Libor Bukata

Parallel Algorithms for Optimization of Production Systems

April 2018

CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Electrical Engineering
Department of Control Engineering

Parallel Algorithms for Optimization of Production Systems

by

Libor Bukata

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Control Engineering

Doctoral Thesis

Supervisor: Doc. Ing. Přemysl Šůcha, Ph.D.
Ph.D. programme: Electrical Engineering and Information Technology
Branch of study: Control Engineering and Robotics
Submission date: April 2018

Libor Bukata: Parallel Algorithms for Optimization of Production Systems, Ph.D. Thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Control Engineering, April 2018, Prague.

Acknowledgement

I would like to express my deep gratitude to doc. Ing. Přemysl Šůcha, Ph.D. for his invaluable advice and assistance during my doctoral studies. I also thank Ing. Pavel Burget, Ph.D. and his team for their work on the verification of optimization results in Škoda Auto company. I cannot forget to give thanks to prof. Dr. Ing. Zdeněk Hanzálek who enabled me to work on interesting topics, and my colleagues who created a pleasant and friendly atmosphere. My special thanks belong to Ing. Antonín Novák for his advice on using special Gurobi simplex method.

I highly appreciate the opportunity to study at the Department of Control Engineering, and I admire the Czech Technical University in Prague for being a good alma mater. The results I achieved during my research would not be possible without my family and girlfriend Bc. Hana Falberová, whose never-ending support and patience I fully appreciate.

Finally, I acknowledge all the grants which funded the research presented in this dissertation. Namely, the work was supported by the Grant Agency of the Czech Republic under the projects GACR P103/12/1994 and FOREST GACR P103-16-23509S, by the Ministry of Education of the Czech Republic under the projects DEMANES 295372 and OK7X 295372, by the Ministry of Industry and Trade of the Czech Republic under the project FV10299, by the ARTEMIS initiative funded by the European Commission, and by the European Regional Development Fund under the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466). I am also grateful to NVIDIA Corporation for the donation of three Tesla K20 graphics cards, and Škoda Auto for the cooperation and support through the contract 830-8301343/13135.

Libor Bukata
Prague, April 2018

Declaration

This doctoral thesis is submitted in partial fulfillment of the requirements for the degree of doctor (Ph.D.). The work submitted in this dissertation is the result of my own investigation, except where otherwise stated. I declare that I worked out this thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis. Moreover I declare that it has not already been accepted for any degree and is also not being concurrently submitted for any other degree.

Libor Bukata
Prague, April 2018

Abstract

The industrial production involves complex processes that directly determine the throughput and manufacturing cost, therefore, it is not surprising that there is a great demand for computer-aided optimization to improve the profitability. Such optimization is, however, typically computationally expensive, and therefore, it is very beneficial to use modern multi-core processors or graphics cards, which can accelerate the optimization about one to two orders of magnitude, in order to find better-optimized processes in a limited time. The transition to the parallel optimization, however, often requires the redesign of the algorithms and good knowledge of architecture. For that reason, it cannot be taken as granted in Operations Research.

In this thesis, we propose novel parallel algorithms to solve two problems that are important to optimize production processes. The first problem is the energy optimization of robotic cells which goal is to minimize the total energy consumption without any deterioration in the throughput. The second problem is the Resource Constrained Project Scheduling Problem that is a universal problem applicable in, e.g., the metallurgical industry and assembly shop scheduling.

The performance of our algorithms was verified on benchmark datasets. The experiments revealed that the Hybrid Heuristic and Branch & Bound algorithm can optimize industrial-sized robotic cells with up to 12 robots, compared to the existing works where 4 robots were considered at maximum. The Tabu Search algorithm, on the other hand, is designed for graphics cards and its performance is superior to other existing Tabu Search implementations.

Besides the benchmarks, the outcomes were also used to optimize an existing robotic cell from Škoda Auto with the result of 20 % energy saving, which indicates that if the optimization is widely used in industry, it will improve the environmental and financial sustainability. The cooperation with industrial partners (Blumenbecker, Škoda Auto) continues within the eRobot project, which main goal is to integrate the proposed algorithms into the digital factory software in order to make the optimization accessible to designers of robotic cells.

Keywords: efficient manufacturing, optimization, parallel algorithms, energy, robotics, Resource Constrained Project Scheduling Problem, Hybrid Heuristic, Branch & Bound, Tabu Search

Abstrakt

Průmyslová výroba je založena na složitých procesech, které přímo ovlivňují výrobní kapacitu a celkové náklady. Není proto divu, že je snaha využít počítače k optimalizaci těchto procesů za účelem zlepšení rentability výroby. Optimalizace výrobních procesů je ovšem typicky velmi náročná na výpočetní zdroje, a tak je velmi výhodné použít vícejádrové procesory nebo grafické karty, od kterých lze typicky očekávat urychlení o jeden či dva řády, aby se nalezly lépe optimalizované procesy v daném časovém horizontu. Přechod k paralelní optimalizaci nicméně často vyžaduje nový návrh algoritmů a velmi dobrou znalost použité architektury. Z těchto důvodů nejsou paralelní algoritmy všeobecně používány v oblasti operačního výzkumu.

Tato dizertační práce se zabývá návrhem nových paralelních algoritmů pro řešení dvou problémů, které jsou důležité pro optimalizaci výrobních procesů. První problém se týká optimalizace spotřeby robotických buněk, kde cíl je minimalizovat celkovou spotřebu energie bez dopadu na kapacitu výroby. Druhý problém je rozvrhování s omezenými zdroji, což je universální problém, jež najde uplatnění například v hutním průmyslu nebo v rozvrhování montážních hal.

Výkonnost algoritmů byla ověřena na testovacích datech obsahujících optimalizační problémy. Experimenty ukázaly výbornou škálovatelnost a paměťovou efektivitu hybridní heuristiky a metody větví a mezí, které lze použít pro optimalizaci robotických buněk až s 12 roboty. Pro porovnání existující práce uvažovaly maximálně 4 roboty. Tabu Search algoritmus naproti tomu byl navržen pro grafické karty a jeho efektivita překonává ostatní existující implementace tohoto algoritmu.

Algoritmy byly také použity k optimalizaci existující robotické buňky ve Škodě Auto. Změřená úspora 20 % energie naznačuje, že pokud by se optimalizace běžně používala v průmyslu, tak pak by to mělo pozitivní dopad na prostředí a ekonomiku. Tento výsledek podnítil další spolupráci s průmyslovými partnery (Blumenbecker, Škoda Auto), s kterými v rámci projektu eRobot pracujeme na integraci navržených algoritmů do softwaru pro virtuální zprovoznění robotických buněk. Integrace umožní snadné použití optimalizace širokému okruhu vývojářů robotických buněk.

Klíčová slova: efektivní výroba, optimalizace, paralelní algoritmy, energie, robotika, rozvrhování s omezenými zdroji, hybridní heuristika, metoda větví a mezí, Tabu Search

Goals and Objectives

The thesis deals with the parallel optimization algorithms for the production systems. Its main goals were determined as follows:

1. Study the existing literature related to the energy optimization of robotic cells and identify possible improvements.
2. Devise a mathematical model that considers important optimization aspects to minimize the energy consumption of industrial robots.
3. Propose heuristic and exact algorithms to solve industrial-sized robotic cells. Both the algorithms should utilize the problem structure and multi-core processors.
4. Design and implement a parallel Tabu Search algorithm to solve the Resource Constrained Project Scheduling Problem on graphics cards.
5. Verify the proposed algorithms on benchmark instances and compare them with the existing works.

Contents

List of Acronyms **1**

1 Introduction **3**
1.1 Closest State-of-the-Art Work . . . 5
1.2 Key Contributions . . . 6
1.3 Potential Impact . . . 7
1.4 Structure of the Thesis . . . 8

2 Energy Optimization of Robotic Cells **9**
2.1 Related Work . . . 11
2.2 Contribution and Outline . . . 12
2.3 Problem Statement . . . 13
2.3.1 Example 1 . . . 15
2.3.2 Example 2 . . . 16
2.4 Mixed-Integer Linear Programming Model . . . 18
2.5 Parallel Heuristic Algorithm . . . 20
2.5.1 Generation of Alternatives . . . 22
2.5.2 Generation of Tuples . . . 22
2.5.3 Reduced Linear Programming Problem . . . 23
2.5.4 Sub-heuristics . . . 25
2.6 Parallel Branch & Bound Algorithm Overview . . . 26
2.7 Node Definition . . . 28
2.8 Branching . . . 28
2.8.1 Order Propagation . . . 29
2.8.2 Propagation of Locations . . . 30
2.8.3 Propagation of Power Saving Modes . . . 31
2.8.4 Fast Feasibility Checks . . . 32
2.9 Energy Evaluator of Nodes . . . 32
2.9.1 Composite Activities . . . 32
2.9.2 Extra Activity Sets . . . 33
2.9.3 Convex Envelopes . . . 34
2.9.4 Lower Bound based on Convex Envelopes . . . 36
2.10 Deep Jumping . . . 40
2.11 Parallelization . . . 42
2.12 Experimental Results . . . 45
2.12.1 Performance Experiments . . . 45
2.12.2 Optimality Experiments . . . 49
2.12.3 Quality Experiments . . . 50

2.13 Case Study from Škoda Auto 54
2.14 Conclusion 56

3 Project Scheduling on Graphics Cards 57
3.1 Related works 58
3.2 Contribution and Outline of the Chapter 60
3.3 CUDA platform 61
3.4 Problem Statement 62
3.4.1 Mathematical Formulation 63
3.4.2 Instance Example 64
3.5 Outline of the Tabu Search meta-heuristic 65
3.6 Exploration of the Solution Space 66
3.6.1 Creating Initial Activity Order 66
3.6.2 Move Transformation 66
3.6.3 Neighborhood Generation 67
3.6.4 Filtering Infeasible Moves 67
3.6.5 Simple Tabu List and Cache 68
3.7 Schedule Evaluation 69
3.7.1 Capacity-Indexed Resources Evaluation 69
3.7.2 Time-Indexed Resources Evaluation 72
3.7.3 Schedule Evaluation Procedure 73
3.7.4 Heuristic Selection of Resources Evaluation Algorithms 73
3.8 Parallel Tabu Search for the CUDA platform 75
3.8.1 Block Cooperation and Distribution of Iterations . . . 77
3.8.2 Memory Model 78
3.9 Experimental Results 78
3.9.1 Evaluation of the Selection Heuristic 82
3.9.2 Demonstration of Convergence 83
3.10 Conclusion 83

4 Conclusion and Future Work 85
4.1 Fulfillment of Goals 85

Bibliography 93

A Nomenclature – Chapter 2 95

B Nomenclature – Chapter 3 99

C Curriculum Vitae 101

D List of Author's Publications 103

List of Figures

1.1 Problems solved in this thesis with their applications. 3
1.2 Typical energy distribution for robotic cells in Škoda Auto. . 4

2.1 Robotic cell with two welding robots in digital factory software. 9
2.2 Robotic cell with two robots that perform welding operations. 15
2.3 An example of a robotic cell from automotive industry. 17
2.4 Flowchart of the parallel Hybrid Heuristic. 21
2.5 Block diagram of the Branch & Bound algorithm. 27
2.6 Example of the order propagation (two activities are selected). 30
2.7 Example of the propagation after selecting locations. 30
2.8 Illustration of the composite activity and its related sets. . . 32
2.9 Example of the construction of the convex envelope. 34
2.10 Example of Deep Jumping. 41
2.11 Parallel Branch & Bound algorithm. 43
2.12 Thread-safe storage of nodes. 43
2.13 Progress of the heuristic and MILP solver on M8_8 instance. . 46
2.14 Scalability graphs for S5, M8, and L12 datasets. 48
2.15 Robotic cell from Škoda Auto. 54

3.1 Integration of the heuristic into the optimization process. . . 57
3.2 Fermi architecture – memory diagram. 61
3.3 Graph of precedences for the example instance. 64
3.4 Utilization of resources for the example instance. 65
3.5 Example of the swap move. 67
3.6 An example of the resource state update. 71
3.7 Example of the decision tree. 74
3.8 Parallel Tabu Search for the CUDA platform. 76
3.9 Graph of convergence for the GPU version. 84

List of Tables

2.1 Performance of LP solvers for the heuristic. 45
2.2 Performance metrics of the Branch & Bound algorithm. . . . 47
2.3 Time to optimality for S3 dataset. 49
2.4 Quality of solutions with $t_{\max} = 30\,\mathrm{s}$ for S5 dataset. 51
2.5 Quality of solutions with $t_{\max} = 1\,\mathrm{h}$ for S5 dataset. 51
2.6 Quality of solutions with $t_{\max} = 600\,\mathrm{s}$ for M8 dataset. 52
2.7 Quality of solutions with $t_{\max} = 1\,\mathrm{h}$ for M8 dataset. 52
2.8 Quality of solutions with $t_{\max} = 3\,\mathrm{h}$ for L12 dataset. 53
2.9 Comparison of lower bounds for M8 dataset. 53
2.10 Dependence of the quality of solutions on the cycle time. . . . 54

3.1 Data of an example instance. 64
3.2 Attributes used for learning. 74
3.3 PTSG parameters and datasets information. 78
3.4 Quality of solutions — J30. 79
3.5 Performance comparison — J30. 79
3.6 Quality of solutions — J60. 80
3.7 Performance comparison — J60. 80
3.8 Quality of solutions — J90. 81
3.9 Performance comparison — J90. 81
3.10 Quality of solutions — J120. 81
3.11 Performance comparison — J120. 81
3.12 Comparison with other heuristics. 82
3.13 Percentage of correctly classified problems for each dataset. . 83
3.14 Effect of the selection heuristic on the PTSG performance. . . 83

List of Algorithms

1 Removing infeasible moves from the reduced neighborhood. . 68
2 Check if a move is in the Simple Tabu List. 68
3 Add a move to the Simple Tabu List. 69
4 Method updates state of resources after adding activity i. . . 71
5 Algorithm calculates the earliest start time of activity i. . . . 72
6 Updating of resources after adding activity i. 73
7 Complete schedule evaluation. 73

List of Acronyms

BaB Branch & Bound 49
CF Convex Functions 46
CPU Central Processing Unit 58
CUDA Compute Unified Device Architecture 61
FSP Flowshop Scheduling Problem 59
GPU Graphics Processing Unit 58
LB Lower Bound 26
LP Linear Programming 20
MILP Mixed Integer Linear Programming 6
PH Parallel Heuristic 50
PTSG Parallel Tabu Search for GPU 75
RCPSP Resource Constrained Project Scheduling Problem 4
STL Simple Tabu List 68
TC Tabu Cache 68
TL Tabu List 58
TS Tabu Search 58
UB Upper Bound 26

Chapter 1

Introduction

Manufacturing of goods involves complex processes that guarantee the desired production rate with limited resources (e.g., robots, machines). Since these processes are extremely difficult to optimize by hand (many variables and combinations), the use of optimization algorithms is crucial to fully utilize machines, improve the throughput, or reduce costs. However, the great complexity of manufacturing problems often makes the finding of the optimal solution impossible even for the state-of-the-art algorithms, therefore, the algorithm typically searches for the best feasible solution in a given time horizon.

Much better solutions can be obtained if the algorithm is accelerated by using modern parallel architectures such as multi-core processors or graphics cards. A recipe for the high performance is to utilize many (slower) execution cores instead of one (fast) core since it is resulting in a significantly better performance per watt. As a good example, consider battery-powered

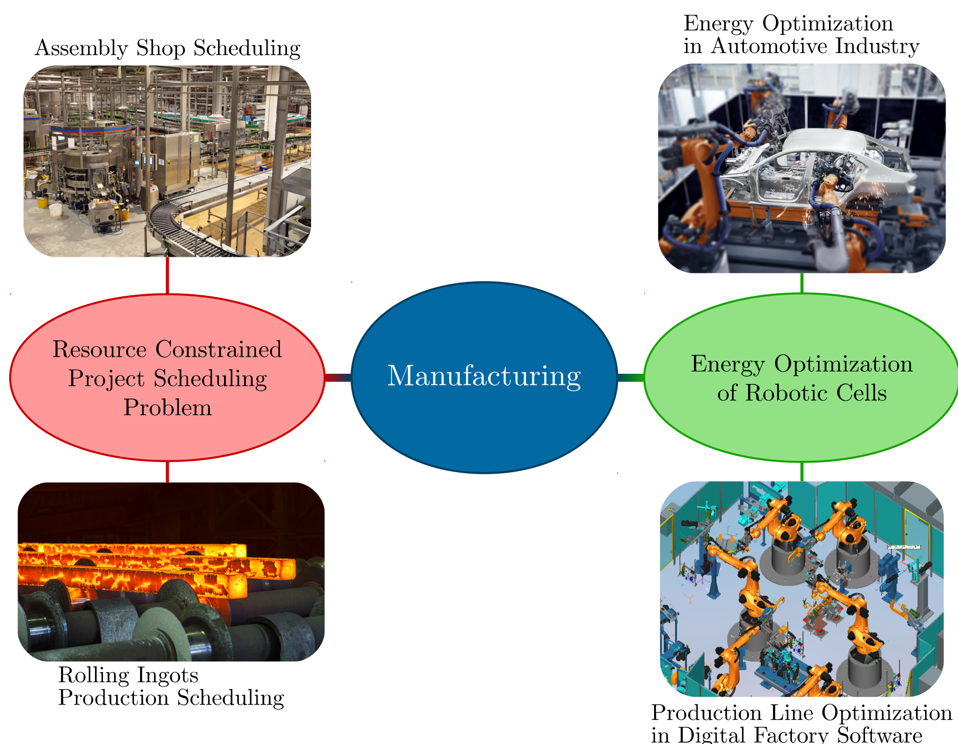


Figure 1.1: Problems solved in this thesis with their applications.[1]

[1]Embedded photographs: ID1974,ruhmal,xieyuliang/Shutterstock.com

smartphones that are usually equipped with multi-core processors. If the algorithm is parallelized and executed on a powerful graphics card or multi-core processor, speedups between one to two orders of magnitude are achievable (see, e.g., [41, 58, 40]) depending on the experience of a programmer, optimization problem, and hardware configuration. The effective parallelization of optimization algorithms is, however, a difficult task that is under the active research not only in Operations Research (see, e.g., [42, 19, 9]).

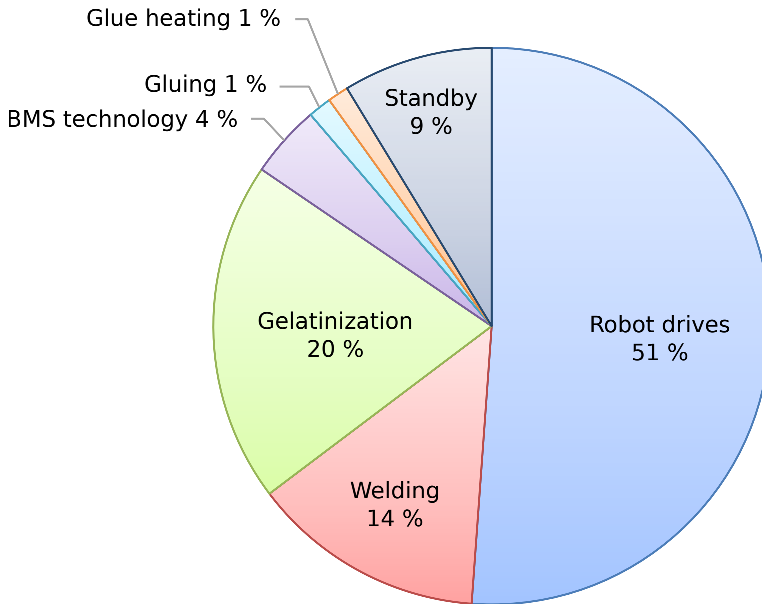


Figure 1.2: Typical energy distribution for robotic cells in Škoda Auto.

This thesis is dedicated to the design and implementation of parallel optimization algorithms for production systems. The proposed algorithms solve two combinatorial problems (see Figure 1.1) that are particularly useful in the manufacturing industry. The first one is the problem of energy optimization of robotic cells that attracted particular attention of industrial partners (Škoda Auto, Blumenbecker) since the reduction of energy consumption significantly decreases manufacturing costs, and robot drives, based on measurements of a robotic cell with 6 robots in Škoda Auto, consume about half of the total energy required by a typical robotic cell in the automotive industry (see Figure 1.2). The second one is the Resource Constrained Project Scheduling Problem (RCPSP) which is a well established and versatile problem with many applications. Despite its name, it can also be used to model manufacturing processes, for example, consider Assembly Shop Scheduling and Rolling Ingots Production Scheduling (see, e.g., [4, 43, 48]).

To solve the aforementioned problems, we proposed and implemented three parallel cutting-edge optimization algorithms. In case of the energy optimization of robotic cells, the parallel Hybrid Heuristic and Branch & Bound algorithms were proposed to minimize the energy consumption of robots by changing their speeds, robotic paths, the order of operations, and applying power saving modes such as brakes or bus-power-off. In case of the RCPSP, the Tabu Search algorithm was designed to find a feasible schedule, i.e., an order of activities, with the minimal completion time (the so-called makespan). In general, all the algorithms utilize the problem structure and the parallelization to find high-quality solutions fast. The Tabu Search algorithm, moreover, uses the graphics card to accelerate the searching process even more than it is possible by using conventional multi-core processors.

1.1 Closest State-of-the-Art Work

The closest works related to the proposed algorithms and solved problems are briefly summarized in this section. Detailed literature reviews can be found in Sections 2.1 and 3.1.

In general, there is a large body of research on the optimization in robotics; however, only a small part is devoted to the energy optimization of robotic cells. If it is the case, most of the works are dedicated to the local optimization of individual trajectories (see, e.g., [47, 46], and [11]) compared to the holistic optimization that is considered in this dissertation (see Chapter 2).

One exception is the pioneering work of Wigström and Lennartson [55], where the authors optimized the energy consumption of robotic cells as a whole by solving a nonlinear mathematical model. The optimal schedule determines the timing of robots such that the synchronization constraints are satisfied and the energy consumption is minimal for the fixed work cycle time. Nevertheless, the approach was verified on the Job Shop scheduling problem.

On the borderland between the local and holistic optimization, there is the work of Meike et al. [39, 38], where the authors proposed to optimize the last movement to the robot home position and the subsequent waiting period to save energy. Compared with the previous study [55], the robot brakes were considered at the robot home position. Although a very small part of the robotic cell was taken into account, it was estimated that 7.3 % of energy can be saved.

The optimization algorithms solving combinatorial problems are, in general, computationally expensive; therefore, it is desirable to utilize modern hardware like multi-core processors or graphics cards to accelerate compu-

tations. The effect of the parallelization may be so significant that it enables to find new best solutions, for example, consider works of Subramanian et al. [50] and Jin et al. [30] where the Vehicle Routing Problem was solved by parallel heuristics.

Recently, graphics cards are becoming popular to accelerate the optimization algorithms because of their huge computational power compared to conventional multi-core processors. However, the design of parallel algorithms for graphics cards is a non-trivial task, for that reason, most of the works up to that time solved rather simpler problems such as the Knapsack (e.g., [8, 34]) or Quadratic Assignment Problem (e.g., [40, 44]).

Fortunately, the first works addressing complex problems are beginning to appear. For example, Czapiński and Barnes [41] implemented the parallel Tabu Search algorithm suitable for graphics cards to solve the Flowshop Scheduling Problem, and Delévacq et al. [18] used the parallel Ant Colony Optimization meta-heuristic to solve the Traveling Salesman Problem. And finally, our work in Chapter 3 proposes the efficient Tabu Search algorithm to deal with an inherently complex combinatorial problem, and compared to work [41], the quality of solutions is investigated as well.

1.2 Key Contributions

The most important contributions of this thesis grouped according to the topics are as follows:

1. **Energy optimization of robotic cells** (Chapter 2)
    - Based on measurements of a small industrial robot (described in [15]), we *identified important aspects* influencing the energy consumption of industrial robots.
    - The aspects were used to formulate a *mathematical model* that minimizes the energy consumption for a fixed robot cycle time.
    - Compared to the work of Wigström and Lennartson [55], our model considers more optimization aspects such as *alternative robotic paths* and application of *power-saving modes* of robots. Moreover, our different definition of the cycle time enables higher production throughput.
    - To solve large robotic cells with up to 12 robots, which sometimes occur in the manufacturing industry, the *parallel Heuristic and Branch & Bound algorithm* were designed and implemented.
    - The proposed algorithms *utilize the problem structure and modern hardware* to find energy-efficient solutions. The performance is superior to the state-of-the-art Mixed Integer Linear Program-

ming (MILP) solvers and existing works where robotic cells with one to four robots were taken into account.
- In cooperation with Pavel Burget's team, we verified the results of the optimization on an existing robotic cell in Škoda Auto. The measured energy savings achieved 20 %, which was close to the estimation of algorithms; therefore, the approach was *verified in industry.*
- The work *attracted the attention of industrial partners* such as Škoda Auto and Blumenbecker, which resulted in the joined project eRobot with Blumenbecker and Brno University of Technology. The goal of the project is to integrate our optimization algorithms into digital factory software to enable user-friendly optimization of robotic cells during the virtual commissioning.

2. **Solving the RCPSP on graphics cards** (Chapter 3)
   - The *Tabu Search algorithm* was designed to solve the RCPSP on graphics cards. It is the first known algorithm solving this complex combinatorial problem on graphics cards.
   - The performance achieved on the graphics card is superior to the performance on the multi-core processor because of the *effective schedule evaluation*, optimized memory access, and homogeneous model. Moreover, the algorithm that evaluates the schedule is dynamically selected based on the properties of the current problem.
   - The experiments confirmed that the algorithm finds better solutions than other Tabu Search implementations in the literature.

1.3 Potential Impact

The potential impact of this thesis can be split into two parts. The first part is the *development of new methods, models, and approaches* that contribute to the current knowledge and may influence the future research directions. The key findings were published in two impacted journals, and currently, one extra journal paper is under the review. Besides the *journal papers*, there is a book chapter, conference paper, and many contributions at international conferences.

