisation methods to the real-time task-scheduling problem for computing systems, since such problems are naturally formulated in terms of mathematical optimisation constructs in the first place.

These arguments represent the starting point for our work. We are motivated by the need for more sophisticated energy-efficient scheduling paradigms across a diverse range of computing systems, and by the desire to demonstrate the enormous impact the optimal control community can make in this field.

We are also encouraged by the recent developments in industry, where the rapid advances in semiconductor technology has led to energy-aware multiprocessor architectures. For instance, ARM has developed a two-type heterogeneous multi-core architecture called big.LITTLE. The big.LITTLE scheme comprises of two processors (clusters) sharing the same instruction set architecture (ISA). The LITTLE processors are designed for maximum power efficiency, whereas the big processors are designed for meeting high performance requirements. The architecture also facilitates practical support for task migration, enabling task schedulers to devise a truly global schedule that is both energy and feasibility optimal (ARM 2013).

1.3 Aims and Objectives

This project concerns the application of feedback methods from control theory towards improving the practical performance of scheduling algorithms. The primary objective here is to develop efficient software implementations of optimisation based real-time task-scheduling algorithms on multiprocessor systems, with an underlying aim of minimising overall energy consumption. Ultimately, we aim to reinforce the natural applicability of robust mathematical optimization methods to task scheduling problems within the computing systems context.

More specifically, this work aims to demonstrate the practical scalability of a custom written interior-point method (IPM) in solving a specific optimal control formulation of the task scheduling problem. In particular, we focus on the computational complexity associated with solving the Karush-Kuhn-Tucker (KKT) based system of equations during each IPM iteration, as this often represents the largest computational bottleneck in practice. The manner in which novel approaches to existing problems scale with an increasing problem size often represents an important feasibility metric. Therefore, the main focus of this project lies on the computational complexities associated with the devised algorithms as opposed to nominal performance results for a given set of parameters.

Our main contribution lies in establishing a worst case complexity result of $\mathcal{O}(N^2n)$,

where N is the number of discrete time steps and n is the number of tasks, associated with solving the aforementioned system of equations. Despite the quadratic increase in the number of time steps, the linear scaling relationship with the number of tasks represents an important result since we expect $n \gg N$ in practice.

1.4 Thesis Outline

This report is organised as follows. After this brief introduction to the subject, we present an extensive overview of literature followed by an outline of interior point methods. Hereafter, the optimal control formulation associated with the task scheduling problem is formally defined in the 'General Model' chapter.

The 'Workload-Partitioning Fixed-Deadline Problem (WP-FDP)' chapter focuses on a particular variation of the task scheduling problem where all tasks are due at a fixed point on the time grid. This chapter develops the fundamental definitions that remain valid throughout the report. We also express a linear program formulation of the optimal control problem that is fully described in terms of vectors and matrices. This step enables us to start solving the task scheduling problem in software. First, we make use of the `linprog` function provided with MATLAB to solve the linear program. Secondly, we develop the necessary mathematics for implementing a custom written interior point method in software. The software implementation details, simulation results and analysis for both approaches is shown.

The 'Workload-Partitioning Different-Deadline Problem (WP-DDP)' chapter concerns the task scheduling problem where tasks do not necessarily have the same deadlines across the time grid. A large proportion of the report is focused on this more involved problem. Firstly, we specify the exact sequence of block eliminations carried out to reduce the dimensionality of the matrix equation that needs to be solved in interior point methods. Hereafter, we prove the worst case complexity result of $\mathcal{O}(N^2n)$ with the help of the Sherman-Morrison formula. The second part of this chapter concerns the corresponding software implementation of the theory developed in the first part. Namely, we explain the workings of the main MATLAB functions that were developed in this project. The implementation details, simulation results and analysis for all methods is presented.

Finally, we finish with the 'Conclusions and Future Work' chapter, where we summarise thesis achievements and discuss a few concepts that could constitute the next stage of this work.

Chapter 2

Background

2.1 Overview of Literature

We have witnessed new interest in the multiprocessor scheduling problem from a wide range of perspectives in recent years. This project approaches this problem with the conviction that modern numerical optimisation methods and feedback control theory possess great promise in solving the broader class of resource allocation problems, as long as they are implemented with the utmost dexterity. This is not a completely novel approach by any means since several researchers have investigated the application of the aforementioned theory to computing systems.

In Papadopoulos et al. (2015) the authors cite the advantages of designing components from the ground up using control theory, especially on the back of promising results seen with uniprocessor schedulers. This paper extends the single core processor model seen in Leva & Maggio (2010) to the multi core processor case. This is primarily achieved through the design of a central thread dispatcher module that is responsible for thread migration. In Maggio et al. (2014), some of the same authors make the case for the incorporation of control theory and its methods within the design of computing systems. They argue that the inclusion of a dynamical system model in the design cycle from the very beginning brings about several advantages, namely the fact that the design can be assessed systematically, providing theoretical guarantees on performance and robustness. Furthermore, a scheduling algorithm titled I+PI is designed and successfully benchmarked against the Round Robin(RR) and Earliest Deadline First (EDF) algorithms. Moreover, Filieri et al. (2015), discusses a control theory based design process within the general software engineering context. The broad range of robust mathematical formulations within control theory are depicted as very relevant amongst the host of different approaches towards constructing self-adaptive software.

The Imes et al. (2015) paper introduces a platform independent, energy aware, resource allocation scheme that is able to meet soft real time constraints. The authors cite the platform/application specificity of several energy minimisation methods on many embedded systems today. They introduce a portable open-source C library appropriately titled (Performance with Optimal Energy Toolkit) POET, and evaluate it using several different benchmarks. Moreover, POET actually uses feedback control techniques in order to al-

locate resources on embedded platforms, and is another example of the growing presence of control theory within the computing systems sphere.

Multiprocessor scheduling algorithms in literature can be generally identified as global or partition based. A global scheme entails non-restrictive job migration across different processors, and often entails a significantly more ambitious scheduler design, especially on heterogeneous multiprocessor systems where the dissimilarities in silicon and/or instruction set architecture need to be accounted for. A partition based approach involves assigning each task in the task set to one of the available processors, after which the problem is reduced to finding a feasible uniprocessor schedule. While both approaches have their respective advantages and disadvantages, a global scheduling paradigm is deemed superior in the theoretical sense (Correa et al. 2012), (Raravi et al. 2014). Note that the uniprocessor task-scheduling problem has been well studied, and is often solved using a greedy algorithm such as the Earliest Deadline First (EDF) (Liu & Layland 1973). However, these techniques do not easily extend to the multiprocessor case without the introduction of overly complex modifications that undermine the simplicity of the original approach.

Some of the well-known global scheduling algorithms for homogeneous multiprocessors are based on the notion of fairness applied to a fluid scheduling model. Figure 2.1 illustrates a fluid task execution path example the remaining execution time is plotted as a linearly decreasing function of the local time. This model represents the *ideal* execution path for a task. Baruah et al. (1996) introduced the first optimal homogeneous multiprocessor scheduling algorithm for periodic tasks: Proportionate-fair (Pfair). This algorithm is based on the fair distribution of processing capacity amongst tasks such that no task is more than one quantum (unit time) away from its fluid path. However, one of the biggest drawbacks associated with Pfair, and other similar approaches, is the potentially unworkable number of task migrations and context switches. Alternatively, the Largest Local Remaining Execution First (LLREF) algorithm as presented in Cho et al. (2006) is quite similar to Pfair. LLREF is based on the fluid scheduling model and a fairness notion, except that the fairness requirement is only imposed at task deadlines (as opposed to at every time instant). This paper also publishes a new graphical concept known as T-L planes that are used to visualise the deadline partitioned fluid schedule.

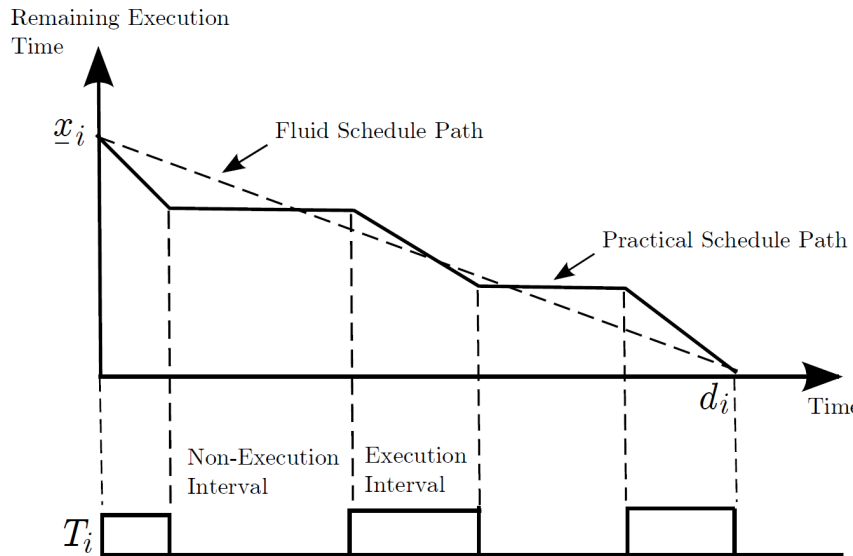


Figure 2.1: Fluid vs Practical Schedule (Cho et al. 2006)

More recently, Levin et al. (2010) attempts to provide a general framework/guidelines, termed DP-FAIR, for algorithms that attempt to utilise the notion of fluid scheduling with deadline partitioning for both periodic and sporadic tasks. Note that deadline partitioning refers to the partitioning of time into a sequence of slices that are marked by two successive task deadlines. The paper also points out how greedy scheduling algorithms such as Earliest Deadline First (EDF) can fail to be feasibility optimal on multiprocessors. The authors also present a scheduling algorithm, DP-WRAP, in conjunction with the DP-FAIR model and effectively based on McNaughton’s wrap-around algorithm (McNaughton 1959). In Funk et al. (2012), the authors build on the theory published in Levin et al. (2010) by presenting a DP-WRAP based algorithm with the aim of reducing power consumption. The authors argue that we can do better than the constant worst case execution times used in the DP-WRAP algorithm. A reduction in power consumption is achieved through the implementation of a Dynamic Voltage Frequency Scaling (DVFS) scheme in conjunction with one of the following two approaches: adjusting execution times within time slices in order to account for early task completion (slack reclamation) or anticipating early task completions and under-allocating local task execution values (aggressive speed selection).

The authors in Chwa et al. (2014) focus on achieving both a feasibility and energy optimal scheduling algorithm. The research is motivated by the advent of ARM’s big.LITTLE architecture that allows practical support for task migration in the form of a common instruction set and a dedicated cross-cluster interconnection bus. The global heterogeneous multi-core scheduling problem is formulated specifically for the big.LITTLE architecture. The overall problem is then split up into a workload assignment and a schedule generation problem. The former involves computing the percentage of each task that should be assigned to one of the available two clusters, whereas the latter concerns constructing a global (task migrations allowed) algorithm by using the solution to the first sub problem. Subsequently, the authors formally present two algorithms that solve the workload assignment and schedule generation sub problems in Chwa et al. (2015). They show that these two algorithms, named Hetero-Split and Hetero-Wrap, have computational complexity of $O(n \log n)$ and $O(n)$ respectively (n is the number of tasks in the task set). They

do not formulate the workload assignment problem as a linear program, arguing that an LP solution requires polynomial time complexity, and that obtaining a particular feasible solution using Hetero-Split facilitates optimal task assignment through the Hetero-Wrap algorithm.

2.2 Interior-Point Methods

In this section, we will present an overview of the primary topic of study in this work - interior-point methods. We start with a brief history of this class of constrained optimisation algorithms. This is followed by a mathematical outline of linear programming problems and primal-dual interior-point methods. This background theory serves as a key precursor to understanding the intuition behind the problem specific implementation of Mehrotra's predictor-corrector method presented in later chapters.

2.2.1 History

Numerical optimisation represents an important mathematical framework within the realm of decision science today. The effectiveness of this framework when applied to a particular real-world problem depends on two key factors. Firstly, a mathematical model of the objective and constraints must be constructed at an appropriate level of abstraction i.e. a formulation that incorporates sufficient detail but is still simple enough to solve quickly in practice. Secondly, an optimisation algorithm that is most applicable must be chosen in order to ensure the efficient computation of an optimal solution.

Academics often identify two key developments that have shaped the modern era of optimisation algorithms. Firstly, the introduction of the simplex method in the late 1940s made the systematic analysis of theoretical models a practical reality (Dantzig 1951). The timely arrival of the first electronic computers during this time enabled researchers to continually improve simplex-based software implementations. The simplex method was the only available method to solve linear programs for several decades after its introduction, and it continues to remain an efficient and reliable algorithm in practice.

The main idea of the simplex method involves traversing the vertices of the feasible region until the optimal solution is reached. However, Klee & Minty (1970) showed that the worst case complexity of this method is exponential time, rendering it a non-polynomial time algorithm in general. This result brings us to the second key development in optimisation theory.

In 1984, Karmarkar demonstrated the polynomial time complexity of an ellipsoid based method that also performed well in practice (Karmarkar 1984). This breakthrough generated huge interest in the optimisation community as researchers were motivated by the prospect of solving large linear programs that were previously thought to be too difficult. These efforts led to a new class of methods known as the interior-point methods (IPMs). An IPM differs from the simplex method by traversing the interior of the feasible region until some stopping condition is met.

In the late 1980s and early 1990s, on the back of work carried out by Mehrotra, Lustig, Marsten, and Shanno to name a few, the practical effectiveness of IMPs in solving very large scale LPs started gaining appreciation (Gondzio 2012).

Within the class of IPMs, the so-called primal-dual methods possess the most appealing theoretical and practical properties. These methods are one of the most useful and commonly utilised class of algorithms today. In fact, most software implementation of IPMs since the 1990s is based on a single primal-dual method: Mehrotra's predictor-corrector algorithm (Mehrotra 1992).

2.2.2 Linear Programming

Linear programming refers to a fundamental class of mathematical optimisation problems where the problem requirements are described by linear relationships only. Linear programming techniques have continued to remain a mainstay across a range of different applications today. This is because of the intrinsic simplicity of the model in conjunction with the existence of several advanced software packages with convergence guarantees.

Consider the linear programming problem in its standard form (2.1). We utilise the notation and terminology as presented in Nocedal & Wright (2006). Here, the n decision variables are contained in the vector $\mathbf{x} \in \mathbb{R}^n$, and the linear coefficients in the objective function to be minimised are represented by the vector $\mathbf{c} \in \mathbb{R}^n$. We impose m linear equality constraints on the decision variables by defining the matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and the vector $\mathbf{b} \in \mathbb{R}^m$. We also impose a positivity requirement on all decision variables in the form of an inequality constraint.

$$\min_{\mathbf{x}} \quad \mathbf{c}^T \mathbf{x} \tag{2.1a}$$

$$\text{s.t.} \quad \mathbf{A} \mathbf{x} = \mathbf{b} \tag{2.1b}$$

$$\mathbf{x} \geq \mathbf{0} \tag{2.1c}$$

Note that any linear program can be converted into this standard form. Moreover, there always exists another linear program associated with every original formulation known as the dual. The dual problem to (2.1) is shown in (2.2). The original minimisation problem (2.1) is referred to as the primal. The components of $\boldsymbol{\lambda} \in \mathbb{R}^m$ and $\mathbf{s} \in \mathbb{R}^n$ are called the dual decision variables and dual slack variables respectively.

$$\max_{\boldsymbol{\lambda}} \quad \mathbf{b}^T \boldsymbol{\lambda} \tag{2.2a}$$

$$\text{s.t.} \quad \mathbf{A}^T \boldsymbol{\lambda} + \mathbf{s} = \mathbf{c} \tag{2.2b}$$

$$\mathbf{s} \geq \mathbf{0} \tag{2.2c}$$

The primal and dual problems are inherently interrelated. One of the important results from duality theory states that the dual objective is always forms a lower bound to the

primal objective i.e. $\mathbf{b}^T \boldsymbol{\lambda} \leq \mathbf{c}^T \mathbf{x}$. Therefore, because the primal problem is a minimisation problem, and the dual a maximisation problem, the equivalence of the two objective functions occurs at the optimal solutions to the two problems. In other words, $\mathbf{b}^T \boldsymbol{\lambda}^* = \mathbf{c}^T \mathbf{x}^*$ when \mathbf{x}^* and $(\boldsymbol{\lambda}^*, \mathbf{s}^*)$ represent the optimal solutions to the primal and dual problem respectively. This case is known as strong duality, as opposed to the case of weak duality where the duality gap $\mathbf{c}^T \mathbf{x}^* - \mathbf{b}^T \boldsymbol{\lambda}^* \geq 0$.

The primal and dual problems shown in (2.1) and (2.2) are convex optimisation problems. Therefore, the first-order Karush-Kuhn-Tucker (KKT) conditions form both the necessary and sufficient conditions for a global optimal solution.

The optimality conditions for the primal problem state that the vector $\mathbf{x}^* \in \mathbb{R}^n$ is a global minimiser of (2.1) if and only if there exist vectors $\boldsymbol{\lambda}^* \in \mathbb{R}^m$ and $\mathbf{s}^* \in \mathbb{R}^n$ such that the first-order KKT conditions in (2.3) are satisfied for $(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{s}) = (\mathbf{x}^*, \boldsymbol{\lambda}^*, \mathbf{s}^*)$.

$$\mathbf{A}^T \boldsymbol{\lambda} + \mathbf{s} - \mathbf{c} = 0 \quad \text{stationarity} \quad (2.3a)$$

$$\mathbf{A} \mathbf{x} - \mathbf{b} = 0 \quad \text{stationarity} \quad (2.3b)$$

$$\mathbf{x} \geq \mathbf{0} \quad \text{primal feasibility} \quad (2.3c)$$

$$\mathbf{s} \geq \mathbf{0} \quad \text{dual feasibility} \quad (2.3d)$$

$$x_i s_i = 0 \quad \forall i \in [1, n] \quad \text{complementary slackness} \quad (2.3e)$$

The complementary slackness condition (2.3e) implies that at least one of the two components x_i and s_i must equal zero for $i = 1, 2, \dots, n$. The manner in which this condition is handled represents an important discriminator of optimisation algorithms.

Note that an important relationship between the primal (2.1) and dual problem (2.2) is revealed by selecting $\boldsymbol{\lambda}$ and \mathbf{s} to represent both the Lagrange multipliers of the primal problem as well as the unknowns in the dual problem: when deriving the optimality conditions for the dual problem, we observe that the first-order KKT conditions (2.3) for the primal problem also apply to the dual.

More specifically, the optimality conditions for the dual problem state that the vectors $\boldsymbol{\lambda}^* \in \mathbb{R}^m$ and $\mathbf{s}^* \in \mathbb{R}^n$ represent the global optimal solutions to (2.2) if and only if there exists a vector $\mathbf{x}^* \in \mathbb{R}^n$ such the optimality conditions in (2.3) are satisfied for $(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{s}) = (\mathbf{x}^*, \boldsymbol{\lambda}^*, \mathbf{s}^*)$. To that end, the set of vectors $(\mathbf{x}^*, \boldsymbol{\lambda}^*, \mathbf{s}^*)$ are referred to as primal-dual solutions.

2.2.3 Primal-Dual Methods

In most iterative optimisation algorithms, we attempt to find a useful search direction alongside an appropriate step length at each iteration.

Primal-dual interior point methods obtain a search direction by repeatedly applying a modified version of Newton's method to the equality conditions (2.3a), (2.3b), (2.3e).

The equality conditions are rewritten in matrix form as follows

$$\mathbf{F}(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{s}) = \begin{bmatrix} \mathbf{A}^T \boldsymbol{\lambda} + \mathbf{s} - \mathbf{c} \\ \mathbf{A}\mathbf{x} - \mathbf{b} \\ \mathbf{X}\mathbf{S}\mathbf{e} \end{bmatrix} = \mathbf{0} \quad (2.4)$$

where $\mathbf{X} = \text{diag}(\mathbf{x})$, $\mathbf{S} = \text{diag}(\mathbf{s})$, and $\mathbf{e} = [1 \ 1 \ \cdots \ 1]^T$.

Newton's method obtains a new search direction $(\Delta\mathbf{x}, \Delta\boldsymbol{\lambda}, \Delta\mathbf{s})$ by forming a linear model of \mathbf{F} around the current iteration and solving the system of linear equations (2.5). The left-hand side matrix here is the Jacobian of \mathbf{F} . The iteration indices have been left out to aid readability.

$$\begin{bmatrix} & \mathbf{A}^T & \mathbf{I} \\ \mathbf{A} & & \\ \mathbf{S} & & \mathbf{X} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x} \\ \Delta\boldsymbol{\lambda} \\ \Delta\mathbf{s} \end{bmatrix} = - \begin{bmatrix} \mathbf{A}^T \boldsymbol{\lambda} + \mathbf{s} - \mathbf{c} \\ \mathbf{A}\mathbf{x} - \mathbf{b} \\ \mathbf{X}\mathbf{S}\mathbf{e} \end{bmatrix} \quad (2.5)$$

The step length is chosen such that the inequality conditions (2.3c), (2.3d) are strictly satisfied at each iteration. However, a full step along the Newton direction is likely to violate these conditions. Therefore, we carry out a line search in order to select the largest possible step length while still maintaining iterate feasibility. Traversing the interior of the feasible set directly along the Newton direction (affine scaling direction) does not work well in practice as this approach leads to very small step lengths, preventing any meaningful progress towards the solution.

Primal-dual methods address this problem by relaxing the complementarity slackness condition. This is achieved by modifying the system of linear equations such that the value of $x_i s_i$ for $i = 1, 2, \dots, n$ is gradually reduced towards zero. This modification produces a less aggressive Newton direction towards the interior of the feasible set during the initial iterations. The modified Newton's method is shown in (2.6).

$$\begin{bmatrix} & \mathbf{A}^T & \mathbf{I} \\ \mathbf{A} & & \\ \mathbf{S} & & \mathbf{X} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x} \\ \Delta\boldsymbol{\lambda} \\ \Delta\mathbf{s} \end{bmatrix} = - \begin{bmatrix} \mathbf{A}^T \boldsymbol{\lambda} + \mathbf{s} - \mathbf{c} \\ \mathbf{A}\mathbf{x} - \mathbf{b} \\ \mathbf{X}\mathbf{S}\mathbf{e} - \sigma\mu\mathbf{e} \end{bmatrix} \quad (2.6)$$

Here, we require the pairwise product $x_i s_i = \sigma\mu$ as opposed to $x_i s_i = 0$ for $i = 1, 2, \dots, n$. The quantity $\mu = (\mathbf{x}^T \mathbf{s})/n$ is known as the duality measure, and represents the average value of the pairwise products $x_i s_i$. The variable $\sigma \in [0, 1]$ is known as the centring parameter.

For $\sigma = 1$, the solution to (2.6) defines a centring direction. This direction is biased towards a point in the interior of the feasibility set where all pairwise products equal the value of μ on the current iteration. Prioritising centrality helps achieve a potentially large reduction in μ on the next iteration. When $\sigma = 0$, the solution to (2.6) represents the standard Newton (affine scaling) direction. Increasing σ from 0 to 1 corresponds to trading off a reduction in μ for improved centrality.

Algorithm (1) summarises the general methodology for primal-dual interior-point methods.

Algorithm 1 General Primal-Dual Framework

Input: $(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{s})$ with $(\mathbf{x}, \mathbf{s}) > \mathbf{0}$, $\epsilon > 0$

- 1: **while** $\mu > \epsilon$ **do** ▷ Repeat until duality measure $<$ tolerance
 - 2: **select** $\sigma \in [0, 1]$
 - 3: **set** $\mu \leftarrow (\mathbf{x})^T \mathbf{s} / n$
 - 4: **solve**
$$\begin{bmatrix} & \mathbf{A}^T & \mathbf{I} \\ \mathbf{A} & & \\ \mathbf{S} & & \mathbf{X} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \boldsymbol{\lambda} \\ \Delta \mathbf{s} \end{bmatrix} = - \begin{bmatrix} \mathbf{A}^T \boldsymbol{\lambda} + \mathbf{s} - \mathbf{c} \\ \mathbf{A} \mathbf{x} - \mathbf{b} \\ \mathbf{X} \mathbf{S} \mathbf{e} - \sigma \mu \mathbf{e} \end{bmatrix}$$
 - 5: **set** $(\mathbf{x}', \boldsymbol{\lambda}', \mathbf{s}') \leftarrow (\mathbf{x}, \boldsymbol{\lambda}, \mathbf{s}) + \alpha(\Delta \mathbf{x}, \Delta \boldsymbol{\lambda}, \Delta \mathbf{s})$
 where $\alpha \in [0, 1]$ such that $(\mathbf{x}', \mathbf{s}') > \mathbf{0}$
 - 6: **update** $(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{s}) \leftarrow (\mathbf{x}', \boldsymbol{\lambda}', \mathbf{s}')$
 - 7: **end while**
-

2.2.4 Mehrotra's Predictor-Corrector Method

Since its introduction in the 1990s, Mehrotra's predictor-corrector method remains one of the most widely implemented primal-dual methods. Mehrotra's contribution was to combine existing concepts with effective heuristics for selecting the centring parameter σ and the step length α at each iteration. The algorithmic details presented in this sub-section are based on Wright (1997) and Rao et al. (1998).

The predictor-corrector method differs from the generic primal-dual framework (1) in two main ways. Firstly, the affine-scaling (predictor) direction is calculated separately from, and prior to, the centring (corrector) direction. This entails solving two systems of linear equations during each iteration. Note that the same left-hand side Jacobian matrix is used in both these computations. Therefore, the left-hand side factorisation cost is spread across two solves and the overall increase in computation is less than twofold.

Secondly, the centring parameter σ is calculated adaptively using the information obtained from the affine-scaling direction. If the affine-search direction is deemed to make good progress towards the solution, a smaller value of the centring parameter σ is chosen at that iteration. Conversely, we use a value of σ closer to 1 when significant centring is required. The complete method is depicted in Algorithm 2.

Algorithm 2 Mehrotra's Predictor-Corrector Method

Input: $(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{s})$ with $(\mathbf{x}, \mathbf{s}) > \mathbf{0}$, $\epsilon > 0$

- 1: **while** $\mu > \epsilon$ **do** ▷ Repeat until duality measure $<$ tolerance
 - 2: **solve** $\begin{bmatrix} & \mathbf{A}^T & \mathbf{I} \\ \mathbf{A} & & \\ \mathbf{S} & & \mathbf{X} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}^{\text{aff}} \\ \Delta \boldsymbol{\lambda}^{\text{aff}} \\ \Delta \mathbf{s}^{\text{aff}} \end{bmatrix} = - \begin{bmatrix} \mathbf{A}^T \boldsymbol{\lambda} + \mathbf{s} - \mathbf{c} \\ \mathbf{A} \mathbf{x} - \mathbf{b} \\ \mathbf{X} \mathbf{S} \mathbf{e} \end{bmatrix}$
 - 3: $\alpha^{\text{aff}} = \arg \max \{ \alpha \in [0, 1] : (\mathbf{x}, \mathbf{s}) + \alpha(\Delta \mathbf{x}^{\text{aff}}, \Delta \mathbf{s}^{\text{aff}}) \geq \mathbf{0} \}$
 - 4: $\mu^{\text{aff}} = (\mathbf{x} + \alpha^{\text{aff}} \Delta \mathbf{x}^{\text{aff}})^T (\mathbf{s} + \alpha^{\text{aff}} \Delta \mathbf{s}^{\text{aff}}) / n$
 - 5: $\sigma = (\mu^{\text{aff}} / \mu)^3$
 - 6: **solve** $\begin{bmatrix} & \mathbf{A}^T & \mathbf{I} \\ \mathbf{A} & & \\ \mathbf{S} & & \mathbf{X} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}^{\text{cc}} \\ \Delta \boldsymbol{\lambda}^{\text{cc}} \\ \Delta \mathbf{s}^{\text{cc}} \end{bmatrix} = - \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \Delta \mathbf{X}^{\text{aff}} \Delta \mathbf{S}^{\text{aff}} \mathbf{e} - \sigma \mu \mathbf{e} \end{bmatrix}$
 - 7: $(\Delta \mathbf{x}, \Delta \boldsymbol{\lambda}, \Delta \mathbf{s}) = (\Delta \mathbf{x}^{\text{aff}}, \Delta \boldsymbol{\lambda}^{\text{aff}}, \Delta \mathbf{s}^{\text{aff}}) + (\Delta \mathbf{x}^{\text{cc}}, \Delta \boldsymbol{\lambda}^{\text{cc}}, \Delta \mathbf{s}^{\text{cc}})$
 - 8: $\alpha = \arg \max \{ \alpha \in [0, 1] : (\mathbf{x}, \mathbf{s}) + \alpha(\Delta \mathbf{x}, \Delta \mathbf{s}) \geq \mathbf{0} \}$
 - 9: $(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{s}) = (\mathbf{x}, \boldsymbol{\lambda}, \mathbf{s}) + \alpha(\Delta \mathbf{x}, \Delta \boldsymbol{\lambda}, \Delta \mathbf{s})$
 - 10: $\mu = \mathbf{x}^T \mathbf{s} / n$
 - 11: **end while**
-

Chapter 3

General Model

In this chapter, we will develop the intuition behind our optimisation based approach to solving the multi-processor task scheduling problem. A high-level description of the feedback and optimal control framework is presented first. Hereafter, we formally define the mathematical model that fully describes the optimisation problem to be solved.

3.1 Feedback Scheduling Architecture

The arguments presented in Thammawichai & Kerrigan (2016) represent an excellent starting point for this project. The authors assert that several fluid-based scheduling algorithms in literature use worst-case parameter estimates, leading to unnecessarily high operating frequencies and sub-optimal energy consumption levels. Furthermore, they cite the regulatory nature of existing *feedback* methods, and propose an optimal feedback control framework instead.

Figure 3.1 illustrates the high-level architecture that was proposed in Thammawichai & Kerrigan (2016). The event-driven scheduler is invoked whenever a task completes its allocated workload, and also when a new task arrives in the system. The feedback scheme is crucial to the optimal operation of the system in an unpredictable scheduling environment. Measured task execution parameters are fed back to the scheduler, and the optimisation problem is re-solved in order to ensure the feasibility and energy optimal control of the system as it evolves in time.

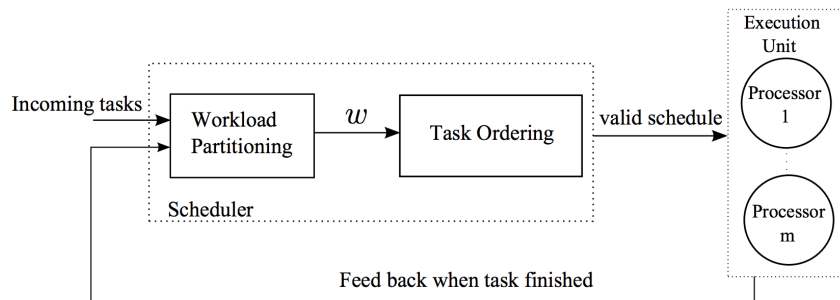


Figure 3.1: Feedback Scheduling Architecture

The scheduling problem is shown to be computationally tractable when divided into two sub-problems: Workload Partitioning and Task Ordering. These two functional blocks are executed sequentially such that the output of the Workload Partitioning module forms the input to the Task Ordering module.

For completeness, we reiterate the following assumptions on the scheduling framework shown in Figure 3.1. Firstly, we assume the availability of multiple homogeneous processors. These processors have the same set of operating frequencies that can be adjusted individually. We assume that no incoming task has any precedence requirements i.e. tasks are ready to start immediately. We also assume that task migration is allowed, and that no delay is incurred in suspending execution on one processor and resuming on another.

3.2 Workload Partitioning

The workload partitioning module is responsible for determining the proportion of a particular time interval that should be dedicated to the execution of each incoming task at every available processor speed level. This module partitions the requested workload such that the devised schedule is both feasibility and energy optimal. It makes decisions using task deadline and power consumption estimates that are continuously updated using feedback information from the Execution Unit.