The second part, which concerns the energy optimization of robotic cells, is the potential *environmental and economic impact* if the results of the algorithms are widely used in the industry. The measurements and experiments indicate that the optimization can significantly reduce the energy consumption of robots, and as a result, relax the requirements placed on

energy sources and improve the profitability. The common usage, however, requires that the algorithms are integrated into digital factory software in order to make the optimization accessible to designers of robotic cells. The *project eRobot*, which I also participate in, will remove this barrier by integrating the optimization algorithms into Siemens Process Simulate (popular in the automotive industry).

1.4 Structure of the Thesis

The rest of the thesis is divided into the chapters as follows. Chapter 2 is dedicated to the energy optimization of robotic cells. The MILP formulation, heuristic, and Branch & Bound algorithm are proposed and verified on benchmark instances and the existing robotic cell from Škoda Auto.

Chapter 3 describes the design and implementation of the parallel Tabu Search algorithm for graphics cards that solves the RCPSP. The core parts of the algorithm such as the schedule evaluation and generation of a neighborhood are presented in details as well as the memory model and optimized data structures. The algorithm is compared with other existing works on benchmark instances.

Since chapters 2 and 3 address different problems, each of them can be read separately in a similar way as a research paper with **nomenclatures** included in appendices A and B, respectively. The dissertation concludes with Chapter 4, where the achieved results are summarized, fulfillment of goals is evaluated, and future work is discussed.

Chapter 2

Energy Optimization of Robotic Cells

Robotic cell is a highly complex system (see Figure 2.1), which is mainly used in industry to increase the productivity and repeatability of the manufacturing process. Compared to human beings, industrial robots can work in dangerous environments with a higher precision, therefore, they are invaluable for heavy industry (e.g., automotive). On the other hand, a high energy consumption may limit a further extension of the robotic production. First of all, the power grid capacity can be insufficient to supply new robots since they are usually programmed to run at the maximal speed to meet the desired production rate. Such performance leads to a power profile that sharply fluctuates during production; for instance, a regular 6-axis industrial robot with a 200 kg payload requires between 0.5 and 20 kW of power (Meike and Ribickis [37]). Even if the capacity is sufficient, the increasing costs of electricity may constitute a deterrent. Besides physical and economical aspects, European Union (see report [1]) and ecological groups put pressure on factories to be more environment-friendly. As a result, there is a great demand for energy-efficient robotic cells.

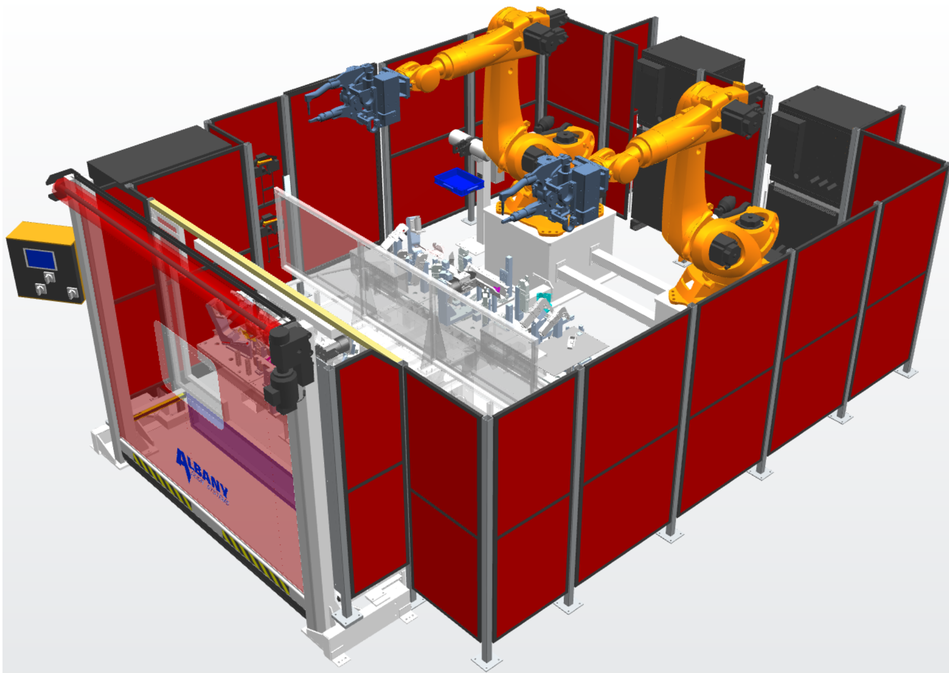


Figure 2.1: Robotic cell with two welding robots in digital factory software.

Robots support power-saving modes to reduce energy consumption. For example, if the robot is stationary, its brakes can slightly alleviate the energy burden if they are released early because the motors do not consume energy in holding the robot in a certain position, and the power needed to keep the brakes open does not apply to the released brakes. On one hand, the frequent use of brakes could result in significant energy savings; on the other hand, caution must be exercised because the number of brake cycles is limited to 5 million, and the expected lifetime of robots is about 15 years (Meike et al. [39]). Some robots, such as those from KUKA, support the PROFIenergy profile [2], which allows controlling the power consumption by sending commands through a PROFINET network. This enables deeper energy-saving modes for robots, such as bus power-off or hibernate. Several experimental facilities were using PROFIenergy, such as the Mercedes-Benz plant in Sindelfingen, in which various experiments were carried out [49].

Besides applying power-saving modes, it is also beneficial to reduce speed of robots without deterioration in the production rate, e.g., avoid a fast movement to a location if the robot waits for a signal after approching it. Our case study (see Section 2.13) indicates that this approach may be resulting in significant energy savings even for existing robotic cells, since the energy consumption was decreased approximately by 20 % for the robotic cell from Škoda Auto.

In this research, we proposed a novel mathematical model that minimizes the energy consumption by changing robot speeds, considering various robot positions (i.e., configurations), applying their power-saving modes (brakes, bus power-off), and selecting an order of operations. The *robot cycle time* is fixed and corresponds to the time interval (∼1/throughput) between two outgoing workpieces after the start-up phase. The robot cycle time is typically shorter than the duration of the start-up phase; thus, there are usually more unfinished workpieces in the robotic cell at a given time. Such parallelism is analogous with the cyclic scheduling of the processor pipeline (see, e.g., [5] and [54]). Note that the optimization is suitable not only for planned robotic cells but also for the existing robotic cells since at least robot speeds can be optimized without complex changes to the structure of the robotic cell, as shown in our case study 2.13. Even in case of planned robotic cells, it is unnecessary to consider all the optimization aspects, which were identified based on measurements on a small industrial KUKA robot, since a designer may select only a subset of them to reduce time to deployment.

In order to optimize the robotic cells with up to 12 robots, we proposed two tailor-made algorithms: a *parallel Hybrid Heuristic*, and *parallel Branch & Bound* algorithm. The merits of these algorithms and their superior performance compared to general MILP solvers will be discussed in Section 2.2.

2.1 Related Work

In general, there is a large body of research on the optimization in robotics; however, only a small part is devoted to the energy optimization since the majority of work concentrates on increasing the throughput of the robotic cells (see, e.g., Dawande et al. [17] for a survey). Meike and Ribickis [37] summarize various methods how to reduce the energy consumption of industrial robots, and estimate their potential to save energy. One approach is to optimize individual robot trajectories, as investigated in [47, 46], and [11]. Another approach is to consider the robotic cell as a whole to reduce the energy consumption. In the pioneering work of Wigström and Lennartson [55], the authors optimized the energy consumption of robotic cells as a whole by solving a nonlinear mathematical model, a solution of which determined an energy efficient timing of robots that satisfies synchronization constraints. The approach was benchmarked on the Job Shop scheduling problem.

In our research [14], we followed their work and proposed a more general model, where robot positions and their power-saving modes are taken into account. Moreover, our formulation enables to consider multiple workpieces in the robotic cell at the same time. In order to optimize robotic cells with up to 12 robots, a parallel Hybrid Heuristic was designed and compared with the MILP solver on publicly available datasets. The algorithms were used to optimize an existing robotic cell from Škoda Auto and achieved results revealed energy savings of up to 20 %.

Gadaleta et al. [22] proposed a method that optimizes the placement of robots for a given set of tasks. An energy model of robots was created in Modelica and used to calculate time/energy maps (contour plots) that indicate the working time of a robot and its consumption for a given placement (base coordinates). The authors demonstrated their approach on two robotic cells and concluded that up to 20 % of energy could be saved if their tool is used to determine placements of robots.

In the work of Meike et al. [39, 38], the authors proposed to optimize the last movement to the robot home position and the subsequent waiting period to save energy. Compared with the work of Wigström and Lennartson [55], in this study, the robot brakes were considered at the robot home position. Although only a relatively small part of the robotic cell was considered, the authors estimated that the energy savings for the robotic cell reached 7.3 %.

In the work of Mashaei and Lennartson [36], an energy model of a pallet-constrained flow shop problem was formulated to find an optimal switching control strategy for achieving the desired throughput and minimal energy consumption. Idle states of machines were also taken into account to reduce the energy consumption when the machine was not working. However, the

model required a line with a particular structure, i.e., a closed-loop pallet system; therefore, it was not generally applicable to robotic cells.

Since the optimization problems are often very difficult to solve, there is an endeavor to accelerate algorithms by using multi-core processors or other modern hardware. For example, Subramanian et al. [50] and Jin et al. [30] proposed parallel heuristics for the Vehicle Routing Problem, and in both the cases, the parallelization enabled them to find new best solutions. Boyer et al. [8] proposed a parallel implementation of the dynamic programming method for the Knapsack Problem, the results of which revealed significant speedups. Our work proposes two parallel tailor-made algorithms for the energy optimization of robotic cells. The outcomes indicate (see Section 2.12) a good scalability and performance. In general, the parallelization enables to find better solutions within a given time limit.

2.2 Contribution and Outline

We cooperated with Škoda Auto to identify key optimization aspects, and these aspects were verified by the measurements on a small KUKA robot. Based on the analysis, the mathematical model was formulated. In contrast to the existing works [55, 56, 57], this model supports the energy-saving modes and alternative positions of robots. Moreover, compared to Wigström and Lennartson [55], we do not limit the work parallelism by the assumption that the whole robotic cell is dedicated to one workpiece during its entire processing time. The validy of the model was verified on benchmark datasets and the robotic cell from Škoda Auto, where the measurements confirmed the potential to save about 20 % of energy for existing robotic cells.

Although the model is solvable by MILP solvers, their performance is insufficient for large instances. Therefore, we implemented a parallel Hybrid Heuristic and parallel Branch & Bound algorithm that utilize the problem structure and provide an additional insight into the problem. The heuristic is very efficient and fast, thanks to the application of a special Gurobi simplex algorithm for piecewise linear convex functions and the parallelization that enables the algorithm to exploit the power of all processor cores. The parallel Branch & Bound algorithm, on the other hand, uses a tight lower bound based on convex envelopes that can be efficiently solved by a special Gurobi simplex algorithm for piecewise linear convex functions. Besides the bounding, the algorithm is very efficient at finding feasible solutions since it uses a Deep Jumping heuristic to guide the search to promising directions in the Branch & Bound tree.

In summary, the key contributions are the mathematical model and efficient algorithms that optimize robotic cells with up to 12 robots. Moreover,

the approach was verified on the existing robotic cell from Škoda Auto, where the energy consumption was reduced approximately by 20 %. The parallel heuristic and *generator* of problem instances are publicly available as *open-source software* and can be downloaded from `https://github.com/CTU-IIG`.

The rest of the chapter is structured as follows. The next section sets out the formal specification of the problem. Two examples are provided to illustrate mathematical formalism. Section 2.4 presents the MILP model, and Section 2.5 introduces the parallel heuristic algorithm for solving the problem. Section 2.6 presents an overview of the parallel Branch & Bound algorithm, and the following sections describe its core parts such as the branching, bounding, search strategy, and parallelization. The performance of the proposed algorithms is tested on benchmark datasets in Section 2.12, and the model is verified on a real-world problem from Škoda Auto in the case study (see Section 2.13). Finally, the last section of this chapter concludes the work and discusses the future directions. In order to simplify reading of the dissertation, a **nomenclature** is in Appendix A.

2.3 Problem Statement

The energy optimization problem of the robotic cell can be defined as follows. There is a set of robots $R = \{r_1, \ldots, r_{|R|}\}$ and set of graphs $G = \{G^{r_1}, \ldots, G^{r_{|R|}}\}$. The graph of robot $r \in R$ is defined as $G^r = (A^r_{\mathcal{S}}, A^r_{\mathcal{D}})$, where nodes $A^r_{\mathcal{S}}$ are *static activities* (e.g., waiting or welding), and edges $A^r_{\mathcal{D}}$ are *dynamic activities*, which define the possible robot moves between static activities. Let $A_{\mathcal{S}} = \bigcup_{\forall r \in R} A^r_{\mathcal{S}}$, $A_{\mathcal{D}} = \bigcup_{\forall r \in R} A^r_{\mathcal{D}}$, and $A^r = A^r_{\mathcal{S}} \cup A^r_{\mathcal{D}}$.

Each static activity $v \in A_{\mathcal{S}}$ has assigned set L_v of possible robot positions, i.e., *locations*, in which the motionless robot either works or waits for a signal. During this stationary phase, one of the robot *power-saving modes* $m \in M^r$, including a dummy power-saving mode for the robot held by motors, can be applied if the activity duration $d_v \geq \underline{d}^m$, where $\underline{d}^m$ is the minimal time to switch the mode on. Furthermore, it has to be satisfied that $\underline{d}_v \leq d_v \leq \overline{d}_v$, $\forall v \in A_{\mathcal{S}}$, where $\underline{d}_v$ and $\overline{d}_v$ are duration limits required by a manufacturing process. The energy consumption of activity v is then $f^m_{v,l}(d_v) = p^m_{v,l} d_v$, where $p^m_{v,l}$ is the input power of robot $r \in R$ at location $l \in L_v$ for mode $m \in M^r$.

The dynamic activity, i.e., edge $e = (v_1, v_2) \in A^r_{\mathcal{D}}$, consists of the set of *trajectories* T_e between v_1 and $v_2 \in A^r_{\mathcal{S}}$, where each trajectory $t = (l_1, l_2) \in T_e$ interconnects two locations $l_1 \in L_{v_1}$ and $l_2 \in L_{v_2}$. The duration of the robot movement along trajectory $t \in T_e$ is from $\underline{d}^t_e$ to $\overline{d}^t_e$. The duration d_e of dynamic activity $e \in A_{\mathcal{D}}$, regardless of the selected trajectory, is

limited to range $< \underline{d}_e, \overline{d}_e >$ where $\underline{d}_e = \min_{\forall t \in T_e} \underline{d}_e^t$ and $\overline{d}_e = \max_{\forall t \in T_e} \overline{d}_e^t$. Energy consumption of activity $e \in A_{\mathcal{D}}$ depends on the selected trajectory $t \in T_e$, duration d_e ($\underline{d}_e^t \leq d_e \leq \overline{d}_e^t$), and convex function $f_e^t(d_e)$. Convex function $f_e^t(d_e)$ maps duration d_e to energy consumption for $\forall e \in A_{\mathcal{D}}, t \in T_e$. The form of the function (see Vergnano et al. [53], p. 389, Eq. 12) is $f_e^t(d_e) = \sum_{i=1}^{5} \alpha_{e,i}^t d_e^{2-i}$, where $\alpha_{e,i}^t$ are constants.

Let $A = A_{\mathcal{S}} \cup A_{\mathcal{D}}$ be the set of all activities; then, S_a and P_a are the set of successors and predecessors of activity $a \in A$, respectively. If $|S_v| > 1$ for $v \in A_{\mathcal{S}}$, then the dynamic activities $e \in S_v$ are optional because only one trajectory $t \in T_e$ will be selected as the leaving one. The *optional activities*, denoted as $A_{\mathcal{O}} \subseteq A_{\mathcal{D}}$, can model alternative orders of operations. Note that $|S_e| = |P_e| = 1$ for $\forall e \in A_{\mathcal{D}}$.

A closed path of robot $r \in R$, including the order of operations, can be represented as a directed Hamiltonian circuit, $\mathcal{HC}_{\text{loc}}^r = (l_1, \ldots, l_{|A_{\mathcal{S}}^r|})$, through the activity locations, where each $v \in A_{\mathcal{S}}^r$ is visited only once during each cycle. The order of operations (activities), regardless of the selected locations, can be expressed in the form of a directed Hamiltonian circuit, $\mathcal{HC}_{\text{act}}^r = (v_1, \ldots, v_{|A_{\mathcal{S}}^r|})$, where the last activity $v_{|A_{\mathcal{S}}^r|}$, denoted as v_h^r, closes the cycle of robot r. The last activity v_h^r, regardless of the work of robot r, contains so-called *home positions* of robot r (i.e., $L_{v_h^r}$), where the robot typically waits for its next cycle. Note that $\mathcal{HC}_{\text{act}}^r$ can be derived from $\mathcal{HC}_{\text{loc}}^r$; however, the converse is not true. The robot cycle time $\overline{C_T}$, i.e., the production cycle time of the robotic cell, has to be fulfilled; in other words, $\sum_{\forall a \in A_{\mathcal{HC}}^r} d_a = \overline{C_T}$ for $\forall r \in R$, where $A_{\mathcal{HC}}^r = A_{\mathcal{S}}^r \cup U_{\mathcal{D}}^r$, $U_{\mathcal{D}}^r \subseteq A_{\mathcal{D}}^r$, and $|U_{\mathcal{D}}^r| = |A_{\mathcal{S}}^r|$. Set $U_{\mathcal{D}}^r$ contains dynamic activities on Hamiltonian circuit $\mathcal{HC}_{\text{act}}^r$. Let $A_{\mathcal{HC}} = \bigcup_{\forall r \in R} A_{\mathcal{HC}}^r$.

To guarantee that the robots cooperate with each other at the right time and location, there are global constraints for *time synchronization* and so-called *spatial compatibility*. Correct timing is ensured by inter-robot time lags $E_{\mathcal{TL}}$, where each time lag $(a_1, a_2) \in E_{\mathcal{TL}}$ has fixed length $l_{a_1,a_2} \in \mathbb{R}$ and height $h_{a_1,a_2} \in \mathbb{Z}$. Length l_{a_1,a_2} denotes a time shift between $a_1 \in A^{r_i}$ and $a_2 \in A^{r_j}$ $(r_i \neq r_j)$, and the height h_{a_1,a_2} enables addressing the current and previous robot cycles. Both terms, which are well-known in cyclic scheduling (see, e.g., [5, 54]), define the time relation as follows: $s_{a_2} \geq s_{a_1} + l_{a_1,a_2} - \overline{C_T} h_{a_1,a_2}$, where s_{a_1} and s_{a_2} are the start times of activities a_1 and a_2, respectively.

To ensure that the workpiece is passed correctly from one robot to another, i.e., that the workpiece is picked up from the same place where it was put, it is necessary to define the spatial compatibility as follows. Let v_1 and $v_2 \in A_{\mathcal{S}}$ be two static activities in which an inter-robot handover is carried out; then, the set of compatible location pairs is $Q_{v_1,v_2} \subseteq L_{v_1} \times L_{v_2}$, and

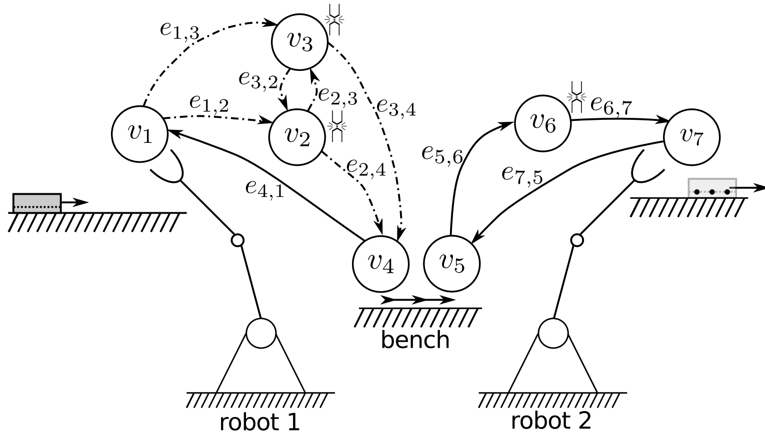


Figure 2.2: Robotic cell with two robots that perform welding operations.

set $Q_{l_i} = \{\forall l_j : (l_i, l_j) \in Q_{v_1,v_2} \text{ or } (l_j, l_i) \in Q_{v_1,v_2}\}$ contains all the locations (robot positions) compatible with location l_i. Finally, because the concurrent work of robots may result in collisions, time disjunctive combinations of trajectories and locations are specified as quadruplets $(a_1, g_1, a_2, g_2) \in K$, where $g_i \in L_{a_i}$ if $a_i \in A_{\mathcal{S}}$ or $g_i \in T_{a_i}$ if $a_i \in A_{\mathcal{D}}$.

The goal of the optimization is to find closed paths of robots (i.e., $\mathcal{HC}^r_{\text{loc}}$ for $\forall r \in R$), determine the timing of activities (i.e., the assignment of s_a and d_a for $\forall a \in A_{\mathcal{HC}}$), and decide which power-saving mode of the robot to apply for each static activity $v \in A_{\mathcal{S}}$ such that the global constraints are satisfied, collisions are prevented, and energy consumption $\sum_{\forall(v,l,m)\in X_1} f^m_{v,l}(d_v) + \sum_{\forall(e,t)\in X_2} f^t_e(d_e)$ is minimized, where set X_1 assigns a location and power-saving mode for $\forall v \in A_{\mathcal{S}}$, and set X_2 provides a selected trajectory for $\forall e \in A_{\mathcal{HC}}, e \in A_{\mathcal{D}}$. MILP formulation of this problem is presented in Section 2.4. In case you need to refresh the meaning of a symbol, please refer to the **nomenclature** in Appendix A.

2.3.1 Example 1

Figure 2.2 shows a robotic cell consisting of two robots: r_1 and $r_2 \in R$. Robot r_1 takes the weldment, carries out two spot-welding operations, and subsequently puts the weldment on a bench. Robot r_2 takes the weldment from the bench, carries out an additional spot-welding operation, and puts the weldment on a conveyor. The process flow of each robot can be expressed as a graph G^r; such as $G^{r_1} = (A^{r_1}_{\mathcal{S}}, A^{r_1}_{\mathcal{D}})$, where $A^{r_1}_{\mathcal{S}} = \{v_1, v_2, v_3, v_4\}$ and $A^{r_1}_{\mathcal{D}} = \{e_{1,2}, e_{1,3}, e_{2,3}, e_{2,4}, e_{3,2}, e_{3,4}, e_{4,1}\}$. Nodes v_2, v_3, v_6 correspond to spot-welding operations, whereas v_1, v_5 and v_4, v_7 are take and put operations, respectively. The edges define how to move from one operation to another, and the edge style determines whether the move is mandatory

(solid line), or optional (dashed line). The dashed edges, i.e., the optional dynamic activities $A_{\mathcal{O}}$, enable modeling alternative orders of operations; in this case, there are two possible orders: (v_1, v_2, v_3, v_4) and (v_1, v_3, v_2, v_4). The graphs shown in Figure 2.2 have activity-level granularity because each edge is a dynamic activity and each node is a static activity. However, a lower-level of granularity is achievable because each node v has locations L_v, and each edge e contains trajectories T_e. Because the weldment has to be passed at the right place at the right time, spatial and time synchronization between robots is necessary. For example, if $L_{v_4} = \{l_{4,1}, l_{4,2}\}$ and $L_{v_5} = \{l_{5,1}, l_{5,2}\}$, i.e., v_4 and v_5 have 2 locations each, then the spatial compatibility can be defined as $Q_{l_{4,1}} = \{l_{5,2}\}$ and $Q_{l_{4,2}} = \{l_{5,1}\}$. To ensure correct timing, two time lags need to be added: $s_{v_5} \geq s_{e_{4,1}} + l_{e_{4,1},v_5}$ and $s_{v_4} \geq s_{e_{5,6}} + l_{e_{5,6},v_4} - \overline{C_T}$. The first time lag ensures that robot r_2 takes the weldment from the bench after robot r_1 has put it there. The second time lag guarantees that robot r_2 has left the bench before robot r_1 puts another weldment there. The lengths of the time lags are set to the required time for a robot to safely leave the bench.

2.3.2 Example 2

Figure 2.3 shows a robotic cell where robots r_1 and $r_2 \in R$ are welding and gluing an automotive body. The process flow of each robot $r_i \in R$, including a conveyor belt that is simulated as a robot, is expressed as graph $G^{r_i} = (A_{\mathcal{S}}^{r_i}, A_{\mathcal{D}}^{r_i})$, where $v_1 \in A_{\mathcal{S}}^{r_1}$, $v_8 \in A_{\mathcal{S}}^{r_2}$, and $v_{12} \in A_{\mathcal{S}}^{r_3}$ are the last (home) activities $v_h^{r_1}$, $v_h^{r_2}$, and $v_h^{r_3}$, respectively, $v_2, v_3, v_4, v_5 \in A_{\mathcal{S}}^{r_1}$ are spot-welding operations, and $v_6, v_7 \in A_{\mathcal{S}}^{r_2}$ are gluing operations. Graph edges $A_{\mathcal{D}}^{r_i}$ determine allowed robot moves between static activities. Based on graph G^{r_1} structure it is obvious that the welding can be divided into two working groups: $\{v_2, v_3\}$ and $\{v_4, v_5\}$, where the first group is finished before the second one. Each group has undecided order of its spot-welding operations (two alternatives), which is modeled by using optional dynamic activities $e \in A_{\mathcal{O}}$ (dashed edges); in this case $A_{\mathcal{O}} = A_{\mathcal{D}}^{r_1}$. In total, there are four orders of operations for robot r_1: $(v_1, v_2, v_3, v_4, v_5)$, $(v_1, v_3, v_2, v_4, v_5)$, $(v_1, v_2, v_3, v_5, v_4)$, and $(v_1, v_3, v_2, v_5, v_4)$. Graphs G^{r_i}, $\forall i \in \{1, 2, 3\}$, have activity-level granularity since each edge is a dynamic activity and each node is a static activity. A lower-level of granularity is achievable (see, e.g., Figure 2.7) as each node v has locations L_v, and each edge e contains trajectories T_e.