Solving the workload partitioning problem described above involves solving an optimisation problem with an energy minimisation objective and appropriate physical constraints. For practical systems with discrete frequency levels, we may formulate the workload partitioning problem as a discrete-time linear program.

We will first introduce the mathematical notation that will be used throughout this project, followed by a formal description and explanation of the optimisation problem that is solved within the workload partitioning block..

3.2.1 Notation

We use the following notation to define the problem size: The variables N , n , and l denote the number of discrete time steps, number of tasks in the incoming task set and the number of available speed levels respectively. The total number of processors is denoted by m .

Because we are working with discrete time steps, integer number of tasks and discrete speed levels, we must define integer sets that help us index through these quantities. Let $K = \{k : k \in [0, N - 1]\}$, $I = \{i : i \in [1, n]\}$ and $Q = \{q : q \in [1, l]\}$.

Each point on the discrete time grid is represented by $\tau[k] \forall k \in K \cup \{N\}$. The τ_k notation is also used in this work to represent the same quantity. The grid comprises of N discrete time steps, and $N + 1$ indices from $k = 0$ to $k = N$. Let $T_i \forall i \in I$ denote each task in the task set. The variable $s^q \forall q \in Q$ represents one of the l normalised speed levels available on every processor.

Let b_i represent the arrival time for each task T_i , and d_i represent the corresponding estimated task duration. Therefore, the arrival and deadline times (in seconds) for each task are denoted by b_i , and $b_i + d_i$ respectively.

The functions $\Phi_b(\cdot)$ and $\Phi_d(\cdot)$ perform the task arrival and task deadline mapping respectively i.e. $\Phi_b(T_i) = k_1$ such that $\tau[k_1] = b_i$, and $\Phi_d(T_i) = k_2$ such that $\tau[k_2] = b_i + d_i$. Therefore, the time-grid $\tau[k]$ is partitioned by the task arrival and deadline times.

The P_{idle} variable models power consumption when the system is performing no work, whereas the $P(s^q)$ term marks the active power consumption quantity at a processor speed of s^q .

3.2.2 Problem Definition

In addition to the notation introduced previously, we must also define the decision variables in our discrete-time optimal control problem as follows. The state $x_i[k]$ represents the remaining execution time of some task T_i at $\tau[k]$. The input $a_i^q[k]$ represents the fraction of the interval $[\tau[k], \tau[k+1]]$ during which some task T_i is determined to be executed at the speed level s^q . With this complete set-up, the discrete-time optimal control problem is written as follows:

$$\min_{x_i[k], a_i^q[k]} \sum_{k=0}^{N-1} (\tau[k+1] - \tau[k]) \cdot \sum_{i=1}^n \sum_{q=1}^l (P(s^q) - P_{idle}) a_i^q[k] \quad (3.1a)$$

subject to:

$$x_i[\Phi_b(T_i)] = \underline{x}_i \quad \forall i \in I \quad (3.1b)$$

$$x_i[\Phi_d(T_i)] = 0 \quad \forall i \in I \quad (3.1c)$$

$$x_i[k+1] = x_i[k] - (\tau[k+1] - \tau[k]) \sum_{q=1}^l s_q a_i^q[k] \quad \forall k \in K, \forall i \in I \quad (3.1d)$$

$$\sum_{q=1}^l a_i^q[k] \leq 1 \quad \forall k \in K, \forall i \in I \quad (3.1e)$$

$$\sum_{i=1}^n \sum_{q=1}^l a_i^q[k] \leq m \quad \forall k \in K \quad (3.1f)$$

$$0 \leq a_i^q[k] \leq 1 \quad \forall k \in K, \forall i \in I, \forall q \in Q \quad (3.1g)$$

In simple terms, this formulation translates to the following: For every task in the task set, find *when*, for *how long* and at *what speed level* each task should execute on the time grid such that the total energy consumption is minimised, and none of the equality and/or inequality constraints are violated.

Equation (3.1a) specifies the cost function we would like to minimise. This is a linear function in terms of the input $a_i^q[k]$ where we are penalising the use of higher processor speed levels that consume more energy.

The six constraints are interpreted as follows:

- (3.1b) When a task first arrives at the scheduler, its remaining execution time is set to its estimated execution time \underline{x}_i .
- (3.1c) The remaining execution time of a task at its deadline must equal zero.
- (3.1d) This constraint defines the discrete time scheduling dynamic. The remaining time of a task at the next immediate time step $x_i[k + 1]$ is equal to the remaining time of the task at the current step $x_i[k]$ subtracted by the length of time the task is determined to be executed in the interval.
- (3.1e) The execution of each task is restricted to a single processor at every time step.
- (3.1f) This constraint requires that the total system workload is not exceeded at any time step.
- (3.1g) The input variable $a_i^q[k]$ must be restricted between 0 and 1 since it represents a proportion of a time step.

Table 3.1, 3.2 and 3.3 summarise the nomenclature introduced in this section.

Expression	Description
N	Number of Discrete Time Steps
n	Number of Tasks
l	Number of Available Speed Levels
m	Number of Available Processors

Table 3.1: Problem Size Parameters

Expression	Description
K	Set of integers $K = \{0, 1, \dots, N - 1\}$ used to index time steps
I	Set of integers $I = \{1, 2, \dots, n\}$ used to index tasks
Q	Set of integers $Q = \{1, 2, \dots, l\}$ used to index normalised speed levels
$\tau[k]$ or τ_k	Time (s) on the discrete time grid at step $k \in K$
T_i	Task $i \in I$
s^q	Normalised speed level where $q \in Q$
b_i	Arrival time for T_i
d_i	Task duration for T_i
$\Phi_b(\cdot)$	Task arrival function
$\Phi_d(\cdot)$	Task deadline function
P_{idle}	Idle power consumption
$P(s^q)$	Active power consumption at speed level s^q

Table 3.2: Additional Problem Parameters

Expression	Description
$x_i[k]$	Remaining execution time of some task T_i at τ_k
$a_i^q[k]$	Fraction of the interval $[\tau_k, \tau_{k+1}]$ during which some task T_i is determined to be executed at the speed level s^q

Table 3.3: Decision Variables

3.3 A Note on Task Ordering

Using the solution to the workload partitioning problem, we must subsequently generate a valid execution schedule on all processors such that no task is executed across the different processors in parallel at any given time. This problem is addressed in Thammawichai & Kerrigan (2016) with the use of the McNaughtons wrap-around algorithm (McNaughton 1959). However, we will not present any discussions regarding task ordering as it falls beyond the scope of this work.

Chapter 4

The Workload-Partitioning Fixed-Deadline Problem (WP-FDP)

4.1 Introduction

In this chapter, we explore the fixed-deadline workload-partitioning problem where all tasks have a fixed deadline at τ_N . We build upon the general form optimal control problem introduced in Chapter 3, and extend those ideas to the formulation of a linear program that is fully defined in terms of linear algebraic operators. This step enables us to design, implement and evaluate the scaling performance of both off-the-shelf and custom written optimisation algorithms. The WP-FDP is first solved using the in-built `linprog` routine from MATLAB. Hereafter, the problem is further explored and solved using a custom written IPM that is based on Mehrotra's predictor-corrector algorithm. Our primary focus remains fixated on driving down the relatively expensive computational cost of calculating search directions (using Newton's method) at each IPM iteration.

4.2 Problem Definition

This first step in solving the WP-FDP in software is to define the linear program in terms of tractable numerical operations. Therefore, we start with an introduction of new linear algebra nomenclature. Hereafter, the WP-FDP is formally stated in terms of this new notation. Note that the additional definitions developed here are valid throughout the rest of this work.

Let $\mathbf{x}_k \in \mathbb{R}^{n \times 1}$ represent the cascaded sequence of state variables $x_i[k]$ for $k = 0, 1, \dots, N$. Each \mathbf{x}_k vector contains n elements corresponding to the remaining execution time of all tasks at time step k . This column vector is depicted in (4.1). Moreover, let $\hat{\mathbf{x}} \in \mathbb{R}^{n \times 1}$ denote the estimated remaining execution time at $k = 0$.

$$\mathbf{x}_k = \begin{bmatrix} x_1[k] & x_2[k] & \dots & x_n[k] \end{bmatrix}^T \quad (4.1)$$

Let $\mathbf{a}_k \in \mathbb{R}^{nl \times 1}$ represent the cascaded sequence of input variables $a_i^q[k]$ for $k = 0, 1, \dots, N-1$. Each \mathbf{a}_k vector contains nl elements that denote the proportion of time each task has been determined to execute (at every available speed level permutation) in the interval $[\tau_k, \tau_{k+1}]$. We group the $a_i^q[k]$ terms by task number first, and then by the speed levels as shown in (4.2).

$$\mathbf{a}_k = [a_1^1[k] \ a_1^2[k] \ \dots \ a_1^l[k] \ a_2^1[k] \ \dots \ a_n^l[k]]^T \quad (4.2)$$

Let $\mathbf{s} \in \mathbb{R}^{l \times 1}$ represent an array of all available speed levels (4.3). These values do not vary with time. As a result, we leave out the time step subscript k on this vector.

$$\mathbf{s} = [s^1 \ s^2 \ \dots \ s^l]^T \quad (4.3)$$

Let $\mathbf{P} \in \mathbb{R}^{l \times 1}$ represent the monotonically increasing sequence of net power consumption values at each speed level (4.4). The power consumption model is assumed to be time invariant.

$$\mathbf{P} = [P(s^1) - P_{idle} \ P(s^2) - P_{idle} \ \dots \ P(s^l) - P_{idle}]^T \quad (4.4)$$

The matrix $\mathbf{B} \in \mathbb{R}^{n \times nl}$ encapsulates the discrete-time scheduling dynamic. This matrix appears in the equality constraint $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{B}\mathbf{a}_k$ for $k = 0, 1, \dots, N-1$. Note that the matrix $\mathbf{I}_n \in \mathbb{R}^{n \times n}$ denotes the identity matrix, and the \otimes operator corresponds to the Kronecker product between two objects. The definition for \mathbf{B} is shown in (4.5).

$$\mathbf{B} = -(\tau_{k+1} - \tau_k)(\mathbf{I}_n \otimes \mathbf{s}^T) \quad (4.5a)$$

$$= \begin{bmatrix} -(\tau_1 - \tau_0)\mathbf{s}^T & & & \\ & -(\tau_2 - \tau_1)\mathbf{s}^T & & \\ & & \ddots & \\ & & & -(\tau_N - \tau_{N-1})\mathbf{s}^T \end{bmatrix} \quad (4.5b)$$

The matrix $\mathbf{D} \in \mathbb{R}^{n_d \times nl}$ and the column vector $\mathbf{d} \in \mathbb{R}^{n_d \times 1}$ define three inequality constraints on the input \mathbf{a}_k in the form $\mathbf{D}\mathbf{a}_k - \mathbf{d} \leq \mathbf{0}_{n_d}$ for $k = 0, 1, \dots, N-1$ as shown in (4.6) and (4.7). The vectors $\mathbf{1}_n \in \mathbb{R}^{n \times 1}$ and $\mathbf{0}_n \in \mathbb{R}^{n \times 1}$ correspond to column vectors populated

with ones and zeros respectively. Note that $n_d := n + 1 + nl$.

$$\mathbf{D} = \begin{bmatrix} \mathbf{I}_n \otimes \mathbf{1}_l^T \\ \mathbf{1}_{nl}^T \\ -\mathbf{I}_{nl} \end{bmatrix} = \begin{bmatrix} 1 & \cdots & 1 & & & & & & & & \\ & & & 1 & \cdots & 1 & & & & & \\ & & & & & & \ddots & & & & \\ & & & & & & & 1 & \cdots & 1 & \\ 1 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 1 \\ -1 & & & & & & & & & & \\ & -1 & & & & & & & & & \\ & & \ddots & & & & & & & & \\ & & & \ddots & & & & & & & \\ & & & & \ddots & & & & & & \\ & & & & & \ddots & & & & & \\ & & & & & & \ddots & & & & \\ & & & & & & & \ddots & & & \\ & & & & & & & & \ddots & & \\ & & & & & & & & & \ddots & \\ & & & & & & & & & & -1 \end{bmatrix} \quad (4.6)$$

$$\mathbf{d} = \begin{bmatrix} \mathbf{1}_n \\ m \\ \mathbf{0}_{nl} \end{bmatrix} = \begin{bmatrix} 1 \\ \vdots \\ 1 \\ m \\ 0 \\ \vdots \\ \vdots \\ 0 \end{bmatrix} \quad (4.7)$$

The first inequality constraint $(\mathbf{I}_n \otimes \mathbf{1}_l^T) \mathbf{a}_k \leq \mathbf{1}_n$ for $k = 0, 1, \dots, N - 1$ segments the input variable \mathbf{a}_k into n sub-vectors of length l , and requires that the sum of each sub-vectors is upper bounded by 1. This implies that each task can be executed for a maximum of one time interval i.e. the parallel execution of any particular task across multiple processors is prohibited.

The second constraint $\mathbf{1}_{nl}^T \mathbf{a}_k \leq m$ for $k = 0, 1, \dots, N - 1$ asserts an upper bound on the sum of each \mathbf{a}_k vector. This is a system capacity constraint in terms of the number of available processors.

The third constraint $-\mathbf{I}_{nl} \mathbf{a}_k \leq \mathbf{0}_{nl}$ for $k = 0, 1, \dots, N - 1$ defines a lower bound of 0 on every component of \mathbf{a}_k at each stage on the time grid. This positivity requirement stems from the fact that the scalar quantity $a_i^q[k]$ is proportion measure.

With this set-up, the discrete-time optimal control problem is formally written as follows:

$$\min_{\mathbf{x}_k, \mathbf{a}_k} \sum_{k=0}^{N-1} (\tau_{k+1} - \tau_k) (\mathbf{1}_n^T \otimes \mathbf{P}^T) \mathbf{a}_k \quad (4.8a)$$

subject to:

$$\mathbf{x}_0 = \hat{\mathbf{x}} \quad (4.8b)$$

$$\mathbf{x}_N = \mathbf{0}_n \quad (4.8c)$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{B}\mathbf{a}_k \quad k = 0, 1, \dots, N-1 \quad (4.8d)$$

$$\mathbf{D}\mathbf{a}_k - \mathbf{d} \leq \mathbf{0}_{n_d} \quad k = 0, 1, \dots, N-1 \quad (4.8e)$$

Equation (4.8a) specifies the cost function we would like to minimise. This quantity is a weighted sum of the input vectors \mathbf{a}_k . It is also a measure of total energy consumption. For completeness, we restate the intended interpretation of the four constraints as follows:

- (4.8b) All n tasks arrive at τ_0 . The remaining execution times of all tasks on arrival \mathbf{x}_0 is set to the vector of estimated execution times $\hat{\mathbf{x}}$.
- (4.8c) All n tasks are due at τ_N . The remaining execution times of all tasks at their deadlines \mathbf{x}_N must equal zero.
- (4.8d) This constraint defines the discrete-time evolution of the remaining execution times for every task. The product $\mathbf{B}\mathbf{a}_k$ corresponds to a vector of times spent executing each task in the interval $[\tau_k, \tau_{k+1}]$. Therefore, the remaining execution time of every task at the next time step \mathbf{x}_{k+1} must equal $\mathbf{x}_k + \mathbf{B}\mathbf{a}_k$.
- (4.8e) This constraint summarises the restrictions placed on the input vectors \mathbf{a}_k into a single inequality. These restrictions prevent parallel computation of tasks across different processors (and place an implicit upper bound of one on the input values as a result), take into account the total number of available processors in the system, and impose a positivity requirement on the input values.

Lastly, note that we will divide up the time grid between the first and last point uniformly. This helps simplify several aspects of the software implementation without loss of generality. To this end, we define a new scalar constant $\bar{\tau} := \tau_{k+1} - \tau_k$ for all $k = 0, 1, \dots, N-1$.

4.3 Solving WP-FDP with LINPROG

In this section, we will obtain a solution to the WP-FDP (4.8) using the `linprog` function from the Optimisation Toolbox in MATLAB. This function solves linear programming problems using one of three available optimisation algorithms. The default is the dual-simplex method, which effectively performs the simplex method on the dual problem. Alternatively, the user can choose from one of the two very similar interior-point methods based on Mehrotra's predictor-corrector method. These options are titled

`interior-point-legacy` and `interior-point`. We make use of the latter choice in all the simulations in this section.

The non-legacy variant performs clever preprocessing to simplify the linear program and select a suitable initial point. The predictor-corrector search directions are obtained by carrying out a series of block eliminations, solving the reduced system of linear equations, and back-substituting the result to construct the full solution. While the algorithm primarily relies on a Cholesky factorisation of the left-hand side matrix to solve the reduced system of linear equations, the exact method used is actually dependent on the sparsity structures of the matrices involved.

The `linprog` function provides an easy-to-use framework for solving linear programs so that the user does not have to worry about the low-level implementation details. In addition to the embedded optimisation algorithm, the MATLAB code comprises of several auxiliary routines in order to extract structural information on the problem. This adaptive approach helps maintain practical times to convergence for a broad range of problems.

4.3.1 Conversion to Standard Form

The `linprog` function solves the linear program shown in (4.9). The decision variable here is denoted by \mathbf{w} . The linear cost function (4.9a) is fully defined by the vector \mathbf{f} . The inequality constraint (4.9b) is defined by the operators \mathbf{A}_{ineq} and \mathbf{b}_{ineq} , whereas the equality constraint (4.9c) is defined by \mathbf{A}_{eq} and \mathbf{b}_{eq} . The vectors $\mathbf{b}_{\text{lower}}$ and $\mathbf{b}_{\text{upper}}$ impose boundary conditions on \mathbf{w} . In order to use the `linprog` function, we must translate the WP-FDP in (4.8) to this standard form.

$$\min_{\mathbf{w}} \quad \mathbf{f}^T \mathbf{w} \quad (4.9a)$$

subject to:

$$\mathbf{A}_{\text{ineq}} \mathbf{w} \leq \mathbf{b}_{\text{ineq}} \quad (4.9b)$$

$$\mathbf{A}_{\text{eq}} \mathbf{w} = \mathbf{b}_{\text{eq}} \quad (4.9c)$$

$$\mathbf{b}_{\text{lower}} \leq \mathbf{w} \leq \mathbf{b}_{\text{upper}} \quad (4.9d)$$

The Objective Function

The first step in translating (4.8) to (4.9) involves defining a new decision variable in an appropriate manner as this has an impact on the sparsity structures of all subsequent definitions. Let $\mathbf{w} \in \mathbb{R}^{\{Nln+(N+1)n\} \times 1}$ represent the concatenated sequence of all input variables \mathbf{a}_k for $k = 0, 1, \dots, N-1$ followed by all state variables \mathbf{x}_k for $k = 0, 1, \dots, N$. This construction is shown in (4.10).

$$\mathbf{w} = [\mathbf{a}_0^T \quad \mathbf{a}_1^T \quad \dots \quad \mathbf{a}_{N-1}^T \mid \mathbf{x}_0^T \quad \mathbf{x}_1^T \quad \dots \quad \mathbf{x}_N^T]^T \quad (4.10)$$

Note that the vertical lines drawn through the definitions in this sub-section are for illustrative purposes only as they help distinguish between the operations on \mathbf{a}_k and \mathbf{x}_k where possible.

The WP-FDP cost function (4.8a) comprises of linear operations on the \mathbf{a}_k vectors only. This is equivalent to including the \mathbf{x}_k vectors and setting the corresponding weights to zero. To this end, we define the vector $\mathbf{f} \in \mathbb{R}^{\{Nln+(N+1)n\} \times 1}$ as shown in (4.11).

$$\mathbf{f} = [\bar{\tau}(\mathbf{1}_n^T \otimes \mathbf{P}^T) \quad \bar{\tau}(\mathbf{1}_n^T \otimes \mathbf{P}^T) \quad \dots \quad \bar{\tau}(\mathbf{1}_n^T \otimes \mathbf{P}^T) \mid \mathbf{0}_n^T \quad \mathbf{0}_n^T \quad \dots \quad \mathbf{0}_n^T]^T \quad (4.11)$$

The Equality Constraint

In the WP-FDP formulation, a total of three equality constraints (4.8b), (4.8c), (4.8d) were specified. We address these constraints separately at first, and then combine the derived linear algebraic objects into the single `linprog` compliant equality constraint (4.9c).

Let the $\mathbf{A}_{\text{eq}}^1 \in \mathbb{R}^{n \times \{Nln+(N+1)n\}}$ and $\mathbf{b}_{\text{eq}}^1 \in \mathbb{R}^{n \times 1}$ operators encode the equality constraint (4.8b) as shown in (4.12). This is a constraint on \mathbf{x}_0 only.

$$\mathbf{A}_{\text{eq}}^1 = [\mathbf{0}_{n \times Nln} \mid \mathbf{I}_n \quad \mathbf{0}_{n \times Nn}] \quad (4.12a)$$

$$\mathbf{b}_{\text{eq}}^1 = \hat{\mathbf{x}} \quad (4.12b)$$

Let the $\mathbf{A}_{\text{eq}}^2 \in \mathbb{R}^{n \times \{Nln+(N+1)n\}}$ and $\mathbf{b}_{\text{eq}}^2 \in \mathbb{R}^{n \times 1}$ operators encode the equality constraint (4.8c) as shown in (4.13). This is a constraint on the state variable \mathbf{x}_N only.

$$\mathbf{A}_{\text{eq}}^2 = [\mathbf{0}_{n \times Nln} \mid \mathbf{0}_{n \times Nn} \quad \mathbf{I}_n] \quad (4.13a)$$

$$\mathbf{b}_{\text{eq}}^2 = \mathbf{0}_n \quad (4.13b)$$

Let the $\mathbf{A}_{\text{eq}}^3 \in \mathbb{R}^{Nn \times \{Nln+(N+1)n\}}$ and $\mathbf{b}_{\text{eq}}^3 \in \mathbb{R}^{Nn \times 1}$ operators describe the equality constraint (4.8d) when rewritten as $\mathbf{x}_k + \mathbf{B}\mathbf{a}_k - \mathbf{x}_{k+1} = \mathbf{0}_n$ for $k = 0, 1, \dots, N-1$. These two quantities are defined in (4.14). This is clearly a constraint on every \mathbf{a}_k and \mathbf{x}_k vector in \mathbf{w} .

$$\mathbf{A}_{\text{eq}}^3 = \left[\begin{array}{ccc|ccc} \mathbf{B} & & & \mathbf{I}_n & -\mathbf{I}_n & \\ & \mathbf{B} & & & \mathbf{I}_n & -\mathbf{I}_n \\ & & \ddots & & & \ddots \\ & & & \mathbf{B} & & \mathbf{I}_n & -\mathbf{I}_n \end{array} \right] \quad (4.14a)$$

$$\mathbf{b}_{\text{eq}}^3 = \mathbf{0}_{Nn} \quad (4.14b)$$

Lastly, the matrix $\mathbf{A}_{\text{eq}} \in \mathbb{R}^{\{n+n+Nn\} \times \{Nln+(N+1)n\}}$ and vector $\mathbf{b}_{\text{eq}} \in \mathbb{R}^{\{n+n+Nn\} \times 1}$ are constructed as shown in (4.15).

$$\mathbf{A}_{\text{eq}} = \begin{bmatrix} \mathbf{A}_{\text{eq}}^1 \\ \mathbf{A}_{\text{eq}}^2 \\ \mathbf{A}_{\text{eq}}^3 \end{bmatrix} = \left[\begin{array}{cccc|cccc} \mathbf{B} & & & & \mathbf{I}_n & & & \\ & \mathbf{B} & & & \mathbf{I}_n & -\mathbf{I}_n & & \\ & & \ddots & & & \mathbf{I}_n & -\mathbf{I}_n & \\ & & & \mathbf{B} & & & & \mathbf{I}_n \\ & & & & & & \ddots & \\ & & & & & & & \mathbf{I}_n \\ & & & & & & & -\mathbf{I}_n \end{array} \right] \quad (4.15a)$$

$$\mathbf{b}_{\text{eq}} = \begin{bmatrix} \mathbf{b}_{\text{eq}}^1 \\ \mathbf{b}_{\text{eq}}^2 \\ \mathbf{b}_{\text{eq}}^3 \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{x}} \\ \mathbf{0}_n \\ \mathbf{0}_{Nn} \end{bmatrix} \quad (4.15b)$$

The Inequality Constraint

Recall that the single inequality constraint (4.8e) in the WP-FDP actually encompasses three separate requirements on the \mathbf{a}_k vectors. We will explicitly encode the first two requirements (non-parallel execution rule and maximum system capacity) as inequality constraints on \mathbf{w} at first, and then combine the derived objects together to form the single `linprog` compliant inequality constraint (4.9b).

Note that the third requirement is a positivity constraint on \mathbf{a}_k . This is implemented through the use of the dedicated upper-bound vector $\mathbf{b}_{\text{upper}}$ instead.

Let the $\mathbf{A}_{\text{ineq}}^1 \in \mathbb{R}^{Nn \times \{Nln+(N+1)n\}}$ and $\mathbf{b}_{\text{ineq}}^1 \in \mathbb{R}^{Nn \times 1}$ operators encode the constraint $(\mathbf{I}_n \otimes \mathbf{1}_l^T) \mathbf{a}_k \leq \mathbf{1}_n$ for $k = 0, 1, \dots, N-1$ as shown in (4.16).

$$\mathbf{A}_{\text{ineq}}^1 = \left[\begin{array}{cccc|} (\mathbf{I}_n \otimes \mathbf{1}_l^T) & & & \\ & (\mathbf{I}_n \otimes \mathbf{1}_l^T) & & \\ & & \ddots & \\ & & & (\mathbf{I}_n \otimes \mathbf{1}_l^T) \end{array} \right] \quad (4.16a)$$

$$\mathbf{b}_{\text{eq}}^1 = \mathbf{1}_{Nn} \quad (4.16b)$$

Let the $\mathbf{A}_{\text{ineq}}^2 \in \mathbb{R}^{N \times \{Nln+(N+1)n\}}$ and $\mathbf{b}_{\text{ineq}}^2 \in \mathbb{R}^{N \times 1}$ operators encode the constraint $\mathbf{1}_{nl}^T \mathbf{a}_k \leq m$ for $k = 0, 1, \dots, N-1$ as shown in (4.17)

$$\mathbf{A}_{\text{ineq}}^2 = \left[\begin{array}{cccc|c} \mathbf{1}_{nl}^T & & & & \\ & \mathbf{1}_{nl}^T & & & \\ & & \ddots & & \\ & & & \mathbf{1}_{nl}^T & \end{array} \right] \quad (4.17a)$$

$$\mathbf{b}_{\text{eq}}^2 = m \cdot \mathbf{1}_N \quad (4.17b)$$

Lastly, the matrix $\mathbf{A}_{\text{ineq}} \in \mathbb{R}^{\{Nn+N\} \times \{Nln+(N+1)n\}}$ and vector $\mathbf{b}_{\text{ineq}} \in \mathbb{R}^{\{Nn+N\} \times 1}$ are constructed as shown in (4.18)

$$\mathbf{A}_{\text{ineq}} = \begin{bmatrix} \mathbf{A}_{\text{ineq}}^1 \\ \mathbf{A}_{\text{ineq}}^2 \end{bmatrix} = \left[\begin{array}{cccc|c} (\mathbf{I}_n \otimes \mathbf{1}_l^T) & & & & \\ & (\mathbf{I}_n \otimes \mathbf{1}_l^T) & & & \\ & & \ddots & & \\ & & & (\mathbf{I}_n \otimes \mathbf{1}_l^T) & \\ \mathbf{1}_{nl}^T & & & & \\ & \mathbf{1}_{nl}^T & & & \\ & & \ddots & & \\ & & & \mathbf{1}_{nl}^T & \end{array} \right] \quad (4.18a)$$

$$\mathbf{b}_{\text{ineq}} = \begin{bmatrix} \mathbf{b}_{\text{ineq}}^1 \\ \mathbf{b}_{\text{ineq}}^2 \end{bmatrix} = \begin{bmatrix} \mathbf{1}_{Nn} \\ m \cdot \mathbf{1}_N \end{bmatrix} \quad (4.18b)$$

The Boundary Conditions

At this point, we complete the conversion to the `linprog` standard form by defining the lower bound vector $\mathbf{b}_{\text{lower}} \in \mathbb{R}^{\{Nln+(N+1)n\} \times 1}$ and the upper bound vector $\mathbf{b}_{\text{upper}} \in \mathbb{R}^{\{Nln+(N+1)n\} \times 1}$ as shown in (4.19).

$$\mathbf{b}_{\text{lower}} = \mathbf{0}_{Nln+(N+1)n} \quad (4.19a)$$

$$\mathbf{b}_{\text{upper}} = \left[\mathbf{1}_{Nln}^T \mid (\mathbf{1}_{(N+1)}^T \otimes \hat{\mathbf{x}}^T) \right]^T \quad (4.19b)$$

4.3.2 Simulation Results and Analysis

Having defined the exact equivalence between the WP-FDP (4.8) and the `linprog` standard form (4.9) in the previous sub-section, we are now in a position to present the corresponding MATLAB simulation results. Firstly, we outline the following technical details that hold true throughout this work.

1. All simulations were carried out within the MATLAB 2016b environment on a 2.5GHz Intel i5 system with 8GB of DDR3 RAM.
2. Sparse matrices were constructed using the MATLAB `sparse` routines in order to limit memory requirements where possible.
3. The initial execution time estimates, denoted by the vector $\hat{\mathbf{x}}$, are randomly generated using a uniform distribution in the range $[1, 6]$ seconds.
4. The time grid is divided uniformly between the arrival and deadline times, and the step length between any two consecutive arbitrary points $\bar{\tau}$ is fixed at 5 seconds.
5. The net power consumption is modelled as a quadratic function of the speed-levels, offset by a fixed DC value.

The focus here lies on identifying how the problem scales with an increasing number of problem size parameters. Namely, we are interested in the number of iterations and total time to convergence for an increasing number of tasks n , number of time-steps N and number of discrete speed levels l .

We begin with a visual demonstration of the feasibility of the `linprog` solution for a set-up that is demanding of the system. This experiment is run for $(n, N, l) = (140, 10, 10)$ with $m = 10$ available processors.

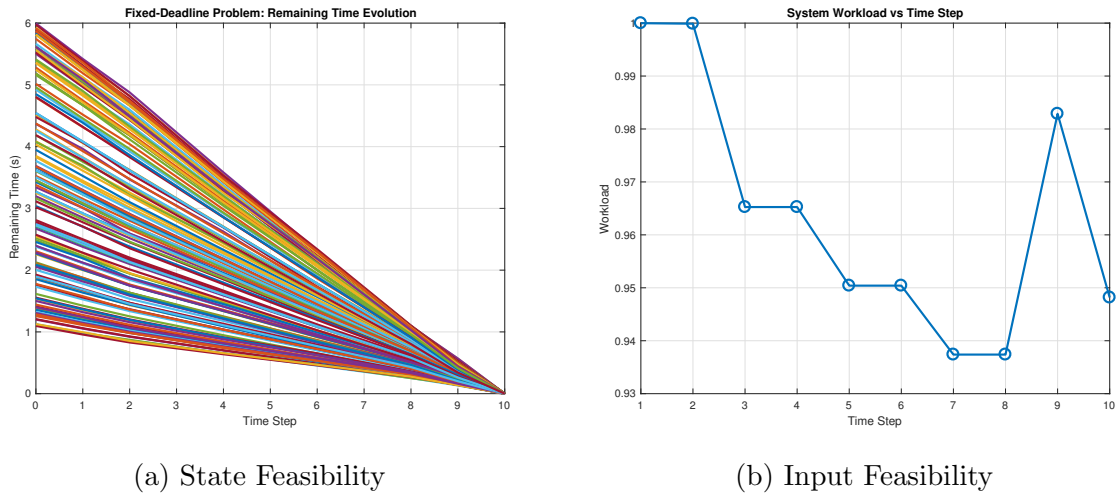


Figure 4.1: Remaining Time-Evolution and System Workload Plots [FDP-LINPROG]

Figure 4.1a depicts a plot of the remaining execution times of all 140 tasks at each point on the time grid. This simple figure provides an intuitive visualisation of the problem. It also serves to show that none of the equality constraints on the state variables are violated at any stage. This in turn proves that the problem has been correctly formulated and translated into the `linprog` standard form.

Figure 4.1b illustrates a plot of the system workload at each point on the time grid. The system workload at some stage k is calculated by summing together the components of the vector \mathbf{a}_k and dividing by the total number of available processors m . This figure

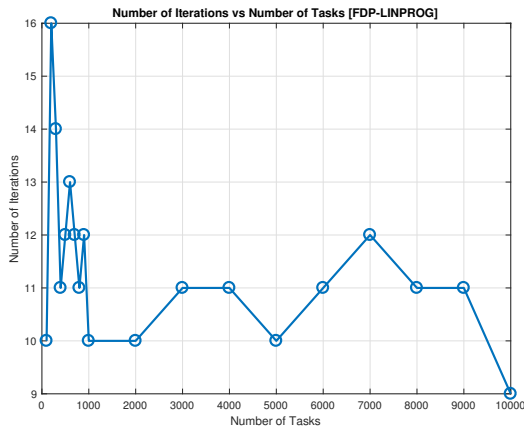
effectively shows that the system capacity constraint (4.17) is not violated, and that the solver does not perform unexpectedly when solutions lie near the boundaries of the feasible set.

We now present results from three separate simulations. In each experiment, we vary one of the the three problem size parameters while keeping the other quantities fixed. Three plots from each simulation are shown. The first two figures show the number of iterations required to arrive within a fixed tolerance of the solution, and the total time taken to solve the problem respectively.

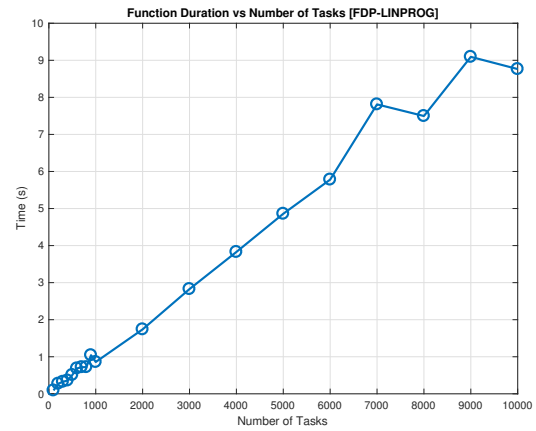
The third figure corresponds to a log-log plot of the time taken per iteration for an increasing level of the particular parameter under investigation. This last figure in each simulation provides an useful insight into how the computation complexity of the `linprog` solver scales with increasingly larger problems.

Increasing Number of Tasks

This experiment is run for $(N, l) = (10, 10)$ with $m = 1000$ available processors. A sufficiently large number of processors is chosen in order to ensure that the problem remains feasible throughout the experiment. A relatively small value for the number of step lengths and speed levels is chosen in order to render a broad range of results within a reasonable length of time, and to to emphasise the impact of an increasing number of tasks alone. We cycle through values of n in the range $[100, 10000]$.



(a) Iteration Count



(b) Function Duration

Figure 4.2: Number of Iterations and Function Duration vs Number of Tasks [FDP-LINPROG]

Figure 4.2a and Figure 4.2b illustrate the iteration count and function duration plots from this simulation. No clear trends are observed in the iteration count plots as the interior-point algorithm appears to loop over roughly 10 – 15 iterations regardless of the number of tasks. The function duration plot suggests a linear relationship between the time to convergence and the number of tasks.

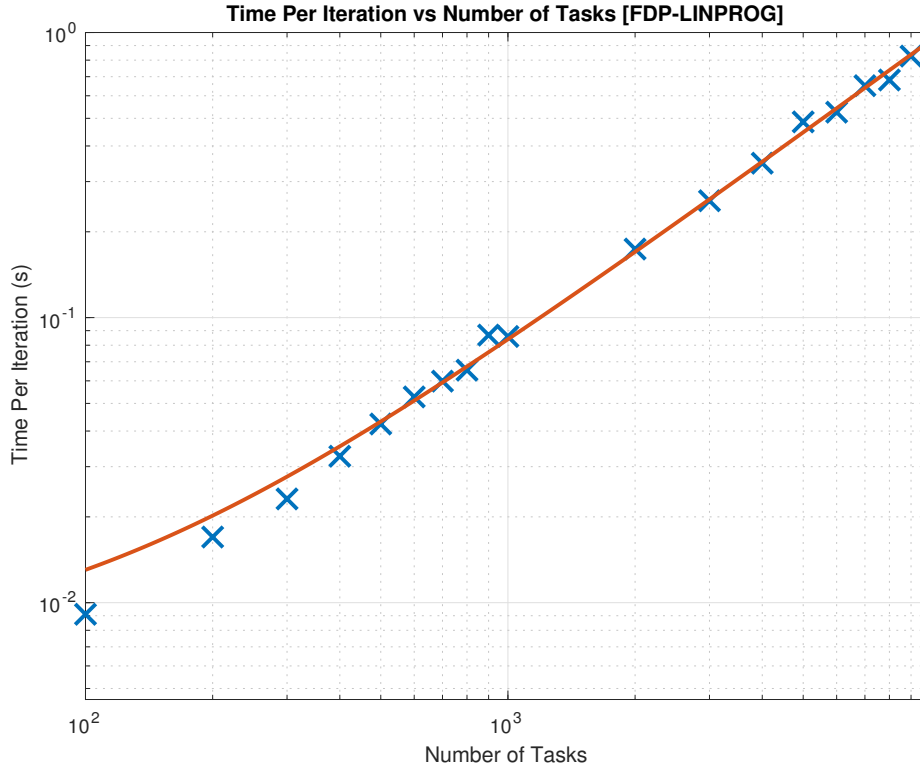


Figure 4.3: Time per iteration vs Number of Tasks [FDP-LINPROG]

Figure 4.3 illustrates a logarithmic plot of the time taken per iteration versus the number of tasks in the system. A roughly linear relationship between the two quantities is observed, particularly at the latter end of the spectrum. The red line corresponds to a line of best fit that is produced using a two-term power series model within the `fit` function. The equation of this line is given as follows

$$\text{Time/Iteration} = 0.000044 \cdot (\# \text{ of Tasks})^{1.080146} + 0.006607 \quad (4.20)$$

An exponent of 1.080146 in conjunction with a comparatively small add-on value of 0.006607 implies that the gradient of the line on the log-log plot is approximately 1 for a very large number of tasks. This in turn reaffirms our linear relationship claim.

Increasing Number of Steps

This experiment is run for $(n, l) = (10, 10)$ with $m = 100$ available processors. We cycle through values of N in the range $[100, 10000]$.

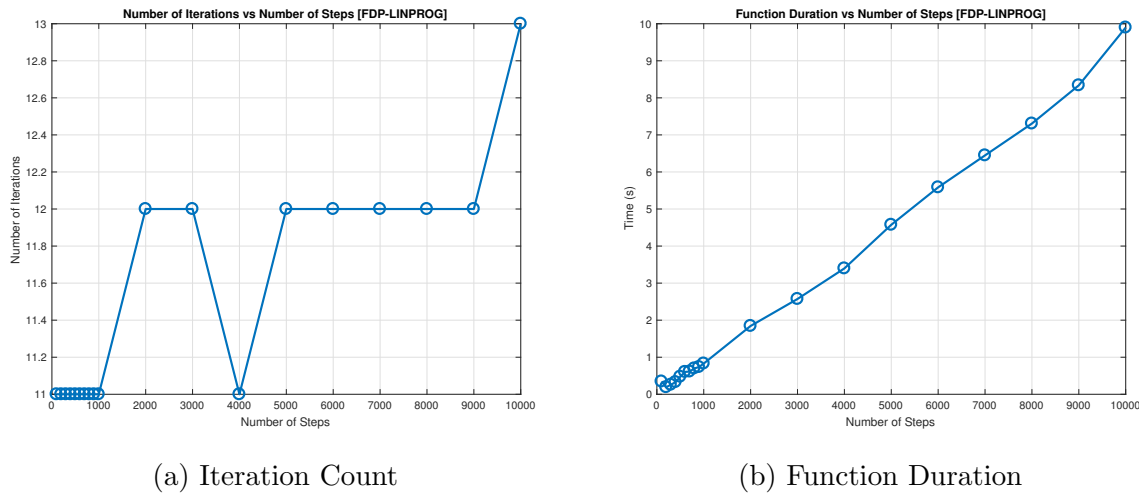


Figure 4.4: Number of Iterations and Function Duration vs Number of Steps [FDP-LINPROG]

Figure 4.4a and Figure 4.4b illustrate the iteration count and function duration plots from this simulation. These plots showcase very similar results to those obtained in the previous experiment. Namely, no stand-out trends emerge in the iteration count plot for this selection of values. Note that the function duration plot shows a clear linearity with an increasing number of tasks.

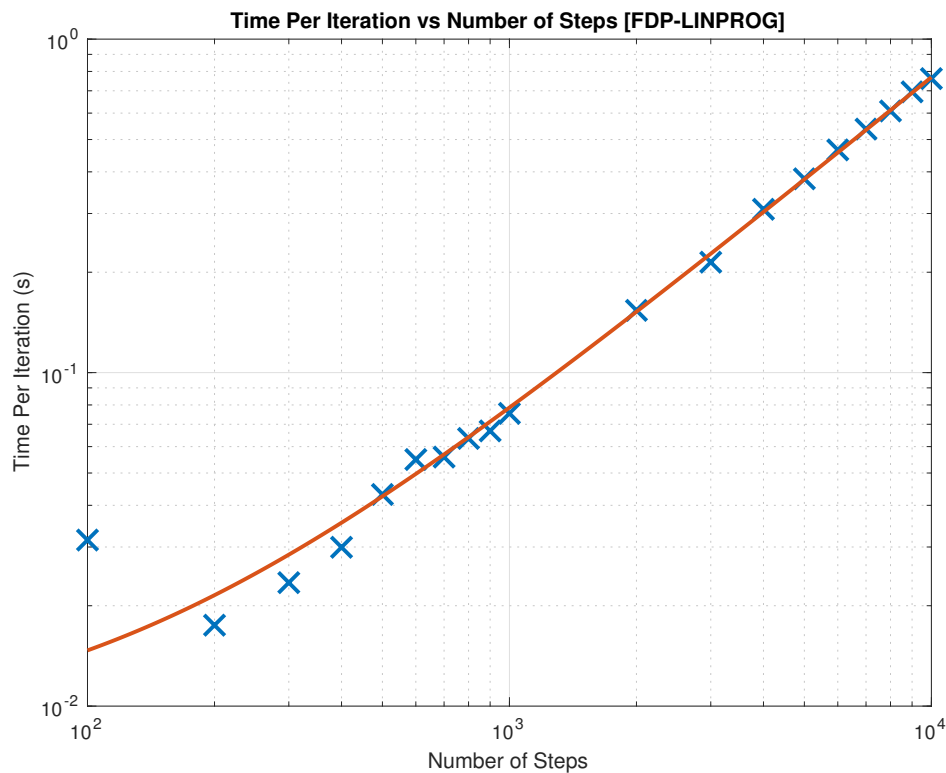


Figure 4.5: Time per iteration vs Number of Steps [FDP-LINPROG]

Figure 4.5 illustrates a logarithmic plot of the time taken per iteration against the number

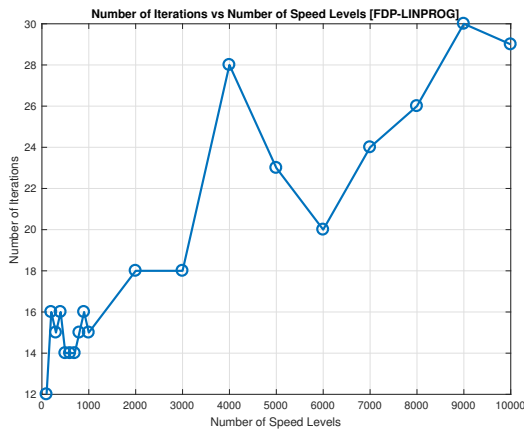
of steps on the time grid. The dotted grid lines help emphasise the almost perfect linear trend between the two variables. The equation for the line of best fit is given as follows

$$\text{Time/Iteration} = 0.000056 \cdot (\# \text{ of Time Steps})^{1.033659} + 0.008177 \quad (4.21)$$

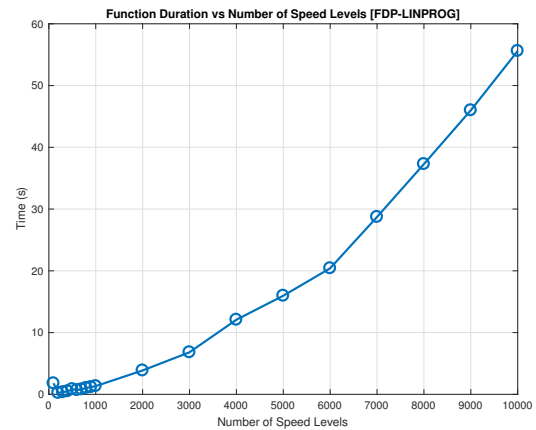
Increasing Number of Speed Levels

This experiment is run for $(n, N) = (10, 10)$ with $m = 100$ available processors. We cycle through 19 values of l in the range $[100, 10000]$.

Figure 4.6a and Figure 4.6b illustrate the iteration count and function duration plots from this simulation. The iteration count plot depicts a comparatively higher variance in the number of iteration values. In general, the number of iterations is not confined to a small range as was the case in the previous two experiments, and the two quantities appear to be positively correlated. The function duration plot resembles a parabolic trend as opposed to a perfectly linear one. The slope of this line is increasing up to $l = 6000$, after which it appears roughly constant.



(a) Iteration Count



(b) Function Duration

Figure 4.6: Number of Iterations and Function Duration vs Number of Speed Levels [FDP-LINPROG]

Figure 4.7 illustrates the relationship between the time taken per iteration versus the number of available speed levels on a logarithmic grid. The equation for the line of best fit is given as follows

$$\text{Time/Iteration} = 0.000006 \cdot (\# \text{ of Speed Levels})^{1.373134} + 0.006684 \quad (4.22)$$

An exponent of 1.373134 points to a complexity result that is neither linear nor quadratic. This indicates an increasing rate at which the time per iteration quantity scales with a growing number of speed levels.

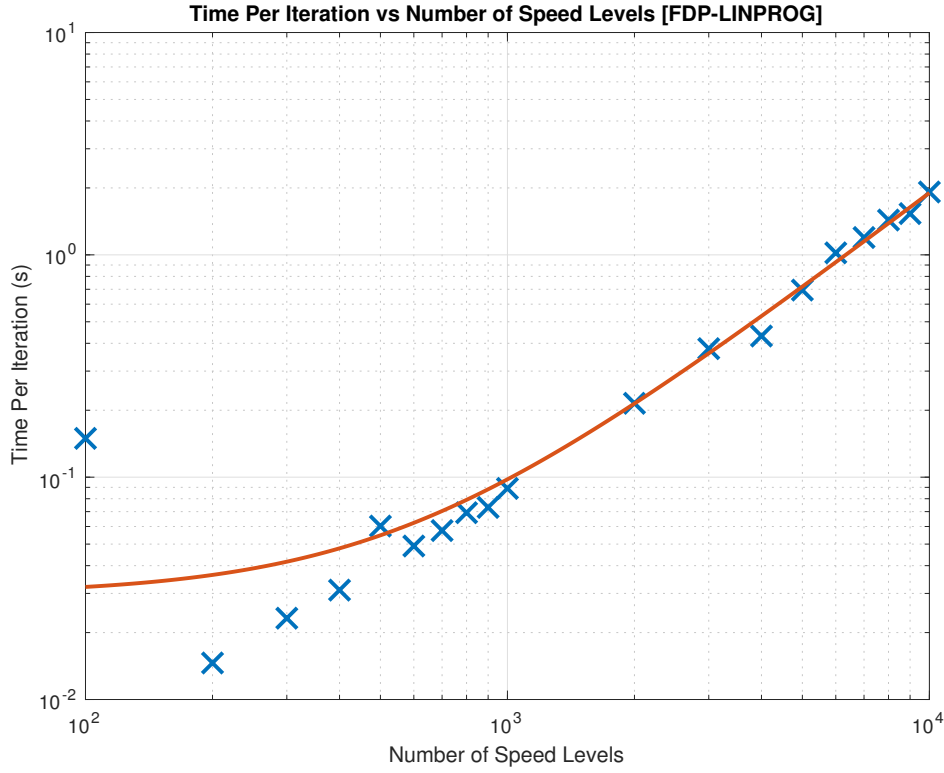


Figure 4.7: Time per iteration vs Number of Speed Levels [FDP-LINPROG]

4.4 The Complete KKT System of Equations

In this section, we formally define the first order conditions of optimality for the WP-FDP problem defined in (4.8), and derive the matrix equation that is solved in Newton's method using these optimality conditions.

The Lagrangian to the WP-FDP in (4.8) is shown in (4.23). The Lagrange multipliers are denoted by $\mathbf{p}_k \in \mathbb{R}^{n \times 1}$, $\boldsymbol{\lambda}_k \in \mathbb{R}^{n_d \times 1}$ and $\boldsymbol{\beta} \in \mathbb{R}^{n \times 1}$. The slack variable is denoted by $\mathbf{t}_k \in \mathbb{R}^{n_d \times 1}$.

$$\begin{aligned}
 \mathcal{L} = & \sum_{k=0}^{N-1} (\tau_{k+1} - \tau_k) (\mathbf{1}_n^T \otimes \mathbf{P}^T) \mathbf{a}_k \\
 & + \sum_{k=0}^{N-1} \mathbf{p}_k^T (\mathbf{x}_k + \mathbf{B} \mathbf{a}_k - \mathbf{x}_{k+1}) \\
 & + \sum_{k=0}^{N-1} \boldsymbol{\lambda}_k^T (\mathbf{D} \mathbf{a}_k - \mathbf{d} + \mathbf{t}_k) \\
 & + \boldsymbol{\beta}^T \mathbf{x}_N
 \end{aligned} \tag{4.23}$$

The Karush-Kuhn-Tucker (KKT) conditions for optimality are derived as follows:

$$\mathbf{r}_k^{\mathbf{x}} = \mathbf{p}_k - \mathbf{p}_{k-1} = \mathbf{0} \quad k = 1, 2, \dots, N-1 \quad (4.24a)$$

$$\mathbf{r}_k^{\mathbf{x}} = \boldsymbol{\beta} - \mathbf{p}_{k-1} = \mathbf{0} \quad k = N \quad (4.24b)$$

$$\mathbf{r}_k^{\mathbf{a}} = (\tau_{k+1} - \tau_k)(\mathbf{1}_{n \times 1} \otimes \mathbf{P}) + \mathbf{B}^T \mathbf{p}_k + \mathbf{D}^T \boldsymbol{\lambda}_k = \mathbf{0} \quad k = 0, 1, \dots, N-1 \quad (4.24c)$$

$$\mathbf{r}_k^{\boldsymbol{\lambda}} = \mathbf{D} \mathbf{a}_k - \mathbf{d} + \mathbf{t}_k = \mathbf{0} \quad k = 0, 1, \dots, N-1 \quad (4.24d)$$

$$\mathbf{r}_k^{\mathbf{p}} = \mathbf{x}_k + \mathbf{B} \mathbf{a}_k - \mathbf{x}_{k+1} = \mathbf{0} \quad k = 0, 1, \dots, N-1 \quad (4.24e)$$

$$\mathbf{r}_k^{\mathbf{t}} = \boldsymbol{\Lambda}_k \mathbf{T}_k \mathbf{e} = \mathbf{0} \quad k = 0, 1, \dots, N-1 \quad (4.24f)$$

$$\mathbf{r}_k^{\boldsymbol{\beta}} = \mathbf{x}_k = \mathbf{0} \quad k = N \quad (4.24g)$$

where $\boldsymbol{\Lambda}_k \in \mathbb{R}^{n_d \times n_d}$ and $\mathbf{T}_k \in \mathbb{R}^{n_d \times n_d}$ denote diagonal matrices with $\boldsymbol{\lambda}_k$ and \mathbf{t}_k elements placed on the main diagonal respectively.

The order of unknowns in Newton's method has an impact on the structure of the left-hand side Jacobian matrix. We arrange the unknown components in a non-condensed interleaved manner that ensures a sparse and banded Jacobian matrix. The corresponding system of linear equations that must be solved to obtain the affine-scaling direction is specified in matrix form as shown in (4.25).

$$\begin{bmatrix} & & \mathbf{D}^T & \mathbf{B}^T & & & \\ \mathbf{D} & & & & \mathbf{I} & & \\ \mathbf{B} & & & & & & \\ & \mathbf{T}_0 & & & & & \\ & & & -\mathbf{I} & & & \\ & & & & \boldsymbol{\Lambda}_0 & & \\ & & & & & & \\ & & & & & & \ddots \end{bmatrix} \begin{bmatrix} \Delta \mathbf{a}_0 \\ \Delta \boldsymbol{\lambda}_0 \\ \Delta \mathbf{p}_0 \\ \Delta \mathbf{t}_0 \\ \Delta \mathbf{x}_1 \\ \vdots \end{bmatrix} = - \begin{bmatrix} \mathbf{r}_0^{\mathbf{a}} \\ \mathbf{r}_0^{\boldsymbol{\lambda}} \\ \mathbf{r}_0^{\mathbf{p}} \\ \mathbf{r}_0^{\mathbf{t}} \\ \mathbf{r}_1^{\mathbf{x}} \\ \vdots \end{bmatrix} \quad (4.25a)$$

$$\begin{bmatrix} \ddots & & & & & & \\ & & & & \mathbf{I} & & \\ & & \mathbf{D}^T & \mathbf{B}^T & & & \\ & \mathbf{D} & & & \mathbf{I} & & \\ \mathbf{I} & \mathbf{B} & & & & & \\ & & \mathbf{T}_k & & & & \\ & & & -\mathbf{I} & & & \\ & & & & \boldsymbol{\Lambda}_k & & \\ & & & & & & \ddots \end{bmatrix} \begin{bmatrix} \vdots \\ \Delta \mathbf{x}_k \\ \Delta \mathbf{a}_k \\ \Delta \boldsymbol{\lambda}_k \\ \Delta \mathbf{p}_k \\ \Delta \mathbf{t}_k \\ \Delta \mathbf{x}_{k+1} \\ \vdots \end{bmatrix} = - \begin{bmatrix} \vdots \\ \mathbf{r}_k^{\mathbf{x}} \\ \mathbf{r}_k^{\mathbf{a}} \\ \mathbf{r}_k^{\boldsymbol{\lambda}} \\ \mathbf{r}_k^{\mathbf{p}} \\ \mathbf{r}_k^{\mathbf{t}} \\ \mathbf{r}_{k+1}^{\mathbf{x}} \\ \vdots \end{bmatrix} \quad (4.25b)$$

$$\begin{bmatrix} \ddots & & & & & & \\ & & & & \mathbf{I} & & \\ & & \mathbf{D}^T & \mathbf{B}^T & & & \\ & \mathbf{D} & & & \mathbf{I} & & \\ \mathbf{I} & \mathbf{B} & & & & & \\ & & \mathbf{T}_{N-1} & & & & \\ & & & -\mathbf{I} & & & \\ & & & & \boldsymbol{\Lambda}_{N-1} & & \\ & & & & & & \mathbf{I} \end{bmatrix} \begin{bmatrix} \vdots \\ \Delta \mathbf{x}_{N-1} \\ \Delta \mathbf{a}_{N-1} \\ \Delta \boldsymbol{\lambda}_{N-1} \\ \Delta \mathbf{p}_{N-1} \\ \Delta \mathbf{t}_{N-1} \\ \Delta \mathbf{x}_N \\ \Delta \boldsymbol{\beta} \end{bmatrix} = - \begin{bmatrix} \vdots \\ \mathbf{r}_{N-1}^{\mathbf{x}} \\ \mathbf{r}_{N-1}^{\mathbf{a}} \\ \mathbf{r}_{N-1}^{\boldsymbol{\lambda}} \\ \mathbf{r}_{N-1}^{\mathbf{p}} \\ \mathbf{r}_{N-1}^{\mathbf{t}} \\ \mathbf{r}_N^{\mathbf{x}} \\ \mathbf{r}^{\boldsymbol{\beta}} \end{bmatrix} \quad (4.25c)$$

The centring direction is computed using the same left-hand side matrix, and apart from the residual vectors corresponding to the slack variable \mathbf{r}_k^t , all remaining components of the modified right-hand side are set to zero. The \mathbf{r}_k^t vectors for $k = 0, 1, \dots, N - 1$ are constructed as per (4.26).

$$\mathbf{r}_k^t = \Delta \mathbf{X}_k^{\text{aff}} \Delta \mathbf{S}_k^{\text{aff}} \mathbf{e} - \sigma \mu \mathbf{e} \quad (4.26)$$

4.5 Solving WP-FDP with Custom IPM

In this section, we will obtain a solution to the WP-FDP using a custom written IPM in MATLAB. This custom implementation is based on Mehrotra's predictor corrector algorithm. As highlighted previously, the overall search direction in this algorithm is obtained using Newton's method twice in each iteration. The two systems of linear equations utilise the same left-hand side Jacobian matrix and only differ in the right-hand side residual vector. The focus remains on solving the KKT system of equations efficiently since they represent the largest computational bottleneck in primal-dual methods.

The optimality conditions derived in the previous section are incorporated into the predictor-corrector algorithm directly. The primary objective in this section is to introduce the KKT system of equations within a correctly functioning IPM implementation that will form the basis of further analysis in the subsequent chapter.

4.5.1 Implementation Details

The first attempt at solving the WP-FDP using a custom written IPM shown here involves constructing the left-hand side Jacobian matrix (4.25) in its entirety in order to compute the predictor and corrector search directions during every iteration. This early implementation does not aim for efficient scaling but rather serves as a springboard for subsequent optimisations.

Figure 4.8a and 4.8b illustrate the structure of the Jacobian matrix for different problem sizes when visualised using the `spy` function in MATLAB. The highly sparse structure arises because the constituent block matrices of the Jacobian are sparse matrices themselves. The Jacobian is also banded, and this is an intended property achieved by appropriately arranging the unknowns in (4.25).

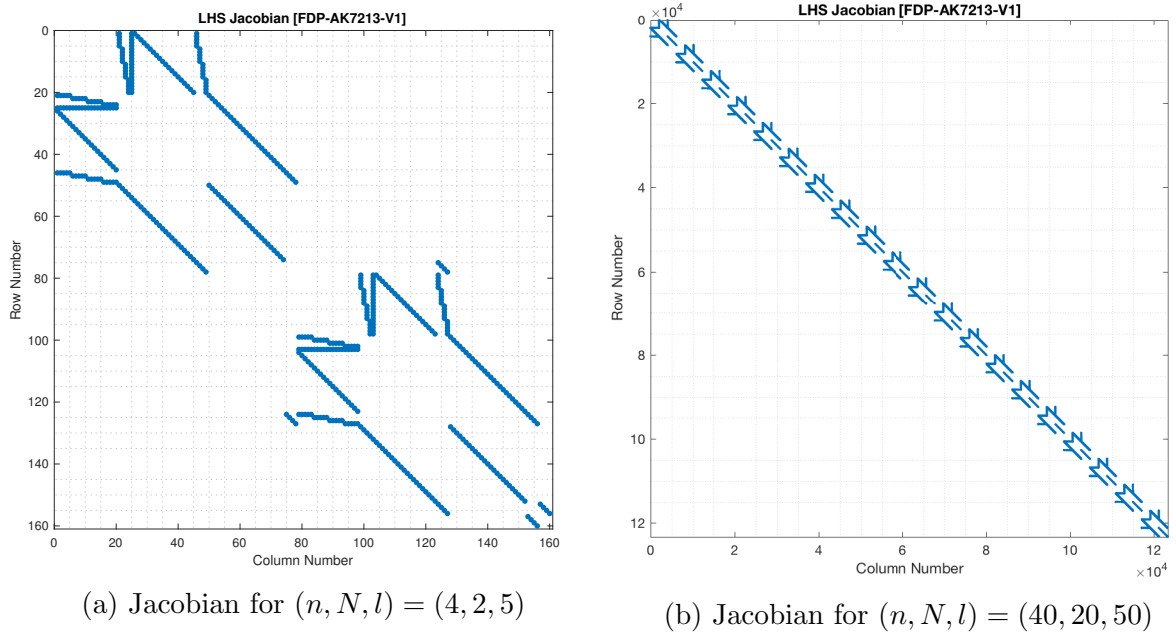


Figure 4.8: Sparsity pattern of Jacobian matrix for different problem sizes
[FDP-AK7213-V1]

The MATLAB procedure that was implemented is described in Algorithm 3. Note that the linear algebraic objects that are constant across every iteration are initialised and stored beforehand in order to circumvent redundant computations. The most important example of this is the Jacobian matrix itself. Since it is only the diagonal matrices $\mathbf{\Lambda}_k$ and \mathbf{T}_k that need to be modified at the start of a new iteration, we pre compute the remaining constituents in advance.

Before carrying out the simulations, the function was profiled using the MATLAB `profile` routine for a problem size of $(n, N, l) = (50, 50, 50)$ and $m = 1000$ available processors. The function returned a feasible solution in 10 iterations. Figure 4.9 illustrates a snippet of the profile report that was generated. The key result here is the significantly high proportion of total time spent on solving the KKT system of equations in order to obtain the predictor and corrector search directions.