Finally, time synchronization is needed to guarantee a correct cooperation between the robots. Consider the following manufacturing process: 1) automotive body gets on a conveyor belt, 2) it is transported to robots r_1 and r_2, 3) its front part is processed (activities v_2, v_3, v_6), 4) the body is moved forward, 5) its rear part is processed (activities v_4, v_5, v_7), and 6) it

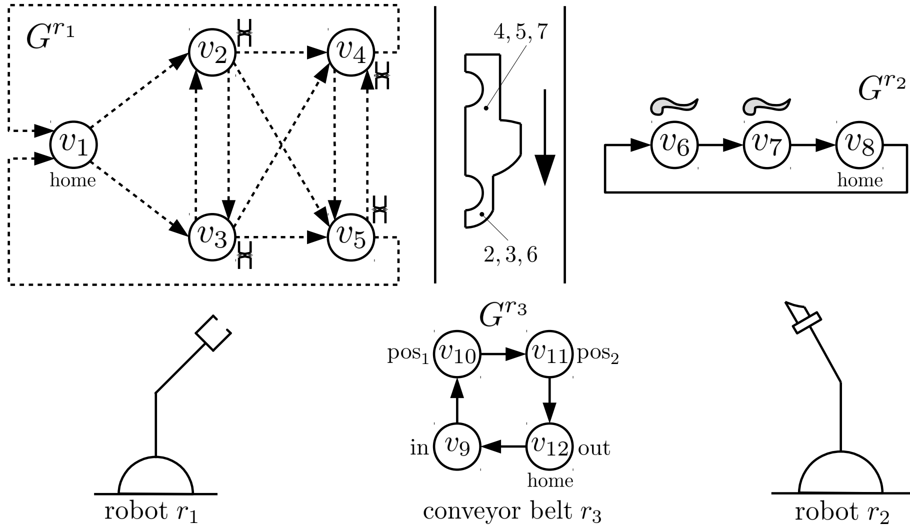


Figure 2.3: An example of a robotic cell from automotive industry.

is transported out of the robotic cell. It is obvious that the robots r_1 and r_2 have to be synchronized with the conveyor belt since the execution of welding and gluing operations depends on the position of the automotive body. In a similar way, the conveyor belt is not allowed to move when some welding/gluing operations are in progress. Fortunately, the conveyor belt can be simulated as an additional robot r_3 with its graph G^{r_3}, where each static activity $v \in A_{\mathcal{S}}^{r_3}$ has zero power consumption (the belt is not moving) and one location ($|L_v| = 1$), each dynamic activity $e \in A_{\mathcal{D}}^{r_3} \setminus (v_{12}, v_9)$ has a constant input power (the belt is moving) and its duration is equal to transportation time, and dynamic activity $(v_{12}, v_9) \in A_{\mathcal{D}}^{r_3}$ has zero duration and zero energy consumption. Robots r_1 and $r_2 \in R$ cannot start the work on the front part of the automotive body until the body is transported to the robots from v_9 (i.e., 'in') to v_{10} (i.e., 'pos$_1$'), therefore activities v_2, v_3, v_6 can start after dynamic activity $(v_9, v_{10}) \in A_{\mathcal{D}}^{r_3}$ is finished. It is resulting in 3 time lags: $s_{v_i} \geq s_{v_{10}} + l_{v_{10},v_i}$ $\forall i \in \{2, 3, 6\}$ where the length is an extra time required by robots r_1 and r_2 to safely enter the working space. After the front part of the body is processed, the belt can transport the body from v_{10} (i.e., 'pos$_1$') to v_{11} (i.e., 'pos$_2$'). It means that dynamic activity $(v_{10}, v_{11}) \in A_{\mathcal{D}}^{r_3}$ can start after the last activity $v \in \{v_2, v_3, v_6\}$, therefore the following time lags are added: $s_{e_2} \geq s_{e_1} + l_{e_1,e_2}$ where $e_1 \in \{S_{v_2} \cup S_{v_3} \cup S_{v_6}\} \setminus \{(v_2, v_3), (v_3, v_2)\}$ and $e_2 = (v_{10}, v_{11})$. The lengths of the time lags correspond to an extra time required by robots r_1 and r_2 to safely leave the working space. In a similar way the work on the rear part of the body requires the following time lags: $s_{v_i} \geq s_{v_{11}} + l_{v_{11},v_i}$ $\forall i \in \{4, 5, 7\}$ and $s_{e_2} \geq s_{e_1} + l_{e_1,e_2}$ where $e_1 \in \{(v_4, v_1), (v_5, v_1), (v_7, v_8)\}$ and $e_2 = (v_{11}, v_{12})$. Having the rear part

of the body processed, the body is transported out of the robotic cell (i.e., $v_{12} \sim$ 'out') and a new cycle starts. Note that both the robots and the conveyor belt have the same production cycle time $\overline{C_T}$.

2.4 Mixed-Integer Linear Programming Model

The problem is intrinsically nonlinear due to f_e^t being convex functions; nevertheless, a MILP model can be formulated if the functions are piecewise linearized. The piecewise linear functions $\hat{f}_e^t$ are propagated through constraints (2.3) to the criterion (2.1), where W_a is the energy consumption of activity $a \in A$, $\overline{W}$ is an upper bound on energy consumption, y_e^t is a binary variable that determines whether or not the trajectory $t \in T_e$ is selected for activity $e \in A_{\mathcal{D}}$, B is a set of indices, and $k_{e,b}^t$, $q_{e,b}^t$ are coefficients of the b-th linear function approximating f_e^t. The number of segments $|B|$ of each $\hat{f}_e^t$ can be adjusted to meet the desired accuracy of the approximation. There is no need to introduce binary variables for each segment of the function because functions f_e^t are convex, and the criterion ensures that the right linear function is active. The energy consumption of static activities is calculated by constraints (2.2), where x_v^l determines whether or not location $l \in L_v$ is selected for $v \in A_{\mathcal{S}}$, and z_v^m is set to true if and only if mode $m \in M^r$ is applied for activity $v \in A_{\mathcal{S}}$.

$$\text{Min} \sum_{\forall a \in A} W_a \tag{2.1}$$

$$s.t. \quad p_{v,l}^m d_v - \overline{W}\left(2 - z_v^m - x_v^l\right) \leq W_v \tag{2.2}$$
$$\forall r \in R, \forall v \in A_{\mathcal{S}}^r, \forall l \in L_v, \forall m \in M^r$$

$$k_{e,b}^t d_e + q_{e,b}^t - \overline{W}\left(1 - y_e^t\right) \leq W_e \tag{2.3}$$
$$\forall e \in A_{\mathcal{D}}, \forall t \in T_e, \forall b \in B$$

Assignment constraints (2.4), (2.5), and (2.6) ensure the correct selection of locations, power saving modes, and trajectories for activities. Note that optional activities $A_{\mathcal{O}}$ are omitted in constraints (2.6) because they may or may not be carried out. If an optional activity $e \in A_{\mathcal{O}}$ is not performed, then none of its trajectories is selected. Therefore, W_e is zero because all the constraints (2.3) for activity e are deactivated, and criterion (2.1) pushes W_e down to zero due to the minimization.

$$\sum_{\forall l \in L_v} x_v^l = 1 \qquad \forall v \in A_{\mathcal{S}} \tag{2.4}$$

$$\sum_{\forall m \in M^r} z_v^m = 1 \qquad \forall r \in R, \forall v \in A_{\mathcal{S}}^r \tag{2.5}$$

$$\sum_{\forall t \in T_e} y_e^t = 1 \qquad \forall e \in A_{\mathcal{D}} \setminus A_{\mathcal{O}} \tag{2.6}$$

Flow constraints (2.7) and (2.8) state that if the robot moves to location l then it also has to move away from the same location.

$$\sum_{\forall e \in P_v} \sum_{\forall t=(l_{\text{from}},l) \in T_e} y_e^t = x_v^l \qquad \forall v \in A_{\mathcal{S}}, \forall l \in L_v \tag{2.7}$$

$$\sum_{\forall e \in S_v} \sum_{\forall t=(l,l_{\text{to}}) \in T_e} y_e^t = x_v^l \qquad \forall v \in A_{\mathcal{S}}, \forall l \in L_v \tag{2.8}$$

Constraints (2.9) to (2.13) enforce the time ordering of activities. Some precedences of activities are mandatory (see constraints (2.9) and (2.10)) due to the structure of graph $G^r = (A_{\mathcal{S}}^r, A_{\mathcal{D}}^r)$, whereas others are optional (see constraints (2.11) to (2.13)) because some $e \in A_{\mathcal{O}}$ do not have to be carried out. The binary variables $w_{e,v}$ determine whether or not activity $e \in A_{\mathcal{O}}$ is performed and which order of operations is selected.

$$s_{a_2} = s_{a_1} + d_{a_1} \qquad \forall a_1 \in A \setminus A_{\mathcal{O}}, \forall a_2 \in S_{a_1}, \nexists v_h^r = a_1 \tag{2.9}$$

$$s_e = s_{v_h^r} + d_{v_h^r} - \overline{C_T} \qquad \forall v_h^r \in A_{\mathcal{S}}, \forall e \in S_{v_h^r} \tag{2.10}$$

$$\sum_{\forall t \in T_e} y_e^t = w_{e,v} \qquad \forall e \in A_{\mathcal{O}}, v \in S_e \tag{2.11}$$

$$s_v + (1 - w_{e,v}) \overline{C_T} \geq s_e + d_e \qquad \forall e \in A_{\mathcal{O}}, v \in S_e \tag{2.12}$$

$$s_v - (1 - w_{e,v}) \overline{C_T} \leq s_e + d_e \qquad \forall e \in A_{\mathcal{O}}, v \in S_e \tag{2.13}$$

Restrictions on the duration of static and dynamic activities are reflected in constraints (2.14) and (2.15), respectively.

$$\max(\underline{d}_v, \underline{d}^m) z_v^m \leq d_v \leq \overline{d}_v \qquad \forall r \in R, \forall v \in A_{\mathcal{S}}^r, \forall m \in M^r \tag{2.14}$$

$$\underline{d}_e^t y_e^t \leq d_e \leq \overline{d}_e^t + \overline{C_T} \left(1 - y_e^t\right) \qquad \forall e \in A_{\mathcal{D}}, \forall t \in T_e \tag{2.15}$$

The problem described by Equations (2.1) to (2.15) is robot independent; i.e., the constraint matrix of the MILP formulation is block diagonal; thus, each robot can be treated separately. In the following constraints, however,

the robots are coupled together with regard to time synchronization and spatial compatibility.

$$s_{a_2} - s_{a_1} \geq l_{a_1,a_2} - \overline{C_T} h_{a_1,a_2} \qquad \forall (a_1, a_2) \in E_{\mathcal{TL}} \tag{2.16}$$

$$x_{v_1}^{l_1} \leq \sum_{\forall l_2 \in L_{v_2}, l_2 \in Q_{l_1}} x_{v_2}^{l_2} \tag{2.17}$$
$$\forall v_1 \in A_{\mathcal{S}}, \forall l_1 \in L_{v_1}, |Q_{l_1}| > 0$$

Collisions between robots are resolved by constraints (2.18) and (2.19) where $u_{a_i}^{g_i}$ is either $x_{a_i}^{g_i}$ or $y_{a_i}^{g_i}$ depending on whether $a_i \in A_{\mathcal{S}}$ or $a_i \in A_{\mathcal{D}}$, and the binary variables c_o^n determine the execution order of a_i and a_j for the multiples of cycle time $\overline{C_T}$ if the collision applies. As an example, consider $x_{a_1}^{g_1}$ and $y_{a_2}^{g_2}$ to be a colliding pair for a multiple of cycle time $n = 2$. If location g_1 and movement g_2 are selected, i.e., if the variables $u_{a_1}^{g_1}$ and $u_{a_2}^{g_2}$ are equal to one, then exactly one of constraints (2.18) and (2.19) is active for this collision pair and multiple $n = 2$; therefore, either $s_{a_2} + 2\overline{C_T} \geq s_{a_1} + d_{a_1}$ or $s_{a_1} \geq s_{a_2} + d_{a_2} + 2\overline{C_T}$ has to be satisfied. In case the collision does not apply, i.e., if location g_1 or movement g_2 is not selected, then these two equations remain inactive.

$$s_{a_2} + n\overline{C_T} + 2\,|R|\,\overline{C_T}(3 - c_o^n - u_{a_1}^{g_1} - u_{a_2}^{g_2}) \geq s_{a_1} + d_{a_1}$$
$$\forall o = (a_1, g_1, a_2, g_2) \in K, \forall n \in \{-|R|, \ldots, |R|\} \tag{2.18}$$
$$s_{a_1} + 2\,|R|\,\overline{C_T}(2 + c_o^n - u_{a_1}^{g_1} - u_{a_2}^{g_2}) \geq s_{a_2} + d_{a_2} + n\overline{C_T}$$
$$\forall o = (a_1, g_1, a_2, g_2) \in K, \forall n \in \{-|R|, \ldots, |R|\} \tag{2.19}$$

All the variables of the model are either nonnegative floats or binary variables, as expressed in Equation (2.20).

$$W_a, s_a, d_a \in \mathbb{R}_{\geq 0} \qquad x_v^l, z_v^m, y_e^t, c_o^n, w_{e,v} \in \mathbb{B} \tag{2.20}$$

2.5 Parallel Heuristic Algorithm

Motivated by the inability of MILP solvers to solve bigger instances, this work proposes a parallel Hybrid Heuristic based on a Linear Programming (LP) solver that iteratively solves partially fixed problems, called *tuples*, for selected locations, power-saving modes, and alternative orders. Although the application of an LP solver to partially fixed problems has been previously studied (see, e.g., [32, 26], and [20]), the present work accelerates the heuristic by using a tailor-made Gurobi simplex algorithm for piecewise linear convex functions and the parallelization.

The flowchart of the heuristic, as shown in Figure 2.4, can be divided into two parts: the *control thread* and *worker threads.* The control thread

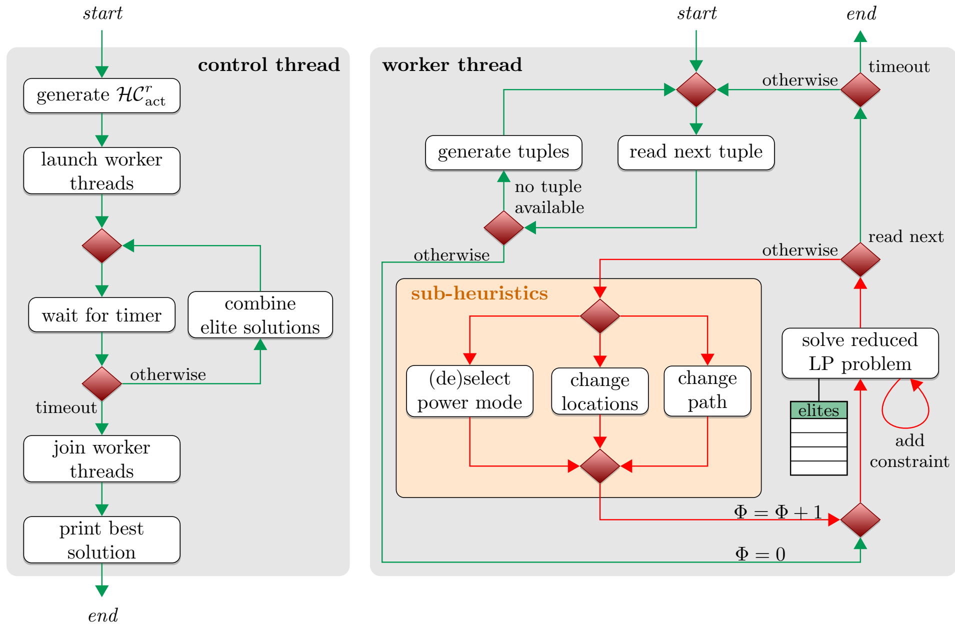


Figure 2.4: Flowchart of the parallel Hybrid Heuristic.

generates various alternative orders in the form of $\mathcal{HC}^r_{\text{act}}$, launches worker threads, and occasionally combines elite solutions to create promising tuples. After the specified time limit t_{max}, the control thread joins the worker threads and prints the best solution if found. The process flow of the worker thread consists of reading and generating tuples and then optimizing them iteratively by alternating between solving a reduced LP problem and carrying out three sub-heuristics. In general, each sub-heuristic performs a small modification of the tuple that may reduce the energy consumption, and an LP solver evaluates a real impact of this modification. The iterative optimization is stopped if there is no significant improvement after Φ_{min} iterations, in which case the next tuple is read, or if the time limit t_{max} is exceeded, in which case the thread is terminated.

The heuristic (see Figure 2.4) is accelerated by using multiple worker threads that independently process tuples one by one from a list. If the list becomes empty, then new tuples are added to it, and the process continues. Because the tuples, alternative orders $\mathcal{HC}^r_{\text{act}}$, and elite solutions are accessible from all the threads, it is necessary to use synchronization primitives to guarantee consistent reads and writes. Fortunately, the introduced overhead is negligible (see the experiments in Section 2.12) because the access time is very short compared with the time needed to solve the reduced LP problem. The key parts of the heuristic are detailed in the following subsections.

2.5.1 Generation of Alternatives

The order of operations of robot r, i.e., an alternative, is encoded as a Hamiltonian circuit $\mathcal{HC}^r_{\text{act}}$ in graph $G^r = (A^r_{\mathcal{S}}, A^r_{\mathcal{D}})$. Because there can be more alternative orders of operations (i.e., circuits), it is necessary to find some of them. For this purpose, the finding of Hamiltonian circuits in graph $G^r = (A^r_{\mathcal{S}}, A^r_{\mathcal{D}})$ is transformed into a search for Hamiltonian paths in graph $G^{r\prime} = (A^r_{\mathcal{S}}{}', A^r_{\mathcal{D}}{}')$ as follows. Nodes $A^r_{\mathcal{S}}{}' = A^r_{\mathcal{S}} \setminus v^r_h \cup \{v_{\rightarrow}, v_{\leftarrow}\}$, where $v_{\rightarrow}, v_{\leftarrow}$ are the starting and ending nodes, respectively. The edges leaving v^r_h are modified such that they start from $v_{\rightarrow}$; similarly the edges entering v^r_h are changed such that they end in $v_{\leftarrow}$. Both the nodes and edges of $G^{r\prime}$ are weighted by the minimum possible duration $\underline{d}_a$ of the related activity; i.e., $\min_{\forall t \in T_e} \underline{d}^t_e$ for $\forall e \in A^r_{\mathcal{D}}{}'$ and $\max(\underline{d}_v, \min_{\forall m \in M^r} \underline{d}^m)$ for $\forall v \in A^r_{\mathcal{S}}{}'$. The goal is then to find random Hamiltonian paths from $v_{\rightarrow}$ to $v_{\leftarrow}$ such that $\sum_{\forall a \in \text{path}} \underline{d}_a \leq \overline{C_T}$. To ensure effective pruning during an iterative random tree search from node $v_{\rightarrow}$, the obviously infeasible or completely searched sub-paths from $v_{\rightarrow}$ are detected and skipped. The sub-path $(v_{\rightarrow}, \ldots, a_i)$ is infeasible if there is an unvisited node a_u that cannot be reached or $|(v_{\rightarrow}, \ldots, a_i)| + C(a_i, a_u) + C(a_u, v_{\leftarrow}) > \overline{C_T}$, where the first term is the sub-path length, and $C(a_i, a_j)$ is the length of the shortest path from a_i to a_j calculated by the Floyd-Warshall algorithm. If a Hamiltonian path is found, its fastest sequence through the activity locations can be obtained by applying a shortest path algorithm for directed acyclic graphs. If the duration of this sequence is longer than the robot cycle time, then the related Hamiltonian path is neglected because it cannot lead to a feasible solution for the original problem. Note that the Hamiltonian path and its fastest sequence through the locations can be easily transformed back to $\mathcal{HC}^r_{\text{act}}$ and $\mathcal{HC}^r_{\text{loc}}$, respectively. Because the algorithm is exact, i.e., it can prove the nonexistence of a Hamiltonian circuit of a given length, it is possible to detect the infeasibility of some instances. Moreover, the generation of alternatives is robot independent; therefore, the work is distributed among threads to accelerate the initialization of the heuristic.

2.5.2 Generation of Tuples

The generation of tuples, which are formally defined below, is crucial for finding initial feasible solutions; therefore, the emphasis is placed on feasibility rather than on energy optimality during the generation. At the beginning of the generation, a random alternative $\mathcal{HC}^r_{\text{act}}$ and its fastest closed path $\mathcal{HC}^r_{\text{loc}}$ through the locations (see Section 2.5.1) are assigned to each robot r. Only the fastest power-saving mode of robot r is considered. The closed paths are subsequently modified to meet the spatial compatibility as

follows. For each pair (v_i, v_j) with the violated spatial compatibility one fixing pair $q \in Q_{v_i,v_j}$ is selected with respect to the prolongation of the related closed paths; i.e., the paths with a minimal duration close to the robot cycle time are penalized. After the spatial compatibility is fixed, the tuple is created as a triple $\mathcal{T} = (\mathscr{A}, \mathscr{P}, \alpha : A_{\mathcal{S}} \rightarrow M)$, where $\mathscr{A}, \mathscr{P}$ are the selected alternatives and their closed paths, respectively, and function α maps each activity $v \in A_{\mathcal{S}}$ to its power mode $m \in M$, where $M = \bigcup_{\forall r \in R} M^r$. The process of tuple generation is applicable to two blocks shown in Figure 2.4: *generate tuples* and *combine elite solutions*. However, in the second block, only the alternatives $\mathcal{HC}^r_{\text{act}}$ used in elite solutions are considered to intensify the search process.

2.5.3 Reduced Linear Programming Problem

The timing of a partial problem, i.e. a tuple, is determined by the reduced LP problem. If the resulting solution is feasible, then it is feasible for the original problem and can be added to the list of elite solutions if it ranks among the top solutions in terms of energy consumption. Otherwise, the solution is infeasible, and one of two cases arises: the related tuple is not modified by any sub-heuristics, in which case another tuple is read; or the previous tuple resulting in a feasible solution is loaded, and the next sub-heuristic is selected.

Before formally describing the reduced LP problem, it is necessary to define some sets extracted from tuple $\mathcal{T}$. Set $A_{\mathcal{D}}(\mathcal{T})$ contains all the dynamic activities in $\forall \mathcal{HC}^r_{\text{act}} \in \mathscr{A}$, where each dynamic activity e has to be carried out (is selected by $\mathcal{T}$) and has assigned its fixed trajectory $t \in T_e$ as a pair $(e, t) \in \mathcal{F}_1(\mathcal{T})$; similarly, each static activity $v \in A_{\mathcal{S}}$ is linked to its location $l \in L_v$ and power mode $m \in M$ as a triple $(v, l, m) \in \mathcal{F}_2(\mathcal{T})$. Then, the reduced LP problem can be stated as follows.

$$\text{Min} \sum_{\forall (e,t) \in \mathcal{F}_1(\mathcal{T})} \hat{f}^t_e(d_e) + \sum_{\forall (v,l,m) \in \mathcal{F}_2(\mathcal{T})} p^m_{v,l} d_v \tag{2.21}$$

subject to: (2.9)*, (2.10)*, (2.14)*, (2.15)*, (2.16)*

$$s_{a_2} + n\overline{C_T} \geq s_{a_1} + d_{a_1} \qquad \forall (a_1, a_2, n) \in \mathcal{D}_{\geq} \tag{2.22}$$

$$s_{a_2} + d_{a_2} + n\overline{C_T} \leq s_{a_1} \qquad \forall (a_1, a_2, n) \in \mathcal{D}_{\leq} \tag{2.23}$$

Criterion (2.21) minimizes the sum of the piecewise linear convex functions $\hat{f}^t_e$ passed to the Gurobi LP solver as a list of function points or alternatively expressed similarly to constraints (2.2) and (2.3) for other solvers. The equations marked with an asterisk are the same as the original ones (without the asterisk); however the sets from which the constraints are generated are different. Constraints (2.9)*, (2.10)*, and (2.16)* establish

precedences between activities $A(\mathcal{T}) = A_{\mathcal{S}} \cup A_{\mathcal{D}}(\mathcal{T})$, and constraints (2.14)* and (2.15)* transform into the domain specification of d_a variables.

All the aforementioned constraints are always present in the above formulation; however, these alone may not ensure feasibility because some collisions could occur. For this reason, additional constraints (2.22) and/or (2.23) could be iteratively added to heuristically resolve active collisions. Thus, if a solution is feasible with respect to the current constraints of the reduced LP problem but is not feasible for the original problem due to some active collisions, then the worst collision (formally defined later) is detected and resolved by adding constraint (2.22) or (2.23). Afterward, the problem is resolved, and the same procedure is repeated if collisions still occur (see Figure 2.4).

To introduce the collision resolution more formally, constraints (2.22) and (2.23) have to be shown as specialized forms of constraints (2.18) and (2.19), respectively. First, the variables $u_{a_i}^{g_i}$ in constraints (2.18) and (2.19) are fixed for tuple $\mathcal{T}$, i.e., $u_{a_i}^{g_i} = 1$ if $(a_i, g_i) \in \mathcal{F}_1(\mathcal{T})$ or $(a_i, g_i, m_i) \in \mathcal{F}_2(\mathcal{T})$; otherwise, $u_{a_i}^{g_i} = 0$. Second, it is sufficient to consider set $K(\mathcal{T}) = \{\forall(a_i, a_j) : u_{a_i}^{g_i} + u_{a_j}^{g_j} = 2, (a_i, g_i, a_j, g_j) \in K, a_i, a_j \in A(\mathcal{T})\}$ instead of K because constraints (2.18) and (2.19) are neither binding nor violated for unselected locations and movements of tuple $\mathcal{T}$, and each activity $a_i \in A(\mathcal{T})$ has an assigned movement or location. Note that if $u_{a_i}^{g_i} + u_{a_j}^{g_j} = 2$, then the variable c_o^n makes only one constraint, either (2.18) or (2.19), active ("big M method") depending on the decision whether $s_{a_2} + n\overline{C_T} \geq s_{a_1} + d_{a_1}$ or $s_{a_1} \geq s_{a_2} + d_{a_2} + n\overline{C_T}$, which guarantees collision-free ordering. However, these decisions correspond exactly to constraints (2.22) and (2.23), in which the decision on ordering is given by adding a triple (a_i, a_j, n) to the related set $\mathcal{D}_{\geq}$ or $\mathcal{D}_{\leq}$, respectively; i.e., a constraint is added to the problem. The constraint added is not removed during the solving of the reduced LP problem; therefore, this approach is heuristic. The key question is how to select which constraint to add and how such constraint is related to the worst collision. To find an answer, the maximal violation Γ of constraints (2.18) and (2.19) for tuple $\mathcal{T}$ is calculated as follows.

$$\upsilon_{a_i,a_j}^n = s_{a_i} + d_{a_i} - s_{a_j} - n\overline{C_T} \tag{2.24}$$

$$\mu_{a_i,a_j}^n = s_{a_j} + d_{a_j} + n\overline{C_T} - s_{a_i} \tag{2.25}$$

$$\Gamma = \max_{\substack{\forall(a_i,a_j)\in K(\mathcal{T}) \\ \forall n\in\{-|R|,\cdots,|R|\}}} \min(\upsilon_{a_i,a_j}^n, \mu_{a_i,a_j}^n) \tag{2.26}$$

Note that if $\Gamma \leq 0$, then there are no active collisions, and no extra constraints are needed; failing that, i.e., $\Gamma > 0$, means that there is a pair of colliding activities that is not time disjunctive. In that case, let (a_i^*, a_j^*, n^*) be an optimal argument of the max function in Equation (2.26) (replace

max with argmax). This triple defines the worst collision, i.e., the collision with the biggest time intersection, occurring in a current solution. Whether this collision should be resolved by adding a constraint of type (2.22) or (2.23) is determined by the $\upsilon^* = \upsilon^{n^*}_{a_i^*, a_j^*}$ and $\mu^* = \mu^{n^*}_{a_i^*, a_j^*}$ values. If $\upsilon^* \leq \mu^*$ ($\mu^* \leq \upsilon^*$), then the worst collision is resolved by adding the triple to $\mathcal{D}_{\geq}$ ($\mathcal{D}_{\leq}$) because it may result in smaller changes in timing after the problem is resolved with the added constraints.

2.5.4 Sub-heuristics

The aim of the sub-heuristics is to modify a given tuple $\mathcal{T}$ and its timing calculated by the reduced LP problem such that the energy consumption would be reduced in successive LP calls. Because the performed modifications can result in a violation of time lags or the occurrence of collisions, a penalty based on the duration of breakage and average input power is added to these modifications. If the modifications of an active sub-heuristic do not lead to significant energy improvements, then the next sub-heuristic is selected in round-robin order.

The goal of the *(de)select power mode* sub-heuristic, which focuses on the application of the power-saving modes of robots, is to find and apply an alternative power-saving mode for some activity $v \in A_{\mathcal{S}}$. To select a suitable mode and activity, the energy consumption is estimated for all applicable modes of each $v \in A_{\mathcal{S}}$ as follows. If a different mode $m \in M^r$ of robot $r \in R$ is applied for activity v', then some activities $v \in A^r_{\mathcal{S}} \setminus v'$ are uniformly shortened/prolonged to meet the cycle time. After the timing is modified, both the energy consumption and the above-mentioned penalty are determined. Finally, the choice falls on mode m and activity v' with the lowest sum of the energy consumption and penalty. Note that the sub-heuristic uses a Tabu list, i.e., a short-term memory with forbidden modifications, to avoid cycling; therefore, some power-saving modes may not be applicable for some activities.

The *change locations* sub-heuristic optimizes the closed paths of robots, i.e., $\forall \mathcal{HC}^r_{\text{loc}} \in \mathscr{P}$, by modifying the go-through locations as follows. Let '$\dashrightarrow l_a \xrightarrow{t_1} l_b \xrightarrow{t_2} l_c \dashrightarrow$' be a part of $\mathcal{HC}^r_{\text{loc}}$, where t_1 and t_2 are the movements between the locations; then, the question arises as to whether the inner part, i.e., '$\xrightarrow{t_1} l_b \xrightarrow{t_2}$', can be replaced so as to achieve a reduction in energy consumption. To find an answer, each viable substitution '$l_a \xrightarrow{t_{\vdash}} l_{\circ} \xrightarrow{t_{\dashv}} l_c$' has to be evaluated in terms of energy by solving the convex problem (2.27) to (2.28), where $t_1, t_{\vdash} \in T_{e_1}$, $t_2, t_{\dashv} \in T_{e_2}$, $e_1, e_2 \in A_{\mathcal{D}}(\mathcal{T})$, $l_b, l_{\circ} \in L_{v_b}$, $v_b \in A_{\mathcal{S}}$, and $\xi = d^{\text{LP}}_{e_1} + d^{\text{LP}}_{e_2}$ and $d^{\text{LP}}_{v_b}$ are constants determined from the LP solution. Because the problem is convex and one-dimensional, the golden

section search algorithm can be used to find the optimal solution. Then, the most energy-friendly substitution not breaking the spatial compatibility is applied, and the process is repeated for other sub-paths.

$$\text{minimize } f_{e_1}^{t_\vdash}(d_{e_1}) + f_{e_2}^{t_\dashv}(\xi - d_{e_1}) + p_{v_b,l_\circ}^{m} d_{v_b}^{\mathrm{LP}} \tag{2.27}$$

$$\max(\underline{d}_{e_1}^{t_\vdash}, \xi - \overline{d}_{e_2}^{t_\dashv}) \leq d_{e_1} \leq \min(\overline{d}_{e_1}^{t_\vdash}, \xi - \underline{d}_{e_2}^{t_\dashv}) \tag{2.28}$$

The last sub-heuristic, called *change path*, diversify the search process to allow exploring some otherwise unreachable $\mathcal{HC}_{\mathrm{loc}}^{r}$. The sub-heuristic selects one $\mathcal{HC}_{\mathrm{loc}}^{r} \in \mathscr{P}$ and randomly changes its go-through locations such that spatial compatibility is achieved and the resulting closed path exists.

2.6 Parallel Branch & Bound Algorithm Overview

The Branch & Bound algorithm, as depicted in Figure 2.5, adopts the standard procedure but the key difference that makes the algorithm efficient is the *Node evaluator* and *Deep Jumping* approach (described later). Figure 2.5 shows the basic functional blocks and process flow of the algorithm, which are described as follows. At the beginning, an instance of the problem is read, and the global Lower Bound (LB) and best Upper Bound (UB) are initialized. The LB is the best currently known lower estimation of energy, and UB is the energy required by the current best solution (the incumbent solution). Afterwards, the root node of the Branch & Bound tree is created and inserted to a node storage (see Section 2.11 for details), and the searching process is started from the root node to explore the tree for the best quality solutions.

Each node $\boldsymbol{n}$ obtained from the storage is tested whether its lower estimation on energy $\mathrm{LB}(\boldsymbol{n}) < \mathrm{UB}$; or in other words, whether node $\boldsymbol{n}$ might be resulting in a better solution than the incumbent one. If it is not the case, the node is pruned and another node is obtained. In case the storage is empty or time limit t_{max} is exceeded, the algorithm stops the searching process and prints the best solution if found; otherwise, node $\boldsymbol{n}$ is processed based on its type: a *leaf node* or *partial node*. In the leaf node, the order, locations, and power-saving modes are already decided, therefore no further branching is required and the energy consumption is evaluated by solving a MILP problem (see Section 2.9 for details). In case the energy consumption $\mathrm{UB}(\boldsymbol{n}) < \mathrm{UB}$, a new incumbent solution is found and $\mathrm{UB} = \mathrm{UB}(\boldsymbol{n})$. Compared to the leaf node, the partial node is not fully decided yet, therefore new nodes are generated by branching on order, locations, or power-saving modes (see Section 2.8 for details).

As it is obvious from Figure 2.5, there are two approaches how to generate new nodes. The first one, i.e., conventional branching, expands one

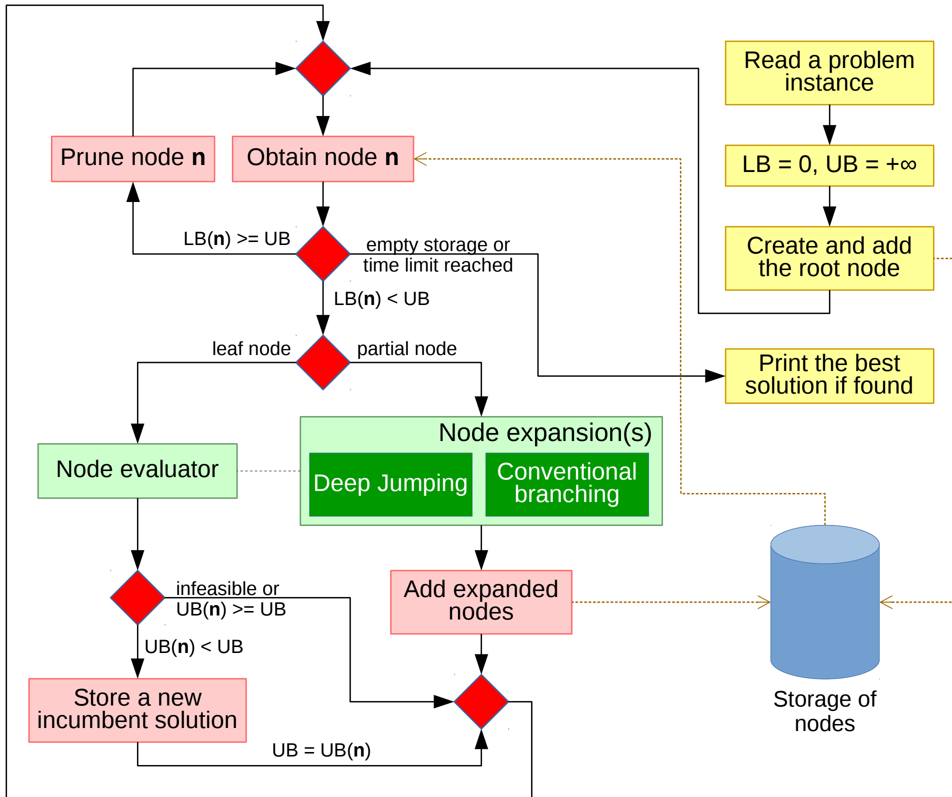


Figure 2.5: Block diagram of the Branch & Bound algorithm.

node and evaluates the generated nodes by using a *tight relaxation* based on convex envelopes (see Section 2.9). The second one, i.e., *Deep Jumping*, can be used if the order is not fully decided, and finding a feasible order of activities is difficult due to the global constraints, especially time lags. The main idea is to employ a heuristic to find a promising immersion in the tree, which could be resulting in good solutions in subsequent expansions. It is accomplished by solving a relaxed MILP problem with a time limit and relative optimality gap, where the cycle time is minimized to make a solution robust to time prolongations (see Section 2.10 for details). From a feasible solution to the relaxed problem, an order of activities is determined and used to create a subtree, where non-expanded nodes (not branched) are evaluated using the tight relaxation (see Section 2.9). If there is a feasible subtree node with the same order of activities as in the feasible solution of the relaxed MILP problem, then this node is preferred for the next expansion (higher priority). The procedure is named Deep Jumping since it transfers the searching process to higher levels of the tree. Regardless of how the nodes are generated, non-expanded nodes are added to the storage.

After the node obtained from the storage is processed, the searching process continues with another node in the same way.

The following sections provide a mathematical background and details related to the basic blocks of the algorithm. To be more particular, the node is defined in the next section, the branching and propagations are described in Section 2.8, the energy evaluator of nodes is presented in Section 2.9, the Deep Jumping approach is proposed in Section 2.10, and the acceleration of the algorithm by using multiple threads and efficient data structures is discussed in Section 2.11.

2.7 Node Definition

The node of the Branch & Bound tree can be defined as a triple

$$\boldsymbol{n} = (\mathcal{P}, \mathcal{L}, \mathcal{M}) \tag{2.29}$$

where $\mathcal{P}$ is a set of subpaths in graphs G with the decided order of activities (see, e.g., Figure 2.6b), $\mathcal{L}$ is a mapping from $v \in A_{\mathcal{S}}$ to selectable locations $L_v(\boldsymbol{n})$, and $\mathcal{M}$ maps each $v \in A^r_{\mathcal{S}}$ to its applicable power-saving modes $M^r_v(\boldsymbol{n})$. The definitions of $\mathcal{L}$ and $\mathcal{M}$ use a new notation, i.e., $L_v(\boldsymbol{n})$ and $M^r_v(\boldsymbol{n})$, to extract only selectable/applicable elements from a set for a given node $\boldsymbol{n}$. For example, $T_e(\boldsymbol{n})$ contains all trajectories $t \in T_e$ that are selected or can be selected in a given (partial) solution determined by node $\boldsymbol{n}$. The node is called the *leaf node* if the order is fixed, i.e., $|\mathcal{P}| = |R|$, and each static activity $v \in A_{\mathcal{S}}$ is assigned respectively unique location $l \in L_v$ and power-saving mode $m \in M^r$ by $\mathcal{L}$ and $\mathcal{M}$ mappings; otherwise, it is a *partial node* that can be branched to generate new nodes.

2.8 Branching

Every partial node can be branched on the order of activities, locations, or power-saving modes. The highest priority is given to fixing an order of activities since it makes the problem difficult with respect to time lags. If the order is fixed, then the spatial compatibility is resolved by selecting locations for the static activities that are involved in inter-robot handovers. Nodes detected to violate the spatial compatibility are pruned by the branching. After the spatial compatibility is resolved, the remaining locations and power-saving modes are assigned. The goal of these empirical rules is to make difficult decisions in lower levels of the tree to increase the likelihood of finding feasible solutions.

The following three subsections describe propagation techniques that are subsequently carried out in expanded nodes to infer new decisions based on

a branching decision (a selected activity/location/mode). The propagation techniques resemble the domain propagation in constraints programming.

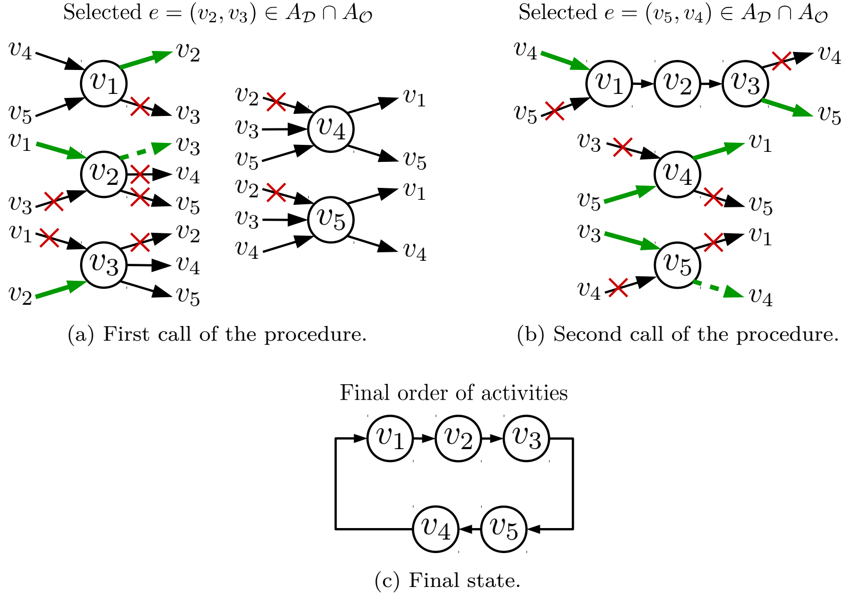
2.8.1 Order Propagation

The primary goal of the order propagation is to determine, based on a branching decision to select $e \in A_{\mathcal{O}}$, which additional optional activities should be selected or removed from the consideration for expanded node $\boldsymbol{n}$. Since the optional activities determine the order of activities, the propagation procedure, in fact, concatenates subpaths $\mathcal{P}$ of node $\boldsymbol{n}$ (see, e.g., Figure 2.6) until no more decisions can be inferred.

Order propagation procedure:

1. Let optional activity $e \in A^r_{\mathcal{O}}$ is selected by a branching decision.
2. Let $e_f = e$, where e_f is the activity to be fixed in the next two steps.
3. Find subpaths $p_1 \in \mathcal{P}^r$ and $p_2 \in \mathcal{P}^r$ of robot r that can be joined by activity e_f, i.e., $e_f \in next(p_1, \boldsymbol{n})$ and $e_f \in prev(p_2, \boldsymbol{n})$, where $next(p_i, \boldsymbol{n})$ and $prev(p_i, \boldsymbol{n})$ are leaving and entering edges ($\subseteq A^r_{\mathcal{O}}$) of subpath p_i for node $\boldsymbol{n}$, respectively.
4. Join subpaths p_1 and p_2 together by activity e_f and create new subpath p_{12}. Set $\mathcal{P}^r = \mathcal{P}^r \setminus \{p_1, p_2\} \cup \{p_{12}\}$, and remove edges $next(p_1, \boldsymbol{n}) \cup prev(p_2, \boldsymbol{n})$ since they cannot be selected after the join.
5. If $\exists p_i \in \mathcal{P}^r, e \in A^r_{\mathcal{O}}$ such that $e \in next(p_i, \boldsymbol{n})$ and $e \in prev(p_i, \boldsymbol{n})$, then remove activity e as it forms a loop on subpath p_i and repeat this step.
6. If $|\mathcal{P}^r| > 1$ and $\exists p_i \in \mathcal{P}^r$ such that $|prev(p_i, \boldsymbol{n})| = 0$ or $|next(p_i, \boldsymbol{n})| = 0$, then Hamiltonian circuit $\mathcal{HC}^r_{\text{act}}$ cannot be closed, i.e., node $\boldsymbol{n}$ is infeasible. Go to step 9.
7. If $\exists p_i \in \mathcal{P}^r$ such that $|prev(p_i, \boldsymbol{n})| = 1$, then infer the decision that activity $e \in prev(p_i, \boldsymbol{n})$ is fixed, i.e., $e_f = e$, and go to step 3.
8. If $\exists p_i \in \mathcal{P}^r$ such that $|next(p_i, \boldsymbol{n})| = 1$, then infer the decision that activity $e \in next(p_i, \boldsymbol{n})$ is fixed, i.e., $e_f = e$, and go to step 3.
9. The propagation is completed and subpaths $\mathcal{P}^r$ are updated if the node is feasible. Notice that if an order of activities of robot r is fixed, then $|\mathcal{P}^r| = 1$ and $|prev(p_i)| = |next(p_i)| = 0$ for $p_i \in \mathcal{P}^r$.

A demonstration of the propagation on robot r_1 from Figure 2.3 is provided in Figure 2.6, where two optional activities are successively selected (branching decisions) to determine the order of activities of robot r_1. After optional activity (v_2, v_3) is selected (dashed in Figure 2.6a), the procedure removes (crossed) and fixes (bold) some edges (see Figure 2.6a) and updates subpaths $\mathcal{P}^{r_1}$ (see Figure 2.6b). Subsequently, another activity (v_5, v_4) is



(a) First call of the procedure.

(b) Second call of the procedure.

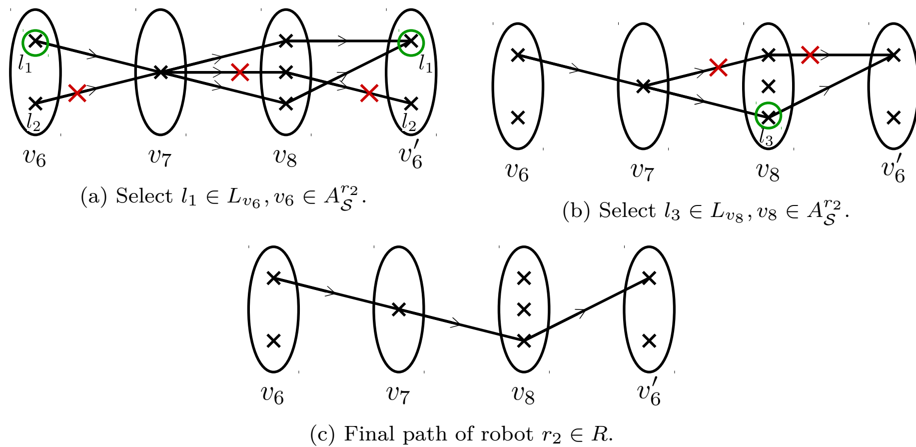
(c) Final state.

Figure 2.6: Example of the order propagation (two activities are selected).

selected and the process is repeated. In the end, the final order of activities (see Figure 2.6c) is used to construct $\mathcal{HC}^{r_1}_{\text{act}}$.

2.8.2 Propagation of Locations

The propagation of locations is executed after a location is selected by a branching decision. It removes trajectories that become unselectable due to the decision and updates $L_v(\boldsymbol{n})$ sets for $\forall v \in A^r_{\mathcal{S}}$ of robot r.



(a) Select $l_1 \in L_{v_6}, v_6 \in A^{r_2}_{\mathcal{S}}$.

(b) Select $l_3 \in L_{v_8}, v_8 \in A^{r_2}_{\mathcal{S}}$.

(c) Final path of robot $r_2 \in R$.

Figure 2.7: Example of the propagation after selecting locations.

Propagation of locations procedure:

1. Location $l_f \in L_v(\boldsymbol{n})$ is selected by a branching decision.
2. Let $prev(l_i, \boldsymbol{n})$ and $next(l_i, \boldsymbol{n})$ are entering and leaving trajectories of location l_i for node $\boldsymbol{n}$, respectively. Remove all trajectories $t \in prev(l_i, \boldsymbol{n}) \cup next(l_i, \boldsymbol{n})$ such that $l_i, l_f \in L_v(\boldsymbol{n})$ and $l_i \neq l_f$.
3. If $\exists l_i \in L_v, v \in A^r_{\mathcal{S}}$ such that $|prev(l_i, \boldsymbol{n})| = 0$ and $|next(l_i, \boldsymbol{n})| > 0$, then remove all trajectories in $next(l_i, \boldsymbol{n})$ and repeat this step.
4. If $\exists l_i \in L_v, v \in A^r_{\mathcal{S}}$ such that $|prev(l_i, \boldsymbol{n})| > 0$ and $|next(l_i, \boldsymbol{n})| = 0$, then remove all trajectories in $prev(l_i, \boldsymbol{n})$ and go to 3.
5. The propagation of locations finished, update $\mathcal{L}$ according to the remaining trajectories and check the feasibility by verifying the existence of $\mathcal{HC}^r_{\text{loc}}$ (see feasibility checks in Section 2.8.4).

The propagation is illustrated in Figure 2.7, where a path of robot r_2 from Figure 2.3 is determined by selecting two locations. The first decision (see Figure 2.7a) is to select location $l_1 \in L_{v_6}$ in activities $v_6 \in A^{r_2}_{\mathcal{S}}$ and v'_6, where v'_6 is a ghost of activity v_6 used to unwrap the cycle. As the other locations of activity v_6 cannot be selected, the propagation can remove the crossed trajectories. Finally, after the second decision (see Figure 2.7b) to select location $l_3 \in L_{v_8}$ is applied, the propagation stops with a fully determined robotic path as shown in Figure 2.7c.

2.8.3 Propagation of Power Saving Modes

After the order of activities and locations are determined, it remains to select power-saving modes. The propagation of power-saving modes updates $M^r_v(\boldsymbol{n})$ sets based on a newly applied power-saving mode. Let say that static activity $v' \in A^r_{\mathcal{S}}$ is branched on its power-saving modes, then each applicable mode $m \in M^r_{v'}(\boldsymbol{n})$ has to satisfy the following equation

$$\underline{d}^m \leq \overline{d}_{v'} \leq \overline{C_T} - \sum_{\substack{\forall v \in \mathcal{HC}^r_{\text{act}} \setminus v' \\ v \in A^r_{\mathcal{S}}}} \max(\underline{d}_v, \min_{\forall m \in M^r_v(\boldsymbol{n})} \underline{d}^m) - \sum_{\substack{\forall e \in \mathcal{HC}^r_{\text{act}} \\ e \in A^r_{\mathcal{D}}}} \min_{\forall t \in T_e(\boldsymbol{n})} \underline{d}^t_e, \quad (2.30)$$

where the right part bounds the maximal possible duration of activity v'. If a decision to apply mode $m \in M^r_{v'}(\boldsymbol{n})$ is taken, i.e., $|M^r_{v'}(\boldsymbol{n})| = 1$, then the propagation can be accomplished by using Equation (2.30) as follows. For each activity $v'' \in A^r_{\mathcal{S}} \setminus v'$ such that $|M^r_{v''}(\boldsymbol{n})| > 1$ filter out all the modes $m \in M^r$ that satisfy $\underline{d}^m > \overline{d}_{v''}$, where $\overline{d}_{v''}$ is calculated as in Equation (2.30) with v' replaced by v''. Modify $\mathcal{M}$ mapping according to the branching decision and updated sets. Note that the propagation is usually stronger if a deeper power-saving mode is applied since it reduces the maximal durations of other activities through the constant cycle time, and as a result, deeper power-saving modes may become inapplicable.