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
309	CSOL = LHS \ CRHS;	10	95.077 s	49.3%	<div style="width: 49.3%;"></div>
234	PSOL = LHS \ PRHS;	10	95.074 s	49.3%	<div style="width: 49.3%;"></div>
185	LHS = sparse([Rows; TLRows], ...	10	1.610 s	0.8%	<div style="width: 0.8%;"></div>
273	if(affDlambdas(j,i) < 0)	1275500	0.063 s	0.0%	
355	if(Dlambdas(j,i) < 0)	1275500	0.060 s	0.0%	
All other lines			0.980 s	0.5%	<div style="width: 0.5%;"></div>
Totals			192.864 s	100%	

Figure 4.9: Function Profile Report [FDP-AK7213-V1]

Algorithm 3 FDP-AK7213-V1

```

1: procedure FDP-AK7213-V1( $n, N, l, m, \epsilon, \bar{\tau}, \mathbf{s}, \mathbf{P}, \hat{\mathbf{x}}$ )
2:   initialise  $\mathbf{x}_k > \mathbf{0} \forall k \in [1, N]$ ,  $(\mathbf{a}_k, \boldsymbol{\lambda}_k, \mathbf{p}_k, \mathbf{t}_k) > \mathbf{0} \forall k \in [0, N-1]$  and  $\boldsymbol{\beta} > \mathbf{0}$ 
3:   define constants  $\mathbf{B}, \mathbf{D}$  and  $\mathbf{d}$  as per (4.5), (4.6) and (4.7) respectively
4:   construct the sparse Jacobian matrix (4.25) without  $\boldsymbol{\Lambda}_k$  and  $\mathbf{T}_k$  components
5:   while  $\mu > \epsilon$  do
6:     update the sparse Jacobian matrix (4.25) with  $\boldsymbol{\Lambda}_k$  and  $\mathbf{T}_k$  components
7:     compute predictor residuals  $\mathbf{r}_k^{\mathbf{x}}, \mathbf{r}_k^{\mathbf{a}}, \mathbf{r}_k^{\boldsymbol{\lambda}}, \mathbf{r}_k^{\mathbf{p}}, \mathbf{r}_k^{\mathbf{t}}, \mathbf{r}_k^{\boldsymbol{\beta}}$  as per (4.24)
8:     solve for predictor step  $(\Delta \mathbf{x}_k^{\text{aff}}, \Delta \mathbf{a}_k^{\text{aff}}, \Delta \boldsymbol{\lambda}_k^{\text{aff}}, \Delta \mathbf{p}_k^{\text{aff}}, \Delta \mathbf{t}_k^{\text{aff}}, \Delta \boldsymbol{\beta}^{\text{aff}})$  as per (4.25)
9:      $\alpha^{\text{aff}} = \arg \max \{ \alpha \in [0, 1] : (\boldsymbol{\lambda}_k, \mathbf{t}_k) + \alpha(\Delta \boldsymbol{\lambda}_k^{\text{aff}}, \Delta \mathbf{t}_k^{\text{aff}}) \geq \mathbf{0} \forall k \in [0, N-1] \}$ 
10:     $\mu^{\text{aff}} = \frac{1}{Nn_d} \sum_{k=0}^{N-1} (\boldsymbol{\lambda}_k + \alpha^{\text{aff}} \Delta \boldsymbol{\lambda}_k^{\text{aff}})^T (\mathbf{t}_k + \alpha^{\text{aff}} \Delta \mathbf{t}_k^{\text{aff}})$ 
11:     $\sigma = (\mu^{\text{aff}} / \mu)^3$ 
12:    compute corrector residuals  $\mathbf{r}_k^{\mathbf{x}}, \mathbf{r}_k^{\mathbf{a}}, \mathbf{r}_k^{\boldsymbol{\lambda}}, \mathbf{r}_k^{\mathbf{p}}, \mathbf{r}_k^{\mathbf{t}}, \mathbf{r}_k^{\boldsymbol{\beta}} = \mathbf{0}$  and  $\mathbf{r}_k^{\mathbf{t}} = \Delta \mathbf{X}_k^{\text{aff}} \Delta \mathbf{S}_k^{\text{aff}} \mathbf{e} - \sigma \mu \mathbf{e}$ 
13:    solve for corrector step  $(\Delta \mathbf{x}_k^{\text{cc}}, \Delta \mathbf{a}_k^{\text{cc}}, \Delta \boldsymbol{\lambda}_k^{\text{cc}}, \Delta \mathbf{p}_k^{\text{cc}}, \Delta \mathbf{t}_k^{\text{cc}}, \Delta \boldsymbol{\beta}^{\text{cc}})$  as per (4.25)
14:     $(\Delta \mathbf{x}_k, \Delta \mathbf{a}_k, \Delta \boldsymbol{\lambda}_k, \Delta \mathbf{p}_k, \Delta \mathbf{t}_k, \Delta \boldsymbol{\beta}) = (\Delta \mathbf{x}_k^{\text{aff}}, \Delta \mathbf{a}_k^{\text{aff}}, \Delta \boldsymbol{\lambda}_k^{\text{aff}}, \Delta \mathbf{p}_k^{\text{aff}}, \Delta \mathbf{t}_k^{\text{aff}}, \Delta \boldsymbol{\beta}^{\text{aff}}) \dots$ 
15:     $\dots + (\Delta \mathbf{x}_k^{\text{cc}}, \Delta \mathbf{a}_k^{\text{cc}}, \Delta \boldsymbol{\lambda}_k^{\text{cc}}, \Delta \mathbf{p}_k^{\text{cc}}, \Delta \mathbf{t}_k^{\text{cc}}, \Delta \boldsymbol{\beta}^{\text{cc}})$ 
16:     $\alpha = \arg \max \{ \alpha \in [0, 1] : (\boldsymbol{\lambda}_k, \mathbf{t}_k) + \alpha(\Delta \boldsymbol{\lambda}_k, \Delta \mathbf{t}_k) \geq \mathbf{0} \forall k \in [0, N-1] \}$ 
17:     $(\mathbf{x}_k, \mathbf{a}_k, \boldsymbol{\lambda}_k, \mathbf{p}_k, \mathbf{t}_k, \boldsymbol{\beta}) = (\mathbf{x}_k, \mathbf{a}_k, \boldsymbol{\lambda}_k, \mathbf{p}_k, \mathbf{t}_k, \boldsymbol{\beta}) \dots$ 
18:     $\dots + \alpha(\Delta \mathbf{x}_k, \Delta \mathbf{a}_k, \Delta \boldsymbol{\lambda}_k, \Delta \mathbf{p}_k, \Delta \mathbf{t}_k, \Delta \boldsymbol{\beta})$ 
19:     $\mu = \frac{1}{Nn_d} \sum_{k=0}^{N-1} \boldsymbol{\lambda}_k^T \mathbf{t}_k$ 
20:  end while
21: end procedure

```

4.5.2 Simulation Results and Analysis

Table 4.1 presents the simulation results for Algorithm 3. In every experiment, the the problem size parameters that were not under consideration were fixed at 10, and a range of $[100, 1000]$ was utilised for the varying parameter. The number of processors was fixed at $m = 1000$ for all experiments.

These results indicate worse scaling performance in comparison to the `linprog` function across every problem size parameter. The number of tasks and the number of speed levels are seen to increase faster than at a quadratic rate. However, because almost all the time to convergence is spent on solving the KKT system of equations, the scaling results can be attributed to the use of the `\function` as a black box. Clearly, the function records superior scaling results for structural changes with an increasing number of time steps. The exact discrepancy between the three exponents cannot be described without an extensive study of the various routines embedded within the `\function`.

Line Of Best Fit: $Y = aX^b + c$

Experiment	<i>a</i>	<i>b</i>	<i>c</i>
Increasing Number of Tasks (n)	0.000005	2.257462	0.314571
Increasing Number of Time Steps (N)	0.000053	1.594641	0.334689
Increasing Number of Speed Levels (l)	0.000006	2.146372	0.405762

Table 4.1: Simulation Results [FDP-AK7213-V1]

Chapter 5

The Workload-Partitioning Different-Deadline Problem (WP-DDP)

5.1 Introduction

In this chapter, we explore the different-deadline workload-partitioning problem where incoming tasks are due at different points on the time grid. We build upon the WP-FDP formulation introduced in the previous chapter, and modify this LP in order to extend the problem from fixed task deadlines to the differing task deadlines. This leads to a different set of optimality conditions and hence a differently structured system of linear equations that must be solved during each IPM iteration.

The WP-DDP and any additional definitions are formally introduced before re-writing the corresponding KKT conditions for optimality. Hereafter, we aim to solve the problem using a custom written IPM that is based on Mehrotra's predictor-corrector algorithm. Our primary focus remains fixated on driving down the relatively expensive computational cost of calculating search directions (using Newtons method) during each IPM iteration.

We set out to achieve this by performing (and keeping track of) block eliminations on the original KKT system of equations. The sparsity structures of the constituent block matrices is exploited in solving the reduced system of equations. The corresponding MATLAB implementation details, alongside an analysis of the simulation results, are also presented in this chapter.

5.2 Problem Definition

We start with the introduction of some additional definitions that help incorporate the different-deadline variation into the WP-FDP formulation seen previously. The additional definitions developed here are valid throughout the remainder of this work.

Let $n_r[k]$ represent the number of remaining tasks at τ_k (including the tasks that are due at τ_k), and $n_j[k]$ represent the number of tasks that are due at time step k . These quantities obey the equations shown in (5.1). Note that $n_r[k]$ and $n_j[k]$ are defined for $k = 1, \dots, N$ since all tasks are assumed to arrive at τ_0 .

$$n_r[1] = n \quad (5.1a)$$

$$n_r[N] = n_j[N] \quad (5.1b)$$

$$n_r[k+1] = n_r[k] - n_j[k] \quad k = 1, 2, \dots, N-1 \quad (5.1c)$$

$$\sum_{k=1}^N n_j[k] = n \quad (5.1d)$$

Let $\mathbf{F}_k \in \mathbb{R}^{n_j[k] \times n}$ represent a selector matrix that requires the appropriate tasks to meet their respective deadlines for $k = 1, 2, \dots, N$. Note that we assume that the components of the estimated execution times vector $\hat{\mathbf{x}}$ are sorted by earliest deadline first. This enables the correct sequential selection of tasks using the \mathbf{F}_k operator as shown in (5.2).

$$\mathbf{F}_k = [\mathbf{0}_{n_j[k] \times n - n_r[k]} \quad \mathbf{I}_{n_j[k] \times n_j[k]} \quad \mathbf{0}_{n_j[k] \times n_r[k] - n_j[k]}] \quad (5.2)$$

With this set up, the modified discrete-time optimal control problem is written as shown in (5.3). The different deadline variation in the problem is accounted for through the introduction of terminal equality constraints for all state variables beyond that corresponding to the arrival time. This is shown in (5.3c) and represents the only modification that is necessary.

$$\min_{\mathbf{x}_k, \mathbf{a}_k} \sum_{k=0}^{N-1} (\tau_{k+1} - \tau_k) (\mathbf{1}_n^T \otimes \mathbf{P}^T) \mathbf{a}_k \quad (5.3a)$$

subject to:

$$\mathbf{x}_0 = \hat{\mathbf{x}} \quad (5.3b)$$

$$\mathbf{F}_k \mathbf{x}_k = \mathbf{0}_{n_j[k] \times 1} \quad k = 1, 2, \dots, N \quad (5.3c)$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{B} \mathbf{a}_k \quad k = 0, 1, \dots, N-1 \quad (5.3d)$$

$$\mathbf{D} \mathbf{a}_k - \mathbf{d} \leq \mathbf{0}_{n_d} \quad k = 0, 1, \dots, N-1 \quad (5.3e)$$

Lastly, note that the time grid between the first and last points is divided uniformly as was the case in the previous chapter. Moreover, we assume that a constant number of tasks due at each time step. This helps simplify the mathematical descriptions as well as the software implementation. To this end, we drop the subscript k on the $n_j[k]$ scalar quantities and redefine $n_j := n_j[k] = n/N$ for $k = 1, 2, \dots, N$.

5.3 The Complete KKT System of Equations

In this section, the optimality conditions for the modified discrete-time problem in (5.3) are re-derived in order to depict the differences in the left-hand side Jacobian matrix that appears in the matrix equation in Newton's method.

The Lagrangian to the WP-DDP in (5.3) is shown in (5.4). The Lagrange multipliers are denoted by $\mathbf{p}_k \in \mathbb{R}^{n \times 1}$, $\boldsymbol{\lambda}_k \in \mathbb{R}^{n_d \times 1}$ and $\boldsymbol{\beta}_k \in \mathbb{R}^{n_j \times 1}$. The slack variable is denoted by $\mathbf{t}_k \in \mathbb{R}^{n_d \times 1}$.

$$\begin{aligned} \mathcal{L} = & \sum_{k=0}^{N-1} (\tau_{k+1} - \tau_k) (\mathbf{1}_n^T \otimes \mathbf{P}^T) \mathbf{a}_k \\ & + \sum_{k=1}^N \boldsymbol{\beta}_k^T (\mathbf{F}_k \mathbf{x}_k) + \sum_{k=0}^{N-1} \mathbf{p}_k^T (\mathbf{x}_k + \mathbf{B} \mathbf{a}_k - \mathbf{x}_{k+1}) \\ & + \sum_{k=0}^{N-1} \boldsymbol{\lambda}_k^T (\mathbf{D} \mathbf{a}_k - \mathbf{d} + \mathbf{t}_k) \end{aligned} \quad (5.4)$$

The Karush-Kuhn-Tucker (KKT) conditions for optimality are derived as follows:

$$\mathbf{r}_k^{\mathbf{x}} = \mathbf{F}_k^T \boldsymbol{\beta}_k + \mathbf{p}_k - \mathbf{p}_{k-1} = \mathbf{0} \quad k = 1, 2, \dots, N-1 \quad (5.5a)$$

$$\mathbf{r}_k^{\mathbf{x}} = \mathbf{F}_k^T \boldsymbol{\beta}_k - \mathbf{p}_{k-1} = \mathbf{0} \quad k = N \quad (5.5b)$$

$$\mathbf{r}_k^{\boldsymbol{\beta}} = \mathbf{F}_k \mathbf{x}_k = \mathbf{0} \quad k = 1, 2, \dots, N \quad (5.5c)$$

$$\mathbf{r}_k^{\mathbf{a}} = (\tau_{k+1} - \tau_k) (\mathbf{1}_{n \times 1} \otimes \mathbf{P}) + \mathbf{B}^T \mathbf{p}_k + \mathbf{D}^T \boldsymbol{\lambda}_k = \mathbf{0} \quad k = 0, 1, \dots, N-1 \quad (5.5d)$$

$$\mathbf{r}_k^{\boldsymbol{\lambda}} = \mathbf{D} \mathbf{a}_k - \mathbf{d} + \mathbf{t}_k = \mathbf{0} \quad k = 0, 1, \dots, N-1 \quad (5.5e)$$

$$\mathbf{r}_k^{\mathbf{p}} = \mathbf{x}_k + \mathbf{B} \mathbf{a}_k - \mathbf{x}_{k+1} = \mathbf{0} \quad k = 0, 1, \dots, N-1 \quad (5.5f)$$

$$\mathbf{r}_k^{\mathbf{t}} = \boldsymbol{\Lambda}_k \mathbf{T}_k \mathbf{e} = \mathbf{0} \quad k = 0, 1, \dots, N-1 \quad (5.5g)$$

where $\boldsymbol{\Lambda}_k \in \mathbb{R}^{n_d \times n_d}$ and $\mathbf{T}_k \in \mathbb{R}^{n_d \times n_d}$ denote diagonal matrices with $\boldsymbol{\lambda}_k$ and \mathbf{t}_k elements placed on the main diagonal respectively.

The corresponding system of linear equations that must be solved in Newton's Method to obtain the affine-scaling direction is shown in (5.6).

$$\begin{bmatrix} & \mathbf{D}^T & \mathbf{B}^T & & & \\ \mathbf{D} & & & \mathbf{I} & & \\ \mathbf{B} & & & & -\mathbf{I} & \\ & \mathbf{T}_0 & & \boldsymbol{\Lambda}_0 & & \\ & & -\mathbf{I} & & & \\ & & & & & \ddots \end{bmatrix} \begin{bmatrix} \Delta \mathbf{a}_0 \\ \Delta \boldsymbol{\lambda}_0 \\ \Delta \mathbf{p}_0 \\ \Delta \mathbf{t}_0 \\ \Delta \mathbf{x}_1 \\ \vdots \end{bmatrix} = - \begin{bmatrix} \mathbf{r}_0^{\mathbf{a}} \\ \mathbf{r}_0^{\boldsymbol{\lambda}} \\ \mathbf{r}_0^{\mathbf{p}} \\ \mathbf{r}_0^{\mathbf{t}} \\ \mathbf{r}_1^{\mathbf{x}} \\ \vdots \end{bmatrix} \quad (5.6a)$$

$$\begin{bmatrix}
\ddots & & & & & & & & \\
& \mathbf{F}_k & \mathbf{F}_k^T & & & & \mathbf{I} & & \\
& & & \mathbf{D}^T & \mathbf{B}^T & & & & \\
& & \mathbf{I} & \mathbf{D} & \mathbf{B} & & \mathbf{I} & & \\
& & & & \mathbf{T}_k & & \mathbf{\Lambda}_k & & \\
& & & & & -\mathbf{I} & & & \\
& & & & & & & \ddots &
\end{bmatrix}
\begin{bmatrix}
\vdots \\
\Delta \mathbf{x}_k \\
\Delta \boldsymbol{\beta}_k \\
\Delta \mathbf{a}_k \\
\Delta \boldsymbol{\lambda}_k \\
\Delta \mathbf{p}_k \\
\Delta \mathbf{t}_k \\
\Delta \mathbf{x}_{k+1} \\
\vdots
\end{bmatrix}
= -
\begin{bmatrix}
\vdots \\
\mathbf{r}_k^{\mathbf{x}} \\
\mathbf{r}_k^{\boldsymbol{\beta}} \\
\mathbf{r}_k^{\mathbf{a}} \\
\mathbf{r}_k^{\boldsymbol{\lambda}} \\
\mathbf{r}_k^{\mathbf{p}} \\
\mathbf{r}_k^{\mathbf{t}} \\
\mathbf{r}_{k+1}^{\mathbf{x}} \\
\vdots
\end{bmatrix} \quad (5.6b)$$

$$\begin{bmatrix}
\ddots & & & & & & & & \\
& \mathbf{F}_{N-1} & \mathbf{F}_{N-1}^T & & & & \mathbf{I} & & \\
& & & \mathbf{D}^T & \mathbf{B}^T & & & & \\
& & \mathbf{I} & \mathbf{D} & \mathbf{B} & & \mathbf{I} & & \\
& & & & \mathbf{T}_{N-1} & & \mathbf{\Lambda}_{N-1} & & \\
& & & & & -\mathbf{I} & & & \\
& & & & & & & \mathbf{F}_N^T & \\
& & & & & & & & \mathbf{F}_N^T
\end{bmatrix}
\begin{bmatrix}
\vdots \\
\Delta \mathbf{x}_{N-1} \\
\Delta \boldsymbol{\beta}_{N-1} \\
\Delta \mathbf{a}_{N-1} \\
\Delta \boldsymbol{\lambda}_{N-1} \\
\Delta \mathbf{p}_{N-1} \\
\Delta \mathbf{t}_{N-1} \\
\Delta \mathbf{x}_N \\
\Delta \boldsymbol{\beta}_N
\end{bmatrix}
= -
\begin{bmatrix}
\vdots \\
\mathbf{r}_{N-1}^{\mathbf{x}} \\
\mathbf{r}_{N-1}^{\boldsymbol{\beta}} \\
\mathbf{r}_{N-1}^{\mathbf{a}} \\
\mathbf{r}_{N-1}^{\boldsymbol{\lambda}} \\
\mathbf{r}_{N-1}^{\mathbf{p}} \\
\mathbf{r}_{N-1}^{\mathbf{t}} \\
\mathbf{r}_N^{\mathbf{x}} \\
\mathbf{r}_N^{\boldsymbol{\beta}}
\end{bmatrix} \quad (5.6c)$$

5.4 The Reduced KKT System of Equations

In this section, we carry out a specific sequence of block eliminations to reduce the dimensionality of the problem since we have already identified solving the KKT system of equations represents the single largest computational bottleneck in primal-dual methods.

The order in which we carry out block eliminations can have an impact on the numerical conditioning of the Jacobian matrix. A common starting point in literature is the elimination of the search direction components associated with the slack variables $\Delta \mathbf{t}_k$ from the problem since it introduces symmetry. The elimination process entails writing down all the equations in (5.3) that contain the $\Delta \mathbf{t}_k$ terms, and reducing these equations to those without the $\Delta \mathbf{t}_k$ terms. The reduced system of equations is derived as shown in Theorem 5.1.

Theorem 5.1. *The KKT system of equations in Newton's method for the WP-DDP (5.6) can be reduced to a new (symmetric) system by eliminating the search direction components associated with the slack variables $\Delta \mathbf{t}_k$*

Proof. The equations associated with $\Delta \mathbf{t}_k$ for $k = 0, 1, \dots, N-1$ are shown in (5.7)

$$\mathbf{T}_k \Delta \boldsymbol{\lambda}_k + \mathbf{\Lambda}_k \Delta \mathbf{t}_k = -\mathbf{r}_k^{\mathbf{t}} \quad (5.7a)$$

$$\mathbf{D} \Delta \mathbf{a}_k + \Delta \mathbf{t}_k = -\mathbf{r}_k^{\boldsymbol{\lambda}} \quad (5.7b)$$

Equation (5.7a) is used to obtain an expression for $\Delta \mathbf{t}_k$ as shown in (5.8a). This is then substituted back into (5.7b) in order to obtain a single expression as outlined in (5.8b) and (5.8c).

$$\Delta \mathbf{t}_k = -\Lambda_k^{-1}(\mathbf{r}_k^{\mathbf{t}} + \mathbf{T}_k \Delta \boldsymbol{\lambda}_k) \quad (5.8a)$$

$$\mathbf{D} \Delta \mathbf{a}_k - \Lambda_k^{-1}(\mathbf{r}_k^{\mathbf{t}} + \mathbf{T}_k \Delta \boldsymbol{\lambda}_k) = -\mathbf{r}_k^{\boldsymbol{\lambda}} \quad (5.8b)$$

$$\mathbf{D} \Delta \mathbf{a}_k - \Lambda_k^{-1} \mathbf{T}_k \Delta \boldsymbol{\lambda}_k = -\mathbf{r}_k^{\boldsymbol{\lambda}} + \Lambda_k^{-1} \mathbf{r}_k^{\mathbf{t}} \quad (5.8c)$$

We may simplify the reduced expression (5.8c) by defining the following two quantities. The diagonal matrix $\boldsymbol{\Sigma}_k \in \mathbb{R}^{n_d \times n_d}$ for $k = 0, 1, \dots, N-1$ and the column vector $\hat{\mathbf{r}}_k^{\boldsymbol{\lambda}} \in \mathbb{R}^{n_d \times 1}$ for $k = 0, 1, \dots, N-1$ are defined as per (5.9a) and (5.9b) respectively.

$$\boldsymbol{\Sigma}_k := -\Lambda_k^{-1} \mathbf{T}_k \quad (5.9a)$$

$$\hat{\mathbf{r}}_k^{\boldsymbol{\lambda}} := \mathbf{r}_k^{\boldsymbol{\lambda}} - \Lambda_k^{-1} \mathbf{r}_k^{\mathbf{t}} \quad (5.9b)$$

Lastly, we rewrite (5.8c) as per (5.10)

$$\mathbf{D} \Delta \mathbf{a}_k + \boldsymbol{\Sigma}_k \Delta \boldsymbol{\lambda}_k = -\hat{\mathbf{r}}_k^{\boldsymbol{\lambda}} \quad (5.10)$$

The modified KKT system of equations in Newton's method is shown in (5.11). The symmetry property is evident through inspection. This concludes the proof.

$$\begin{bmatrix} & & \mathbf{D}^{\mathbf{T}} & \mathbf{B}^{\mathbf{T}} & & \\ \mathbf{D} & \boldsymbol{\Sigma}_0 & & & & \\ \mathbf{B} & & & -\mathbf{I} & & \\ & & & & -\mathbf{I} & \\ & & & & & \ddots \end{bmatrix} \begin{bmatrix} \Delta \mathbf{a}_0 \\ \Delta \boldsymbol{\lambda}_0 \\ \Delta \mathbf{p}_0 \\ \Delta \mathbf{x}_1 \\ \vdots \end{bmatrix} = - \begin{bmatrix} \mathbf{r}_0^{\mathbf{a}} \\ \hat{\mathbf{r}}_0^{\boldsymbol{\lambda}} \\ \mathbf{r}_0^{\mathbf{p}} \\ \mathbf{r}_1^{\mathbf{x}} \\ \vdots \end{bmatrix} \quad (5.11a)$$

$$\begin{bmatrix} \ddots & & & & & \\ & \mathbf{F}_k & \mathbf{F}_k^{\mathbf{T}} & & \mathbf{I} & \\ & & & \mathbf{D}^{\mathbf{T}} & \mathbf{B}^{\mathbf{T}} & \\ & & \mathbf{D} & \boldsymbol{\Sigma}_k & & \\ \mathbf{I} & & \mathbf{B} & & & -\mathbf{I} \\ & & & & & -\mathbf{I} \\ & & & & & \ddots \end{bmatrix} \begin{bmatrix} \vdots \\ \Delta \mathbf{x}_k \\ \Delta \boldsymbol{\beta}_k \\ \Delta \mathbf{a}_k \\ \Delta \boldsymbol{\lambda}_k \\ \Delta \mathbf{p}_k \\ \Delta \mathbf{x}_{k+1} \\ \vdots \end{bmatrix} = - \begin{bmatrix} \vdots \\ \mathbf{r}_k^{\mathbf{x}} \\ \mathbf{r}_k^{\boldsymbol{\beta}} \\ \mathbf{r}_k^{\mathbf{a}} \\ \hat{\mathbf{r}}_k^{\boldsymbol{\lambda}} \\ \mathbf{r}_k^{\mathbf{p}} \\ \mathbf{r}_{k+1}^{\mathbf{x}} \\ \vdots \end{bmatrix} \quad (5.11b)$$

$$\begin{bmatrix}
\ddots & & & & & & & & \\
& \mathbf{F}_{N-1} & \mathbf{F}_{N-1}^T & & & & & & \\
& & & \mathbf{I} & & & & & \\
& & & & \mathbf{D}^T & \mathbf{B}^T & & & \\
& & & & \mathbf{D} & \boldsymbol{\Sigma}_{N-1} & & & \\
& & \mathbf{I} & & \mathbf{B} & & & & \\
& & & & & & -\mathbf{I} & & \\
& & & & & & & -\mathbf{I} & \\
& & & & & & & & \mathbf{F}_N^T
\end{bmatrix}
\begin{bmatrix}
\vdots \\
\Delta \mathbf{x}_{N-1} \\
\Delta \boldsymbol{\beta}_{N-1} \\
\Delta \mathbf{a}_{N-1} \\
\Delta \boldsymbol{\lambda}_{N-1} \\
\Delta \mathbf{p}_{N-1} \\
\Delta \mathbf{x}_N \\
\Delta \boldsymbol{\beta}_N
\end{bmatrix}
= -
\begin{bmatrix}
\vdots \\
\mathbf{r}_{N-1}^{\mathbf{x}} \\
\mathbf{r}_{N-1}^{\boldsymbol{\beta}} \\
\mathbf{r}_{N-1}^{\mathbf{a}} \\
\hat{\mathbf{r}}_{N-1}^{\boldsymbol{\lambda}} \\
\mathbf{r}_{N-1}^{\mathbf{p}} \\
\mathbf{r}_N^{\mathbf{x}} \\
\mathbf{r}_N^{\boldsymbol{\beta}}
\end{bmatrix} \quad (5.11c)$$

□

The next logical step concerns eliminating the search direction components for the Lagrange multipliers associated with the single inequality constraint $\Delta \boldsymbol{\lambda}_k$. This is because the equations associated with the other unknowns are coupled together across different stages and therefore more difficult to work with.

Theorem 5.2. *The reduced KKT system of equations in Newton's method for the WP-DDP (5.11) can be further scaled down to a new (symmetric) system by eliminating the search direction components associated with the Lagrange multiplier $\Delta \boldsymbol{\lambda}_k$*

Proof. The equations associated with $\Delta \boldsymbol{\lambda}_k$ for $k = 0, 1, \dots, N-1$ are shown in (5.12).

$$\mathbf{D}^T \Delta \boldsymbol{\lambda}_k + \mathbf{B}^T \Delta \mathbf{p}_k = -\mathbf{r}_k^{\mathbf{a}} \quad (5.12a)$$

$$\mathbf{D} \Delta \mathbf{a}_k + \boldsymbol{\Sigma}_k \Delta \boldsymbol{\lambda}_k = -\hat{\mathbf{r}}_k^{\boldsymbol{\lambda}} \quad (5.12b)$$

Equation (5.12b) is used to obtain an expression for $\Delta \boldsymbol{\lambda}_k$ as shown in (5.13a). This is then substituted back into (5.12a) in order to obtain a single expression as outlined in (5.13b) and (5.13c).

$$\Delta \boldsymbol{\lambda}_k = -(\boldsymbol{\Sigma}_k)^{-1}(\hat{\mathbf{r}}_k^{\boldsymbol{\lambda}} + \mathbf{D} \Delta \mathbf{a}_k) \quad (5.13a)$$

$$-\mathbf{D}^T(\boldsymbol{\Sigma}_k)^{-1}(\hat{\mathbf{r}}_k^{\boldsymbol{\lambda}} + \mathbf{D} \Delta \mathbf{a}_k) + \mathbf{B}^T \Delta \mathbf{p}_k = -\mathbf{r}_k^{\mathbf{a}} \quad (5.13b)$$

$$-\mathbf{D}^T(\boldsymbol{\Sigma}_k)^{-1} \mathbf{D} \Delta \mathbf{a}_k + \mathbf{B}^T \Delta \mathbf{p}_k = -\mathbf{r}_k^{\mathbf{a}} + \mathbf{D}^T(\boldsymbol{\Sigma}_k)^{-1} \hat{\mathbf{r}}_k^{\boldsymbol{\lambda}} \quad (5.13c)$$

We may simplify the reduced expression (5.13c) by defining the following two quantities. The dense matrix $\mathbf{R}_k \in \mathbb{R}^{nl \times nl}$ for $k = 0, 1, \dots, N-1$ and the column vector $\hat{\mathbf{r}}_k^{\mathbf{a}} \in \mathbb{R}^{nl \times 1}$ for $k = 0, 1, \dots, N-1$ are defined as per (5.14a) and (5.14b) respectively.

$$\mathbf{R}_k := -\mathbf{D}^T(\boldsymbol{\Sigma}_k)^{-1} \mathbf{D} \quad (5.14a)$$

$$\hat{\mathbf{r}}_k^{\mathbf{a}} := \mathbf{r}_k^{\mathbf{a}} - \mathbf{D}^T(\boldsymbol{\Sigma}_k)^{-1} \hat{\mathbf{r}}_k^{\boldsymbol{\lambda}} \quad (5.14b)$$

Lastly, we rewrite (5.13c) as per (5.15)

$$\mathbf{R}_k \Delta \mathbf{a}_k + \mathbf{B}^T \Delta \mathbf{p}_k = -\hat{\mathbf{r}}_k^{\mathbf{a}} \quad (5.15)$$

The new system of equations that needs to be solved is shown in (5.16). This completes the proof.

$$\begin{bmatrix} \mathbf{R}_0 & \mathbf{B}^T & & \\ \mathbf{B} & & -\mathbf{I} & \\ & & & \ddots \end{bmatrix} \begin{bmatrix} \Delta \mathbf{a}_0 \\ \Delta \mathbf{p}_0 \\ \Delta \mathbf{x}_1 \\ \vdots \end{bmatrix} = - \begin{bmatrix} \hat{\mathbf{r}}_0^{\mathbf{a}} \\ \mathbf{r}_0^{\mathbf{p}} \\ \mathbf{r}_1^{\mathbf{x}} \\ \vdots \end{bmatrix} \quad (5.16a)$$

$$\begin{bmatrix} \ddots & & & & & \\ & \mathbf{F}_k & \mathbf{F}_k^T & & \mathbf{I} & \\ & & & \mathbf{R}_k & \mathbf{B}^T & \\ & \mathbf{I} & & \mathbf{B} & & -\mathbf{I} \\ & & & & & \ddots \end{bmatrix} \begin{bmatrix} \vdots \\ \Delta \mathbf{x}_k \\ \Delta \beta_k \\ \Delta \mathbf{a}_k \\ \Delta \mathbf{p}_k \\ \Delta \mathbf{x}_{k+1} \\ \vdots \end{bmatrix} = - \begin{bmatrix} \vdots \\ \mathbf{r}_k^{\mathbf{x}} \\ \mathbf{r}_k^{\beta} \\ \hat{\mathbf{r}}_k^{\mathbf{a}} \\ \mathbf{r}_k^{\mathbf{p}} \\ \mathbf{r}_{k+1}^{\mathbf{x}} \\ \vdots \end{bmatrix} \quad (5.16b)$$

$$\begin{bmatrix} \ddots & & & & & \\ & \mathbf{F}_{N-1} & \mathbf{F}_{N-1}^T & & \mathbf{I} & \\ & & & \mathbf{R}_{N-1} & \mathbf{B}^T & \\ & \mathbf{I} & & \mathbf{B} & & -\mathbf{I} \\ & & & & & \ddots \end{bmatrix} \begin{bmatrix} \vdots \\ \Delta \mathbf{x}_{N-1} \\ \Delta \beta_{N-1} \\ \Delta \mathbf{a}_{N-1} \\ \Delta \mathbf{p}_{N-1} \\ \Delta \mathbf{x}_N \\ \Delta \beta_N \end{bmatrix} = - \begin{bmatrix} \vdots \\ \mathbf{r}_{N-1}^{\mathbf{x}} \\ \mathbf{r}_{N-1}^{\beta} \\ \hat{\mathbf{r}}_{N-1}^{\mathbf{a}} \\ \mathbf{r}_{N-1}^{\mathbf{p}} \\ \mathbf{r}_N^{\mathbf{x}} \\ \mathbf{r}_N^{\beta} \end{bmatrix} \quad (5.16c)$$

□

Theorem 5.3. *The reduced KKT system of equations in Newton's method for the WP-DDP (5.16) can be further scaled down and expressed in terms of the search direction components associated with the Lagrange multiplier $\Delta \beta_k$ only.*

Proof. The reduced system of equations in (5.16) are written out in (5.17).