2.8.4 Fast Feasibility Checks

Since the evaluation of generated nodes is time-consuming (see Section 2.9 for details), an additional check is carried out to verify that cycle time $\overline{C_T}$ can be satisfied. In case the order of activities is not fully decided, the following constraint has to be held for each robot $r \in R$.

$$\overline{C_T} \geq \sum_{\forall p \in \mathcal{P}^r} \sum_{\forall a \in p} \underline{d}_a + \max\Big(\sum_{\forall p \in \mathcal{P}^r} \min_{\forall e \in prev(p, \boldsymbol{n})} \underline{d}_e, \sum_{\forall p \in \mathcal{P}^r} \min_{\forall e \in next(p, \boldsymbol{n})} \underline{d}_e \Big) \quad (2.31)$$

It states that the minimal length of subpaths $\mathcal{P}^r$ in terms of duration plus the minimal duration of entering/leaving optional activities (one for each subpath) cannot exceed the cycle time. After the order of activities is decided, a tighter lower bound on cycle time $\underline{C_T^r}$ of robot r can be obtained as a length of a shortest path in an acyclic location-level graph of robot r (see, e.g., Figure 2.7), where each edge $t \in T_e(\boldsymbol{n})$ of activity e is weighted by $\underline{d}_e^t + \max(\underline{d}_v, \min_{\forall m \in M_v^r(\boldsymbol{n})} \underline{d}^m)$ where $v \in P_e$. To be more particular, for each location $l \in L_v(\boldsymbol{n})$, the shortest path from $l \in L_v(\boldsymbol{n})$ to $l \in L_{v'}(\boldsymbol{n})$ with length $\underline{C_T^{r,l}}$ is calculated (infinity if not exists), where v is the first activity of subpath $p \in \mathcal{P}^r$ and v' is its ghost in a similar way as in Figure 2.7. The minimal length of a shortest path corresponds to the lower bound, i.e., $\underline{C_T^r} = \min_{\forall l \in L_v(\boldsymbol{n})} \underline{C_T^{r,l}}$. Obviously, if $\max_{\forall r \in R} \underline{C_T^r} > \overline{C_T}$, then node $\boldsymbol{n}$ is infeasible and can be discarded.

2.9 Energy Evaluator of Nodes

To formulate the tight lower bound on energy consumption, this section introduces composite activities, extra activity sets, and convex envelopes. Most of used symbols are listed in **nomenclature** in Appendix A.

2.9.1 Composite Activities

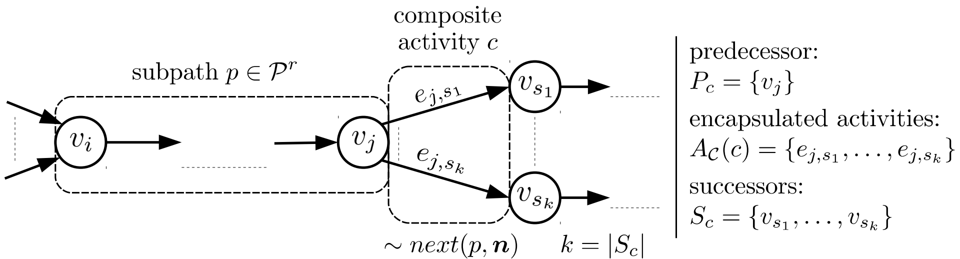


Figure 2.8: Illustration of the composite activity and its related sets.

In order to deal with an undecided order of activities, a new type of activity, i.e., a *composite activity*, is introduced. The composite activity encapsulates leaving edges $next(p, \boldsymbol{n})$ of subpath $p \in \mathcal{P}^r$ (see, e.g., Figure 2.8), from which anyone but one edge $e \in A_{\mathcal{O}}$ can be selected in any solution derived from $\boldsymbol{n}$. In other words, each subpath $p \in \mathcal{P}^r$ such that $|next(p, \boldsymbol{n})| > 1$ is used to create composite activity $c \in A^r_{\mathcal{C}}(\boldsymbol{n})$, where $A^r_{\mathcal{C}}(\boldsymbol{n})$ is a set of composite activities of robot r for a partial solution given by node $\boldsymbol{n}$. Notice that if an order is fully decided, then no composite activities are needed, and set $A_{\mathcal{C}}(\boldsymbol{n}) = \bigcup_{\forall r \in R} A^r_{\mathcal{C}}(\boldsymbol{n})$ is empty.

Let $A_{\mathcal{D}_u}(c)$ is a set of edges (activities) encapsulated by activity $c \in A_{\mathcal{C}}(\boldsymbol{n})$, then duration limits of activity $c \in A_{\mathcal{C}}(\boldsymbol{n})$ are determined as $\underline{d}_c = \min_{\forall e \in A_{\mathcal{D}_u}(c)} \underline{d}_e$ and $\overline{d}_c = \max_{\forall e \in A_{\mathcal{D}_u}(c)} \overline{d}_e$, respectively. The composite activity has a unique predecessor determined as a shared predecessor of all the encapsulated edges, i.e., $P_c = \bigcup_{\forall e \in A_{\mathcal{D}_u}(c)} P_e$. In a similar way, successors of the composite activity (at least two) are the successors of the encapsulated edges, i.e., $S_c = \bigcup_{\forall e \in A_{\mathcal{D}_u}(c)} S_e$.

2.9.2 Extra Activity Sets

Extra activity sets, used in the MILP formulation of the node evaluator (see Section 2.9.4) to calculate the lower bound, are defined in this subsection. Let divide the dynamic activities of robot r that are/can be selected in node $\boldsymbol{n}$ into sets $A^r_{\mathcal{D}_f}(\boldsymbol{n})/A^r_{\mathcal{D}_u}(\boldsymbol{n})$, respectively, where subsubscript f/u means fixed/undecided. Fixed activities are selected in every feasible solution resulting from node $\boldsymbol{n}$, compared to undecided activities from which only a subset is selected. Let $A_{\mathcal{D}_f}(\boldsymbol{n}) = \bigcup_{\forall r \in R} A^r_{\mathcal{D}_f}(\boldsymbol{n})$ and $A_{\mathcal{D}_u}(\boldsymbol{n}) = \bigcup_{\forall r \in R} A^r_{\mathcal{D}_u}(\boldsymbol{n})$, then $A_{\mathcal{D}}(\boldsymbol{n}) = A_{\mathcal{D}_f}(\boldsymbol{n}) \cup A_{\mathcal{D}_u}(\boldsymbol{n}) \subseteq A_{\mathcal{D}}$ and $A_{\mathcal{D}_u}(\boldsymbol{n}) \subseteq A_{\mathcal{O}}$. Since every optional activity $e \in A_{\mathcal{D}_u}(\boldsymbol{n})$ (i.e., edge) is in a composite activity, then $A_{\mathcal{D}_u}(\boldsymbol{n}) = \bigcup_{\forall c \in A_{\mathcal{C}}(\boldsymbol{n})} A_{\mathcal{D}_u}(c)$.

$$A^r_1(\boldsymbol{n}) = A^r_{\mathcal{S}} \cup A^r_{\mathcal{D}_f}(\boldsymbol{n}) \cup A^r_{\mathcal{D}_u}(\boldsymbol{n}) \tag{2.32}$$

$$A_1(\boldsymbol{n}) = \bigcup_{\forall r \in R} A^r_1(\boldsymbol{n}) \tag{2.33}$$

$$A^r_2(\boldsymbol{n}) = A^r_{\mathcal{S}} \cup A^r_{\mathcal{D}_f}(\boldsymbol{n}) \cup A^r_{\mathcal{C}}(\boldsymbol{n}) \tag{2.34}$$

$$A_2(\boldsymbol{n}) = \bigcup_{\forall r \in R} A^r_2(\boldsymbol{n}) \tag{2.35}$$

Equations (2.32) to (2.35) define sets that are used to select eligible (fixed + undecided) activities for node $\boldsymbol{n}$. Set $A_1(\boldsymbol{n})$ contains static and eligible dynamic activities. Since some optional activities $e \in A_{\mathcal{O}}$ may not be selectable due to the branching decisions, it holds that $A_1(\boldsymbol{n}) \subseteq A$. Set

$A_2(\boldsymbol{n})$ is similar to set $A_1(\boldsymbol{n})$ but it uses composite activities to encapsulate (undecided) optional activities $A_{\mathcal{D}_u}(\boldsymbol{n})$.

2.9.3 Convex Envelopes

Convex envelopes, used to calculate the valid lower bound of node $\boldsymbol{n}$ (see Section 2.9.4), approximate the energy consumption of individual activities based on their durations and a (partial) solution of node $\boldsymbol{n}$, where an order of activities, locations, and power-saving modes may not be fully decided. In other words, each activity $a \in A_2(\boldsymbol{n})$ is assigned convex envelope $f_a(d_a)$ that is defined as a function mapping duration d_a of activity $a \in A_2(\boldsymbol{n})$ to a lower estimation of its energy consumption. The next three paragraphs describe the creation of envelopes for static, dynamic, and composite activities.

In case of static activity $v \in A^r_{\mathcal{S}}$, $r \in R$, convex envelope $f_v(d_v)$ is a linear function with a slope equal to $\min_{\forall l \in L_v(\boldsymbol{n}), \forall m \in M^r_v(\boldsymbol{n})} p^m_{v,l}$ and zero offset. The slope is the least possible input power of activity v for node $\boldsymbol{n}$.

In case of dynamic activity $e \in A_{\mathcal{D}_f}(\boldsymbol{n})$, convex envelope $f_e(d_e)$ is calculated from all functions $f^t_e(d_e)$ where $t \in T_e(\boldsymbol{n})$. If robotic paths are determined, i.e., $|L_v(\boldsymbol{n})| = 1$ for $\forall v \in A_{\mathcal{S}}$ and $A_{\mathcal{D}_u}(\boldsymbol{n}) = \emptyset$, then each dynamic activity $e \in A_{\mathcal{D}_f}(\boldsymbol{n})$ is assigned one trajectory t and its convex envelope $f_e(d_e)$ is equivalent to energy function $f^t_e(d_e)$.

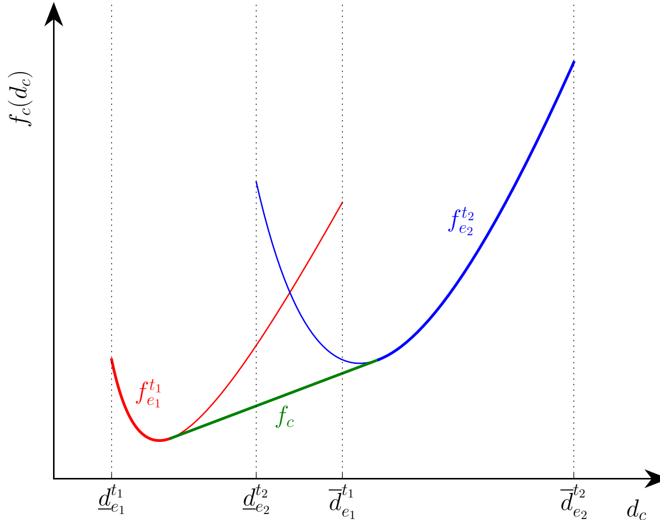


Figure 2.9: Example of the construction of the convex envelope.

In case of composite activity $c \in A_{\mathcal{C}}(\boldsymbol{n})$, convex envelope $f_c(d_c)$ is calculated from all functions $f^t_e(d_e)$ where $e \in A_{\mathcal{D}_u}(c)$ and $t \in T_e(\boldsymbol{n})$. Notice

that the construction of the envelope uses trajectories from two or more activities in $A_{\mathcal{D}_u}(c)$ ($\subseteq A_{\mathcal{O}}$), in comparison to dynamic activity $e \in A_{\mathcal{D}_f}(\boldsymbol{n})$ where only trajectories $T_e(\boldsymbol{n})$ are considered.

As an example consider the robotic cell in Figure 2.3 and robot r_1 with the undecided order of activities. One of five composite activities is c with $A_{\mathcal{D}_u}(c) = \{(v_1, v_2), (v_1, v_3)\}$, since it is not decided whether robot r_1 will move from v_1 to v_2 or to v_3. Assume that $|L_v| = 1$ for $\forall v \in A_{\mathcal{S}}^{r_1}$, then it is not clear whether to evaluate function $f_{e_1}^{t_1}(d_{e_1})$ or $f_{e_2}^{t_2}(d_{e_2})$, where $e_1 = (v_1, v_2)$, $t_1 \in T_{e_1}$, $e_2 = (v_1, v_3)$, and $t_2 \in T_{e_2}$. For this reason, convex envelope $f_c(d_c)$ that provides a lower estimation on the energy consumption is constructed as illustrated in Figure 2.9. In this case, the convex envelope (bold line) is a function with domain $[\underline{d}_{e_1}^{t_1}, \overline{d}_{e_2}^{t_2}]$ such that $f_c(d_c) \leq f_{e_1}^{t_1}(d_c)$ for $d_c \in [\underline{d}_{e_1}^{t_1}, \overline{d}_{e_1}^{t_1}]$ and $f_c(d_c) \leq f_{e_2}^{t_2}(d_c)$ for $d_c \in [\underline{d}_{e_2}^{t_2}, \overline{d}_{e_2}^{t_2}]$.

Theorem 1. *Convex envelope provides a valid lower estimation on energy.*

Proof. In case of static activity $v \in A_{\mathcal{S}}$, it trivially follows since the slope of convex envelope $f_v(d_v)$ (linear function) is calculated as the minimal input power for all selectable locations and power-saving modes.

The situation is much more complex for dynamic activities $A_{\mathcal{D}_f}(\boldsymbol{n})$ and composite activities $A_{\mathcal{C}}(\boldsymbol{n})$. Although the composite activity is different from the dynamic one since it encapsulates at least two dynamic activities, both types have the common property that in an arbitrary feasible solution just one trajectory, i.e., function $f_a^{t_i}(d_a)$, is selected for each $a \in A_{\mathcal{D}_f}(\boldsymbol{n}) \cup A_{\mathcal{C}}(\boldsymbol{n})$. As a result, the convex envelope is constructed in the same way and it suffices to present one proof.

If activity $a \in A_{\mathcal{D}_f}(\boldsymbol{n}) \cup A_{\mathcal{C}}(\boldsymbol{n})$ is assigned one energy function $f_a^{t_1}(d_a)$ by the branching, then convex envelope $f_a(d_a)$ is equal to $f_a^{t_1}(d_a)$. Otherwise, the proof by induction starts with two convex (energy) functions $f_a^{t_1}(d_a)$ and $f_a^{t_2}(d_a)$, i.e., $n = 2$, from which just one function (trajectory) is selected in any feasible solution with its associated duration. This is stated as the following energy optimization subproblem, where y_1 and y_2 are binary variables switching between energy functions, and d'_a and d''_a are the related durations.

$$\min f_a^{t_1}(d'_a) y_1 + f_a^{t_2}(d''_a) y_2 = \gamma_1 \tag{2.36}$$

$$s.t. \quad y_1 + y_2 = 1 \tag{2.37}$$

$$\underline{d}_a^{t_1} \leq d'_a \leq \overline{d}_a^{t_1} \tag{2.38}$$

$$\underline{d}_a^{t_2} \leq d''_a \leq \overline{d}_a^{t_2} \tag{2.39}$$

$$y_1, y_2 \in \mathbb{B} \qquad d'_a, d''_a \in \mathbb{R}_{\geq 0} \tag{2.40}$$

If the integrality of y_1 and y_2 is relaxed and domains of two functions are extended (defined on $\mathbb{R}_{>0}$), then a lower bound on the energy of the subproblem can be obtained as follows.

$$\min f_a^{t_1}(d'_a)y_1 + f_a^{t_2}(d''_a)y_2 = \gamma_2 \tag{2.41}$$

$$s.t. \quad y_1 + y_2 = 1 \tag{2.42}$$

$$\min(\underline{d}_a^{t_1}, \underline{d}_a^{t_2}) \leq d'_a \leq \max(\overline{d}_a^{t_1}, \overline{d}_a^{t_2}) \tag{2.43}$$

$$\min(\underline{d}_a^{t_1}, \underline{d}_a^{t_2}) \leq d''_a \leq \max(\overline{d}_a^{t_1}, \overline{d}_a^{t_2}) \tag{2.44}$$

$$y_1, y_2, d'_a, d''_a \in \mathbb{R}_{\geq 0} \tag{2.45}$$

Observe that the criterion is determined by the convex combination of two functions; therefore it forms a convex hull from which only a lower envelope is required due to the minimization. As a result, the final formulation is

$$\min f_a(d_a) = \gamma_3 \tag{2.46}$$

$$s.t. \quad \underline{d}_a \leq d_a \leq \overline{d}_a \tag{2.47}$$

$$d_a \in \mathbb{R}_{\geq 0} \tag{2.48}$$

where $f_a(d_a)$ is a convex envelope of activity $a \in A_{\mathcal{D}_f}(\boldsymbol{n}) \cup A_{\mathcal{C}}(\boldsymbol{n})$, $\underline{d}_a = \min(\underline{d}_a^{t_1}, \underline{d}_a^{t_2})$, and $\overline{d}_a = \max(\overline{d}_a^{t_1}, \overline{d}_a^{t_2})$. Due to the relaxations applied at each step it holds that $\gamma_1 \geq \gamma_2 \geq \gamma_3$, and therefore, the convex envelope $f_a(d_a)$ provides a valid lower estimation of energy for $\underline{d}_a \leq d_a \leq \overline{d}_a$.

So far, it was shown that the relaxation is valid for two functions, i.e. $n = 2$, in the remaining part, a generalization to arbitrarily many functions is presented. Let say that convex envelope $f_a(d_a)$ is already created from functions $f_a^{t_i}(d_a)$ for $\forall i \in \{1, \cdots, k\}$. The next step $(n = k+1)$ is to extend the convex envelope by adding function $f_a^{t_{k+1}}(d_a)$. Since convex envelope $f_a(d_a)$ is also a convex function, then the same procedure can be repeated on functions $f_a(d_a)$ and $f_a^{t_{k+1}}(d_a)$ and the result immediately follows. □

2.9.4 Lower Bound based on Convex Envelopes

The energy evaluator of nodes is a part of the Branch & Bound algorithm which primary purpose is to provide valid lower bounds for partial nodes. The function of the node evaluator (see Figure 2.5) can be briefly described as follows. First, the MILP formulation of the lower bound problem is created based on node $\boldsymbol{n}$. Subsequently, the MILP problem is solved to obtain a lower bound on energy and the bound is assigned to partial node $\boldsymbol{n}$. In the remaining part of this section, the MILP formulation is described in details.

Optimization criterion is to minimize the overall energy consumption as expressed in Equation (2.49) where functions $\hat{f}_a(d_a)$ are discretized convex envelopes $f_a(d_a)$. Each discretized envelope $\hat{f}_a(d_a)$ is a piece-wise linear convex function approximating convex envelope $f_a(d_a)$ from below since the lower bound cannot be overshot. As the criterion is a summation of piece-wise linear convex functions, it is possible to use a special simplex method in Gurobi solver to significantly accelerate the algorithm (see experiments in Section 2.12).

$$\min \sum_{\forall a \in A_2(\boldsymbol{n})} \hat{f}_a(d_a) \tag{2.49}$$

$$s.t. \sum_{\forall a \in A_2^r(\boldsymbol{n})} d_a = C_T \qquad \forall r \in R \tag{2.50}$$

Constraint (2.50) states that the total duration of activities in a robot cycle is equal to cycle time C_T, which is set to the desired value $\overline{C_T}$ in Equation 2.64. Two forms of the cycle time, i.e., a constant and variable, are introduced to enable reuse of constraints for the Deep Jumping approach. If an order of activities is not fully decided, then the duration of not yet selected (optional) activities can be replaced by the duration of the related composite activities since exactly one optional activity $e \in A_{\mathcal{D}_u}(c)$ is selected for each activity $c \in A_{\mathcal{C}}(\boldsymbol{n})$, and each static activity $v \in A_{\mathcal{S}}$ is assigned just one successor $e \in S_v$ in any feasible solution. Note that even Equations (2.49) and (2.50) provide a very good relaxation of the problem in terms of energy estimation, however, the efficient bounding requires that the lower bound meets the upper bound in leaf nodes if the discretization is neglected.

Auxiliary function $H(v)$ is defined in Equation (2.51). Its purpose is to indicate by a value whether static activity $v \in A_{\mathcal{S}}^r$ is the last activity in the cycle of robot r, or in other words, whether activity v closes the cycle.

$$H(v) = \begin{cases} 1 & \text{if } \exists v_h^r = v \\ 0 & \text{otherwise} \end{cases} \tag{2.51}$$

Constraints (2.52) to (2.55) determine the order of activities and their timing. Each fixed activity $e \in A_{\mathcal{D}_f}(\boldsymbol{n})$ adds two precedences: from its predecessor to e (Equation (2.52)), and from e to its successor (Equation (2.53)). Each composite activity $c \in A_{\mathcal{C}}(\boldsymbol{n})$ has a unique predecessor, thus, a precedence from its predecessor to c is added (Equation (2.54)). Composite activity c has at least two possible successors, from which just one successor can start without delay after activity c, therefore, constraints (2.55) select one of many possible precedences to the successors of activity c. The selection is accomplished by using binary variables x_{v_1,v_2}, where $v_1 \in P_c$ and

x_{v_1,v_2} is equal to one if precedence from c to v_2 is selected; otherwise, the precedence is not selected and the value is zero.

$$s_{v_1} + d_{v_1} - C_T H(v_1) = s_e \qquad \forall e \in A_{\mathcal{D}_f}(\boldsymbol{n}), v_1 \in P_e \tag{2.52}$$

$$s_e + d_e = s_{v_2} \qquad \forall e \in A_{\mathcal{D}_f}(\boldsymbol{n}), v_2 \in S_e \tag{2.53}$$

$$s_{v_1} + d_{v_1} - C_T H(v_1) = s_c \qquad \forall c \in A_{\mathcal{C}}(\boldsymbol{n}), v_1 \in P_c \tag{2.54}$$

$$s_c + d_c \leq s_{v_2} + \overline{C_T}(1 - x_{v_1,v_2}) \qquad \forall c \in A_{\mathcal{C}}(\boldsymbol{n}), v_1 \in P_c, v_2 \in S_c \tag{2.55}$$

Constraints (2.56) state that every composite activity has one successor selected. In a similar way, constraints (2.57) ensure that each static activity without a fixed predecessor $e \in A_{\mathcal{D}_f}(\boldsymbol{n})$ is assigned one predecessor from optional activities $A_{\mathcal{D}_u}(\boldsymbol{n})$. Constraints (2.56) and (2.57) state that each static activity is entered/left just once, which together with the timing constraints (2.52) to (2.55) guarantees that Hamiltonian circuit $\mathcal{HC}^r_{\text{act}}$ of each robot $r \in R$ is one graph component (no division to multiple circuits).

$$\sum_{\forall v_2 \in S_c} x_{v_1,v_2} = 1 \qquad \forall c \in A_{\mathcal{C}}(\boldsymbol{n}), v_1 \in P_c \tag{2.56}$$

$$\sum_{\forall v_1 \in B_{v_2}} x_{v_1,v_2} = 1 \qquad \forall v_2 \in A_{\mathcal{S}}, |B_{v_2}| > 0 \tag{2.57}$$
$$B_{v_2} = \{\forall v_1 : (v_1, v_2) \in A_{\mathcal{D}_u}(\boldsymbol{n})\}$$

The time synchronization between robots is modeled by using time lags as expressed by constraints (2.58). Time lag $(a_1, a_2) \in E_{\mathcal{TL}}$ is active if both activities a_1 and a_2 are selected, it means that each activity a_i is either fixed, i.e., $a_i \in A_{\mathcal{S}} \cup A_{\mathcal{D}_f}(\boldsymbol{n})$, or is selected by x_{v_1,v_2} variable, i.e., $a_i \in A_{\mathcal{D}_u}(\boldsymbol{n})$. Timing of optional dynamic activities $A_{\mathcal{D}_u}(\boldsymbol{n})$ is replaced by timing of the related composite activities as it is indicated by setting a'_i values in Equation (2.58), and optional dynamic activities of the time lag are stored in set J. If all activities $e \in J$ are selected by x_{v_1,v_2} variables, then the related constraint (2.58) is active; otherwise, it is disabled.

$$s_{a'_2} - s_{a'_1} + |R|\overline{C_T}(|J| - \sum_{\forall (v_1,v_2) \in J} x_{v_1,v_2}) \geq l_{a_1,a_2} - C_T h_{a_1,a_2} \tag{2.58}$$
$$\forall (a_1, a_2) \in E_{\mathcal{TL}}, a_1 \in A_1^{r_i}(\boldsymbol{n}), a_2 \in A_1^{r_j}(\boldsymbol{n}), r_i \neq r_j$$
$$\text{if } a_i \notin A_{\mathcal{D}_u}(\boldsymbol{n}) \text{ then } a'_i = a_i \text{ else } a'_i = c_i, c_i \in A_{\mathcal{C}}(\boldsymbol{n}), a_i \in A_{\mathcal{D}_u}(c_i)$$

$$J = \{\forall e : e \in A_{\mathcal{D}_u}(\boldsymbol{n}), e = a_1 \text{ or } e = a_2\}$$

Constraints (2.59) and (2.60) are required to avoid possible collisions between robots for a partial solution of node $\boldsymbol{n}$. Collision $(a_1, g_1, a_2, g_2) \in K$ may happen if $a_i \in A_{\mathcal{D}_f}(\boldsymbol{n}), g_i \in T_{a_i}(\boldsymbol{n}), |T_{a_i}(\boldsymbol{n})| = 1$ or $a_i \in A_{\mathcal{S}}, g_i \in L_{a_i}(\boldsymbol{n}), |L_{a_i}(\boldsymbol{n})| = 1$ for $i \in \{1, 2\}$. Set $K(\boldsymbol{n})$ contains all collisions that may happen in node $\boldsymbol{n}$. Each collision $(a_1, g_1, a_2, g_2) \in K(\boldsymbol{n})$ is resolved by making activities a_1 and a_2 time disjunctive for multiples of cycle time C_T. In other words, for each multiple of the cycle time, binary variable k_o^i decides whether a_1 precedes a_2 or vice versa. The constraints are added gradually as further trajectories and locations are fixed by propagations (see Section 2.8).

$$s_{a_2} + iC_T + 2\,|R|\,\overline{C_T}(1 - k_o^i) \geq s_{a_1} + d_{a_1} \quad \forall o = (a_1, g_1, a_2, g_2) \in K(\boldsymbol{n}), \forall i \in \{-\,|R|\,, \ldots, |R|\} \tag{2.59}$$

$$s_{a_1} + 2\,|R|\,\overline{C_T}k_o^i \geq s_{a_2} + d_{a_2} + iC_T \quad \forall o = (a_1, g_1, a_2, g_2) \in K(\boldsymbol{n}), \forall i \in \{-\,|R|\,, \ldots, |R|\} \tag{2.60}$$

The duration of each activity $e \in A_{\mathcal{D}_f}(\boldsymbol{n})$ is determined by its selectable trajectories $T_e(\boldsymbol{n})$ as expressed in constraints (2.61). Similarly, constraints (2.62) define the duration of each static activity $v \in A_{\mathcal{S}}$, where the duration depends on the operation (e.g., spot-welding) and applicable power-saving modes. Constraints (2.63) state that the duration of each composite activity $c \in A_{\mathcal{C}}(\boldsymbol{n})$ is linked with the duration of a selected activity $e \in A_{\mathcal{D}_u}(c)$, and constraint (2.64) sets the cycle time according to the desired production rate. To alleviate the computational effort, the integrality of the binary variables x_{v_1,v_2} is relaxed to intervals between 0 and 1 as obvious from Equations (2.65) and (2.66). The domains of other variables are shown in Equation (2.66).

$$\min_{\forall t \in T_e(\boldsymbol{n})} \underline{d}_e^t \leq d_e \leq \max_{\forall t \in T_e(\boldsymbol{n})} \overline{d}_e^t \qquad \forall e \in A_{\mathcal{D}_f}(\boldsymbol{n}) \tag{2.61}$$

$$\max(\underline{d}_v, \min_{\forall m \in M_v^r(\boldsymbol{n})} \underline{d}^m) \leq d_v \leq \overline{d}_v \qquad \forall r \in R, \forall v \in A_{\mathcal{S}}^r \tag{2.62}$$

$$\sum_{\substack{\forall e \in A_{\mathcal{D}_u}(c) \\ e = (v_1, v_2)}} \underline{d}_e x_{v_1,v_2} \leq d_c \leq \sum_{\substack{\forall e \in A_{\mathcal{D}_u}(c) \\ e = (v_1, v_2)}} \overline{d}_e x_{v_1,v_2} \qquad \forall c \in A_{\mathcal{C}}(\boldsymbol{n}) \tag{2.63}$$

$$C_T = \overline{C_T} \tag{2.64}$$

$$0 \leq x_{v_1,v_2} \leq 1 \qquad \forall (v_1, v_2) \in A_{\mathcal{D}_u}(\boldsymbol{n}) \tag{2.65}$$

$$s_a, d_a, C_T, x_{v_1,v_2} \in \mathbb{R}_{\geq 0} \qquad k_o^i \in \mathbb{B} \tag{2.66}$$

It is interesting to see how the model changes if a) order of activities is partially fixed, b) some locations are fixed, or c) some power-saving modes

are applied. In all the cases, convex envelopes $\hat{f}_a(d_a)$ get tighter since they are formed based on the branching decisions as described in previous Section 2.9.3. Besides the modified criterion, in case a), the number of composite activities and x_{v_1,v_2} variables decreases since fixing the order means rejecting some optional activities from a partial solution. After the order is fixed, constraints (2.54–2.57, 2.63) are removed and there are no composite activities and variables x_{v_1,v_2}. In case b), new constraints (2.59) and (2.60) are added if required to avoid collisions, and in case c), the minimal duration of the static activity can be increased due to its assigned power-saving mode, see Equation (2.62).

Finally, two important properties of the model should be emphasized. The first one is its abstraction with respect to the selection of locations and trajectories. Only envelopes, activity domains, and set $K(\boldsymbol{n})$ are influenced and no other extra variables are required. The second one is its property that a solution in a leaf node is also a solution to the original problem as defined in the problem statement (see Section 2.3). The only difference is that discretized envelopes $\hat{f}_a(d_a)$ approximate convex envelopes $f_a(d_a)$ from above to get a valid upper bound.

2.10 Deep Jumping

Deep Jumping is an approach that is useful if a feasible order of activities is difficult to find due to time synchronization between robots and spatial compatibility. Deep Jumping employs a heuristic that searches a promising order of activities. This order is used to immerse into the Branch & Bound tree to quickly find good feasible solutions by the subsequent fixing of locations and power-saving modes. The approach comprises of three parts: 1) searching a promising order of activities, 2) construction of a subtree based on the order, 3) immersion into the subtree and marking the node with the fixed order as a preferred one for the subsequent branching.

In the first part, an order of activities is obtained by solving the following MILP problem with a time limit and relative optimality gap.

$$\min C_T \tag{2.67}$$

subject to: (2.50) to (2.63)

$$\overline{C_T} \geq C_T \tag{2.68}$$

$$s_a, d_a, C_T \in \mathbb{R}_{\geq 0} \qquad k_o^i, x_{v_1,v_2} \in \mathbb{B} \tag{2.69}$$

Criterion (2.67) is to minimize the cycle time since it makes a solution robust to time prolongations with respect to further fixing of locations and

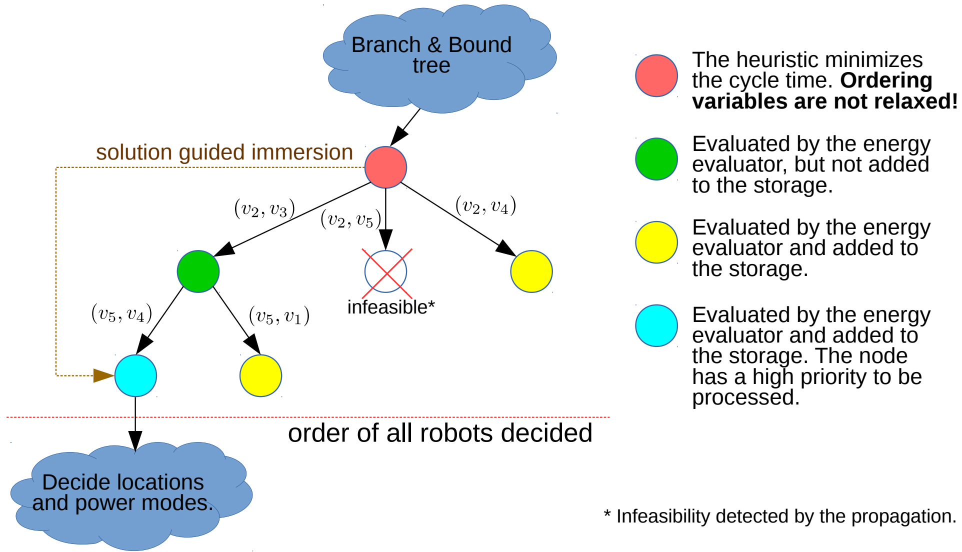


Figure 2.10: Example of Deep Jumping.

power-saving modes. Constraint (2.68) renders solutions exceeding the desired cycle time infeasible. Compared to the energy evaluation, ordering variables x_{v_1,v_2} are not relaxed for $\forall(v_1, v_2) \in A_{\mathcal{D}_u}(\boldsymbol{n})$. If a feasible solution is found, the second part uses these variables to construct a subtree from node $\boldsymbol{n}$; otherwise, node $\boldsymbol{n}$ is branched as described in Section 2.8.

Construction of the subtree:

1. Set current node $\boldsymbol{n}_*$ to $\boldsymbol{n}$.
2. Select an arbitrary composite activity $c \in A_{\mathcal{C}}(\boldsymbol{n}_*)$.
3. Determine its selected activity $e_u = (v_1, v_2) \in A_{\mathcal{D}_u}(c)$ by searching a variable x_{v_1,v_2} with the value equal to 1.
4. Branch node $\boldsymbol{n}_*$ on optional activities $A_{\mathcal{D}_u}(c)$, i.e., on order of activities, and evaluate the child nodes by the tight lower bound (see Section 2.9.4).
5. Find a child node $\boldsymbol{n}'$ such that $e_u \in A_{\mathcal{D}_f}(\boldsymbol{n}')$. If not exists (discussed later), the subtree is not fully expanded and go to 7.
6. Set $\boldsymbol{n}_* = \boldsymbol{n}'$, and go to step 2 if $|A_{\mathcal{C}}(\boldsymbol{n}')| > 0$.
7. A (partial) subtree has been created.

There are two remarkable facts about the method. The first one is that the depth of the subtree can be less than $|A_{\mathcal{C}}(\boldsymbol{n})|$ since the propagation on the order of activities (see Section 2.8) may infer another decisions. The second one is that the subtree might not be fully expanded since collisions with undecided activities $A_{\mathcal{D}_u}(\boldsymbol{n})$ (trajectories) are not resolved in the

subtree root, and newly considered collisions in descendant nodes (added constraints (2.59) and (2.60)) may break the feasibility while the order is being fixed.

In the third part, if the subtree is fully expanded, then the node with the fixed promising order, i.e., node $\boldsymbol{n}_*$ after the method finishes, is marked as a preferred one for the subsequent branching. All feasible nodes with $\mathrm{LB}(\boldsymbol{n}) < \mathrm{UB}$ that are not branched in the subtree are added to the storage. Note that the optimality of the exact algorithm is preserved since Deep Jumping does not ignore any nodes that could be resulting in an optimal solution.

Deep Jumping approach is illustrated (see Figure 2.10) on the robotic cell from Figure 2.3. Let say that $\boldsymbol{n}$ is the root node of the Branch & Bound tree, and the heuristic finds a feasible order $(v_1, v_2, v_3, v_5, v_4)$ for robot r_1, i.e., variables x_{v_1,v_2}, x_{v_2,v_3}, x_{v_3,v_5}, x_{v_5,v_4}, and x_{v_4,v_1} are set to one. In case of robot r_2, there is nothing to decide. For the sake of simplicity, consider the same scheme as in Figure 2.6, that is optional activities (v_2, v_3) and (v_5, v_4) are fixed in order. Node $\boldsymbol{n}_* = \boldsymbol{n}$ is expanded on optional activities $A_{\mathcal{D}_u}(c) = \{(v_2, v_3), (v_2, v_4), (v_2, v_5)\}$, where $c \in A_{\mathcal{C}}(\boldsymbol{n}_*)$ is the corresponding composite activity. One of child nodes is infeasible, see Figure 2.10, since the order propagation detected the infeasibility that circuit $\mathcal{HC}^{r_1}_{\text{act}}$ cannot be closed. Based on the solution, node $\boldsymbol{n}_*$ is updated to $\boldsymbol{n}'$ such that $(v_2, v_3) \in A_{\mathcal{D}_f}(\boldsymbol{n}')$, and the process is repeated in a similar way for (v_5, v_4). The final subtree and subpath $((v_2, v_3), (v_5, v_4))$ determine the node with high priority. Notice how the order propagation effectively reduced the depth of the subtree to 2 compared to the initial number of composite activities $|A_{\mathcal{C}}(\boldsymbol{n})| = 5$. Although the root node was used in the example, in practice, it is desirable to have a minimal number of nodes in the storage to guarantee a good scalability.

2.11 Parallelization

In order to find good solutions faster, the Branch & Bound algorithm was parallelized to exploit the power of modern processors. The parallelization scheme (see Figure 2.11) is based on the division of the work between *worker threads*, where each worker thread searches a different part of the Branch & Bound tree to accelerate the algorithm. A *control thread* manages the data initialization, creation and destruction of worker threads, and prints a result. Besides a thread-safe storage of nodes, there are two variables that require atomic access: variable UB, and a stop flag. The stop flag is used to terminate worker threads if a time limit t_{max} is reached or the tree has been explored. The incumbent solution is protected by a lock to avoid race conditions. Other blocks of the algorithm are the same as in Figure 2.5.

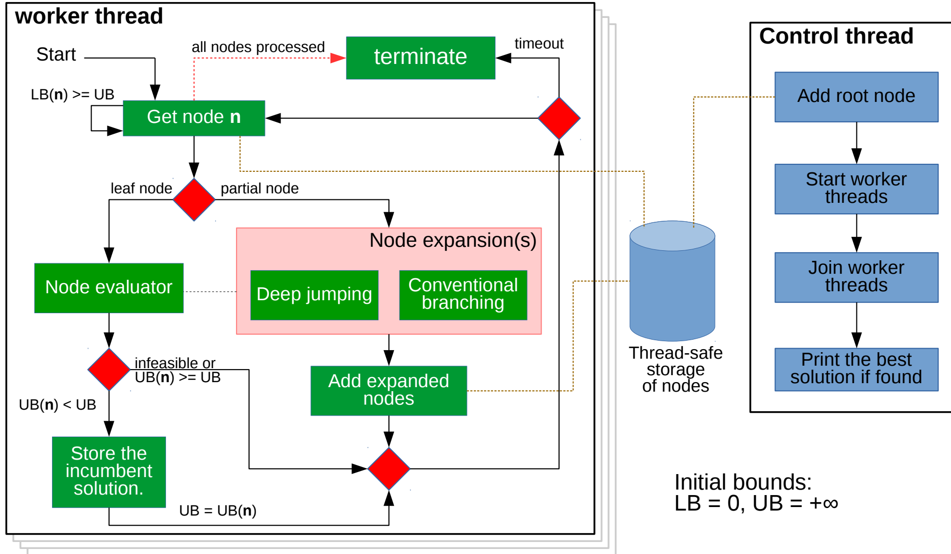


Figure 2.11: Parallel Branch & Bound algorithm.

The *thread-safe storage* keeps unexplored nodes of the Branch & Bound tree. Since the storage is accessed by many threads concurrently, it is designed in a way that minimizes the thread contention. As a result, the sharing of the nodes has a minimal impact on scalability, as shown by experiments in Section 2.12. The structure of the storage, as depicted in Figure 2.12, consists of a shared priority queue, which is protected by a mutex, and a local stack for each worker thread.

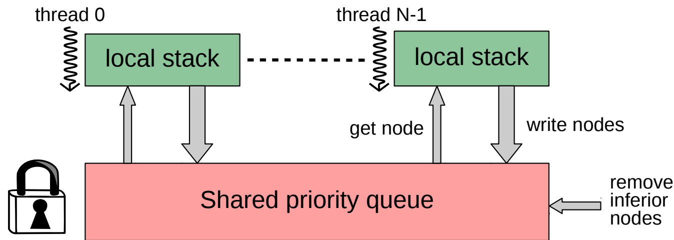


Figure 2.12: Thread-safe storage of nodes.

The queue serves as a sorted pool of nodes that is accessed if: a) a worker thread gets a new node to be explored, b) if a worker thread writes partial nodes generated during the search, or c) if a new incumbent solution is found. In case a), a worker thread gets exclusive access to the queue to get (read and remove) unexplored node $\boldsymbol{n}$ with minimal LB($\boldsymbol{n}$). From this initial node, a depth search with a limit on a number of expansions, empirically set to $3 \times$ (maximal known depth of the tree $+ 1$), is carried out. The depth

search is used, at a price of slower improvements of the global LB, instead of the breadth search since it is more efficient at finding feasible solutions and its memory footprint is smaller. The memory footprint is reduced even more by using a copy-on-write technique, i.e., only data to be modified by the branching are copied before their change. The worker thread maintains the working set of nodes in its local stack until the limit on the number of expansions is reached. Afterwards, i.e., case b), the unexplored nodes are copied back to the queue (exclusive access) and the local stack is emptied. In case c), a worker thread that found a new incumbent solution gets exclusive access to the queue to remove all the nodes which lower bound is greater or equal to a new best upper bound UB. Note that every worker thread iteratively performs a) and b) until the stop flag is set.

The efficiency of the storage resides not only in the separated local stacks but also in underlying data structures. The priority queue corresponds to `multimap<double, n>` from C++11 standard library, and a local stack is `vector<n>`. The multimap contains unexplored nodes sorted according to their lower bounds. With respect to the time complexity of the queue access, in case a), the first element (node) can be obtained in an amortized constant time, in case b), the complexity is linear if elements in a local stack are sorted according to lower bounds before their insertion, and in case c), inferior nodes can be removed in a linear time. Obviously, the complexity defined by C++11 standard is very favorable to alleviate the contention between threads.

Finally, if the tree is completely searched, it is necessary to detect it and set the stop flag. It is accomplished by counting the nodes that are in the storage or in the process atomically. In particular, the counter is set to 1 (a root node) before the worker threads start. Each working thread modifies the counter according to the following rules. If a node is obtained from the storage, the counter is not changed since one node is removed and one extra node is in the process. When the node is processed, the counter is increased by the number of generated nodes that are inserted, and subsequently, the processed node is destroyed and the counter is decreased. If a new incumbent solution is found, the counter is decreased by the number of inferior nodes that are removed. If the counter hits zero, then a worker thread sets the flag to signal that the tree was completely searched.

Note that it is desirable to avoid calling Deep Jumping until there are enough nodes, e.g., two times the number of hardware threads, in the storage since it may limit the scalability of the algorithm. Deep jumping, which is called from time to time, can be considered to some extent as a depth search with solution guided immersion, therefore no extra handling is required.

2.12 Experimental Results

To evaluate the effectiveness and performance of the proposed algorithms, they were tested on benchmark instances with 3, 5, 8, and 12 robots and compared with a parallel MILP solver. The experiments were carried out on a Linux server with two Intel Xeon E5-2620 v2 2.10GHz processors (2 x 6 physical cores + hyper-threading) and 64 GB of DDR3 memory. Gurobi 6.0.4 (the heuristic and MILP solver) and Gurobi 7.0.2 (the Branch & Bound) were installed to solve MILP problems. The performance of the state-of-the-art MILP solvers was compared with lp_solve 5.5.2.0, which is an open-source MILP solver. The algorithms, written in C++11 and compiled by GCC 4.9.3, use OpenMP 4.0 library for the parallelization, and their efficiency was verified on three groups of experiments. The first one deals with the scalability and performance, the second one investigates the ability of two exact algorithms to find optimal solutions, and the last one measures the quality of solutions for bigger instances and compares the proposed lower bound (see Section 2.9.4) with the LP relaxation of the MILP formulation from Section 2.4.

2.12.1 Performance Experiments

The first experiments measure the effect of the parallelization and the special simplex method on the performance of the Hybrid Heuristic and Branch & Bound algorithm. For the purpose of experiments, the following publicly available datasets from `https://github.com/CTU-IIG` were benchmarked: S5 (small, 5 robots), M8 (medium, 8 robots), and L12 (large, 12 robots). Each dataset contains 10 instances of the problem, from which an average performance is calculated for each setting, i.e., for a given number of worker threads and a used solver.

		lp_solve	Gurobi	Gurobi CF
S5	sequential	23.1	78.5	365
	parallel	304	1111	4708
M8	sequential	7.9	44.2	170
	parallel	93.4	488	2134
L12	sequential	3.3	30.8	97.9
	parallel	42.6	371	1084

Table 2.1: Performance of LP solvers for the heuristic.

In case of the heuristic, a good indicator of the performance is the number of LP evaluations per second because more than 90 % of the computational time is consumed by LP evaluations, i.e., the building of the LP problem, its optimization, and the extraction of a solution. Table 2.1 shows

the average number of LP evaluations per second for each dataset, where each energy function f_e^t was approximated by 10 linear functions. The minimal number of optimization iterations per tuple $\Phi_{\min}$ was set to 100, 600, and 1000 for the S5, M8, and L12 datasets, respectively, for all experiments in Section 2.12. The figures indicate that the parallel heuristic is about 12 times faster than the sequential one, and the specialized Gurobi simplex method, denoted as 'Gurobi CF' (abbreviation for Gurobi Convex Functions), accelerates the heuristic about 3 to 4 times compared with the regular one; therefore, an overall speedup of about 36 to 48 can be expected for the aforementioned configuration. Table 2.1 presents the corresponding values for the lp_solve open-source solver for comparison.

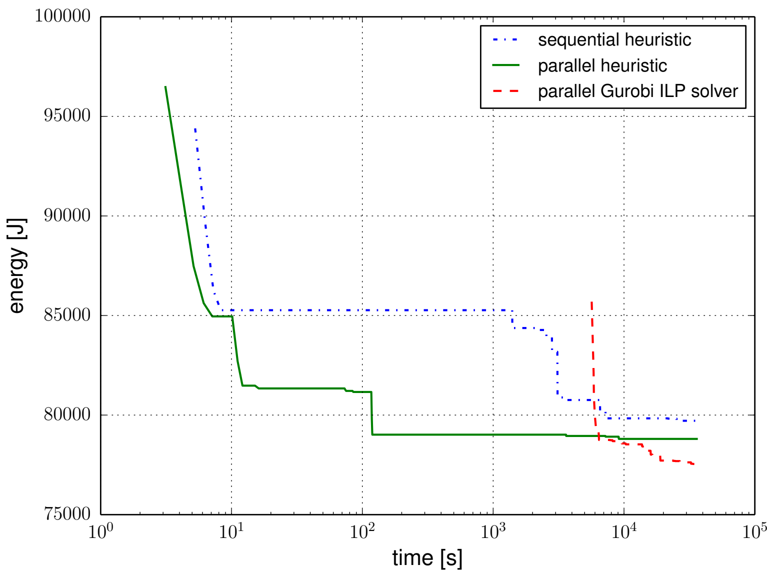


Figure 2.13: Progress of the heuristic and MILP solver on M8_8 instance.

To show that a higher number of evaluated tuples also has a positive impact on the quality of solutions, the dependence of the criterion value, i.e., energy consumption, on the time limit was plotted in Figure 2.13. The results on an instance with 8 robots revealed that the parallel heuristic with 24 threads (12 cores + hyper-threading) converged significantly faster than the sequential version; therefore, a similar solution quality was achievable in a fraction of the time. In comparison to the Gurobi MILP solver, the heuristic seems to be stronger in finding feasible solutions, compare 1.5 h with 3.1 s (5.3 s) required by the parallel (sequential) heuristic, and is more suitable for a short-term (re)optimization. On the other hand, if the MILP solver is given enough time, then better solutions may be found for the medium instances (see Figure 2.13). The same experiment was repeated

on L12_9 instance with 12 robots. The parallel (sequential) heuristic found the first feasible solution in 10 m (4 h) and the best criterion was 177276 J (186855 J) after the 10-h time limit. The Gurobi MILP solver had run out of memory (installed 64 GB of memory) in less than 4 hours without having any feasible solution.

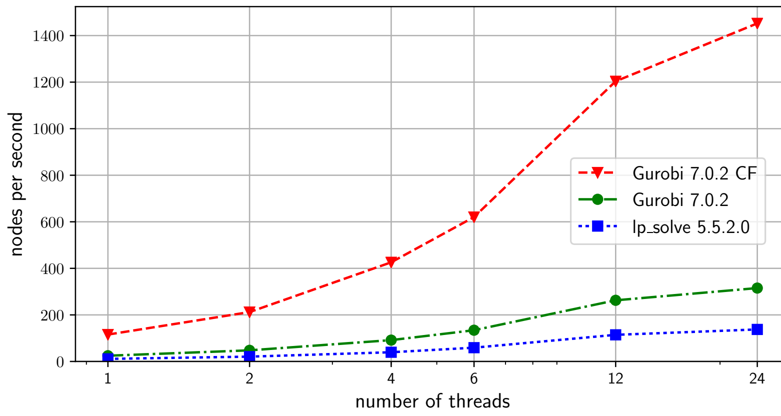
In case of the Branch & Bound algorithm, the performance is measured in the number of evaluated nodes per second since it directly determines the number of explored nodes for a given time limit. The results are shown in scalability graphs in Figure 2.14. The convex envelopes were approximated by 10 linear functions and Deep Jumping was disabled since it negatively influences the precision of results due to its indeterminism. In case of lp_solve and Gurobi (without 'CF') solvers, the criterion was emulated by adding extra constraints in a similar way as in Equations (2.1)–(2.3).

The results on S5 dataset (see Figure 2.14a) indicate a good scalability for all solvers since the parallelization accelerated the algorithm about the factor of 12.4/12.8/12.6 for lp_solve/Gurobi/Gurobi CF, respectively. Moreover, Gurobi with the special simplex method enabled to evaluate 4.6 times more nodes than a regular Gurobi simplex method. In total, the parallelization and the simplex method increased the throughput from 24.6 to 1452.4 nodes/s for Gurobi solver. Similar results were obtained on dataset M8 (see Figure 2.14b), that is speedups 12.4/10.8/13.5 for lp_solve/Gurobi/Gurobi CF, respectively, and the acceleration about the factor of 4.3 for the special simplex method. The total throughput for Gurobi solver was increased from 8.4 to 393 nodes/s. In case of the biggest dataset L12 the achieved speedups were 13.2/13.3/13.9 for lp_solve/Gurobi/Gurobi CF, respectively. Nevertheless, as seen from Figure 2.14c, the impact of the special simplex method is very limited and the acceleration is only about the factor 1.4. As a result, a throughput increase from 1.44 to 27.5 nodes/s is less significant.

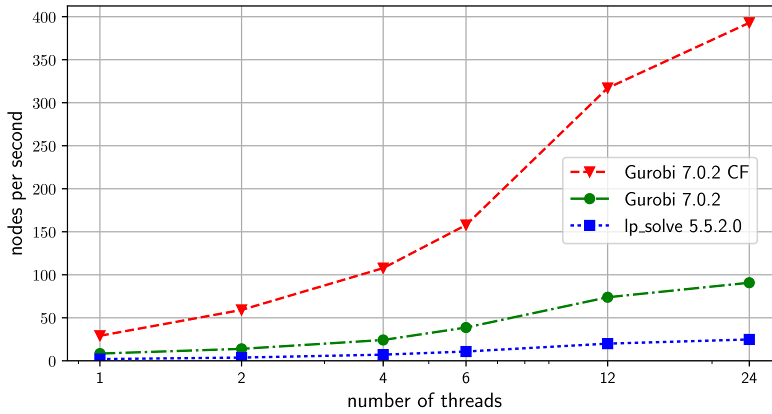
	Instructions per cycle	L1I miss rate	L2 miss rate	LLC miss rate	Branch rate	Misprediction rate
S5	1.81	0.35 %	0.92 %	0.01 %	20.89 %	0.79 %
M8	1.88	0.24 %	1.00 %	0.01 %	20.57 %	0.71 %
L12	2.25	0.04 %	1.01 %	0.005 %	19.86 %	0.31 %

Table 2.2: Performance metrics of the Branch & Bound algorithm.