For $k = 0$

$$\mathbf{R}_0 \Delta \mathbf{a}_0 + \mathbf{B}^T \Delta \mathbf{p}_0 = -\hat{\mathbf{r}}_0^{\mathbf{a}} \quad (5.17a)$$

$$\mathbf{B} \Delta \mathbf{a}_0 - \Delta \mathbf{x}_1 = -\mathbf{r}_0^{\mathbf{p}} \quad (5.17b)$$

For $k = 1, 2, \dots, N - 1$

$$-\Delta \mathbf{p}_{k-1} + \mathbf{F}_k^T \Delta \boldsymbol{\beta}_k + \Delta \mathbf{p}_k = -\mathbf{r}_k^{\mathbf{x}} \quad (5.17c)$$

$$\mathbf{F}_k \Delta \mathbf{x}_k = -\mathbf{r}_k^{\boldsymbol{\beta}} \quad (5.17d)$$

$$\mathbf{R}_k \Delta \mathbf{a}_k + \mathbf{B}^T \Delta \mathbf{p}_k = -\hat{\mathbf{r}}_k^{\mathbf{a}} \quad (5.17e)$$

$$\Delta \mathbf{x}_k + \mathbf{B} \Delta \mathbf{a}_k - \Delta \mathbf{x}_{k+1} = -\mathbf{r}_k^{\mathbf{p}} \quad (5.17f)$$

For $k = N$

$$-\Delta \mathbf{p}_{N-1} + \mathbf{F}_N^T \Delta \boldsymbol{\beta}_N = -\mathbf{r}_N^{\mathbf{x}} \quad (5.17g)$$

$$\mathbf{F}_N \Delta \mathbf{x}_N = -\mathbf{r}_N^{\boldsymbol{\beta}} \quad (5.17h)$$

Let us first consider the $\Delta \mathbf{a}_k$ terms in (5.17a) and (5.17e). These equations are not coupled to any other quantities in the previous or subsequent stages. This allows the direct (as opposed to iterative) computation of the $\Delta \mathbf{a}_k$ values for $k = 0, 1, \dots, N - 1$ as shown in (5.18).

$$\Delta \mathbf{a}_k = -(\mathbf{R}_k)^{-1}(\hat{\mathbf{r}}_k^{\mathbf{a}} + \mathbf{B}^T \Delta \mathbf{p}_k) \quad (5.18)$$

Let us now consider the $\Delta \mathbf{x}_k$ terms. It is important to note that using (5.17d) and (5.17h) to obtain the expression $\Delta \mathbf{x}_k = -(\mathbf{F}_k)^{-1} \mathbf{r}_k^{\boldsymbol{\beta}}$ is unacceptable since the selector matrix \mathbf{F}_k is generally a flat matrix that represents an under-determined system with no unique solution.

Therefore, we make use of (5.17f) and (5.17b) to derive the forward-substitution equation (5.19a) and the initial condition (5.19b) respectively. Hereafter, we can specify a decoupled expression for $\Delta \mathbf{x}_k$ for $k = 1, 2, \dots, N$ as shown in (5.19c).

$$\Delta \mathbf{x}_{k+1} = \Delta \mathbf{x}_k + \mathbf{B} \Delta \mathbf{a}_k + \mathbf{r}_k^{\mathbf{p}} \quad (5.19a)$$

$$\Delta \mathbf{x}_1 = \mathbf{B} \Delta \mathbf{a}_0 + \mathbf{r}_0^{\mathbf{p}} \quad (5.19b)$$

$$\Delta \mathbf{x}_k = \sum_{j=0}^{k-1} (\mathbf{B} \Delta \mathbf{a}_j + \mathbf{r}_j^{\mathbf{p}}) \quad (5.19c)$$

Lastly, we must account for the $\Delta \mathbf{p}_k$ terms. We use (5.17c) and (5.17g) to derive the back-substitution equation (5.20a) and the terminal condition (5.20b) respectively. The direct expression for calculating $\Delta \mathbf{p}_k$ for $k = 0, 1, \dots, N - 1$ is shown in (5.20c)

$$\Delta \mathbf{p}_{k-1} = \Delta \mathbf{p}_k + \mathbf{F}_k^T \Delta \boldsymbol{\beta}_k + \mathbf{r}_k^{\mathbf{x}} \quad (5.20a)$$

$$\Delta \mathbf{p}_{N-1} = \mathbf{F}_N^T \Delta \boldsymbol{\beta}_N + \mathbf{r}_N^{\mathbf{x}} \quad (5.20b)$$

$$\Delta \mathbf{p}_k = \sum_{i=k+1}^N (\mathbf{F}_i^T \Delta \boldsymbol{\beta}_i + \mathbf{r}_i^{\mathbf{x}}) \quad (5.20c)$$

At this point, we are able to start substituting the derived expressions for \mathbf{x}_k , \mathbf{a}_k and \mathbf{p}_k into $\mathbf{F}_k \Delta \mathbf{x}_k = -\mathbf{r}_k^\beta$ for $k = 1, 2, \dots, N$.

The \mathbf{x}_k terms are eliminated in (5.21)

$$\mathbf{F}_k \Delta \mathbf{x}_k = \mathbf{F}_k \sum_{j=0}^{k-1} (\mathbf{B} \Delta \mathbf{a}_j + \mathbf{r}_j^\mathbf{p}) = -\mathbf{r}_k^\beta \quad (5.21)$$

The \mathbf{a}_j terms are eliminated in (5.22)

$$\mathbf{F}_k \sum_{j=0}^{k-1} (\mathbf{B} \Delta \mathbf{a}_j + \mathbf{r}_j^\mathbf{p}) = \mathbf{F}_k \sum_{j=0}^{k-1} (-\mathbf{B}(\mathbf{R}_j)^{-1}(\hat{\mathbf{r}}_j^\mathbf{a} + \mathbf{B}^\mathbf{T} \Delta \mathbf{p}_j) + \mathbf{r}_j^\mathbf{p}) = -\mathbf{r}_k^\beta \quad (5.22a)$$

which can be simplified to

$$\mathbf{F}_k \sum_{j=0}^{k-1} (\mathbf{B}(\mathbf{R}_j)^{-1} \mathbf{B}^\mathbf{T}) \Delta \mathbf{p}_j = \mathbf{r}_k^\beta + \mathbf{F} \sum_{j=0}^{k-1} (\mathbf{r}_j^\mathbf{p} - \mathbf{B}(\mathbf{R}_j)^{-1} \hat{\mathbf{r}}_j^\mathbf{a}) \quad (5.22b)$$

The \mathbf{p}_j terms are eliminated in (5.23)

$$\begin{aligned} \mathbf{F}_k \sum_{j=0}^{k-1} (\mathbf{B}(\mathbf{R}_j)^{-1} \mathbf{B}^\mathbf{T}) \Delta \mathbf{p}_j &= \mathbf{F}_k \sum_{j=0}^{k-1} (\mathbf{B}(\mathbf{R}_j)^{-1} \mathbf{B}^\mathbf{T}) \sum_{i=j+1}^N (\mathbf{F}_i^\mathbf{T} \Delta \beta_i + \mathbf{r}_i^\mathbf{x}) \\ &= \mathbf{r}_k^\beta + \mathbf{F} \sum_{j=0}^{k-1} (\mathbf{r}_j^\mathbf{p} - \mathbf{B}(\mathbf{R}_j)^{-1} \hat{\mathbf{r}}_j^\mathbf{a}) \end{aligned} \quad (5.23a)$$

which can be simplified to

$$\begin{aligned} \mathbf{F}_k \sum_{j=0}^{k-1} (\mathbf{B}(\mathbf{R}_j)^{-1} \mathbf{B}^\mathbf{T}) \sum_{i=j+1}^N \mathbf{F}_i^\mathbf{T} \Delta \beta_i &= \mathbf{r}_k^\beta + \mathbf{F} \sum_{j=0}^{k-1} (\mathbf{r}_j^\mathbf{p} - \mathbf{B}(\mathbf{R}_j)^{-1} \hat{\mathbf{r}}_j^\mathbf{a}) \\ &\quad - \mathbf{F}_k \sum_{j=0}^{k-1} (\mathbf{B}(\mathbf{R}_j)^{-1} \mathbf{B}^\mathbf{T}) \sum_{i=j+1}^N \mathbf{r}_i^\mathbf{x} \end{aligned} \quad (5.23b)$$

The final result of the block eliminations carried out in this proof is the reduced set of equations comprised of the $\Delta \beta_k$ terms only. The equation in (5.23b), for $k = 1, 2, \dots, N$, fully defines the reduced system of equations. This concludes the proof. \square

Admittedly, the equation in (5.23b) is neither pleasing on the eye nor is it very intuitive. We now discuss the approach taken to summarise this result into a new matrix equation (5.24) in order to devise a computationally tractable algorithm.

$$\mathbf{\Pi} \Delta \bar{\boldsymbol{\beta}} = \boldsymbol{\pi} \quad (5.24)$$

The column vector $\Delta\bar{\boldsymbol{\beta}} \in \mathbb{R}^{n \times 1}$ represents the concatenated sequence of the all the unknown $\Delta\boldsymbol{\beta}_k \in \mathbb{R}^{n_j \times 1}$ terms for $k = 1, 2, \dots, N$. This is shown in (5.25).

$$\Delta\bar{\boldsymbol{\beta}} = [\Delta\boldsymbol{\beta}_1^T \quad \Delta\boldsymbol{\beta}_2^T \quad \dots \quad \Delta\boldsymbol{\beta}_N^T]^T \quad (5.25)$$

The square matrix $\boldsymbol{\Pi} \in \mathbb{R}^{n \times n}$ represents a sum of N increasingly sparse $\mathbf{B}(\mathbf{R}_k)^{-1}\mathbf{B}^T$ terms. The combined effect of the selector matrices \mathbf{F}_k and \mathbf{F}_k^T on the left hand-side in (5.23b) can be observed by meticulously expanding the two summations for $k = 1, 2, \dots, N$, and appropriately regrouping the resulting quantities into matrix form. To this end, we define a new selector matrix that summarises the overall selection operation.

The square matrix $\mathbf{H}_k \in \mathbb{R}^{n \times n}$ for $k = 0, 1, \dots, N-1$ represents a selector matrix is used to encode the selected elements of the $\mathbf{B}(\mathbf{R}_k)^{-1}\mathbf{B}^T$ terms, and is defined as per (5.26). More specifically, for some fixed value of k , this operator drives the first kn_j rows and columns of the operand to zero.

$$\mathbf{H}_k = \begin{bmatrix} \mathbf{0}_{kn_j \times n} \\ \mathbf{F}_{k+1} \\ \mathbf{F}_{k+2} \\ \vdots \\ \mathbf{F}_N \end{bmatrix} \Leftrightarrow \begin{bmatrix} \mathbf{0}_{kn_j \times kn_j} & \\ & \mathbf{I}_{(N-k)n_j \times (N-k)n_j} \end{bmatrix} \quad (5.26)$$

With this set up, the left-hand side $\boldsymbol{\Pi} \in \mathbb{R}^{n \times n}$ is constructed as per (5.27). Note that we should not carry out explicit multiplications involving the selector matrices in practice since we can carry out this selection in software without the need for a matrix-matrix multiplication.

$$\boldsymbol{\Pi} = \sum_{k=0}^{N-1} \mathbf{H}_k \mathbf{B}(\mathbf{R}_k)^{-1} \mathbf{B}^T \quad (5.27)$$

We follow a similar procedure for defining a simplified mathematical expression for the column vector $\boldsymbol{\pi} \in \mathbb{R}^{n \times 1}$ from the right-hand side terms in (5.23b). We define this vector as a sum of four separate terms. The complete definition is shown in (5.28)

$$\boldsymbol{\pi} = \begin{bmatrix} \mathbf{r}_1^\beta \\ \mathbf{r}_2^\beta \\ \vdots \\ \mathbf{r}_N^\beta \end{bmatrix} + \sum_{k=0}^{N-1} \mathbf{H}_k \mathbf{r}_j^p - \sum_{k=0}^{N-1} (\mathbf{H}_k \mathbf{B}(\mathbf{R}_k)^{-1} \hat{\mathbf{r}}_k^a) - \sum_{k=0}^{N-1} \mathbf{H}_k (\mathbf{B}(\mathbf{R}_k)^{-1} \mathbf{B}^T) \sum_{j=k+1}^N \mathbf{r}_j^x \quad (5.28)$$

5.4.1 Application of the Sherman-Morrison Formula

In this sub-section, we focus on the efficient computation of the factorisation steps that produce the reduced system of linear equations defined in (5.24).

Firstly, note that the most computationally expensive aspect of constructing both the left-hand side matrix $\mathbf{\Pi}$ and the right-hand side vector $\boldsymbol{\pi}$ is the calculation of the $(\mathbf{R}_k)^{-1} \in \mathbb{R}^{nl \times nl}$ terms for $k = 0, 1, \dots, N - 1$. At first glance, this is particularly worrisome since the direct computation of \mathbf{R}_k definition produces a dense square matrix, and the computational complexity associated with performing the corresponding inverse is $\mathcal{O}((nl)^3)$. This is unacceptable bound given that we have to perform N such inverses twice during each iteration of the IPM to just factorise the system of equations.

However, we may devise an alternative expression for \mathbf{R}_k by exploring the structural properties of its constituent terms. Namely, we claim that the dense matrix \mathbf{R}_k can actually be rewritten as the sum of a block diagonal and rank-1 matrix. This is a promising development since we can now utilise the Sherman-Morrison formula to calculate $(\mathbf{R}_k)^{-1}$. This formula provides an efficient way of calculating the rank-1 update of some square matrix whose inverse is already known. In our case, we propose the repeated application of the Sherman-Morrison formula to compute the inverse of the block diagonal matrix also, since each block in the block diagonal matrix can be rewritten as a sum of a diagonal matrix and a rank-1 matrix. The formula is specified as shown below.

For some square invertible matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, and column vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^{n \times 1}$, the expression $(\mathbf{A} + \mathbf{u}\mathbf{v}^T)$ is invertible if and only if $1 + \mathbf{v}^T \mathbf{A}^{-1} \mathbf{u} \neq 0$. The inverse is computed using Equation (5.29).

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^T)^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1} \mathbf{u} \mathbf{v}^T \mathbf{A}^{-1}}{1 + \mathbf{v}^T \mathbf{A}^{-1} \mathbf{u}} \quad (5.29)$$

An alternative expression for \mathbf{R}_k

Let us reconsider the definitions of $\mathbf{R}_k \in \mathbb{R}^{nl \times nl}$ for $k = 0, 1, \dots, N - 1$ and its constituent components as shown in (5.30)

$$\mathbf{R}_k = -\mathbf{D}^T (\boldsymbol{\Sigma}_k)^{-1} \mathbf{D} \quad (5.30a)$$

where

$$\mathbf{D} = \begin{bmatrix} \mathbf{I}_n \otimes \mathbf{1}_l^T \\ \mathbf{1}_{nl}^T \\ -\mathbf{I}_{nl} \end{bmatrix} \quad \mathbf{D}^T = [\mathbf{I}_n \otimes \mathbf{1}_l \quad \mathbf{1}_{nl} \quad -\mathbf{I}_{nl}] \quad (5.30b)$$

$$-(\boldsymbol{\Sigma}_k)^{-1} = (\mathbf{T}_k)^{-1} \boldsymbol{\Lambda}_k \quad (5.30c)$$

Recall that $\mathbf{D} \in \mathbb{R}^{n_d \times nl}$ represents a highly sparse matrix with unit valued non-zero elements, and $\boldsymbol{\Sigma}_k \in \mathbb{R}^{n_d \times n_d}$ are defined as diagonal matrices for $k = 0, 1, \dots, N - 1$. At this point, we introduce new notation that will help us split the definition for \mathbf{R}_k into the sum of three different terms.

Firstly, let $\sigma_k^j = t_k^j \lambda_k^j$ represent the j^{th} diagonal element of $-(\boldsymbol{\Sigma}_k)^{-1}$ for $j = 1, 2, \dots, n_d$ and $k = 0, 1, \dots, N - 1$.

The $\boldsymbol{\Gamma}_k \in \mathbb{R}^{n \times n}$ object represents a diagonal matrix containing the first n diagonal elements of $-(\boldsymbol{\Sigma}_k)^{-1}$ i.e. from σ_k^1 to σ_k^n for $k = 0, 1, \dots, N - 1$. This matrix is depicted in

(5.31). Note that won't actually use this matrix directly, but its definition helps segregate the σ_k^j terms in line with the dimensions of the three sub matrices in \mathbf{D} .

$$\mathbf{\Gamma}_k = \begin{bmatrix} \sigma_k^1 & & & \\ & \sigma_k^2 & & \\ & & \ddots & \\ & & & \sigma_k^n \end{bmatrix} \quad (5.31)$$

Let $\mathbf{\Phi}_k^j \in \mathbb{R}^{l \times l}$ for $j = 1, 2, \dots, n$ represent the diagonal matrix that contains the values in the range $[\sigma_k^{n+(j-1)l+2}, \sigma_k^{n+jl+1}]$ for $k = 0, 1, \dots, N-1$. We use these matrices to split up the last nl diagonal elements of $-(\mathbf{\Sigma}_k)^{-1}$ i.e. from σ_k^{n+2} to $\sigma_k^{n_d}$ into n sub matrices of size l . Two examples of these matrices are depicted in (5.32)

$$\begin{aligned} \mathbf{\Phi}_k^1 &= \begin{bmatrix} \sigma_k^{n+2} & & & \\ & \sigma_k^{n+3} & & \\ & & \ddots & \\ & & & \sigma_k^{n+l+1} \end{bmatrix} \dots \\ \dots \mathbf{\Phi}_k^n &= \begin{bmatrix} \sigma_k^{n+(n-1)l+2} & & & \\ & \sigma_k^{n+(n-1)l+3} & & \\ & & \ddots & \\ & & & \sigma_k^{n_d} \end{bmatrix} \end{aligned} \quad (5.32)$$

With this set-up, we can write down a new expression for \mathbf{R}_k as shown in (5.33)

$$\mathbf{R}_k = [\mathbf{I}_n \otimes \mathbf{1}_l \quad \mathbf{1}_{nl} \quad -\mathbf{I}_{nl}] \begin{bmatrix} \mathbf{\Gamma}_k & & & \\ & \sigma_k^{n+1} & & \\ & & \mathbf{\Phi}_k^1 & \\ & & & \mathbf{\Phi}_k^2 \\ & & & & \ddots \\ & & & & & \mathbf{\Phi}_k^n \end{bmatrix} \begin{bmatrix} \mathbf{I}_n \otimes \mathbf{1}_l^T \\ \mathbf{1}_{nl}^T \\ -\mathbf{I}_{nl} \end{bmatrix} \quad (5.33a)$$

$$= (\mathbf{I}_n \otimes \mathbf{1}_l) \mathbf{\Gamma}_k + \sigma_k^{n+1} \mathbf{1}_{nl} \mathbf{1}_{nl}^T + \begin{bmatrix} \mathbf{\Phi}_k^1 & & & \\ & \mathbf{\Phi}_k^2 & & \\ & & \ddots & \\ & & & \mathbf{\Phi}_k^n \end{bmatrix} \quad (5.33b)$$

$$:= \mathbf{M}_k + \sigma_k^{n+1} \mathbf{1}_{nl} \mathbf{1}_{nl}^T \quad (5.33c)$$

The only reason \mathbf{R}_k appears dense when computed this way is the addition of the outer product term $\sigma_k^{n+1} \mathbf{1}_{nl} \mathbf{1}_{nl}^T$. The sum of the remaining two terms happens to represent a block diagonal matrix that is denoted by $\mathbf{M}_k \in \mathbb{R}^{nl \times nl}$ for $k = 0, 1, \dots, N-1$. This matrix

is comprised of n dense square matrices of size l . We denote each one of these sub blocks by $\mathbf{M}_k^j \in \mathbb{R}^{l \times l}$ for $j = 1, 2, \dots, n$ and $k = 0, 1, \dots, N - 1$. The block diagonal matrix \mathbf{M}_k and the definition of \mathbf{M}_k^j is depicted in (5.34).

$$\mathbf{M}_k = \begin{bmatrix} \mathbf{M}_k^1 & & & \\ & \mathbf{M}_k^2 & & \\ & & \ddots & \\ & & & \mathbf{M}_k^n \end{bmatrix} \quad (5.34a)$$

$$= \begin{bmatrix} \Phi_k^1 + \sigma_k^1 \mathbf{1}_l \mathbf{1}_l^T & & & \\ & \Phi_k^2 + \sigma_k^2 \mathbf{1}_l \mathbf{1}_l^T & & \\ & & \ddots & \\ & & & \Phi_k^n + \sigma_k^n \mathbf{1}_l \mathbf{1}_l^T \end{bmatrix} \quad (5.34b)$$

Computing $(\mathbf{R}_k)^{-1}$ using the Sherman-Morrison formula

Now that we have shown the decomposition of the matrix \mathbf{R}_k as the sum of a block diagonal and a rank-1 matrix, we utilise the Sherman-Morrison formula in order to devise an expression for $(\mathbf{R}_k)^{-1}$ as shown in (5.35)

$$\begin{aligned} (\mathbf{R}_k)^{-1} &= (\mathbf{M}_k + \sigma_k^{n+1} \mathbf{1}_{nl} \mathbf{1}_{nl}^T)^{-1} \\ &= (\mathbf{M}_k)^{-1} - \frac{\sigma_k^{n+1}}{1 + \sigma_k^{n+1} \mathbf{1}_{nl}^T (\mathbf{M}_k)^{-1} \mathbf{1}_{nl}} \cdot [(\mathbf{M}_k)^{-1} \mathbf{1}_{nl}] [\mathbf{1}_{nl}^T (\mathbf{M}_k)^{-1}] \end{aligned} \quad (5.35)$$

The inverse of the block diagonal matrix \mathbf{M}_k can be calculated by computing the inverse of each of each constituent block \mathbf{M}_k^j . This is shown in (5.36)

$$(\mathbf{M}_k)^{-1} = \begin{bmatrix} (\mathbf{M}_k^1)^{-1} & & & \\ & (\mathbf{M}_k^2)^{-1} & & \\ & & \ddots & \\ & & & (\mathbf{M}_k^n)^{-1} \end{bmatrix} \quad (5.36)$$

Because each block matrix \mathbf{M}_k^j is defined as the sum of a diagonal matrix Φ_k^j and a rank-1 matrix $\sigma_k^j \mathbf{1}_l \mathbf{1}_l^T$, its inverse is computed using the Sherman-Morrison formula again. This is shown in (5.37).

$$\begin{aligned} (\mathbf{M}_k^j)^{-1} &= (\Phi_k^j + \sigma_k^j \mathbf{1}_l \mathbf{1}_l^T)^{-1} \\ &= (\Phi_k^j)^{-1} - \frac{\sigma_k^j}{1 + \sigma_k^j \mathbf{1}_l^T (\Phi_k^j)^{-1} \mathbf{1}_l} \cdot [(\Phi_k^j)^{-1} \mathbf{1}_l] [\mathbf{1}_l^T (\Phi_k^j)^{-1}] \end{aligned} \quad (5.37)$$

The inverse of a diagonal matrix is constructed by computing the reciprocals of elements on the diagonal. The expression for $(\Phi_k^j)^{-1}$ is shown in (5.38) for completeness.

$$(\Phi_k^j)^{-1} = \begin{bmatrix} 1/\sigma_k^{n+(j-1)l+2} & & & \\ & 1/\sigma_k^{n+(j-1)l+3} & & \\ & & \ddots & \\ & & & 1/\sigma_k^{n+jl+1} \end{bmatrix} \quad (5.38)$$

5.4.2 Application of the Conjugate Gradient Method

The conjugate gradient method is one of the most widely used iterative methods for solving large systems of linear equations, and is particularly suited for use with sparse systems where the left-hand side matrix is symmetric positive-definite. We do not present a detailed background of the intuition behind this method in this section. However, the interested reader is encouraged to look through Shewchuk et al. (1994) for an intuitive overview.

Algorithm 4 specifies the steps carried out in the conjugate gradient method when used to solve the system of equations denoted by $\mathbf{\Pi}\Delta\bar{\boldsymbol{\beta}} = \boldsymbol{\pi}$. Note that left-hand side matrix $\mathbf{\Pi} \in \mathbb{R}^{n \times n}$ as defined in (5.29) is symmetric positive-definite.

Algorithm 4 The Conjugate Gradient Method

Input: $(\mathbf{\Pi}, \boldsymbol{\pi})$, $\epsilon > 0$ **Output:** $\Delta\bar{\boldsymbol{\beta}}$

- 1: **initialise** $\mathbf{d}_0 = \mathbf{r}_0 = \boldsymbol{\pi} - \mathbf{\Pi}\Delta\bar{\boldsymbol{\beta}}$, $\Delta\bar{\boldsymbol{\beta}}_0 = \boldsymbol{\pi}$
 - 2: **for** $j = 0, 1, \dots, n-1$ **do** ▷ Method is complete in n iterations
 - 3: $\alpha_j = (\mathbf{r}_j^T \mathbf{r}_j) / (\mathbf{d}_j^T \mathbf{\Pi} \mathbf{d}_j)$
 - 4: $\Delta\bar{\boldsymbol{\beta}}_{j+1} = \Delta\bar{\boldsymbol{\beta}}_j + \alpha_j \mathbf{d}_j$
 - 5: $\mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j \mathbf{\Pi} \mathbf{d}_j$
 - 6: $\beta_j = (\mathbf{r}_{j+1}^T \mathbf{r}_{j+1}) / (\mathbf{r}_j^T \mathbf{r}_j)$
 - 7: $\mathbf{d}_{j+1} = \mathbf{r}_{j+1} + \beta_j \mathbf{d}_j$
 - 8: **end for**
-

At this point, we are in a position to demonstrate the natural applicability of the conjugate gradient method to our reduced system of equations. The following developments represent the primary contribution of this work.

Theorem 5.4. *The conjugate gradient method facilitates a worst-case computational complexity of $\mathcal{O}(N^2n)$ in solving the reduced system of linear equations defined in (5.29), as long as the low-level constituent elements of the left-hand side matrix (as opposed to its precomputed dense formulation) are utilised within the algorithm*

Proof. The first step involves substituting the expression for $(\mathbf{R}_k)^{-1}$ from (5.35) into the left-hand side matrix $\mathbf{\Pi}$. This is shown in (5.39)

$$\mathbf{\Pi} = \sum_{k=0}^{N-1} \mathbf{H}_k \mathbf{B}(\mathbf{R}_k)^{-1} \mathbf{B}^T \quad (5.39a)$$

$$= \sum_{k=0}^{N-1} \mathbf{H}_k \mathbf{B} \{ (\mathbf{M}_k)^{-1} - \rho_k [(\mathbf{M}_k)^{-1} \mathbf{1}_{nl}] [\mathbf{1}_{nl}^T (\mathbf{M}_k)^{-1}] \} \mathbf{B}^T \quad (5.39b)$$

$$= \sum_{k=0}^{N-1} \mathbf{H}_k \mathbf{B} (\mathbf{M}_k)^{-1} \mathbf{B}^T - \sum_{k=0}^{N-1} \rho_k \mathbf{H}_k \{ \mathbf{B} [(\mathbf{M}_k)^{-1} \mathbf{1}_{nl}] \} \{ [\mathbf{1}_{nl}^T (\mathbf{M}_k)^{-1}] \mathbf{B}^T \} \quad (5.39c)$$

$$:= \sum_{k=0}^{N-1} \mathbf{H}_k \mathbf{\Gamma}_k - \sum_{k=0}^{N-1} \mathbf{H}_k \mathbf{v}_k \mathbf{v}_k^T \quad (5.39d)$$

We define three new terms in order to simplify the equations above. Firstly, let $\mathbf{\Gamma}_k \in \mathbb{R}^{n \times n}$ represent a diagonal matrix as shown in (5.40a). Secondly, let $\mathbf{v}_k \in \mathbb{R}^{n \times 1}$ denote a column vector as shown in (5.40b). Lastly, the scalar ρ_k is defined as per (5.40c). All these quantities are defined for $k = 0, 1, \dots, N-1$.

$$\mathbf{\Gamma}_k = \mathbf{B}(\mathbf{M}_k)^{-1} \mathbf{B}^T \quad (5.40a)$$

$$\mathbf{v}_k = \mathbf{B}[(\mathbf{M}_k)^{-1} \mathbf{1}_{nl}] \quad (5.40b)$$

$$\rho_k = \frac{\sigma_k^{n+1}}{1 + \sigma_k^{n+1} \mathbf{1}_{nl}^T (\mathbf{M}_k)^{-1} \mathbf{1}_{nl}} \quad (5.40c)$$

We have now obtained a useful expression for the left-hand side matrix. Namely, $\mathbf{\Pi}$ is expressed as a sum of N diagonal matrices and rank-1 matrices that become increasingly sparse for higher values of k due to the leading H_k selection operators.

The key step in this proof is in using the final expression (5.39d) instead of (5.39a) when performing the relevant multiplications in the conjugate gradient algorithm. Note in particular lines 3 and 5 from Algorithm 4, where the matrix-vector product $\mathbf{\Pi} \mathbf{d}_j$ is carried out. Using (5.39a) to compute this expression generates a quadratic complexity in the number of tasks n .

Alternatively, imposing the order of multiplication shown in (5.41a) leads the computation of products that scales linearly in the number of tasks n . This is because of the linear complexity associated with computing the diagonal matrix-vector product $\mathbf{\Pi}_k \mathbf{d}_j$ and the linear complexity associated with the inner product $\mathbf{v}_k^T \mathbf{d}_j$.

$$\mathbf{\Pi} \mathbf{d}_j = \sum_{k=0}^{N-1} \mathbf{H}_k (\mathbf{\Gamma}_k \mathbf{d}_j) - \sum_{k=0}^{N-1} \mathbf{H}_k \mathbf{v}_k (\mathbf{v}_k^T \mathbf{d}_j) \quad (5.41a)$$

Note that the summation over N multiplications leads to a complexity of $\mathcal{O}(Nn)$ for (5.41a). Moreover, because the conjugate gradient method is complete after n iterations in general, the overall computational complexity is given by $\mathcal{O}(Nn^2)$.

However, we make use of Theorem 10.2.5 from Golub & Van Loan (2012), which shows that the conjugate gradient method does not always require n iterations. Consider the sum of two arbitrary n dimensional square matrices such that one of the matrices is symmetric positive-definite, and the other is rank deficient with rank $q < n$. Under these conditions, the conjugate gradient method will terminate in at most $q + 1$ iterations.

These conditions apply exactly to the left-hand side matrix $\mathbf{\Pi}$ in our problem. As highlighted before, this matrix is effectively a sum of two n -dimensional matrices: a diagonal matrix (symmetric positive-definite) and a dense matrix constructed out of N outer products (rank $N < n$). Therefore, we can claim that the conjugate gradient method will terminate in at most $N + 1$ iterations instead of n . This results in an overall complexity of $\mathcal{O}(N^2n)$ as opposed to $\mathcal{O}(Nn^2)$, and completes the proof. \square

5.5 Solving WP-DDP with Custom IPM

In this section, we will obtain a solution to the WP-DDP using a custom written IPM in MATLAB. The primary objective here is to demonstrate the practical relevance of the algebraic manipulations carried out on the system of linear equations in the preceding sections. Most of the fundamental block matrices defined in this work are sparse matrices with unit valued non-zero elements. We aim to incorporate this structural information within low-level multiplication and addition operations in order to enhance scaling performance.

5.5.1 Implementation Details

We first formally present the algorithms that were implemented in MATLAB. The focus in this chapter has lied on reducing the dimensionality of the overall problem whilst keeping track of the specific elimination procedures. The different iterations on the custom IPM solver also follow this trend. More specifically, the following MATLAB functions were implemented in order to both substantiate and reinforce theoretical reasoning.

DDP-AK7213-V2

This MATLAB routine represents the first attempt at solving the WP-DDP using a custom written IPM. We do not solve the complete KKT system of equations in this function. Instead, we start with the reduced system depicted in (5.16) where the $\Delta \mathbf{t}_k$ and $\Delta \mathbf{\lambda}_k$ terms have been eliminated.

The DDP-AK7213-V2 procedure that was implemented in MATLAB is fully described in Algorithm 5. During each iteration, we eliminate the $\Delta \mathbf{t}_k$ and $\Delta \mathbf{\lambda}_k$ terms in order to solve the reduced system of equations, and recover them in order to facilitate the remaining aspects of the predictor-corrector algorithm.

Algorithm 5 DDP-AK7213-V2

```

1: procedure DDP-AK7213-V2( $n, N, l, m, \epsilon, \bar{\tau}, \mathbf{s}, \mathbf{P}, \hat{\mathbf{x}}$ )
2:   initialise  $(\mathbf{x}_k, \boldsymbol{\beta}_k) > \mathbf{0} \forall k \in [1, N]$ ,  $(\mathbf{a}_k, \boldsymbol{\lambda}_k, \mathbf{p}_k, \mathbf{t}_k) > \mathbf{0} \forall k \in [0, N - 1]$ 
3:   define constants  $\mathbf{B}, \mathbf{D}$  and  $\mathbf{d}$  as per (4.5), (4.6) and (4.7) respectively
4:   construct the reduced Jacobian matrix (5.16) without the  $\mathbf{R}_k$  block matrices.
5:   while  $\mu > \epsilon$  do
6:     compute the  $\mathbf{R}_k$  terms as per (5.14a)
7:     update the reduced Jacobian matrix (5.16) with the  $\mathbf{R}_k$  terms
8:     compute predictor residuals  $\mathbf{r}_k^{\mathbf{x}}, \mathbf{r}_k^{\mathbf{a}}, \mathbf{r}_k^{\boldsymbol{\lambda}}, \mathbf{r}_k^{\mathbf{p}}, \mathbf{r}_k^{\mathbf{t}}, \mathbf{r}_k^{\boldsymbol{\beta}}$  as per (5.5)
9:     compute  $\hat{\mathbf{r}}_k^{\boldsymbol{\lambda}}, \hat{\mathbf{r}}_k^{\mathbf{a}}$  as per (5.9b) and (5.14b) respectively.
10:    solve for the predictor step  $(\Delta \mathbf{x}_k^{\text{aff}}, \Delta \mathbf{a}_k^{\text{aff}}, \Delta \mathbf{p}_k^{\text{aff}}, \Delta \boldsymbol{\beta}_k^{\text{aff}})$  as per (5.16)
11:    recover  $\Delta \boldsymbol{\lambda}_k^{\text{aff}}, \Delta \mathbf{t}_k^{\text{aff}}$  as per (5.13a) and (5.8a) respectively
12:     $\alpha^{\text{aff}} = \arg \max \{ \alpha \in [0, 1] : (\boldsymbol{\lambda}_k, \mathbf{t}_k) + \alpha(\Delta \boldsymbol{\lambda}_k^{\text{aff}}, \Delta \mathbf{t}_k^{\text{aff}}) \geq \mathbf{0} \forall k \in [0, N - 1] \}$ 
13:     $\mu^{\text{aff}} = \frac{1}{Nn_d} \sum_{k=0}^{N-1} (\boldsymbol{\lambda}_k + \alpha^{\text{aff}} \Delta \boldsymbol{\lambda}_k^{\text{aff}})^T (\mathbf{t}_k + \alpha^{\text{aff}} \Delta \mathbf{t}_k^{\text{aff}})$ 
14:     $\sigma = (\mu^{\text{aff}} / \mu)^3$ 
15:    compute corrector residuals  $\mathbf{r}_k^{\mathbf{x}}, \mathbf{r}_k^{\mathbf{a}}, \mathbf{r}_k^{\boldsymbol{\lambda}}, \mathbf{r}_k^{\mathbf{p}}, \mathbf{r}_k^{\boldsymbol{\beta}} = \mathbf{0}$  and  $\mathbf{r}_k^{\mathbf{t}} = \Delta \mathbf{X}_k^{\text{aff}} \Delta \mathbf{S}_k^{\text{aff}} \mathbf{e} - \sigma \mu \mathbf{e}$ 
16:    compute  $\hat{\mathbf{r}}_k^{\boldsymbol{\lambda}}, \hat{\mathbf{r}}_k^{\mathbf{a}}$  as per (5.9b) and (5.14b) respectively.
17:    solve for the corrector step  $(\Delta \mathbf{x}_k^{\text{cc}}, \Delta \mathbf{a}_k^{\text{cc}}, \Delta \mathbf{p}_k^{\text{cc}}, \Delta \boldsymbol{\beta}_k^{\text{cc}})$  as per (5.16)
18:    recover  $\Delta \boldsymbol{\lambda}_k^{\text{cc}}, \Delta \mathbf{t}_k^{\text{cc}}$  as per (5.13a) and (5.8a) respectively
19:     $(\Delta \mathbf{x}_k, \Delta \mathbf{a}_k, \Delta \boldsymbol{\lambda}_k, \Delta \mathbf{p}_k, \Delta \mathbf{t}_k, \Delta \boldsymbol{\beta}_k) = (\Delta \mathbf{x}_k^{\text{aff}}, \Delta \mathbf{a}_k^{\text{aff}}, \Delta \boldsymbol{\lambda}_k^{\text{aff}}, \Delta \mathbf{p}_k^{\text{aff}}, \Delta \mathbf{t}_k^{\text{aff}}, \Delta \boldsymbol{\beta}_k^{\text{aff}}) \dots$ 
20:     $\dots + (\Delta \mathbf{x}_k^{\text{cc}}, \Delta \mathbf{a}_k^{\text{cc}}, \Delta \boldsymbol{\lambda}_k^{\text{cc}}, \Delta \mathbf{p}_k^{\text{cc}}, \Delta \mathbf{t}_k^{\text{cc}}, \Delta \boldsymbol{\beta}_k^{\text{cc}})$ 
21:     $\alpha = \arg \max \{ \alpha \in [0, 1] : (\boldsymbol{\lambda}_k, \mathbf{t}_k) + \alpha(\Delta \boldsymbol{\lambda}_k, \Delta \mathbf{t}_k) \geq \mathbf{0} \forall k \in [0, N - 1] \}$ 
22:     $(\mathbf{x}_k, \mathbf{a}_k, \boldsymbol{\lambda}_k, \mathbf{p}_k, \mathbf{t}_k, \boldsymbol{\beta}_k) = (\mathbf{x}_k, \mathbf{a}_k, \boldsymbol{\lambda}_k, \mathbf{p}_k, \mathbf{t}_k, \boldsymbol{\beta}_k) \dots$ 
23:     $\dots + \alpha(\Delta \mathbf{x}_k, \Delta \mathbf{a}_k, \Delta \boldsymbol{\lambda}_k, \Delta \mathbf{p}_k, \Delta \mathbf{t}_k, \Delta \boldsymbol{\beta}_k)$ 
24:     $\mu = \frac{1}{Nn_d} \sum_{k=0}^{N-1} \boldsymbol{\lambda}_k^T \mathbf{t}_k$ 
25:  end while
26: end procedure

```

DDP-AK7213-V3

This MATLAB function carries out further block eliminations until we are left with equations in terms of $\Delta \boldsymbol{\beta}_k$ only. More specifically, we solve the n dimensional system of equations depicted in (5.24).

Note that version makes no attempt to perform efficient factorisations of the reduced system. The constituent $(\mathbf{R}_k)^{-1}$ terms are precomputed at the start of every iteration, using the `\function`. These terms are then used to construct the left-hand side matrix $\boldsymbol{\Pi}$ and the right-hand side vector π for both the predictor and corrector computations. The system of equations is solved using the `\function` as well. The complete procedure is depicted in Algorithm (6)

Algorithm 6 DDP-AK7213-V3

```

1: procedure DDP-AK7213-V3( $n, N, l, m, \epsilon, \bar{\tau}, \mathbf{s}, \mathbf{P}, \hat{\mathbf{x}}$ )
2:   initialise  $(\mathbf{x}_k, \boldsymbol{\beta}_k) > \mathbf{0} \forall k \in [1, N]$ ,  $(\mathbf{a}_k, \boldsymbol{\lambda}_k, \mathbf{p}_k, \mathbf{t}_k) > \mathbf{0} \forall k \in [0, N - 1]$ 
3:   define constants  $\mathbf{B}, \mathbf{D}$  and  $\mathbf{d}$  as per (4.5), (4.6) and (4.7) respectively
4:   while  $\mu > \epsilon$  do
5:     compute the  $\mathbf{R}_k$  terms as per (5.14a)
6:     compute the  $(\mathbf{R}_k)^{-1}$  terms using the \function
7:     compute the left-hand side matrix  $\boldsymbol{\Pi}$  as per (5.27)
8:     compute predictor residuals  $\mathbf{r}_k^{\mathbf{x}}, \mathbf{r}_k^{\mathbf{a}}, \mathbf{r}_k^{\boldsymbol{\lambda}}, \mathbf{r}_k^{\mathbf{p}}, \mathbf{r}_k^{\mathbf{t}}, \mathbf{r}_k^{\boldsymbol{\beta}}$  as per (5.5)
9:     compute the predictor right-hand side  $\boldsymbol{\pi}$  as per (5.28)
10:    solve for the predictor step  $\Delta \bar{\boldsymbol{\beta}}_k^{\text{aff}}$  with the \operator
11:    recover  $\Delta \mathbf{p}_k^{\text{aff}}, \Delta \mathbf{a}_k^{\text{aff}}, \Delta \mathbf{x}_k^{\text{aff}}$  as per (5.20), (5.18) and (5.19) respectively
12:    recover  $\Delta \boldsymbol{\lambda}_k^{\text{aff}}, \Delta \mathbf{t}_k^{\text{aff}}$  as per (5.13a) and (5.8a) respectively
13:     $\alpha^{\text{aff}} = \arg \max \{ \alpha \in [0, 1] : (\boldsymbol{\lambda}_k, \mathbf{t}_k) + \alpha(\Delta \boldsymbol{\lambda}_k^{\text{aff}}, \Delta \mathbf{t}_k^{\text{aff}}) \geq \mathbf{0} \forall k \in [0, N - 1] \}$ 
14:     $\mu^{\text{aff}} = \frac{1}{Nn_d} \sum_{k=0}^{N-1} (\boldsymbol{\lambda}_k + \alpha^{\text{aff}} \Delta \boldsymbol{\lambda}_k^{\text{aff}})^{\text{T}} (\mathbf{t}_k + \alpha^{\text{aff}} \Delta \mathbf{t}_k^{\text{aff}})$ 
15:     $\sigma = (\mu^{\text{aff}} / \mu)^3$ 
16:    compute corrector residuals  $\mathbf{r}_k^{\mathbf{x}}, \mathbf{r}_k^{\mathbf{a}}, \mathbf{r}_k^{\boldsymbol{\lambda}}, \mathbf{r}_k^{\mathbf{p}}, \mathbf{r}_k^{\mathbf{t}} = \mathbf{0}$  and  $\mathbf{r}_k^{\boldsymbol{\beta}} = \Delta \mathbf{X}_k^{\text{aff}} \Delta \mathbf{S}_k^{\text{aff}} \mathbf{e} - \sigma \mu \mathbf{e}$ 
17:    compute the corrector right-hand side  $\boldsymbol{\pi}$  as per (5.28)
18:    solve for the corrector step  $\Delta \bar{\boldsymbol{\beta}}_k^{\text{cc}}$  with the \operator
19:    recover  $\Delta \mathbf{p}_k^{\text{cc}}, \Delta \mathbf{a}_k^{\text{cc}}, \Delta \mathbf{x}_k^{\text{cc}}$  as per (5.20), (5.18) and (5.19) respectively
20:    recover  $\Delta \boldsymbol{\lambda}_k^{\text{cc}}, \Delta \mathbf{t}_k^{\text{cc}}$  as per (5.13a) and (5.8a) respectively
21:     $(\Delta \mathbf{x}_k, \Delta \mathbf{a}_k, \Delta \boldsymbol{\lambda}_k, \Delta \mathbf{p}_k, \Delta \mathbf{t}_k, \Delta \boldsymbol{\beta}_k) = (\Delta \mathbf{x}_k^{\text{aff}}, \Delta \mathbf{a}_k^{\text{aff}}, \Delta \boldsymbol{\lambda}_k^{\text{aff}}, \Delta \mathbf{p}_k^{\text{aff}}, \Delta \mathbf{t}_k^{\text{aff}}, \Delta \boldsymbol{\beta}_k^{\text{aff}}) \dots$ 
22:     $\dots + (\Delta \mathbf{x}_k^{\text{cc}}, \Delta \mathbf{a}_k^{\text{cc}}, \Delta \boldsymbol{\lambda}_k^{\text{cc}}, \Delta \mathbf{p}_k^{\text{cc}}, \Delta \mathbf{t}_k^{\text{cc}}, \Delta \boldsymbol{\beta}_k^{\text{cc}})$ 
23:     $\alpha = \arg \max \{ \alpha \in [0, 1] : (\boldsymbol{\lambda}_k, \mathbf{t}_k) + \alpha(\Delta \boldsymbol{\lambda}_k, \Delta \mathbf{t}_k) \geq \mathbf{0} \forall k \in [0, N - 1] \}$ 
24:     $(\mathbf{x}_k, \mathbf{a}_k, \boldsymbol{\lambda}_k, \mathbf{p}_k, \mathbf{t}_k, \boldsymbol{\beta}_k) = (\mathbf{x}_k, \mathbf{a}_k, \boldsymbol{\lambda}_k, \mathbf{p}_k, \mathbf{t}_k, \boldsymbol{\beta}_k) \dots$ 
25:     $\dots + \alpha(\Delta \mathbf{x}_k, \Delta \mathbf{a}_k, \Delta \boldsymbol{\lambda}_k, \Delta \mathbf{p}_k, \Delta \mathbf{t}_k, \Delta \boldsymbol{\beta}_k)$ 
26:     $\mu = \frac{1}{Nn_d} \sum_{k=0}^{N-1} \boldsymbol{\lambda}_k^{\text{T}} \mathbf{t}_k$ 
27:  end while
28: end procedure

```

DDP-AK7213-V4

This MATLAB function incorporates an efficient factorisation of $\boldsymbol{\Pi}$ through the use of the Sherman-Morrison formula. In this algorithm, we no longer pre-compute the $(\mathbf{R}_k)^{-1}$ terms. Instead, we pre-compute the $(\mathbf{M}_k)^{-1}$ terms that appear in Sherman-Morrison formula for $(\mathbf{R}_k)^{-1}$ as shown in (5.35). Recall that the block diagonal sub matrices of $(\mathbf{M}_k)^{-1}$ can be expressed as the sum of a diagonal matrix and a rank-1 matrix, which in turn enables the fast calculation of the overall inverse through the repeated use of the Sherman-Morrison formula as shown in (5.37).

Note that there exist four separate expressions involving $(\mathbf{R}_k)^{-1}$ that arise when solving the reduced system of equations. The $\mathbf{B}(\mathbf{R}_k)^{-1}\mathbf{B}^T$ and $\mathbf{B}(\mathbf{R}_k)^{-1}\hat{\mathbf{r}}_k^{\mathbf{a}}$ terms are used to compute $\mathbf{\Pi}$ and $\mathbf{\pi}$, whereas the $-(\mathbf{R}_k)^{-1}\hat{\mathbf{r}}_k^{\mathbf{a}}$ and $-(\mathbf{R}_k)^{-1}(\mathbf{B}^T\Delta\mathbf{p}_k)$ terms are used in the reconstruction of $\Delta\mathbf{a}_k$. Therefore, all four of these expressions must be restated in terms of $(\mathbf{M}_k)^{-1}$ in order to construct an algorithm without the explicit computation of $(\mathbf{R}_k)^{-1}$ during each iteration. We do not repeat the detailed derivations that were carried out in performing these transformations since the restating of very low-level algebraic manipulations does not necessarily add value to the fundamental argument for performing them in the first place. Note that as a result, we refrain from formally stating the DDP-AK7213-V4 MATLAB routine in the body of this work. The function details have been provided in Appendix A instead.

The system of equations in this algorithm is solved using the `\function`. However, we have also written a custom conjugate gradient method that accepts the constituent elements of the left-hand side matrix. The theoretical computational complexity bounds that were derived in the previous section motivate this effort. The use of a custom conjugate-gradient method requires a slight modification to the MATLAB procedure such that the matrix $\mathbf{\Pi}$ is never actually constructed, but only available implicitly in terms of its spectral components.

Overall, the direct incorporation of the Sherman-Morrison formula in conjunction with a custom written conjugate gradient method represents an effective modification to Algorithm 6.

DDP-AK7213-V5

This MATLAB function represents an implementation of DDP-AK7213-V4 where no expensive inverses are precomputed at the start of every IPM iteration. This algorithm performs a carefully ordered sequence of algebraic operations at the lowest possible implementation level. In fact, the function was written with the aim of expressing every expression as a scalar-vector product and/or an inner product at most. As with DDP-AK7213-V4, this function is also written in a way that facilitates the use of the custom conjugate gradient method. Further details are provided in Appendix A.

5.5.2 Simulation Results and Analysis

DDP-AK7213-V2

First, we present the timing results for a relatively small problem size of $(n, N, l) = (20, 20, 20)$ with $m = 1000$ available processors. The `\function` was used to solve the system of linear equations, and a duality gap threshold of 0.0001 was used to impose a reasonably fair stopping criterion. The algorithm returned a feasible solution in 21 iterations.

The timing analysis is carried out with the help of the MATLAB `profile` routine. Figure 5.1 illustrates a snippet of the profile report that was generated. This result is in line with our expectations, since we have previously seen a large proportion of time allocated

towards obtaining a solution to the KKT system of equations in its complete form. Clearly, the preliminary eliminations of the $\Delta \mathbf{t}_k$ and $\Delta \mathbf{\lambda}_k$ terms does not help drive down the absolute times to convergence.

Lines where the most time was spent




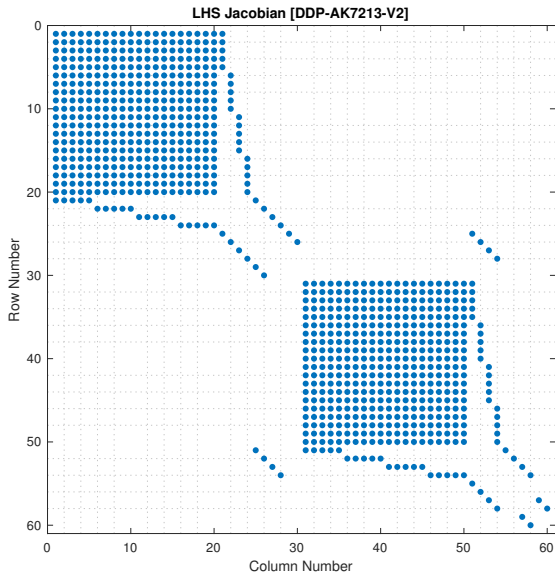
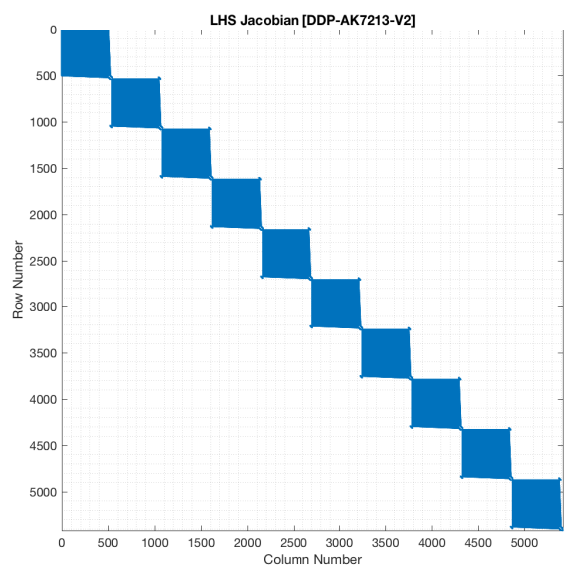
Line Number	Code	Calls	Total Time	% Time	Time Plot
308	CSOL = LHS \ CRHS;	21	170.455 s	48.6%	
220	PSOL = LHS \ PRHS;	21	170.398 s	48.6%	
170	LHS = sparse([Rows; RRows], [...	21	7.770 s	2.2%	
168	[~,~,RVals(:,k)] = find(-D' *...	420	1.381 s	0.4%	
166	RVals = zeros(nl^2,N);	21	0.217 s	0.1%	
All other lines			0.731 s	0.2%	
Totals			350.952 s	100%	

Figure 5.1: Function Profile Report [DDP-AK7213-V2]

Figure 5.2a and 5.2b illustrate the structure of the reduced left-hand side matrix for different problem sizes when visualised using the `spy` function in MATLAB. Notice the presence of the dense \mathbf{R}_k matrices along the main diagonal. A reduction in the dimensions of the problem is seen to have reduced the sparsity of the left-hand side matrix as well.



(a) Jacobian for $(n, N, l) = (4, 2, 5)$



(b) Jacobian for $(n, N, l) = (20, 10, 25)$

Figure 5.2: Sparsity pattern of the reduced Jacobian matrix for different problem sizes [DDP-AK7213-V2]

Obtaining simulation results for this algorithm proved difficult due to the unpredictably lengthy periods of time spent within the `\function` (note the 350 seconds required to compute a relatively small test problem). Employing iterative methods such as `minres` and `cgs` did not have a significant impact on times to convergence either.

The biggest takeaway from the testing procedure for this function is that general purpose linear algebra solvers are not very good at exploiting the sparsity structures present in very large systems of equations. In order to achieve encouraging times to convergence, we must first reduce the dimensionality of the system and solve it in manner that exploits the structural properties of the complete Jacobian matrix. To this end, we proceed to the analysis of MATLAB routines motivated by this ethos.

DDP-AK7213-V3

For the purposes of a direct comparison between two consecutive function iterations, we begin with the timing analysis of this method for the same problem size parameters i.e. only the function handle was modified within the simulation environment. Figure 5.3 illustrates a snippet of the profile report that was subsequently generated. This function returned a feasible solution in 22 iterations that required a total of 5.712s to complete. This is a stark contrast to the 350.952s required by the DDP-AK7213-V2 function. Recall that the primary difference between the two functions is the dimensionality of the system of equations, and the reduced size is seen to have a clear positive impact on the time to convergence.

Lines where the most time was spent







Line Number	Code	Calls	Total Time	% Time	Time Plot
63	<code>invRs(:, :, k) = R \ eye(size(R)...</code>	420	3.293 s	57.6%	
62	<code>R = D'*diag(t(:, k).^-1 .* lamb...</code>	420	0.517 s	9.1%	
225	<code>gg = (B*invRs(:, :, k)) * (B'*su...</code>	420	0.238 s	4.2%	
125	<code>gg = (B*invRs(:, :, k)) * (B'*su...</code>	420	0.235 s	4.1%	
69	<code>G = B * invRs(:, :, k) * B';</code>	420	0.213 s	3.7%	
All other lines			1.216 s	21.3%	
Totals			5.712 s	100%	

Figure 5.3: Function Profile Report [DDP-AK7213-V3]

The largest proportion of time is spent on pre-calculating the $(\mathbf{R}_k)^{-1}$ terms at the start of every iteration. The reduced system of equations in Newton's method accounts for a comparatively negligible amount. This is because we compute a total of N inverses of the nl -dimensional \mathbf{R}_k terms during each IPM iteration. On the other hand, the n -dimensional system of equations needs to be solved only twice per iteration.

Figure 5.4 illustrates the relationship between the time taken per iteration versus the number of tasks on a logarithmic grid. This experiment was run for $(N, l) = (10, 10)$ with $m = 1000$ available processors. We cycle through values of n in the range $[10, 100]$. The line of best fit is given by the equation below. An exponent of 2.734315 indicates an almost cubic complexity relationship with the number of tasks. This is most likely due to

the computation of the inverses of the dense nl -dimensional \mathbf{R}_k terms.

$$\text{Time/Iteration} = 0.000006 \cdot (\# \text{ of Tasks})^{2.734315} - 0.004489 \quad (5.42)$$

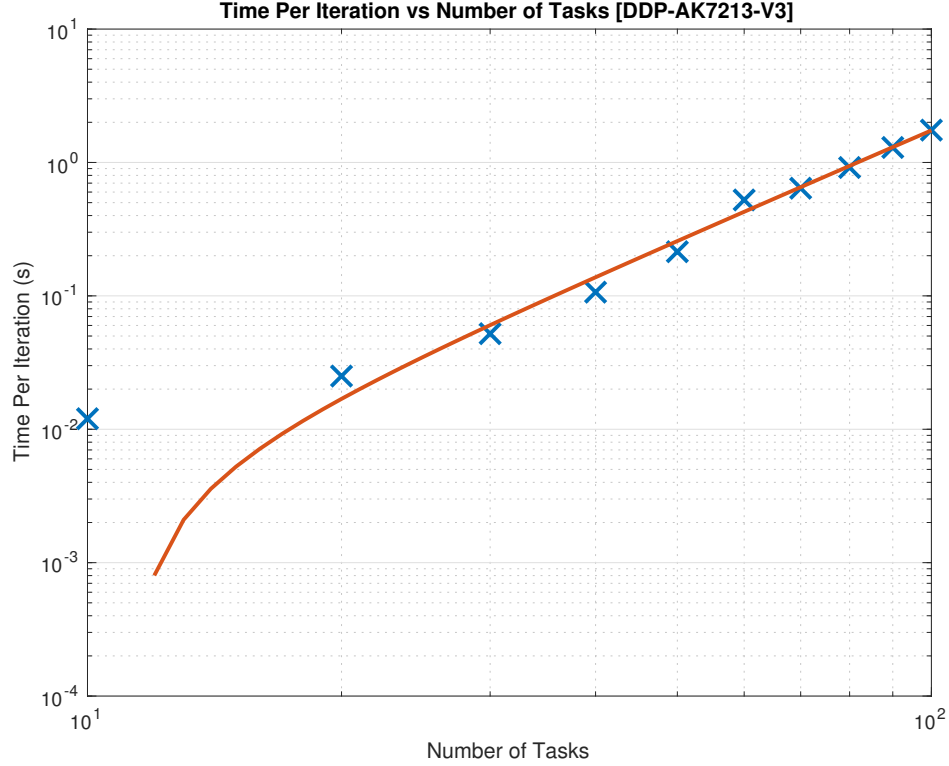


Figure 5.4: Time per iteration vs Number of Tasks [DDP-AK7213-V3]

Figure 5.5 illustrates the relationship between the time taken per iteration versus the number of time steps on a logarithmic grid. This experiment was run for $(n, l) = (100, 10)$ with $m = 1000$ available processors. We cycle through values of N in the range $[10, 100]$ such that the number of tasks due at each time step n_j is an integer number. The line of best fit is given by the equation below. These results show that the time per iteration grows approximately linearly with an increasing number of time steps.

$$\text{Time/Iteration} = 0.178007 \cdot (\# \text{ of Time Steps})^{0.985120} - 0.504123 \quad (5.43)$$

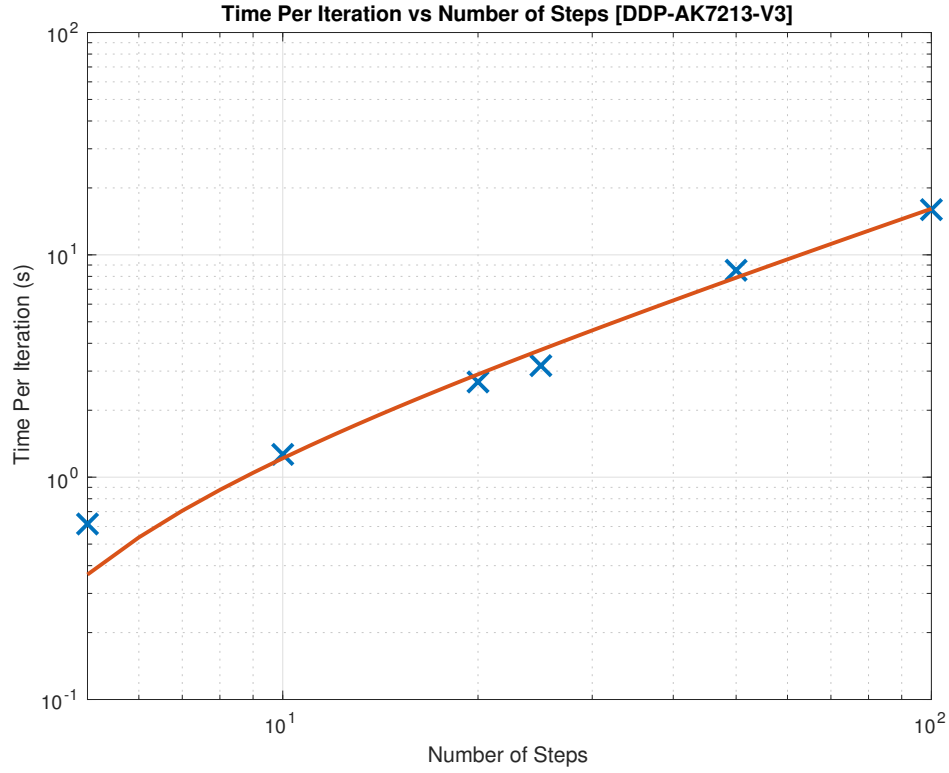


Figure 5.5: Time per iteration vs Number of Time Steps [DDP-AK7213-V3]

Figure 5.6 illustrates the relationship between the time taken per iteration versus the number of available speed levels on a logarithmic grid. This experiment was run for $(n, N) = (10, 10)$ with $m = 1000$ available processors. We cycle through values of l in the range $[10, 100]$. The line of best fit is given by the equation below. The exponent of 2.482951 can be explained using the same argument made for the relationship with the number of tasks: the inefficient computation of the \mathbf{R}^{-1} terms jeopardises the practical relevance of this algorithm.

$$\text{Time/Iteration} = 0.000015 \cdot (\# \text{ of Speed Levels})^{2.482951} - 0.017728 \quad (5.44)$$

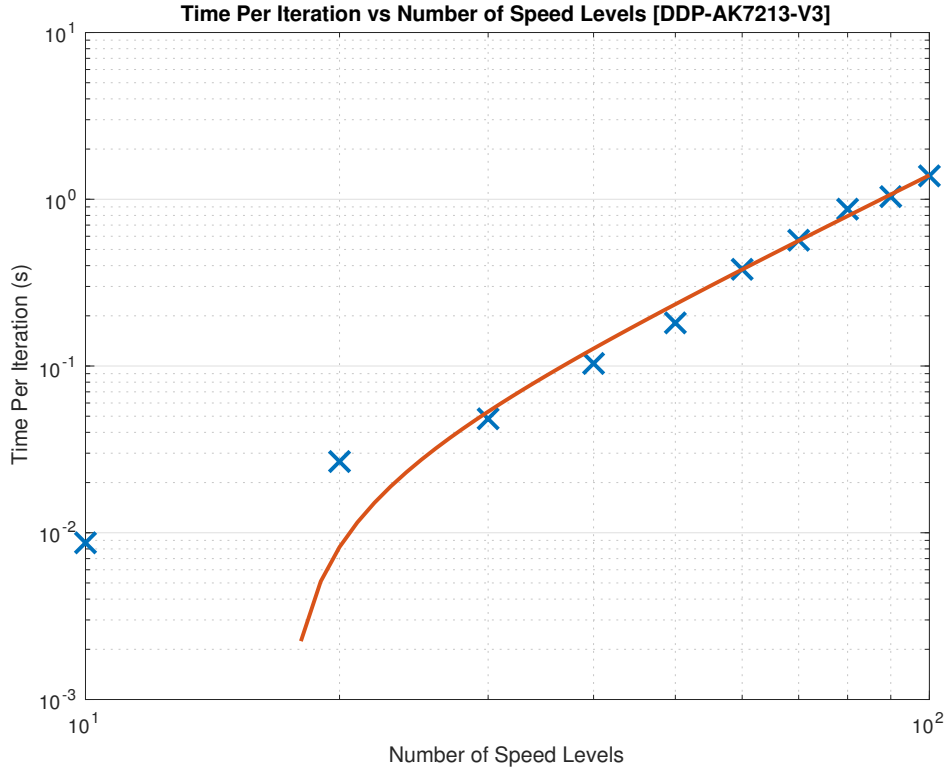


Figure 5.6: Time per iteration vs Number of Speed Levels [DDP-AK7213-V3]

DDP-AK7213-V4

Next, we consider an algorithm that avoids the direct calculation of any inverses where possible. The inverses are computed using the Sherman-Morrison formula instead. The reduced system is solved using a custom written conjugate gradient method. The details of this routine are presented in Appendix A.