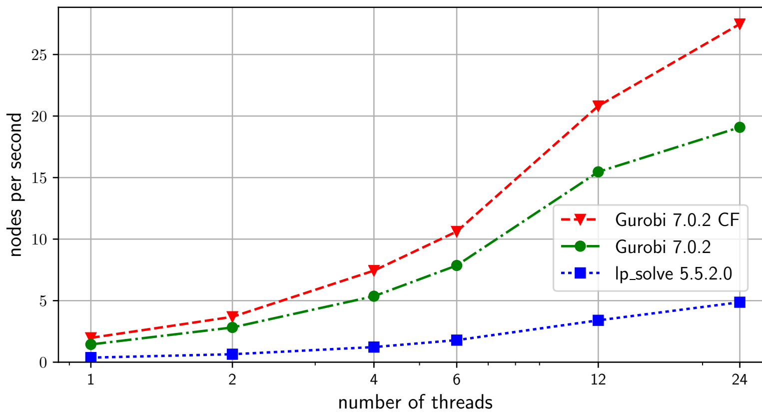
It remains to explain the lower efficiency of the special simplex method for large instances. The first hypothesis is that the method started to use the resources of the processors less efficiently, e.g., because of cache misses. In order to support or reject the hypothesis, the performance counters have been analyzed by likwid 4.2 [51], and the results are in Table 2.2. The number of instructions per cycle per physical core is a key indicator showing



(a) Performance graphs for S5 dataset.



(b) Performance graphs for M8 dataset.



(c) Performance graphs for L12 dataset.

Figure 2.14: Scalability graphs for S5, M8, and L12 datasets.

utilization of the processor. The achieved value is about 2, which means that the processor is utilized by roughly 50 %, which is a very good result. The percentage of instructions that cause cache misses is also very low, therefore, the cache is used efficiently. Although the code contains many direct/indirect branch instructions (each fifth instruction, see 'Branch rate' column), they are well predicted and the overhead is not significant. To summarize, the first hypothesis is rejected since the analysis did not reveal any deterioration of the processor utilization.

The other hypothesis is that the second (optimization) phase of the simplex method takes significantly less time compared to the first phase (finding a feasible solution) for L12 dataset. If it is the case, then the effect of the special simplex method is limited since this method accelerates only the second phase. A few experiments, where nodes were evaluated with and without the criterion, confirmed that the fraction of time spent in the first phase is significantly higher for large instances than for small and medium ones. As a result, there is only a marginal effect of using a special simplex method for L12 dataset and the hypothesis is supported.

2.12.2 Optimality Experiments

To analyze the Branch & Bound algorithm with respect to its ability to find optimal solutions, experiments on new instances with 3 robots were carried out, and the results were compared with the MILP solver. For the purpose of the experiments, S3 dataset with 5 instances was generated and uploaded to `https://github.com/CTU-IIG`, and each energy function was approximated by 20 linear functions. The achieved results are summarized in Table 2.3, where BaB and MILP are abbreviations for the Branch & Bound algorithm and MILP solver, respectively, and each measured value (average from 10 runs) is stated with the standard deviation in order to indicate the error.

inst.	opt. energy	threads	BaB runtime	MILP runtime	BaB eval. nodes
S3_0	16898.5	24	29.8 ± 1.29 s	59.3 ± 0.64 s	68114 ± 2837
		1	371.8 ± 18.6 s	220 ± 0.4 s	66927 ± 3085
S3_1	16923.2	24	> 3600 s	162 ± 0.78 s	6367408 ± 25976
		1	> 3600 s	> 3600 s	497746 ± 3332
S3_2	14010.0	24	1.72 ± 0.23 s	2.3 ± 0.04 s	798 ± 288
		1	1.98 ± 0.49 s	13.1 ± 0.01 s	507 ± 73
S3_3	19661.5	24	3.90 ± 0.75 s	31.2 ± 0.71 s	7310 ± 1383
		1	34.9 ± 3.5 s	37.2 ± 0.1 s	5882 ± 542
S3_4	11081.9	24	0.63 ± 0.02 s	9.42 ± 0.12 s	1140 ± 134
		1	2.46 ± 0.38 s	9.06 ± 0.02 s	698 ± 80

Table 2.3: Time to optimality for S3 dataset.

In case of the Branch & Bound algorithm, the parallelization seems to be more efficient for instances that generate a large number of nodes. For example, the optimal solution was found 12.5 times faster for instance S3_0 (more than 60 000 evaluated nodes) compared to instance S3_2 (more than 400 evaluated nodes) where the speedup is only 1.15. Notice that the bounding is also more efficient for larger instances since the number of evaluated nodes of the parallel version is much closer to the sequential one. As a result, it can be expected that the performance speedups in terms of the number of evaluated nodes per second will be close to the quality speedups (time to optimality) for instances with 5 or more robots. In case of the MILP solver, the parallelization is advantageous for larger instances.

The parallel Branch & Bound algorithm outperformed the parallel MILP solver in 4 of 5 cases. In case of instance S3_1, the algorithm was not able to close the gap in one hour. One of the reasons may be the depth search strategy that focuses on the feasibility and memory consumption rather than the fast improvement in the lower bound. Nevertheless, the parallel Branch & Bound algorithm found the optimal solution of instance S3_1 in 8 of 10 cases (without the optimality certificate) with the average proved gap 2.2 %.

2.12.3 Quality Experiments

The following experiments investigate the quality of obtained solutions within a given time limit for S5, M8, and L12 datasets. The quality of solutions, i.e., energy consumption, was measured from 10 runs for each instance[1]of S5 and M8 datasets, and the best, average, and the worst criterion values are stated for the heuristic and Branch & Bound algorithm. The average quality of solutions is compared with the results achieved by the parallel Gurobi MILP solver, which surprisingly had a deterministic behavior and thus provided the same quality of solutions for all runs. That is why only the average is stated for the Gurobi MILP solver. The Parallel Heuristic is abbreviated as PH in tables.

Since the biggest dataset L12 is too computationally demanding, only 3 measurements were carried out for each instance, and the criterion values were stated explicitly. Energy functions in the criterion were approximated by 10 linear functions each, and Deep Jumping is executed with 15 % probability in every viable node. Deep Jumping employs the MILP solver as a heuristic where the gap from optimality is set to 1/3/5 % and the time limit is 120/300/1200 s for the S5/M8/L12 dataset, respectively.

The results for S5 dataset in Table 2.4 indicate that the Branch & Bound algorithm is comparable with the heuristic even if a short time limit ($t_{\max}$

[1]Instance S5_2 is skipped in tables 2.4 and 2.5 since it was proved to be infeasible.

inst.	best BaB	avg BaB	worst BaB	best PH	avg PH	worst PH	avg MILP
S5_0	40377.5	**40559.2**	40787.1	40654.8	40788.5	40936.2	42521.9
S5_1	30804.8	**30899.8**	31074.8	31195.1	31335.2	31479.2	34582.6
S5_3	47134.7	47997.1	50375.8	47482.8	**47665.5**	47861.6	–
S5_4	48298.4	50911.0	52641.8	47826.3	**48039.4**	48326.6	–
S5_5	41669.0	42245.3	43414.9	41692.0	**41902.7**	42081.2	–
S5_6	34928.7	**35008.5**	35068.7	34920.3	35043.7	35128.2	36315.6
S5_7	42482.5	**42754.0**	43126.2	42511.6	42821.7	42987.6	45670.2
S5_8	38751.1	39577.0	40725.8	39181.0	**39423.6**	39602.4	41440.2
S5_9	38495.1	**38676.2**	38853.1	38864.9	39067.6	39235.9	39721.5

Table 2.4: Quality of solutions with $t_{\max} = 30\,\mathrm{s}$ for S5 dataset.

inst.	best BaB	avg BaB	worst BaB	best PH	avg PH	worst PH	avg MILP
S5_0	40200.1	**40225.5**	40268.7	40494.9	40637.9	40776.6	40272.8
S5_1	30607.0	30649.0	30717.3	31052.9	31111.8	31184.6	**30623.3**
S5_3	46873.4	**46994.4**	47091.8	47434.1	47483.3	47674.9	–
S5_4	46887.1	**46928.6**	46991.0	47541.1	47739.1	48067.6	–
S5_5	41394.3	**41454.0**	41541.3	41654.7	41700.0	41741.0	41498.8
S5_6	34655.4	**34698.7**	34726.3	34760.0	34856.4	34918.9	34720.9
S5_7	41992.4	42103.7	42297.3	42384.4	42515.8	42627.6	**42033.0**
S5_8	38394.1	38530.6	38643.6	39174.0	39301.4	39464.1	**38364.8**
S5_9	38201.0	38314.3	38350.2	38649.8	38840.3	38924.8	**38283.8**

Table 2.5: Quality of solutions with $t_{\max} = 1\,\mathrm{h}$ for S5 dataset.

$= 30\,\mathrm{s}$) is used. The MILP solver provided inferior solutions and failed to solve 3 instances. If the time limit is increased to one hour (see Table 2.5), the Branch & Bound algorithm finds significantly better solutions than the heuristic and the achieved quality of solutions is comparable with the MILP solver. On the one hand, the prolonged time limit enabled the MILP solver to obtain higher quality solutions, on the other hand, the solver was unable to find a feasible solution in two cases. Note that the best solutions found by the Branch & Bound algorithm were close to optimality since the average optimality gap is about 1.5 % from the lower bounds calculated in Bukata et al. [14].

Table 2.6 reveals the results for M8 dataset with the time limit of 10 minutes. The figures show that the Branch & Bound algorithm clearly outperforms both the heuristic and the MILP solver since the heuristic found a better solution only in one case and the MILP solver failed to find a feasible solution for 5 instances. The Branch & Bound algorithm is very effective in finding feasible solutions by virtue of the Deep Jumping approach. In case of one-hour time limit (see Table 2.7), the Branch & Bound algorithm still dominates with the exception of two instances where the MILP solver provided better solutions. The heuristic is strong in providing feasible solutions compared to the MILP solver that failed in 4 cases.

inst.	best BaB	avg BaB	worst BaB	best PH	avg PH	worst PH	avg MILP
M8_0	85986.0	**86824.4**	87788.7	86831.6	90580.7	95333.8	–
M8_1	86641.0	**86945.1**	87226.0	88098.9	88491.5	88883.8	89297.4
M8_2	88946.6	**89464.3**	90480.4	90048.7	90668.0	91463.6	92991.0
M8_3	82444.6	**82907.9**	83383.2	83412.6	83948.5	84942.6	–
M8_4	75394.6	78412.9	80252.4	76914.7	**77549.8**	78191.6	82881.8
M8_5	88448.3	**89133.2**	89756.4	89239.4	90337.2	91486.0	–
M8_6	94706.8	**95324.2**	95975.4	95314.5	96021.3	96629.4	97875.4
M8_7	82186.9	**83983.8**	85230.9	83777.2	85218.1	86951.5	–
M8_8	77887.6	**78744.7**	80273.3	78514.4	79220.3	80530.7	–
M8_9	90488.9	**91882.2**	92763.4	92875.6	93309.6	93736.4	93951.9

Table 2.6: Quality of solutions with $t_{\max} = 600\,\mathrm{s}$ for M8 dataset.

inst.	best BaB	avg BaB	worst BaB	best PH	avg PH	worst PH	avg MILP
M8_0	85444.0	**85955.1**	86378.4	87365.4	90258.8	94670.5	–
M8_1	86399.4	**86704.5**	86934.8	87999.7	88236.5	88488.0	89182.7
M8_2	88623.5	**89130.5**	89531.9	89939.7	90600.3	91804.2	89744.0
M8_3	82293.2	**82401.4**	82666.2	83224.7	83709.9	84741.3	83621.6
M8_4	76580.5	77103.6	77657.8	77058.9	77400.7	77723.0	**76582.8**
M8_5	87983.9	**88554.0**	89023.2	89369.4	89873.9	90545.3	–
M8_6	94382.3	94983.8	95248.7	95638.5	95836.3	96134.6	**94918.1**
M8_7	81745.9	**82523.3**	83301.7	83016.5	84881.7	86266.5	–
M8_8	77600.6	**77939.1**	78310.9	78465.8	78859.4	79342.4	–
M8_9	90531.8	**91020.7**	91755.9	91960.6	92942.0	93654.1	92521.9

Table 2.7: Quality of solutions with $t_{\max} = 1\,\mathrm{h}$ for M8 dataset.

In case of the robotic cells with 12 robots (see Table 2.8), the Branch & Bound algorithm was able to find a feasible solution in 4 of 10 cases compared to the heuristic that found a solution in 8 cases. As a result, the heuristic seems to be slightly more powerful in providing feasible solutions. The MILP solver, however, did not find any feasible solution for L12 dataset.

To show that the proposed lower bound based on convex envelopes is very tight, it was compared with the bounds obtained by the MILP solver that solved the formulation presented in Section 2.4. The results for M8 dataset are listed in Table 2.9 where the superscript of LB determines whether it is a lower bound in the *root* node or the *best* proved lower bound after one hour. Lower bound $\mathrm{LB}^{\mathrm{root}}_{\mathrm{MILP}}$ is obtained by solving the original MILP problem with the relaxation on integrality. On the one hand, the experiment revealed that even the initial bound of the Branch & Bound algorithm is significantly better than the best proved lower bound of the parallel MILP solver. On the other hand, the subsequent improvements during the course of the algorithm are negligible due to the depth search strategy and a relatively small optimality gap between the best solution and

inst.	BaB	PH
L12_0	–, –, –	–, –, 204464
L12_1	172054, 170805, 171619	–, 177731, –
L12_2	–, –, –	211019, –, 211377
L12_3	–, 188905, 191238	–, –, 188527
L12_4	172193, 174205, 173080	187163, –, 171860
L12_5	–, –, –	–, –, –
L12_6	–, –, –	–, –, –
L12_7	–, –, –	190060, 202737, –
L12_8	–, 204405, –	218844, –, –
L12_9	–, –, –	173743, 176969, 176567

Table 2.8: Quality of solutions with $t_{\max} = 3\,\text{h}$ for L12 dataset.

inst.	$\text{LB}^{\text{root}}_{\text{BaB}}$	$\text{LB}^{\text{best}}_{\text{BaB}}$	$\text{LB}^{\text{root}}_{\text{MILP}}$	$\text{LB}^{\text{best}}_{\text{MILP}}$
M8_0	75805.4	75925.3	4069.9	60950.0
M8_1	79115.4	79234.2	6155.8	67257.5
M8_2	82072.5	82162.1	6119.7	71255.0
M8_3	77268.8	77373.5	4095.5	66962.3
M8_4	70324.0	70450.5	2182.3	57265.8
M8_5	81477.6	81600.1	5155.5	71969.6
M8_6	86625.4	86706.5	9921.6	74089.5
M8_7	75140.7	75245.1	4161.1	63365.5
M8_8	71355.6	71453.6	3507.2	59463.3
M8_9	83673.6	83803.5	7931.1	71798.0

Table 2.9: Comparison of lower bounds for M8 dataset.

the lower bound. The results for S5 and L12 datasets yield similar findings, therefore, they are not listed.

The last experiment shows how the robot cycle time, which was scaled by 1.0, 1.1, and 1.2 factors, respectively, influences the performance of the heuristic and MILP solver. The summary of results for the S5 dataset is in Table 2.10, where each figure is the average criterion value from 10 runs with $t_{\max} = 600\,\text{s}$. The outcomes indicate that the heuristic outperforms the MILP solver if the cycle time is tight, i.e., a feasible solution is hard to find due to the limited time for the movements and operations. However, if the cycle time is prolonged, then the MILP solver gets ahead because its ability to find optimal solutions by a systematic search becomes dominant for the given time limit.

The approach proposed in this study cannot be directly compared with existing works [57, 53, 56] since the robot cycle time is considered instead of the work cycle time. However, bigger instances were solved compared to the aforementioned works that considered one to four robots per robotic cell. Besides, our approach deals with the additional optimization aspects, such as the robot power-saving modes and locations (robot positions).

	$\overline{C_T}$		$1.1 * \overline{C_T}$		$1.2 * \overline{C_T}$	
inst.	PH	MILP	PH	MILP	PH	MILP
S5_0	40717	**40355**	40435	**40022**	41381	**40895**
S5_1	31198	**30812**	29410	**29269**	29433	**29239**
S5_3	**47579**	–	**44177**	44338	43846	**43284**
S5_4	**47749**	–	**43993**	44233	43577	**43400**
S5_5	**41730**	–	38248	**38227**	37644	**37566**
S5_6	**34880**	34980	32359	**32186**	31744	**31630**
S5_7	**42560**	42619	42230	**41867**	42737	**42571**
S5_8	39360	**38761**	38407	**38045**	39051	**38432**
S5_9	38935	**38412**	38769	**38233**	39695	**39302**

Table 2.10: Dependence of the quality of solutions on the cycle time.

2.13 Case Study from Škoda Auto

This case study shows a potential impact of the optimization on the energy consumption of existing robotic cells by considering a long-operating robotic cell from Škoda Auto (see Figure 2.15 for a screenshot from the simulation), in which a part of an automotive body is welded, glued, and assembled by 6 industrial robots with a robot cycle time of 56 seconds. The timing of individual robotic operations was obtained from robotic programs.

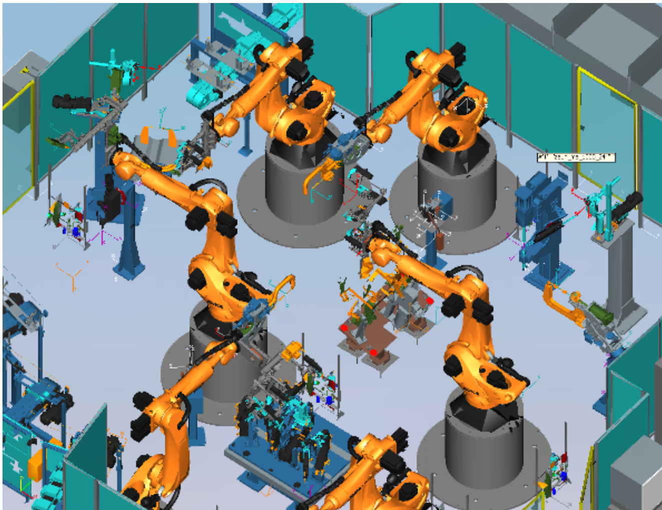


Figure 2.15: Robotic cell from Škoda Auto.

An alternative way is to measure and subsequently identify the movements. The optimized speed of movements can be easily entered into the robotic programs. The energy function f_e^t of a trajectory was fitted from the points obtained from simulations in Siemens Tecnomatix Process Simulate

(digital factory software), in which the robot controller supported the calculation of the energy consumption of movements according to the Realistic Robot Simulation standard. It is also possible to carry out the measurements with a physical robot to obtain the energy functions; however, this is impractical in existing robotic cells. To ensure the repeatability of the production process in terms of output quality, the welding, gluing, and assembling operations remained the same, i.e., the fixed duration and energy consumption were extracted from the measured power profiles. Only the robot speeds and power saving-modes (at home position) were addressed in the optimization because minimal intervention is desirable for existing robotic cells. More information about the structure of the robotic cell, the timing, and synchronizations can be obtained from instance files available on `https://github.com/CTU-IIG/GeneratorOfRoboticCells/tree/master/datasets`.

Based on the results, it was estimated that the original energy consumption of 500 kJ (maximal speeds, without power-saving modes) could be decreased to 391 kJ (reduced speeds, with power-saving modes) per cycle, resulting in about 20 % energy savings. The power-saving modes of robots saved about 2.4 % of energy, whereas the remaining savings were attributed to the optimization of speeds. If the cycle time is extended to 70 s and 80 s, then the application of energy-saving modes would improve the consumption by about 6.8 % and 12 %, respectively, compared with their nonuse. This finding may be particularly useful during over production or production cuts. Note that the inclusion of the bus power-off mode is not always straightforward because it requires interaction between the robot and a superior controller, which may not be ready in existing cells. However, the effort to implement such interaction is not high either.

The outcomes of the optimization were used to modify robot programs, and the measurements of the robotic cell confirmed the saving of 20 % of energy. This supports the claim that significant energy cuts are achievable for existing robotic cells and that even more can be expected for planned robotic cells that will fully utilize the potential of the optimization algorithms.

2.14 Conclusion

Energy optimization is undoubtedly a current and important problem for the industry since such optimization could lead to a significant decrease in costs. Besides being able to save money, an involved company could also improve its green credentials and become more competitive.

This chapter presents a holistic approach to the energy optimization of robotic cells that considers many optimization aspects. A universal mathematical model for describing robotic cells is proposed, from which is derived an MILP formulation that is directly solvable by MILP solvers. However, these solvers can solve only small instances. Therefore, the parallel Hybrid Heuristic and Branch & Bound algorithm are devised for instances with up to 12 robots. Both approaches scale almost linearly up to 12 cores; thus, significant acceleration is achievable on modern processors. Moreover, they use a specialized Gurobi simplex method for piecewise linear convex functions that is significantly faster than broadly used simplex methods.

The heuristic proved to be strong in finding feasible solutions since it solved 8 of 10 instances with 12 robots (L12 dataset). In case of small instances, the heuristic enables fast reoptimization since good feasible solutions are found in a fraction of time compared to the MILP solver.

The Branch & Bound algorithm combines the benefits of the heuristic and MILP solver, since it is very efficient in finding near-to-optimal solutions in a short time. This is possible thanks to the tight lower bound based on convex envelopes and the Deep Jumping approach that searches promising immersions in the tree. The experimental results revealed that the algorithm clearly outperforms the MILP solver and finds better solutions than the heuristic if the time limit is one hour.

The proposed approach was used to optimize the existing robotic cell from Škoda Auto, the measurements of which confirmed the energy savings of up to 20 % merely by changing the robot speeds and applying power-saving modes. In order to simplify optimization of other industrial robotic cells, our algorithms are being integrated into the industrial software.

Chapter 3

Project Scheduling on Graphics Cards

Resource Constrained Project Scheduling Problem (RCPSP), which has a wide range of applications in logistics, manufacturing and project management [21], is a universal and well-known problem in the operations research domain. The problem can be briefly described using a set of activities and a set of precedence constraints describing the relationships among activities. Each activity requires a defined amount of the resources and every resource has a limited capacity. The objective is to find the best feasible schedule according to a criterion. The RCPSP was proved to be NP-hard in the strong sense when the criterion is the makespan [6]. For that reason, only small instances (approximately up to 30 activities) can be reliably solved by exact methods like Branch & Bound [16]; therefore a heuristic or a meta-heuristic is required to solve the problem satisfactorily. In this work, the parallel Tabu Search heuristic is designed and implemented to solve the RCPSP on graphics cards.

The heuristic is the core part of a typical optimization process as illustrated in Figure 3.1. It starts with the analysis of the manufacturing process, from which the data are extracted and transformed to the desired form such as a graph and table. The created optimization problem is solved by the Tabu Search heuristic, and the resulting schedule is used to optimize the production. The rest of the introduction presents the motivation for the parallel computing on graphics cards.

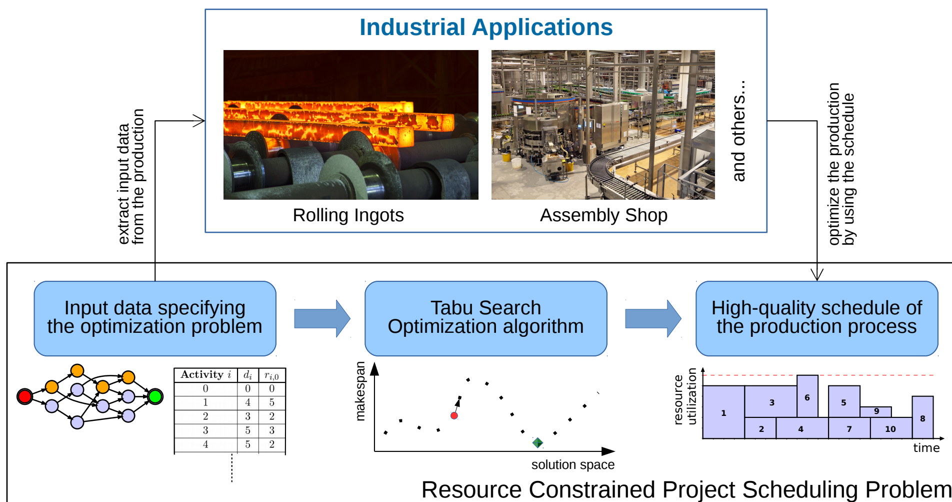


Figure 3.1: Integration of the heuristic into the optimization process.[1]

[1]Embedded photographs: ruhmal/Shutterstock.com and ID1974/Shutterstock.com

In recent times, there is an increased interest in using graphics cards to solve difficult combinatorial problems (e.g. [18, 40, 10]), since a modern graphics card is usually much more powerful than a current multi-core Central Processing Unit (CPU). Although the graphics cards have some restrictions (e.g., high-latency global memory access), the new Graphics Processing Unit (GPU) architectures like Kepler and Fermi can significantly reduce these bottlenecks. As a consequence modern GPUs are applicable to the problems which were solvable only on CPUs previously.

Not only the high computational power makes graphics cards attractive to researchers and practitioners, but also the mature Nvidia CUDA framework which enables us to create GPU programs in an effective and relatively easy way since it extends standard languages like C/C++ by adding GPU specific functions and language keywords. Nevertheless, the CUDA is only designed for the Nvidia graphics cards.

From the implementation point of view, there are two models. The first one is called a *homogeneous model* where all required data structures are stored in a GPU at the beginning of an algorithm and the results are read at the end of the algorithm. There is no communication between the CPU and the GPU during the computations. The second approach is a *heterogeneous model*. The main logic of an algorithm runs on the CPU and the GPU is used only for the most computationally intensive tasks. The disadvantage of the heterogeneous model is the frequent communication during computations, therefore, the communication bandwidth can state a bottleneck. However, the heterogeneous model is usually simpler to implement.

3.1 Related works

The Tabu Search (TS) meta-heuristic was proposed by Glover in 1986 [24]. Hundreds of publications have been written since that time. The basic concept of the TS meta-heuristic is clarified in Gendreau [23]. The author has described the basic terms of the TS, as a Tabu List (TL), aspiration criteria, diversification, intensification, etc.