Figure 5.7 depicts the log-log plot of time per iteration against the number of tasks. This experiment was run for $(N, l) = (10, 10)$ with $m = 1000$ available processors. We cycle through values of n in the range $[10, 1000]$. The line of best fit is given by the equation below.

This method is seen to grow approximately linearly with the number of tasks. This is an important practical result that represents a clear improvement over the DDP-AK7213-V3 function.

$$\text{Time/Iteration} = 0.000109 \cdot (\# \text{ of Tasks})^{1.168254} + 0.020837 \quad (5.45)$$

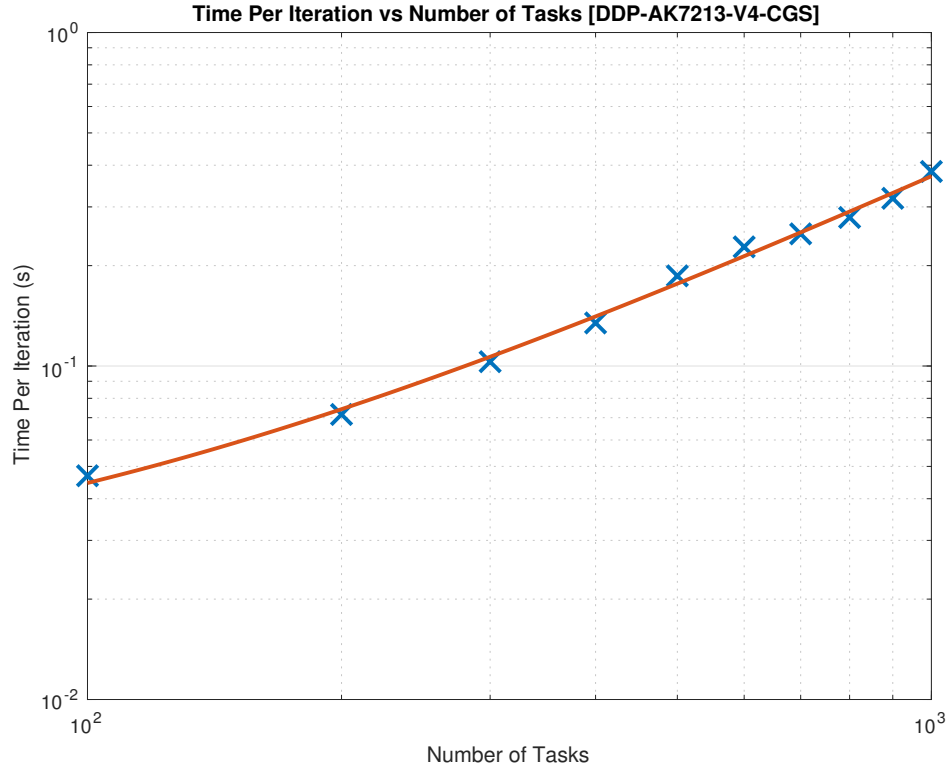


Figure 5.7: Time per iteration vs Number of Tasks [DDP-AK7213-V4CGS]

Figure 5.8 depicts the log-log plot of time per iteration against the number of time steps. This experiment was run for $(n, l) = (100, 10)$ with $m = 1000$ available processors. We cycle through values of N in the range $[10, 100]$ such that the number of tasks due at each time step n_j is an integer number. The line of best fit is given by the equation below. An exponent of 1.012967 shows that linear relationship with the number of time steps is maintained.

$$\text{Time/Iteration} = 0.003333 \cdot (\# \text{ of Time Steps})^{1.012967} + 0.005426 \quad (5.46)$$

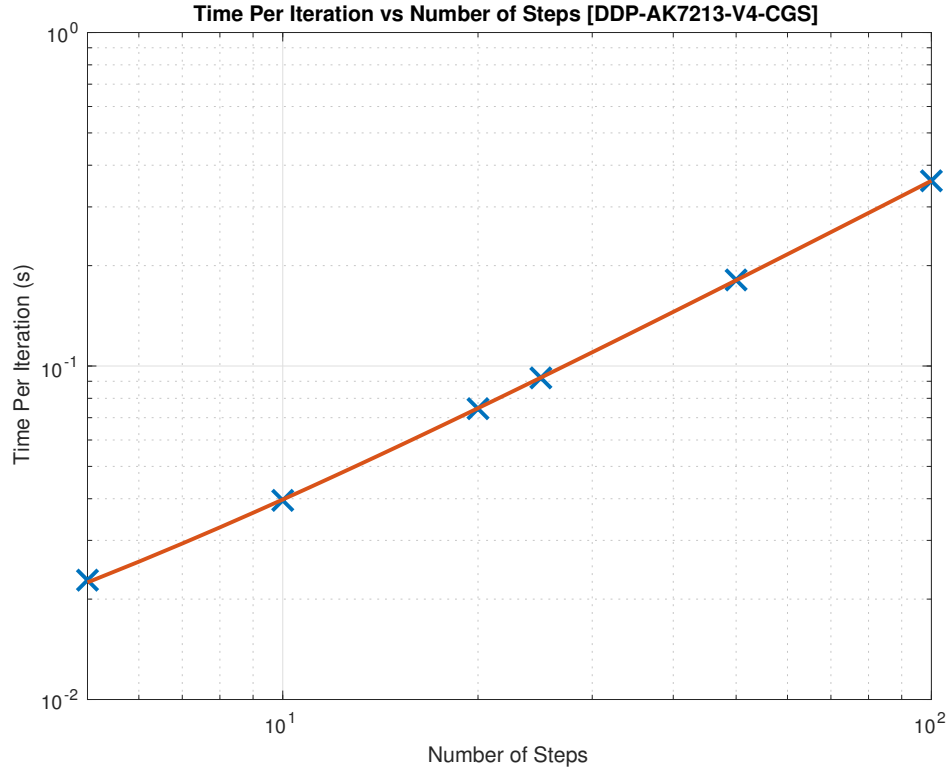


Figure 5.8: Time per iteration vs Number of Time Steps [DDP-AK7213-V4CGS]

Figure 5.9 depicts the relationship between the time taken per iteration versus the number of available speed levels on a log-log plot. This experiment was run for $(n, N) = (10, 10)$ with $m = 1000$ available processors. We cycle through values of l in the range $[100, 1000]$. The line of best fit is given by the equation below. An exponent of 2.053266 does not represent a huge improvement over DDP-AK7213-V3 for an increasing number of speed levels.

$$\text{Time/Iteration} = 0.000004 \cdot (\# \text{ of Speed Levels})^{2.053266} - 0.012082 \quad (5.47)$$

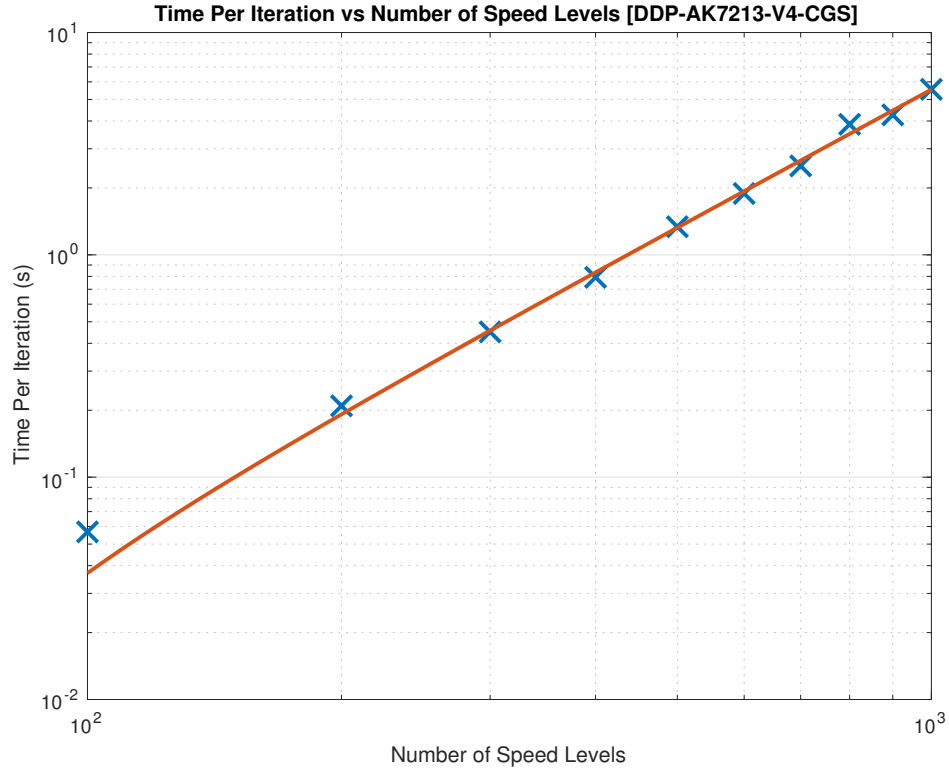


Figure 5.9: Time per iteration vs Number of Speed Levels [DDP-AK7213-V4CGS]

DDP-AK7213-V5

Lastly, we consider our most intricate algorithm that performs algebraic operations at the lowest possible level of implementation.

Figure 5.10 depicts the log-log plot of time per iteration against the number of tasks. This experiment was run for $(N, l) = (10, 10)$ with $m = 1000$ available processors. We cycle through values of n in the range $[10, 1000]$. The line of best fit is given by the equation below. This algorithm shares a linear relationship with the number of tasks.

$$\text{Time/Iteration} = 0.001486 \cdot (\# \text{ of Tasks})^{1.020554} + 0.021423 \quad (5.48)$$

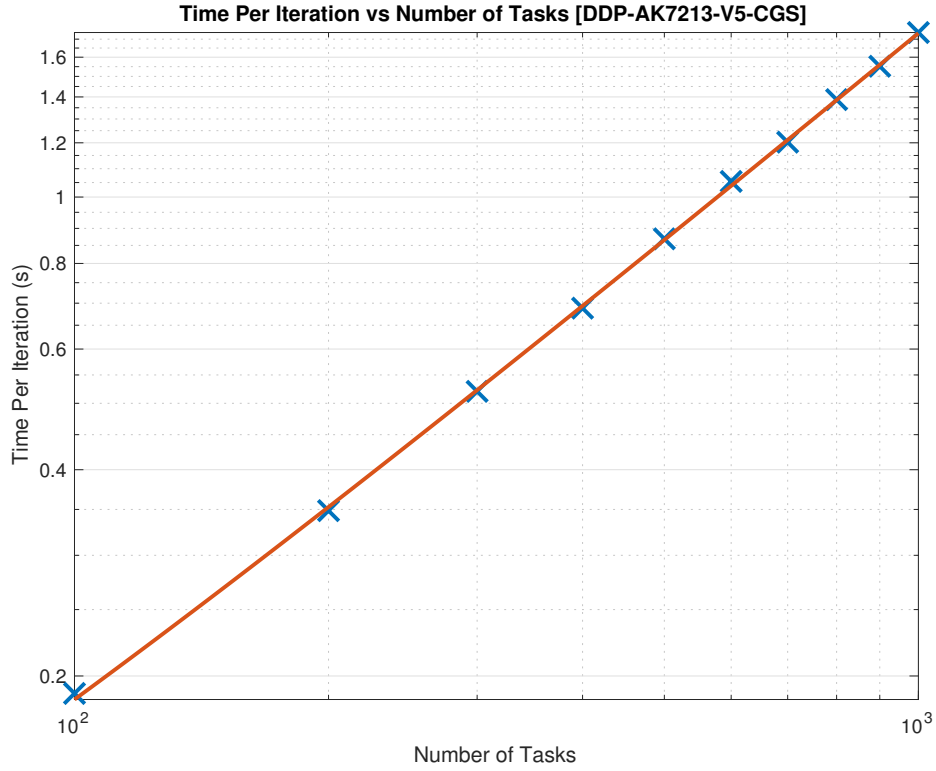


Figure 5.10: Time per iteration vs Number of Tasks [DDP-AK7213-V5CGS]

Figure 5.11 depicts the log-log plot of time per iteration against the number of time steps. This experiment was run for $(n, l) = (100, 10)$ with $m = 1000$ available processors. We cycle through values of N in the range $[10, 100]$ such that the number of tasks due at each time step n_j is an integer number. The line of best fit is given by the equation below. These results show that the time per iteration also grows linearly with an increasing number of time steps.

$$\text{Time/Iteration} = 0.014267 \cdot (\# \text{ of Time Steps})^{1.040153} + 0.034283 \quad (5.49)$$

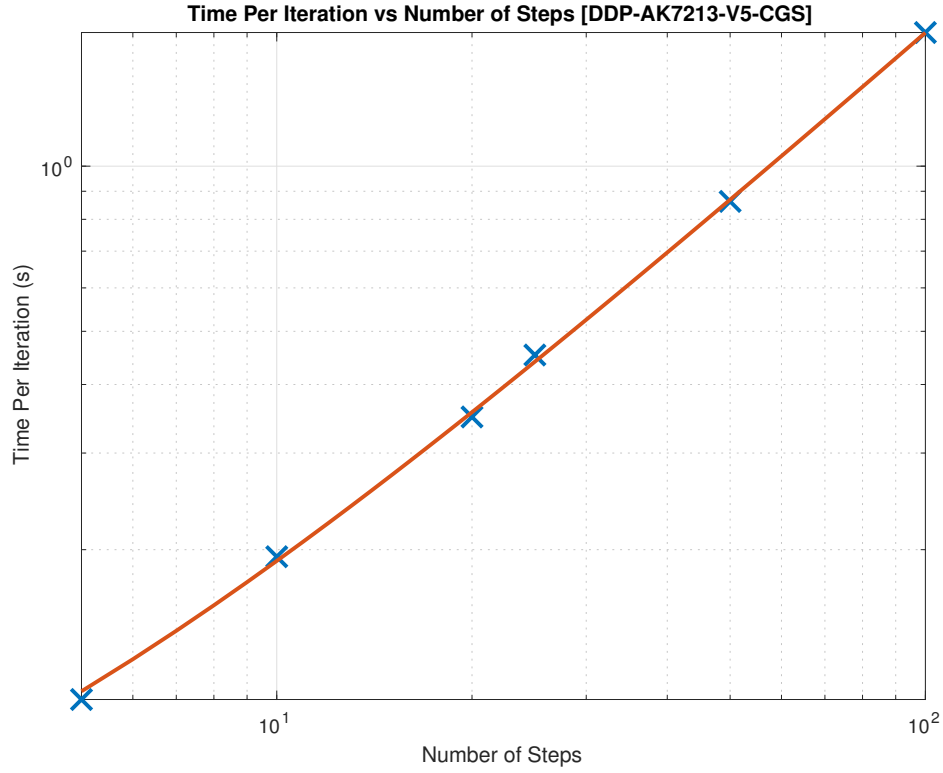


Figure 5.11: Time per iteration vs Number of Time Steps [DDP-AK7213-V5CGS]

Figure 5.12 depicts the relationship between the time taken per iteration versus the number of available speed levels on a log-log plot. This experiment was run for $(n, N) = (10, 10)$ with $m = 1000$ available processors. We cycle through values of l in the range $[100, 1000]$. The line of best fit is given by the equation below.

An exponent of 0.993197 defines a clear linear relationship with the number of speed levels. This is the only algorithm that achieves this result.

$$\text{Time/Iteration} = 0.000098 \cdot (\# \text{ of Speed Levels})^{0.993197} + 0.025436 \quad (5.50)$$

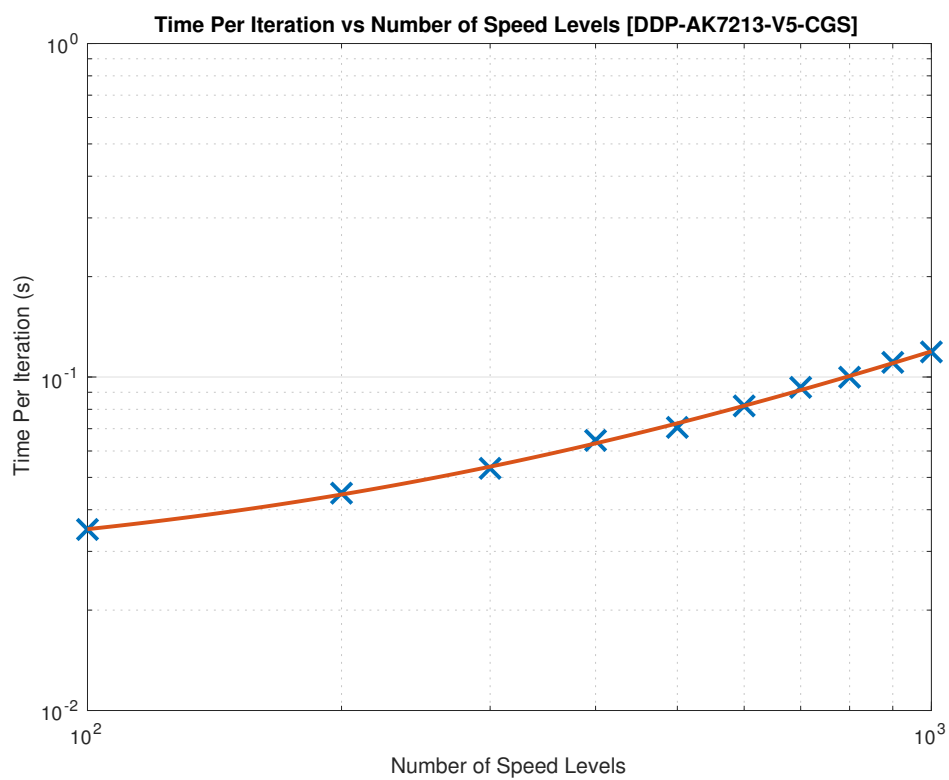


Figure 5.12: Time per iteration vs Number of Speed Levels [DDP-AK7213-V5CGS]

Chapter 6

Conclusions and Future Work

6.1 Summary of Thesis Achievements

The primary objective of this project was to reinforce the natural applicability of robust mathematical optimization methods to real-time multiprocessor task scheduling problems. The derivation of the $\mathcal{O}(N^2n)$ complexity bound helps show that optimisation methods do have a role to play in the design on task scheduling algorithms. This claim is reinforced by the promising practical scaling results associated with the software implementations presented in this work.

Moreover, this project also helps develop a clearer understanding of how it may be possible to tackle large systems of linear equations with similar structural properties. The mathematical framework presented serves as an example of how various tools and techniques from the broad field of linear algebra can be utilised to perform the bare minimum number of operations in order to achieve the same end goal.

6.2 Future Work

Firstly, it is important to note that the approach taken in this project to solve the KKT system of equations represents is in no way definitive. There may exist an alternative permutation on the sequence of block eliminations that organises the constituent sparse matrices in a more workable manner. Alternatively, it might be the case that a slight modification to the original linear programming formulation clears the path for a complexity bound that is linear in terms of both the number of tasks *and* the number of time steps.

However, one particular area that remains unexplored by this project is the formulation of the dual to the discrete-time optimal problem. Addressing the linear program from this complementary perspective could lead to the simplification of the mathematics, which in turn leads to the more general applicability of the methods employed.

Note that the task ordering aspect of the feedback scheduling architecture has been left out of this project. Currently, McNaughton's wrap-around algorithm is utilised to generate

a valid execution schedule for the solution generated by the workload partitioning sub-block. However, there may well exist newer procedures that are better suited within the task scheduling context. A more efficient task ordering sub-block will only help enhance the appeal of the overall scheduling architecture, and this only strengthens the argument for the wider integration of optimal control based methods in computing systems.

Appendix A

Software Package

The complete set of files relevant to this work is available at the following public GitHub repository: <https://github.com/akasture1/EE4FYP.git>. The development and testing of the various routines provided in this repository was carried out entirely in MATLAB 2016b. There exists a total of six unique solvers for the different-deadline problem, and two for the fixed-deadline problem. In addition to this, several simulation scripts and auxiliary functions written to supplement the primary optimisation routines are also provided. Nonetheless, we will present the DDP-AK7213-V4-CGS and DDP-AK7213-V5-CGS algorithms as part of this report. This algorithm solves the block eliminated system of equations with a custom written conjugate gradient method.

A.1 DDP-AK7213-V5-CGS

```
1 function [aOpt, xOpt, ipmIter] = DDP_AK7213_V5_CGS(n, N, l, m, s, ...  
    step, P, xBar, ipmTol, ipmMaxIter, solver, solverTol)  
2  
3 %% Function Argument Definitions  
4 % n:          scalar number of tasks  
5 % N:          scalar number of discrete time steps  
6 % l:          scalar number of speed levels  
7 % m:          scalar number of processors  
8 % s:          column vector of speed levels [1x1]  
9 % step:       fixed time step between each point on the time grid  
10 % P:          column vector of net power consumption values ...  
    corresponding to each speed level [1x1]  
11 % xBar:       column vector containing minimum execution time of ...  
    tasks [nx1]  
12 % ipmTol:     interior-point method stopping threshold  
13 % ipmMaxIter: maximum number of interior-point method iterations  
14 % solver:     system of linear equations solver  
15 % solverTol:  used for iterative solvers like minres and cgs  
16  
17 %% IPM Solver Setup  
18 fprintf('Starting DDP_AK7213_V5_CGS for n=%d; N=%d; l=%d; ...  
    m=%d\n', n, N, l, m);  
19
```

```

20 nl = n * l;
21 nd = n + 1 + nl;
22 ipmIter = 1;
23
24 nj = ceil(n/N);
25 nr = zeros(N,1);           % k = 1,...,N
26 nr(1) = n;
27 for k = 1:N-1
28     nr(k+1) = nr(k) - nj;
29 end
30
31 xOpt = step*rand(n,N);     % k = 1,...,N
32 aOpt = rand(nl,N);        % k = 0,...,N-1
33 lambda = ones(nd,N);     % k = 0,...,N-1
34 p = ones(n,N);           % k = 0,...,N-1
35 t = ones(nd,N);          % k = 0,...,N-1
36 beta = ones(nj,N);       % k = 1,...,N
37
38 %% Debugging/Performance Flags
39 alphaVec = zeros(ipmMaxIter,1);
40 sigmaVec = zeros(ipmMaxIter,1);
41 dualGap = zeros(ipmMaxIter+1,1);
42
43 %% Constant LHS Componentsd = [ones(n,1); m; zeros(nl,1)];
44 d = [ones(n,1); m; zeros(nl,1)];
45
46 D1 = kron(speye(n), ones(1,1));
47 D2 = ones(1,nl);
48 D3 = -1 * speye(nl);
49 D = [D1; D2; D3];
50
51 s = -step*(1/l:1/l:1)';
52 B = kron(speye(n), s');
53
54 %% Start IPM iterations
55 dualGap(1) = (lambda(:)' * t(:))/(nd*N);
56 while dualGap(ipmIter) > ipmTol && ipmIter ≤ ipmMaxIter
57     %fprintf('%d] Duality Gap: %3.8f\n',ipmIter, dualGap(ipmIter));
58
59     %% Compute Sigma, Eta, Rho
60     Sigma = (t.^-1).*lambda;
61     Eta = getEta(n,N,l,Sigma);
62     Rho = getRho(n,N,l,Sigma,Eta);
63
64     %% Compute LHS (Jacobian)
65     U = zeros(n,N);
66     V = zeros(n,N);
67     for k = 1:N
68         for j = 1+(k-1)*nj:n
69             sel = 1+(j-1)*l:j*l;
70             EInv = Sigma(n+1+sel,k).^-1;
71
72             %Term 1 - Diagonal
73             U(j,k) = U(j,k) + sum((s.^2).*EInv) - ...
74                 Eta(j,k) * ((s'*EInv).^2);
75
76             %Term 2 - Rank 1 Matrix (v*v')
77             V(j,k) = V(j,k) + s'*EInv - Eta(j,k)*sum(EInv)*(s'*EInv);

```

```

77         end
78     end
79
80     %% Construct Predictor RHS
81     rx = zeros(n,N);           % k = 1,...,N
82     ra = zeros(nl,N);         % k = 0,...,N-1
83     rp = zeros(n,N);          % k = 0,...,N-1
84     rbeta = zeros(nj,N);      % k = 1,...,N
85     rlambda = zeros(nd,N);    % k = 0,...,N-1
86     rt = zeros(nd,N);         % k = 0,...,N-1
87
88     rlambdaHat = zeros(nd,N);
89     raHat = zeros(nl,N);
90
91     %% rx
92     for k = 1:N-1
93         sel = 1+(k-1)*nj:k*nj;
94         bb = zeros(n,1); bb(sel) = beta(:,k);
95         rx(:,k) = -bb - p(:,k+1) + p(:,k);
96     end
97     sel = 1+n-nj:n;
98     bb = zeros(n,1); bb(sel) = beta(:,k);
99     rx(:,N) = -bb + p(:,N) ;
100
101     %% rbeta
102     I = eye(n);
103     for k = 1:N
104         sel = 1+(k-1)*nj:k*nj;
105         rbeta(:,k) = -xOpt(sel,k);
106     end
107
108     %% rp
109     rp(:,1) = -xBar - B*aOpt(:,1) + xOpt(:,1);
110     for k = 1:N-1
111         rp(:,k+1) = -xOpt(:,k) - B*aOpt(:,k+1) + xOpt(:,k+1);
112     end
113
114     %% ra
115     for k = 1:N
116         ra(:,k) = -step*kron(ones(n,1),P) - B'*p(:,k) - D'*lambda(:,k);
117     end
118
119     %% rlambda, rt, rlambdaHat, raHat
120     for k = 1:N
121         rlambda(:,k) = -( D*aOpt(:,k) - d + t(:,k));
122         rt(:,k) = -( lambda(:,k).*t(:,k) );
123
124         rlambdaHat(:,k) = rlambda(:,k) - (lambda(:,k).^-1) .* rt(:,k);
125         raHat(:,k) = ra(:,k) + D' * (t(:,k).^-1 .* lambda(:,k) .* ...
126             rlambdaHat(:,k));
127     end
128
129     %% Compute PRHS
130     PRHS = zeros(n,1);
131     % Part 1: B*(R.^-1)*ra
132     for k = 1:N
133         u = zeros(n,1);
134         v = zeros(n,1);

```

```

134     kap = 0;
135     for j = 1:n
136         sel = 1+(j-1)*1:j*1;
137         EInv = Sigma(n+1+sel,k).^-1;
138         u(j) = sum(s.*EInv.*raHat(sel,k)) - ...
            Eta(j,k)*(s'*EInv)*(EInv'*raHat(sel,k));
139         v(j) = s'*EInv - Eta(j,k)*sum(EInv)*(s'*EInv);
140         kap = kap + sum(EInv.*raHat(sel,k)) - ...
            Eta(j,k)*sum(EInv)*(EInv'*raHat(sel,k));
141     end
142     PRHS = PRHS + cropVector(u,nj,k) - Rho(k)*kap*cropVector(v,nj,k);
143 end
144
145 % Part 2: B*(R.^-1)*B'*sum(rx)
146 for k = 1:N
147     u = zeros(n,1);
148     v = zeros(n,1);
149     for j = 1:n
150         sel = 1+(j-1)*1:j*1;
151         EInv = Sigma(n+1+sel,k).^-1;
152         u(j) = sum((s.^2).*EInv) - Eta(j,k)*((s'*EInv).^2);
153         v(j) = s'*EInv - Eta(j,k)*sum(EInv)*(s'*EInv);
154     end
155     PRHS = PRHS + (cropVector(u,nj,k).*sum(rx(:,k:N),2)) - ...
        Rho(k)*(v'*sum(rx(:,k:N),2))*cropVector(v,nj,k);
156 end
157
158 % Part 3: rp
159 for k = 1:N
160     PRHS = PRHS - cropVector(rp(:,k),nj,k);
161 end
162
163 %Part 4: rbeta
164 PRHS = PRHS - rbeta(:);
165
166 %% Solve Predictor Problem using Custom CGS
167 PSOL = cgsCustom(U,V,PRHS,Rho,solverTol);
168
169 %% Set Affine Parameters from Predictor Solution
170 affDx = zeros(n,N); % k = 1,...,N
171 affDa = zeros(nl,N); % k = 0,...,N-1
172 affDlambda = zeros(nd,N); % k = 0,...,N-1
173 affDp = zeros(n,N); % k = 0,...,N-1
174 affDt = zeros(nd,N); % k = 0,...,N-1
175 affDbeta = zeros(nj,N); % k = 1,...,N
176
177 %% affDbeta
178 affDbeta = reshape(PSOL, [nj,N]);
179
180 %% affDp
181 sel = 1+n-nj:n;
182 bb = zeros(n,1); bb(sel) = affDbeta(:,k);
183 affDp(:,N) = bb - rx(:,N);
184 for k = N-1:-1:1
185     sel = 1+(k-1)*nj:k*nj;
186     bb = zeros(n,1); bb(sel) = affDbeta(:,k);
187     affDp(:,k) = bb - rx(:,k) + affDp(:,k+1);
188 end

```

```

189
190 %% affDa
191 for k = 1:N
192     u = zeros(nl,1);
193     v = zeros(nl,1);
194     y = raHat(:,k) - B'*affDp(:,k);
195     for j = 1:n
196         sel = 1+(j-1)*l:j*l;
197         EInv = Sigma(n+1+sel,k).^-1;
198         u(sel) = (EInv.*y(sel)) - Eta(j,k)*EInv*(EInv'*y(sel));
199         v(sel) = EInv - Eta(j,k)*EInv*(sum(EInv));
200     end
201     kap = sum(u);
202     affDa(:,k) = u - Rho(k)*kap*v;
203 end
204
205 %% affDx
206 affDx(:,1) = B*affDa(:,1) - rp(:,1);
207 for k = 2:N
208     affDx(:,k) = affDx(:,k-1) + B*affDa(:,k) - rp(:,k);
209 end
210
211 %% affDlambda, affDt
212 for k = 1:N
213     affDlambda(:,k) = (t(:,k).^-1 .* lambda(:,k)) .* ...
214         (-rlambdaHat(:,k) + D*affDa(:,k));
215     affDt(:,k) = lambda(:,k).^-1 .* (rt(:,k) - t(:,k) .* ...
216         affDlambda(:,k));
217 end
218
219 %% Calculate Affine Step Length
220 % only need to consider the cases where affDlambda and affDt is ...
221 % negative
222 affAlpha = 1;
223 for i = 1:N
224     for j = 1:nd
225         if( affDlambda(j,i) < 0 )
226             affAlpha = min(affAlpha, -lambda(j,i)/affDlambda(j,i));
227         end
228         if( affDt(j,i) < 0 )
229             affAlpha = min(affAlpha, -t(j,i)/affDt(j,i));
230         end
231     end
232 end
233 affAlphaVec(ipmIter) = affAlpha;
234 %fprintf('%d] Affine Alpha (Step Length): %3.8f\n',ipmIter, ...
235 %    affAlpha);
236
237 %% Calculate Sigma
238 % sigma ~ 0 indicates that the affine dir. is a good search dir.
239 affDualGap = ( (lambda(:) + (affAlpha * affDlambda(:)))' * (t(:) ...
240     + (affAlpha * affDt(:))) )/(nd*N);
241 % fprintf('%d] Affine Duality Gap: %3.8f\n',ipmIter, affDualGap);
242
243 sigma = (affDualGap/dualGap(ipmIter))^3;
244 sigmaVec(ipmIter) = sigma;
245 %fprintf('%d] Sigma: (%3.8f/%3.8f)^3 = %3.8f\n',ipmIter, ...
246     affDualGap, dualGap(ipmIter),sigma);

```

```

241
242 %% Construct Corrector RHS
243 rx = zeros(n,N); % k = 1,...,N
244 ra = zeros(nl,N); % k = 0,...,N-1
245 rlambda = zeros(nd,N); % k = 0,...,N-1
246 rp = zeros(n,N); % k = 0,...,N-1
247 rt = -(affDlambda.*affDt) + (sigma * dualGap(ipmIter) * ones(nd, ...
    N)); % k = 0,...,N-1
248 rbeta = zeros(nj,N); % k = 1,...,N
249
250 rlambdaHat = zeros(nd,N);
251 raHat = zeros(nl,N);
252
253 for k = 1:N
254     rlambdaHat(:,k) = rlambda(:,k) - (lambda(:,k).^-1) .* rt(:,k);
255     raHat(:,k) = ra(:,k) + D' * (t(:,k).^-1 .* lambda(:,k) .* ...
        rlambdaHat(:,k));
256 end
257
258 %% Compute CRHS
259 CRHS = zeros(n,1);
260 % Part 1: B*(R.^-1)*ra
261 for k = 1:N
262     u = zeros(n,1);
263     v = zeros(n,1);
264     kap = 0;
265     for j = 1:n
266         sel = 1+(j-1)*l:j*l;
267         EInv = Sigma(n+1+sel,k).^-1;
268         u(j) = sum(s.*EInv.*raHat(sel,k)) - ...
            Eta(j,k)*(s'*EInv)*(EInv'*raHat(sel,k));
269         v(j) = s'*EInv - Eta(j,k)*sum(EInv)*(s'*EInv);
270         kap = kap + sum(EInv.*raHat(sel,k)) - ...
            Eta(j,k)*sum(EInv)*(EInv'*raHat(sel,k));
271     end
272     CRHS = CRHS + cropVector(u,nj,k) - Rho(k)*kap*cropVector(v,nj,k);
273 end
274
275 % Part 2: B*(R.^-1)*B'*sum(rx)
276 for k = 1:N
277     u = zeros(n,1);
278     v = zeros(n,1);
279     for j = 1:n
280         sel = 1+(j-1)*l:j*l;
281         EInv = Sigma(n+1+sel,k).^-1;
282         u(j) = sum((s.^2).*EInv) - Eta(j,k)*((s'*EInv).^2);
283         v(j) = s'*EInv - Eta(j,k)*sum(EInv)*(s'*EInv);
284     end
285     CRHS = CRHS + (cropVector(u,nj,k).*sum(rx(:,k:N),2)) - ...
        Rho(k)*(v'*sum(rx(:,k:N),2))*cropVector(v,nj,k);
286 end
287
288 % Part 3: rp
289 for k = 1:N
290     CRHS = CRHS - cropVector(rp(:,k),nj,k);
291 end
292
293 %Part 4: rbeta

```

```

294 CRHS = CRHS - rbeta(:);
295 %fprintf('%d] Corrector RHS complete: %3.5fs\n',ipmIter, ...
    tCRHS(ipmIter));
296
297 %% Solve Corrector Problem using Custom CGS
298 CSOL = cgsCustom(U,V,CRHS,Rho,solverTol);
299
300 %% Set Corrector Parameters from Predictor Solution
301 ccDx = zeros(n,N); % k = 1,...,N
302 ccDa = zeros(nl,N); % k = 0,...,N-1
303 ccDlambda = zeros(nd,N); % k = 0,...,N-1
304 ccDp = zeros(n,N); % k = 0,...,N-1
305 ccDt = zeros(nd,N); % k = 0,...,N-1
306 ccDbeta = zeros(nj,N); % k = 1,...,N
307
308 %% ccDbeta
309 ccDbeta = reshape(CSOL, [nj,N]);
310
311 %% ccDp
312 sel = 1+n-nj:n;
313 bb = zeros(n,1); bb(sel) = ccDbeta(:,k);
314 ccDp(:,N) = bb - rx(:,N);
315 for k = N-1:-1:1
316     sel = 1+(k-1)*nj:k*nj;
317     bb = zeros(n,1); bb(sel) = ccDbeta(:,k);
318     ccDp(:,k) = bb - rx(:,k) + ccDp(:,k+1);
319 end
320
321 %% ccDa
322 for k = 1:N
323     u = zeros(nl,1);
324     v = zeros(nl,1);
325     y = raHat(:,k) - B'*ccDp(:,k);
326     for j = 1:n
327         sel = 1+(j-1)*l:j*l;
328         EInv = Sigma(n+1+sel,k).^-1;
329         u(sel) = (EInv.*y(sel)) - Eta(j,k)*EInv*(EInv'*y(sel));
330         v(sel) = EInv - Eta(j,k)*EInv*(sum(EInv));
331     end
332     kap = sum(u);
333     ccDa(:,k) = u - Rho(k)*kap*v;
334 end
335
336 %% ccDx
337 ccDx(:,1) = B*ccDa(:,1) - rp(:,1);
338 for k = 2:N
339     ccDx(:,k) = ccDx(:,k-1) + B*ccDa(:,k) - rp(:,k);
340 end
341
342 %% ccDlambda, ccDt
343 for k = 1:N
344     ccDlambda(:,k) = (t(:,k).^-1 .* lambda(:,k)) .* ...
        (-rlambdaHat(:,k) + D*ccDa(:,k));
345     ccDt(:,k) = lambda(:,k).^-1 .* (rt(:,k) - t(:,k) .* ...
        ccDlambda(:,k));
346 end
347
348 %% Predictor-Corrector Search Direction

```

```

349     Dx = affDx + ccDx;
350     Da = affDa + ccDa;
351     Dlambd = affDlambd + ccDlambd;
352     Dp = affDp + ccDp;
353     Dt = affDt + ccDt;
354     Dbeta = affDbeta + ccDbeta;
355
356     %% Calculate Predictor-Corrector Step Length
357     alpha = realmax;
358     for i = 1:N
359         for j = 1:nd
360             if( Dlambd(j,i) < 0 )
361                 alpha = min(alpha, -lambd(j,i)/Dlambd(j,i));
362             end
363             if( Dt(j,i) < 0 )
364                 alpha = min(alpha, -t(j,i)/Dt(j,i));
365             end
366         end
367     end
368     alphaVec(ipmIter) = alpha;
369     %fprintf('%d] Alpha (Step Length): %3.8f\n',ipmIter, alpha);
370
371     %% Update Solution
372     gamma = 1 - (1/(ipmIter+5)^2);
373     %gamma = 0.99;
374
375     xOpt = xOpt + (gamma * alpha * Dx);
376     aOpt = aOpt + (gamma * alpha * Da);
377     lambd = lambd + (gamma * alpha * Dlambd);
378     p = p + (gamma * alpha * Dp);
379     t = t + (gamma * alpha * Dt);
380     beta = beta + (gamma * alpha * Dbeta);
381
382     dualGap(ipmIter+1) = ( lambd(:)' * t(:) )/(nd*N);
383     %fprintf('End of iteration [%d] - DualGap: %3.8f -> ...
384         %3.8f\n',ipmIter, dualGap(ipmIter), dualGap(ipmIter+1));
385     ipmIter = ipmIter+1;
386 end
387 if dualGap(ipmIter) > ipmTol
388     fprintf('Failed to find Solution\n')
389 end

```


A.2 DDP-AK7213-V4-CGS

```

1 function [ aOpt, xOpt, ipmIter ] = DDP_AK7213_V4_CGS( n, N, l, m, s, ...
    step, P, xBar, ipmTol, ipmMaxIter, solver, solverTol )
2
3 %% Function Argument Definitions
4 % n:          scalar number of tasks
5 % N:          scalar number of discrete time steps
6 % l:          scalar number of speed levels
7 % m:          scalar number of processors
8 % s:          column vector of speed levels [1x1]
9 % step:       fixed time step between each point on the time grid
10 % P:          column vector of net power consumption values ...
    corresponding to each speed level [1x1]
11 % xBar:       column vector containing minimum execution time of ...
    tasks [nx1]
12 % ipmTol:     interior-point method stopping threshold
13 % ipmMaxIter: maximum number of interior-point method iterations
14 % solver:     choose system of linear equations solver (not ...
    applicable here)
15 % solverTol:  used for iterative solvers like minres and cgs
16
17 %% IPM Solver Setup
18 fprintf('Starting DDP_AK7213_V4_CGS for n=%d; N=%d; l=%d; ...
    m=%d\n', n, N, l, m);
19
20 nl = n * l;
21 nd = n + l + nl;
22 ipmIter = 1;
23
24 nj = ceil(n/N);
25 nr = zeros(N,1);          % k = 1,...,N
26 nr(1) = n;
27 for k = 1:N-1
28     nr(k+1) = nr(k) - nj;
29 end
30
31 xOpt = step*rand(n,N);    % k = 1,...,N
32 aOpt = rand(nl,N);        % k = 0,...,N-1
33 lambda = ones(nd,N);     % k = 0,...,N-1
34 p = ones(n,N);           % k = 0,...,N-1
35 t = ones(nd,N);          % k = 0,...,N-1
36 beta = ones(nj,N);       % k = 1,...,N
37
38 %% Debugging/Performance Flags
39 alphaVec = zeros(ipmMaxIter,1);
40 sigmaVec = zeros(ipmMaxIter,1);
41 dualGap = zeros(ipmMaxIter+1,1);
42
43 %% Constant LHS Componentsd = [ones(n,1); m; zeros(nl,1)];
44 d = [ones(n,1); m; zeros(nl,1)];
45
46 D1 = kron(speye(n), ones(1,1));
47 D2 = ones(1,nl);
48 D3 = -1 * speye(nl);
49 D = [D1; D2; D3];

```

```

50
51 B = -kron(step * speye(n), s');
52
53 %% Start IPM iterations
54 dualGap(1) = (lambda(:)' * t(:))/(nd*N);
55 while dualGap(ipmIter) > ipmTol && ipmIter ≤ ipmMaxIter
56     %fprintf('%d] Duality Gap: %3.8f\n',ipmIter, dualGap(ipmIter));
57
58     %% Compute Inverse-Rs
59     invMs = cell(N,1);
60     Rho = zeros(N,1);
61     for k = 1:N
62         [invMs{k}, Rho(k)] = getRInvBlocks(lambda(:,k), t(:,k), n, 1);
63     end
64
65     %% Compute LHS
66     U = zeros(n,N);
67     V = zeros(n,N);
68     for k = 1:N
69         invM = invMs{k};
70         U(:,k) = spdiags(B * invM * B',0);
71         V(:,k) = B*sum(invM,2);
72     end
73
74     %% Construct Predictor RHS
75     rx = zeros(n,N);           % k = 1,...,N
76     ra = zeros(nl,N);          % k = 0,...,N-1
77     rp = zeros(n,N);           % k = 0,...,N-1
78     rbeta = zeros(nj,N);       % k = 1,...,N
79     rlambd = zeros(nd,N);       % k = 0,...,N-1
80     rt = zeros(nd,N);          % k = 0,...,N-1
81
82     rlambdHat = zeros(nd,N);
83     raHat = zeros(nl,N);
84
85
86     %% rx
87     for k = 1:N-1
88         HSelect = 1+(k-1)*nj:k*nj;
89         bb = zeros(n,1); bb(HSelect) = beta(:,k);
90         rx(:,k) = -bb - p(:,k+1) + p(:,k);
91     end
92     HSelect = 1+n-nj:n;
93     bb = zeros(n,1); bb(HSelect) = beta(:,k);
94     rx(:,N) = -bb + p(:,N) ;
95
96     %% rbeta
97     I = eye(n);
98     for k = 1:N
99         HSelect = 1+(k-1)*nj:k*nj;
100         rbeta(:,k) = -xOpt(HSelect,k);
101     end
102
103     %% rp
104     rp(:,1) = -xBar - B*aOpt(:,1) + xOpt(:,1);
105     for k = 1:N-1
106         rp(:,k+1) = -xOpt(:,k) - B*aOpt(:,k+1) + xOpt(:,k+1);
107     end

```

```

108
109 %% ra
110 for k = 1:N
111     ra(:,k) = -step*kron(ones(n,1),P) - B'*p(:,k) - D'*lambda(:,k);
112 end
113
114 %% rlambda, rt, rlambdaHat, raHat
115 for k = 1:N
116     rlambda(:,k) = -( D*aOpt(:,k) - d + t(:,k) );
117     rt(:,k) = -( lambda(:,k).*t(:,k) );
118
119     rlambdaHat(:,k) = rlambda(:,k) - (lambda(:,k).^-1) .* rt(:,k);
120     raHat(:,k) = ra(:,k) + D' * (t(:,k).^-1 .* lambda(:,k) .* ...
121         rlambdaHat(:,k));
122 end
123
124 %% Compute PRHS
125 PRHS = zeros(n,1);
126 for k = 1:N
127     invM = invMs{k};
128     v = B*sum(invM,2);
129
130     jj = B*(invM*raHat(:,k)) - Rho(k)*v*(sum(invM,1)*raHat(:,k));
131     gg = B*(invM*(B'*sum(rx(:,k:N),2))) - ...
132         Rho(k)*(v*(v'*sum(rx(:,k:N),2)));
133
134     PRHS = PRHS - cropVector(rp(:,k),nj,k) ...
135         + cropVector(jj,nj,k) ...
136         + cropVector(gg,nj,k);
137 end
138 PRHS = PRHS - rbeta(:);
139
140 %% Solve Predictor Problem with Custom CGS
141 PSOL = cgsCustom(U,V,PRHS,Rho,solverTol);
142
143 %% Set Affine Parameters from Predictor Solution
144 affDx = zeros(n,N); % k = 1,...,N
145 affDa = zeros(nl,N); % k = 0,...,N-1
146 affDlambda = zeros(nd,N); % k = 0,...,N-1
147 affDp = zeros(n,N); % k = 0,...,N-1
148 affDt = zeros(nd,N); % k = 0,...,N-1
149 affDbeta = zeros(nj,N); % k = 1,...,N
150
151 %% affDbeta
152 affDbeta = reshape(PSOL, [nj,N]);
153
154 %% affDp
155 HSelect = 1+n-nj:n;
156 bb = zeros(n,1); bb(HSelect) = affDbeta(:,k);
157 affDp(:,N) = bb - rx(:,N);
158 for k = N-1:-1:1
159     HSelect = 1+(k-1)*nj:k*nj;
160     bb = zeros(n,1); bb(HSelect) = affDbeta(:,k);
161     affDp(:,k) = bb - rx(:,k) + affDp(:,k+1);
162 end
163
164 %% affDa

```

```

164     for k = 1:N
165         invM = invMs{k};
166         v = sum(invM,2);
167         affDa(:,k) = invM*(raHat(:,k) - B'*affDp(:,k)) ...
168             - Rho(k)*(v*(v'*(raHat(:,k) - B'*affDp(:,k))));
169     end
170
171     %% affDx
172     affDx(:,1) = B*affDa(:,1) - rp(:,1);
173     for k = 2:N
174         affDx(:,k) = affDx(:,k-1) + B*affDa(:,k) - rp(:,k);
175     end
176
177     %% affDlambda, affDt
178     for k = 1:N
179         affDlambda(:,k) = (t(:,k).^-1 .* lambda(:,k)) .* ...
180             (-rlambdaHat(:,k) + D*affDa(:,k));
181         affDt(:,k) = lambda(:,k).^-1 .* (rt(:,k) - t(:,k) .* ...
182             affDlambda(:,k));
183     end
184
185     %% Calculate Affine Step Length
186     % only need to consider the cases where affDlambda and affDt is ...
187     % negative
188     affAlpha = 1;
189     for i = 1:N
190         for j = 1:nd
191             if( affDlambda(j,i) < 0 )
192                 affAlpha = min(affAlpha, -lambda(j,i)/affDlambda(j,i));
193             end
194             if( affDt(j,i) < 0 )
195                 affAlpha = min(affAlpha, -t(j,i)/affDt(j,i));
196             end
197         end
198     end
199     affAlphaVec(ipmIter) = affAlpha;
200     %fprintf('%d Affine Alpha (Step Length): %3.8f\n',ipmIter, ...
201     %    affAlpha);
202
203     %% Calculate Sigma
204     % sigma > 0 indicates that the affine dir. is a good search dir.
205     affDualGap = ( (lambda(:) + (affAlpha * affDlambda(:)))' * (t(:) ...
206         + (affAlpha * affDt(:))) )/(nd*N);
207     % fprintf('%d Affine Duality Gap: %3.8f\n',ipmIter, affDualGap);
208
209     sigma = (affDualGap/dualGap(ipmIter))^3;
210     sigmaVec(ipmIter) = sigma;
211     %fprintf('%d Sigma: (%3.8f/%3.8f)^3 = %3.8f\n',ipmIter, ...
212     %    affDualGap, dualGap(ipmIter),sigma);
213
214     %% Construct Corrector RHS
215     rx = zeros(n,N);           % k = 1,...,N
216     ra = zeros(nl,N);         % k = 0,...,N-1
217     rlambda = zeros(nd,N);    % k = 0,...,N-1
218     rp = zeros(n,N);          % k = 0,...,N-1
219     rt = -(affDlambda.*affDt) + (sigma * dualGap(ipmIter) * ones(nd, ...
220         N)); % k = 0,...,N-1
221     rbeta = zeros(nj,N);      % k = 1,...,N

```

```

215
216     rlambdaHat = zeros(nd,N);
217     raHat = zeros(nl,N);
218
219     for k = 1:N
220         rlambdaHat(:,k) = rlambda(:,k) - (lambda(:,k).^-1) .* rt(:,k);
221         raHat(:,k) = ra(:,k) + D' * (t(:,k).^-1 .* lambda(:,k) .* ...
                rlambdaHat(:,k));
222     end
223
224     %% Compute CRHS
225     CRHS = zeros(n,1);
226     for k = 1:N
227         invM = invMs{k};
228         v = B*sum(invM,2);
229
230         jj = B*(invM*raHat(:,k)) - Rho(k)*v*(sum(invM,1)*raHat(:,k));
231         gg = B*(invM*(B'*sum(rx(:,k:N),2))) - ...
                Rho(k)*(v*(v'*sum(rx(:,k:N),2)));
232
233         CRHS = CRHS - cropVector(rp(:,k),nj,k) ...
                + cropVector(jj,nj,k) ...
                + cropVector(gg,nj,k);
234     end
235     CRHS = CRHS - rbeta(:);
236     fprintf('%d] Corrector RHS complete: %3.5fs\n',ipmIter, ...
            tcRHS(ipmIter));
237
238     %% SOLVE CORRECTOR
239     CSOL = cgsCustom(U,V,CRHS,Rho,solverTol);
240
241     %% Set Corrector Parameters from Predictor Solution
242     ccDx = zeros(n,N);           % k = 1,...,N
243     ccDa = zeros(nl,N);         % k = 0,...,N-1
244     ccDlambda = zeros(nd,N);    % k = 0,...,N-1
245     ccDp = zeros(n,N);          % k = 0,...,N-1
246     ccDt = zeros(nd,N);         % k = 0,...,N-1
247     ccDbeta = zeros(nj,N);      % k = 1,...,N
248
249     %% ccDbeta
250     ccDbeta = reshape(CSOL, [nj,N]);
251
252     %% ccDp
253     HSelect = 1+n-nj:n;
254     bb = zeros(n,1); bb(HSelect) = ccDbeta(:,k);
255     ccDp(:,N) = bb - rx(:,N);
256     for k = N-1:-1:1
257         HSelect = 1+(k-1)*nj:k*nj;
258         bb = zeros(n,1); bb(HSelect) = ccDbeta(:,k);
259         ccDp(:,k) = bb - rx(:,k) + ccDp(:,k+1);
260     end
261
262     %% ccDa
263     for k = 1:N
264         invM = invMs{k};
265         v = sum(invM,2);
266         ccDa(:,k) = invM*(raHat(:,k) - B'*ccDp(:,k)) ...
                - Rho(k)*(v*(v'*(raHat(:,k) - B'*ccDp(:,k))));
267     end

```

```

270     end
271
272     %% ccDx
273     ccDx(:,1) = B*ccDa(:,1) - rp(:,1);
274     for k = 2:N
275         ccDx(:,k) = ccDx(:,k-1) + B*ccDa(:,k) - rp(:,k);
276     end
277
278     %% ccDlambdax, ccDt
279     for k = 1:N
280         ccDlambdax(:,k) = (t(:,k).^-1 .* lambda(:,k)) .* ...
281             (-rlambdaHat(:,k) + D*ccDa(:,k));
282         ccDt(:,k) = lambda(:,k).^-1 .* (rt(:,k) - t(:,k) .* ...
283             ccDlambdax(:,k));
284     end
285
286     %% Determine Search Direction
287     Dx = affDx + ccDx;
288     Da = affDa + ccDa;
289     Dlambdax = affDlambdax + ccDlambdax;
290     Dp = affDp + ccDp;
291     Dt = affDt + ccDt;
292     Dbeta = affDbeta + ccDbeta;
293
294     %% Calculate Predictor-Corrector Step Length
295     alpha = realmax;
296     for i = 1:N
297         for j = 1:nd
298             if( Dlambdax(j,i) < 0 )
299                 alpha = min(alpha, -lambda(j,i)/Dlambdax(j,i));
300             end
301             if( Dt(j,i) < 0 )
302                 alpha = min(alpha, -t(j,i)/Dt(j,i));
303             end
304         end
305     end
306     alphaVec(ipmIter) = alpha;
307     %fprintf('%d] Alpha (Step Length): %3.8f\n',ipmIter, alpha);
308
309     %% Update Solution
310     gamma = 1 - (1/(ipmIter+5)^2);
311     %gamma = 0.99;
312
313     xOpt = xOpt + (gamma * alpha * Dx);
314     aOpt = aOpt + (gamma * alpha * Da);
315     lambda = lambda + (gamma * alpha * Dlambdax);
316     p = p + (gamma * alpha * Dp);
317     t = t + (gamma * alpha * Dt);
318     beta = beta + (gamma * alpha * Dbeta);
319
320     dualGap(ipmIter+1) = ( lambda(:)' * t(:) )/(nd*N);
321     %fprintf('End of iteration [%d] - DualGap: %3.8f -> ...
322         %3.8f\n',ipmIter, dualGap(ipmIter), dualGap(ipmIter+1));
323     ipmIter = ipmIter+1;
324 end
325
326 if dualGap(ipmIter) > ipmTol
327     fprintf('Failed to find Solution\n')
328 end

```

325 end

A.3 Auxiliary Functions

```

1 function [ MInv, mu ] = getRInvBlocks( lambda, t, n, l )
2     Sigma = (t.^-1).*lambda;
3     EBar = Sigma(n+2:end);
4     MInvs = cell(n,1);
5     for j = 1:n
6         EInv = EBar(1+(j-1)*l:j*l).^-1;
7         beta = Sigma(j)/(1 + Sigma(j)*sum(EInv));
8         MInvs{j} = sparse(diag(EInv) - beta*(EInv*EInv'));
9     end
10    MInv = blkdiag(MInvs{:});
11    mu = Sigma(n+1)/(1 + Sigma(n+1)*sum(sum(MInv)));
12 end

```

```

1 function [ x, iter ] = cgsCustom(U,V,b,Mu,tol)
2     N = size(U,2);
3     x = b;
4
5     % Compute F = A*x
6     F = zeros(length(b),1);
7     for k = 1:N
8         F = F + (U(:,k).*x) - Mu(k)*(V(:,k)'.*x)*V(:,k);
9     end
10
11    r1 = b - F; d1 = r1;
12    if norm(r1) < tol
13        return
14    end
15
16    for iter = 1:length(b)
17        % Compute F = A*d1
18        F = zeros(length(b),1);
19
20        for k = 1:N
21            F = F + (U(:,k).*d1) - Mu(k)*(V(:,k)'.*d1)*V(:,k);
22        end
23
24        alpha = (r1'*r1)/(d1'*F);
25        x = x + alpha*d1;
26        r2 = r1 - alpha*F;
27        beta = (r2'*r2)/(r1'*r1);
28        d2 = r2 + beta*d1;
29
30        if( norm(r2) < tol )
31            return;
32        end
33
34        r1 = r2; d1 = d2;
35    end

```

```
36 end
```

```
1 function [ A ] = cropMatrix( A, nj, k )
2     if k > 1
3         s = nj*(k-1);
4         [r,c,v] = find(A);
5         v((c≤s)|(r≤s)) = 0;
6         A = sparse(r,c,v);
7     %         A(1:s,:) = 0;
8     %         A(:,1:s) = 0;
9     end
10 end
```

```
1 function [ v ] = cropVector( v, nj, k )
2     if k > 1
3         s = nj*(k-1);
4         v(1:s) = 0;
5     end
6 end
```


Bibliography

ARM (2013), ‘big.LITTLE Technology: The future of mobile’.

URL: http://arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf

Baruah, S. K., Cohen, N. K., Plaxton, C. G. & Varvel, D. A. (1996), ‘Proportionate progress: A notion of fairness in resource allocation’, *Algorithmica* **15**(6), 600–625.

URL: <http://dx.doi.org/10.1007/BF01940883>

Cho, H., Ravindran, B. & Jensen, E. D. (2006), An optimal real-time scheduling algorithm for multiprocessors, in ‘2006 27th IEEE International Real-Time Systems Symposium (RTSS’06)’, pp. 101–110.

Chwa, H. S., Seo, J., Lee, J. & Shin, I. (2015), Optimal real-time scheduling on two-type heterogeneous multicore platforms, in ‘Real-Time Systems Symposium, 2015 IEEE’, IEEE, pp. 119–129.

Chwa, H. S., Seo, J., Yoo, H., Lee, J. & Shin, I. (2014), ‘Energy and feasibility optimal global scheduling framework on big. little platforms’, *Department of Computer Science, KAIST and Department of Computer Science and Engineering, Sungkyunkwan University, Republic of Korea, Tech. Rep.*

Correa, J. R., Skutella, M. & Verschae, J. (2012), ‘The power of preemption on unrelated machines and applications to scheduling orders’, *Math. Oper. Res.* **37**(2), 379–398.

URL: <http://dx.doi.org/10.1287/moor.1110.0520>

Dantzig, G. B. (1951), ‘Maximization of a linear function of variables subject to linear inequalities, in activity analysis of production and allocation’.

Filieri, A., Maggio, M., Angelopoulos, K., D’Ippolito, N., Gerostathopoulos, I., Hempel, A. B., Hoffmann, H., Jamshidi, P., Kalyvianaki, E., Klein, C., Krikava, F., Misailovic, S., Papadopoulos, A. V., Ray, S., Sharifloo, A. M., Shevtsov, S., Ujma, M. & Vogel, T. (2015), Software engineering meets control theory, in ‘Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems’, SEAMS ’15, IEEE Press, Piscataway, NJ, USA, pp. 71–82.

URL: <http://dl.acm.org/citation.cfm?id=2821357.2821370>

Funk, S., Berten, V., Ho, C. & Goossens, J. (2012), A global optimal scheduling algorithm for multiprocessor low-power platforms, in ‘Proceedings of the 20th International Conference on Real-Time and Network Systems’, RTNS ’12, ACM, New York, NY, USA, pp. 71–80.

URL: <http://doi.acm.org/10.1145/2392987.2392996>

GeSI (2014), ‘Smarter 2030 - ICT solutions for 21st century challenges’.

URL: http://smarter2030.gesi.org/downloads/Full_report.pdf

Golub, G. H. & Van Loan, C. F. (2012), *Matrix computations*, Vol. 3, JHU Press.

Gondzio, J. (2012), ‘Interior point methods 25 years later’, *European Journal of Operational Research* **218**(3), 587–601.

Heddeghem, W. V., Lambert, S., Lannoo, B., Colle, D., Pickavet, M. & Demeester, P. (2014), ‘Trends in worldwide ICT electricity consumption from 2007 to 2012’, *Computer Communications* **50**, 64 – 76.

URL: <http://sciencedirect.com/science/article/pii/S0140366414000619>

Imes, C., Kim, D. H. K., Maggio, M. & Hoffmann, H. (2015), Poet: a portable approach to minimizing energy under soft real-time constraints, in ‘21st IEEE Real-Time and Embedded Technology and Applications Symposium’, pp. 75–86.

Karmarkar, N. (1984), A new polynomial-time algorithm for linear programming, in ‘Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing’, STOC ’84, ACM, New York, NY, USA, pp. 302–311.

URL: <http://doi.acm.org/10.1145/800057.808695>

Kerrigan, E. C. (2015), ‘Feedback and time are essential for the optimal control of computing systems’, *IFAC-PapersOnLine* **48**(23), 380 – 387. 5th IFAC Conference on Nonlinear Model Predictive Control NMPC 2015, Seville, Spain.

URL: <http://sciencedirect.com/science/article/pii/S2405896315025938>

Klee, V. & Minty, G. J. (1970), How good is the simplex algorithm, Technical report, DTIC Document.

Leva, A. & Maggio, M. (2010), ‘Feedback process scheduling with simple discrete-time control structures’, *IET Control Theory Applications* **4**(11), 2331–2342.

Levin, G., Funk, S., Sadowski, C., Pye, I. & Brandt, S. (2010), Dp-fair: A simple model for understanding optimal multiprocessor scheduling, in ‘2010 22nd Euromicro Conference on Real-Time Systems’, pp. 3–13.

Liu, C. L. & Layland, J. W. (1973), ‘Scheduling algorithms for multiprogramming in a hard-real-time environment’, *J. ACM* **20**(1), 46–61.

URL: <http://doi.acm.org/10.1145/321738.321743>

Maggio, M., Terraneo, F. & Leva, A. (2014), ‘Task scheduling: A control-theoretical viewpoint for a general and flexible solution’, *ACM Trans. Embed. Comput. Syst.* **13**(4), 76:1–76:22.

URL: <http://doi.acm.org/10.1145/2560015>

McNaughton, R. (1959), ‘Scheduling with deadlines and loss functions’, *Management Science* **6**(1), 1–12.

Mehrotra, S. (1992), ‘On the implementation of a primal-dual interior point method’, *SIAM Journal on optimization* **2**(4), 575–601.

- Naffziger, S. & Koomey, J. (2015), ‘Moore’s law might be slowing down, but not energy efficiency’.
URL: <http://spectrum.ieee.org/computing/hardware/moores-law-might-be-slowing-down-but-not-energy-efficiency>
- Nocedal, J. & Wright, S. (2006), *Numerical optimization*, Springer Science & Business Media.
- Papadopoulos, A. V., Carone, R., Maggio, M. & Leva, A. (2015), A control-theoretical approach to thread scheduling for multicore processors, *in* ‘2015 IEEE Conference on Control Applications (CCA)’, pp. 1103–1110.
- Rao, C. V., Wright, S. J. & Rawlings, J. B. (1998), ‘Application of interior-point methods to model predictive control’, *Journal of optimization theory and applications* **99**(3), 723–757.
- Raravi, G., Andersson, B., Nélis, V. & Bletsas, K. (2014), ‘Task assignment algorithms for two-type heterogeneous multiprocessors’, *Real-Time Systems* **50**(1), 87–141.
URL: <http://dx.doi.org/10.1007/s11241-013-9191-3>
- Shewchuk, J. R. et al. (1994), ‘An introduction to the conjugate gradient method without the agonizing pain’.
- Thammawichai, M. & Kerrigan, E. C. (2016), Feedback scheduling for energy-efficient real-time homogeneous multiprocessor systems, *in* ‘2016 IEEE 55th Conference on Decision and Control (CDC)’, pp. 1643–1648.
- Vereecken, W., Heddeghem, W. V., Colle, D., Pickavet, M. & Demeester, P. (2010), Overall ICT footprint and green communication technologies, *in* ‘2010 4th International Symposium on Communications, Control and Signal Processing (ISCCSP)’, pp. 1–6.
- Wright, S. J. (1997), *Primal-dual interior-point methods*, SIAM.