From the Tabu Search parallelization point of view, James et al. [29] proposed a sophisticated solution. The authors use a circular buffer where the size of the buffer is equivalent to the number of the started threads (often the number of CPUs cores). Every location (i.e., an index of a thread) has different parameters (a tabu tenure, stopping criteria). At the beginning of the search, each thread initializes its location by a short TS operator, i.e., a modified version of the Taillard's robust tabu search. Then the asynchronous parallel tabu search is started. Every thread independently reads a solution and parameters from the location, possibly makes a diversifica-

tion, runs the TS operator on the solution, and writes back and sets an UPDATE flag if an improving solution is found. Subsequently, the thread location index is circularly incremented. Diversification takes place if the read solution does not have the UPDATE flag set. Every best global solution is copied to half of the locations of the circular buffer to propagate elite solutions. Since the circular buffer is shared by many threads, the access has to be as short as possible and the locations have to be protected by critical sections.

Relatively many authors tried to use a GPU for solving combinatorial problems. For example, Czapiński and Barnes [41] implemented a GPU version of TS to solve the Flowshop Scheduling Problem (FSP). The success of the implementation illustrates an achieved speedup against the CPU version. The GPU version was up to 89.01 times faster than the CPU (Intel Xeon 3.0 GHz, 2 GB memory, Nvidia Tesla C1060 GPU). Nevertheless, the quality of solutions was not investigated.

The FSP was also solved by Zajíček and Šůcha [58]. The authors implemented a GPU version of an island based genetic algorithm. Islands are used for migration of individuals among sub-populations, where each subpopulation is a subset of solutions and an individual corresponds to a specific solution. Sub-population can be evaluated, mutated and crossed over independently of other sub-populations, therefore, huge parallelization can be achieved. It should be noted that a homogeneous model was used. The maximal speedup against the CPU was 110 for 100 activities and 5 machines (AMD Phenom II X4 945 3.0 GHz, Nvidia Tesla C1060).

Czapiński [40] proposed a parallel Multi-start Tabu Search for the Quadratic Assignment Problem. The main idea is to start several parallel Tabu Search instances with different parameters and initial solutions. All Tabu Search instances should terminate approximately at the same time since the synchronization is required to get the most promising solutions. When a stop criterion is met, the modified solutions are read back and the most promising solutions are used as the initial solutions in the next run. The Tabu Search instance runs entirely on the GPU, therefore communication overheads are reduced to the minimum. The achieved results reveal the effectiveness of the implementation since Nvidia GTX 480 is up to 70 times faster than a six-core Intel Core i7-980x 3.33 GHz.

Delévacq et al. [18] used the parallel Ant Colony Optimization metaheuristic to solve the Traveling Salesman Problem. The authors implemented the Max-Min Ant System algorithm combined with the 3-opt local search using either *parallel ants* or *multiple ant colonies* parallel approach. The experiments show that Ant Colony Optimization can be effectively implemented on the GPU regarding the quality of solutions and performance.

Hofmann et al. [28] investigated the suitability of graphics cards for

genetic algorithms. The authors selected two problems to solve, namely the Weierstrass function minimization and the Traveling Salesman Problem. The first problem can be very effectively implemented on the GPU since the Weierstrass function is comprised of floating-point operations and trigonometric functions that are directly supported by the GPU hardware. The Nvidia GTX 480 graphics card was up to 210 times faster than the multi-core Intel Xeon X5650 processor. In contrast to the first problem, the second problem was not tailor-made for the GPU, therefore, the multi-core CPU was able to compete with the GPU. The authors suggest that all parts of a genetic algorithm should be performed on the Fermi or newer GPUs (i.e., homogeneous model).

Boyer et al. [8] used dynamic programming to solve the knapsack problem on a GPU. An effective data compression was proposed to reduce memory occupancy. The achieved results show that the Nvidia GTX 260 graphics card was up to 26 faster than Intel Xeon 3.0 GHz. The same problem was also solved by Mohamed et al. [34] on a GPU. The authors used a Branch & Bound algorithm to find optimal solutions.

The above mentioned combinatorial problems have something in common. The solution evaluation is quite simple since it is usually a "simple sum". On the other hand, the RCPSP requires much more complicated schedule evaluation methods and data structures.

3.2 Contribution and Outline of the Chapter

The proposed solution is the first known GPU algorithm for the RCPSP. The performed experiments revealed that the GPU outperforms the CPU version in both performance speedup and the quality of solutions. This is possible thanks to an effective schedule evaluation and a GPU-optimized Simple Tabu List. In addition, the required data transfers are reduced to the minimum due to the homogeneous model. Our parallel Tabu Search is able to outperform other Tabu Search implementations in the quality of the resulting solutions.

The chapter is structured as follows: The following section briefly introduces the CUDA platform and the architecture of Nvidia GPUs. Section 3.4 introduces the RCPSP mathematical formulation and notation. The Tabu Search meta-heuristic is briefly described in Section 3.5. Our proposed parallel Tabu Search algorithm for the CUDA platform is described in detail in Sections 3.6, 3.7, and 3.8. The performed experiments are located in Section 3.9, and the last section concludes this work. To simplify reading, a **nomenclature** is included in Appendix B.

3.3 CUDA platform

Compute Unified Device Architecture (CUDA) [45] is a general purpose parallel computing architecture that was introduced by the Nvidia corporation in 2006 to support GPU computing. Since the CUDA is created and maintained by Nvidia, only Nvidia GPUs are capable of running CUDA programs.

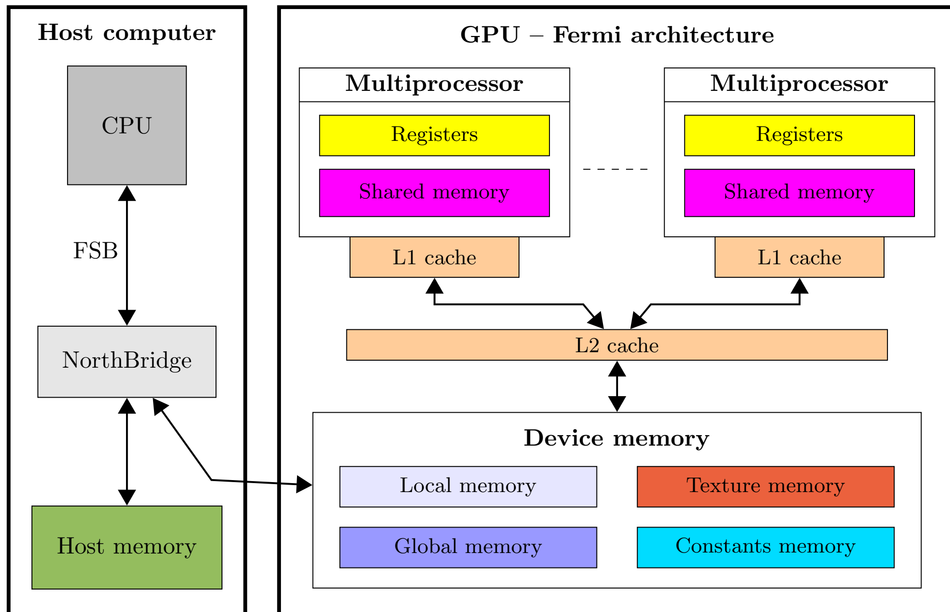


Figure 3.2: Fermi architecture – memory diagram.

Graphics cards, in general, are highly parallel devices capable of running thousands of threads at once. On the other hand, in order to take control of the high computational power of modern graphics cards, knowledge of the GPU memory model and architecture is necessary. A typical graphics card consists of a *device memory* and *streaming multiprocessors* (see Figure 3.2). The device memory is the only memory that can be used to exchange data among CPUs and GPUs. It is the largest memory that a GPU can offer since its size is usually several gigabytes but its latency is from 200 to 800 cycles, therefore effective access patterns have to be used. A streaming multiprocessor is a computational unit that schedules, plans, and controls thousands of *lightweight threads*, which are scheduled in groups of 32 threads called warps. Warps are executed on *streaming processors*.

The CUDA divides the GPU device memory into the following specialized memory types – *global memory*, *texture memory*, *local memory*, and *constant memory*. The global memory is a universal GPU memory which

can be used efficiently if a *coalescing technique* is used. The technique is based on combining multiple read or write accesses into one memory transaction if the coalescing requirements are met.

The *texture memory* is suitable for read-only data like textures and images since it is optimized for a closed spatial access pattern. The access to the memory is accelerated by a 6-8 kB cache; therefore, long read latencies can be especially reduced if data fits into the cache.

The *local memory* stores thread local variables like arrays and structures that are too big to fit in multiprocessor registers. It has similar characteristics as the global memory but with different coalescing requirements. If all threads in the warp access an array with the same relative index, then coalescing takes place.

The *constant memory* is optimized for read-only program constants. In contrast to the texture memory, the access is supposed to be random. To achieve a decent memory bandwidth and latency, it is very desirable to fit data into the 8 kB cache.

Since the device memory is slow, each multiprocessor is equipped with *registers* and a *shared memory*. The registers are the fastest GPU memory that is used for small local thread variables and constants. The number of 32-bit registers per multiprocessor depends on the CUDA capability, e.g., Kepler graphics cards have 65536 registers per multiprocessor.

The *shared memory* is usually used for data exchange among a group of threads. To utilize the full bandwidth of the shared memory, the bank conflicts have to be taken into account according to the CUDA programming guide [45]. The shared memory is almost as fast as the registers if no bank conflicts occur and its capacity is typically between 16 and 48 kB.

To saturate the graphics card, thousands of threads have to be divided among multi-processors. In the CUDA framework, a *block* is an inseparable group of threads that can communicate with each other through the shared memory. Every thread in the block is identified by its X, Y, Z coordinates. Blocks are placed into a multi-dimensional grid and loaded to the device memory from which they are read to multiprocessors. According to the thread coordinate in a block and block coordinate in a grid, calculations are divided among threads. Generally, it is convenient to launch as many threads as possible to partially hide the long latency of the device memory. The latency is also reduced by the caches of the global and local memory.

3.4 Problem Statement

The classification for the RCPSP is PS$|prec|C_{max}$ [13] according to the standard notation. A project can be described as follows: There is a set of

activities $V = \{0, \dots, N-1\}$ with durations $D = \{d_0, \dots, d_{N-1}\}$ where N is the number of activities. There are two dummy activities 0 and $N-1$ such that $d_0 = d_{N-1} = 0$. Activity 0 is a predecessor of all other activities and activity $N-1$ is the end activity of a project. A schedule of the RCPSP is represented as a vector of activities' start times $S = \{s_0, \dots, s_{N-1}\}$ where $s_i \in \mathbb{N}$. Alternatively, a schedule can be expressed as an order of activities $W = \{w_0, \dots, w_{N-1}\} \in \mathcal{W}$ where w_u is the u-th activity of the schedule and $\mathcal{W}$ is a set of all feasible solutions.

The RCPSP can be represented as Direct Acyclic Graph $G(V, E)$ where nodes V are activities and edges E are precedence relations. If there is edge $(i, j) \in E$ then $s_i + d_i \leq s_j$ since activity j has to be scheduled after activity i.

Each activity requires some amount of renewable resources. The number of project resources is denoted as M, and a set of resources capacities is $\mathcal{R} = \{R_0, \dots, R_{M-1}\}$ where $R_k \in \mathbb{N}$. Maximal resource capacity R_{max} is equal to $\max_{k=0}^{M-1} R_k$. Activity resource requirement $r_{i,k} \in \mathbb{N}$ means that activity i requires $r_{i,k} \leq R_k$ resource units of resource k during its execution. As s_i and d_i values are positive integers, the resulting schedule length C_{max} (i.e. the project makespan) will be an integer as well.

A lower bound of the project makespan can be found by neglecting resources. For each activity $i \in V$, all outgoing edges $(i, j) \in E$ are weighted by its duration d_i. The longest path from 0 to $N-1$ in graph $G(V, E)$ corresponds to the *critical path*. Its length is equal to the optimal project makespan on the condition that all resources have an unlimited capacity. All the symbols used in this chapter are listed in the **nomenclature** that is located in Appendix B.

3.4.1 Mathematical Formulation

$$\text{minimize } C_{max} \tag{3.1}$$

$$\text{s.t.} \quad C_{max} = \max_{\forall i \in V} (s_i + d_i) = s_{N-1} \tag{3.2}$$

$$s_j \geq s_i + d_i \qquad \forall (i, j) \in E \tag{3.3}$$

$$\max_{t=0}^{C_{max}} \left(\sum_{i \in F_t} r_{i,k} \right) \leq R_k \qquad \forall k \in \{0, \dots, M-1\} \tag{3.4}$$

$$F_t = \{ i \in V | s_i \leq t < s_i + d_i \}$$

The objective of the RCPSP is to find a feasible schedule W with the minimal schedule length C_{max}. The schedule length is the latest finish time of any activity (Equations (3.1) and (3.2)). Equation (3.3) ensures that all

precedence relations are satisfied. A schedule is feasible if all precedence relations are satisfied and the resources are not overloaded, i.e., the activities requirements do not exceed the capacity of any resource at any time (Equation (3.4)).

3.4.2 Instance Example

The data of an example instance are shown in Table 3.1. In the project, there are 10 non-dummy activities and 2 renewable resources with maximal capacity 6. The corresponding graph of precedences is shown in Figure 3.3. The critical path is highlighted by bold lines and its length is 16. One of the feasible solutions of the instance is the activity order $W = \{0, 1, 2, 3, 4, 6, 5, 7, 9, 10, 8, 11\}$ with $C_{max} = 22$. The resource utilization for this order is depicted in Figure 3.4.

Activity i	d_i	$r_{i,0}$	$r_{i,1}$	**Successors**
0	0	0	0	$\{1, 2\}$
1	4	5	3	$\{3, 6\}$
2	3	2	1	$\{4, 5\}$
3	5	3	2	$\{5, 10\}$
4	5	2	3	$\{7\}$
5	3	3	4	$\{8, 9\}$
6	2	4	1	$\{7, 9\}$
7	4	2	2	$\{8, 10\}$
8	2	4	5	$\{11\}$
9	3	1	2	$\{11\}$
10	4	2	2	$\{11\}$
11	0	0	0	$\{\}$

Table 3.1: Data of an example instance.

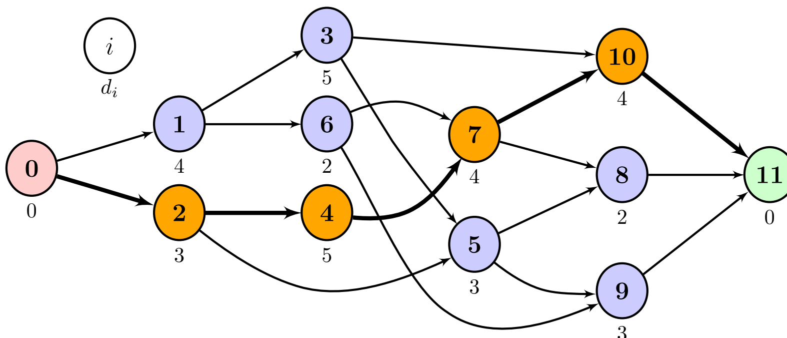


Figure 3.3: Graph of precedences for the example instance.

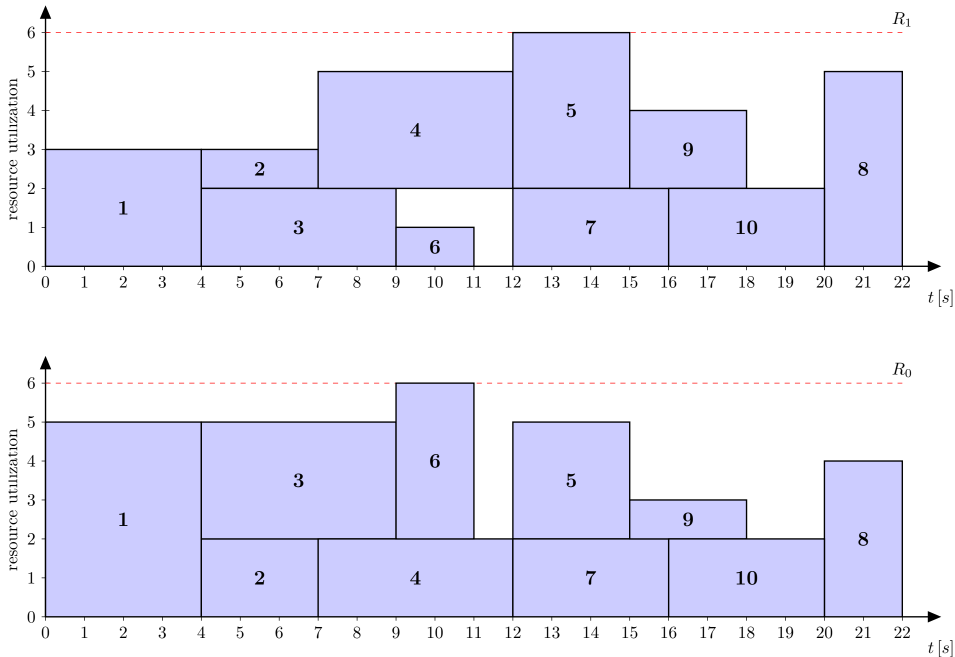


Figure 3.4: Utilization of resources for the example instance.

3.5 Outline of the Tabu Search meta-heuristic

To move through solution space $\mathcal{W}$, a transformation of the current solution to a neighborhood solution is required. This transformation is called a *move* which can be seen as a light solution modification like a swap of two elements in an order, etc.

The Tabu Search meta-heuristic was proposed by Glover [24] as an improvement of the *local search* technique [23]. A local search algorithm starts with the *initial solution* and iteratively improves this solution by applying the best neighborhood moves until a local optimum is reached, whereas the Tabu Search introduces a short-term memory called *Tabu List* which reduces the probability of getting stuck in a local optimum or plateau by forbidding the previously visited solutions. As a consequence, not only improving solutions are permitted and the search process can climb to the hills in the search space $\mathcal{W}$ if it is necessary.

Due to efficiency, the Tabu List usually contains only parts of solutions or several previously applied *moves*. As moves or parts of solutions do not have to be unique, it is possible that a forbidden move leads to the best solution. In this case, it is reasonable to permit the move since the resulting solution was not visited before. In general, exceptions allowing forbidden

moves are called *aspiration criteria* [23].

The quality of the resulting solutions can be further improved by a suitable search strategy. For example, if a current location in the solution space is promising, i.e., the best solution was found recently, then a more thorough search is performed – *intensification*. It can be accomplished by concentrating more computational power to this locality of the space. Opposite to that, if a current location is unpromising, i.e., only poor solutions were found, then *diversification* is performed. The diversification moves the current search location to another one where better solutions could be found. It is often realized by applying a few random moves.

The Tabu Search process is stopped if a stop criterion is met. The stop criterion can be the number of iterations, achieved quality of the best found solution, the maximal number of iterations since the last best solution was found, etc.

3.6 Exploration of the Solution Space

3.6.1 Creating Initial Activity Order

Our Tabu Search algorithm starts from initial feasible solution $W^{init} \in \mathcal{W}$ which is created in the following way: First of all, the longest paths in graph G from the start activity 0 to all other activities are found. The weight of each graph edge $(i, j) \in E$ is set to 1. Activities with the same maximal distance from the start activity are grouped to *levels*. The level l_k corresponds to all activities with maximal distance k from the start activity, therefore, activity 0 is at level l_0 and activity $N-1$ is at level l_{max} where subscript max corresponds with the last level number. The final feasible schedule can be created from levels such that $W = \{\{l_0\}, \ldots, \{l_{max}\}\}$. Alternative feasible schedules can be created by shuffling the activities on the same level.

3.6.2 Move Transformation

Schedule order W is changed in our Tabu Search algorithm by a *swap* move. A simple example is illustrated in Figure 3.5. Two dummy activities (0 and $N-1$) cannot be swapped due to precedence constraints, therefore, the activity at w_0 is always 0 and the activity at w_{N-1} is always $N-1$. The swap move is defined as swap(u, v) where u and v are swapped indices. As swap(u, v) modifies a schedule in the same way as swap(v, u) only swaps with $u < v$ are taken into account without loss of generality.

Let $W \in \mathcal{W}$ be a feasible activity order. A feasible order means that there is not a violated precedence relation. A *feasible move* is a move which does not violate any precedence relation, and therefore if this move is applied

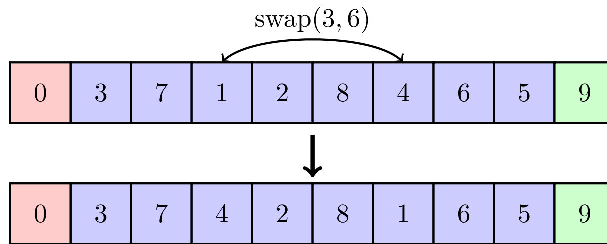


Figure 3.5: Example of the swap move.

to a feasible schedule, then the modified schedule will be feasible as well. Move swap(u, v) is feasible if the following equations are satisfied.

$$(w_u, w_x) \notin E \qquad \forall x \in \{u+1, \ldots, v\} \tag{3.5}$$

$$(w_x, w_v) \notin E \qquad \forall x \in \{u, \ldots, v-1\} \tag{3.6}$$

The First Equation (3.5) means that there are no edges from activity w_u to the activities at indices from $u+1$ to v. If there is an edge, then activity w_u cannot be moved to position v without a precedence violation. In a similar way, Equation (3.6) states that activity w_v cannot be moved to index u if there is a precedence relation that becomes violated.

3.6.3 Neighborhood Generation

Full neighborhood $\mathcal{N}_{full}(W) \subseteq \mathcal{W}$ of schedule W is a set of schedules obtained by applying all feasible moves. Since the full neighborhood is usually too large to be evaluated in a reasonable time only a subset of the neighborhood is usually taken into account. Such a subset will be called as a *reduced neighborhood* denoted $\mathcal{N}_{reduced}(W)$. In the reduced neighborhood, moves are restricted to all swap(u, v), where $u < v$ and $|v - u| \leq \delta$. Value δ is the maximal distance between the swapped activities in order W. The size of the neighborhood $|\mathcal{N}_{reduced}(W)|$ is parametrized by δ.

There are two reasons why only feasible moves are applied. The neighborhood size is reduced without noticeable deterioration of the project makespan, and there is no need to check the feasibility of schedules.

3.6.4 Filtering Infeasible Moves

In order to saturate a GPU, feasible schedules in $\mathcal{N}_{reduced}$ should be evaluated in a parallel way by dividing the schedules equally among the threads. Since the evaluation of a schedule is much more time-consuming than checking whether a move is feasible, it is advantageous to filter out all infeasible

moves before the neighborhood evaluation. It reduces the branch divergency of warps; hence the overall performance of the resources evaluation is improved.

In Algorithm 1 is shown how infeasible moves are filtered out from the neighborhood. The filter works in two phases since it was discovered that it is more effective due to the lower branch divergency than to filter out all the infeasible moves at once. In the end, only part of the array with feasible moves is taken into account in the neighborhood evaluation.

Algorithm 1 Removing infeasible moves from the reduced neighborhood.

Require: $\mathcal{N}_{reduced}(W)$
Ensure: It filters out infeasible moves from the reduced neighborhood.
1: Let *MovesArray* be an array with all potential swaps in $\mathcal{N}_{reduced}(W)$.
2: All moves not satisfying Equation (3.5) are removed, i.e., set empty.
3: Reorder *MovesArray* such that all empty moves are in the end.
4: Remove moves which do not satisfy Equation (3.6).
5: Move all feasible moves to the beginning of *MovesArray*.

3.6.5 Simple Tabu List and Cache

The tabu list in [41] is not suitable for a GPU since it is necessary to go through all moves in the tabu list to decide whether a move is in the tabu list or not. As a consequence, it places higher demands on the device memory bandwidth. To avoid this a simple and efficient Simple Tabu List (STL) with constant algorithmic complexity is proposed. Access to the STL is performed like access to a circular buffer. Its size is fixed and is equal to $|tabuList|$. In our case, the STL stores *swap* moves. Each swap(u, v) is stored to the STL as a pair of swapped indices (u, v). A special value is used for an empty move, e.g., swap$(0, 0)$. At each iteration of the TS algorithm, one move is added (see Algorithm 3) and the oldest one is removed if the STL is full.

Algorithm 2 Check if a move is in the STL.

Require: *tabuCache* – STL cache.
Require: (u, v) – Swap move indices.
Ensure: It returns true if the move is in the STL, otherwise false.
1: **return** $tabuCache[u, v]$

The Tabu Cache (TC) was proposed for effective checking if the move is in the STL. It is illustrated in Algorithm 2. Checking if a swap is in the STL occurs much more often than adding a new move since a move is added only once per iteration and check if the move is in the STL occurs for every neighborhood schedule. The TC is implemented as a 2-dimensional $N \times N$

boolean array which is synchronized with the STL. A check if a move is in the STL requires one read operation, thus, the required memory bandwidth is very low. It is obvious that a check if the move is in the STL has $\mathcal{O}(1)$ algorithmic complexity.

Algorithm 3 Add a move to the STL.

Require: *tabuList* – Fixed size array.
Require: *tabuCache* – STL cache.
Require: *writeIndex* – Current write position.
Require: (u, v) – Swap move indices.
Ensure: Add move to STL and update TC.
1: $(u_{old}, v_{old}) = tabuList[writeIndex]$
2: $tabuCache[u_{old}, v_{old}] =$ **false**
3: $tabuList[writeIndex] = (u, v)$
4: $tabuCache[u, v] =$ **true**
5: $writeIndex = (writeIndex + 1)\ \%\ |tabuList|$

3.7 Schedule Evaluation

During the evaluation of W, precedence relations and resource constraints have to be taken into account to calculate activities start times s_i and C_{max}. The precedence earliest start time es_i^{prec} of activity i can be calculated as $\max_{\forall(j,i)\in E}(s_j + d_j)$, where j are predecessors of activity i. The resources earliest start time es_i^{res} can be computed using either a *time-indexed* or *capacity-indexed* resources evaluation algorithm. The capacity-indexed algorithm is a completely new approach to the best of our knowledge, whereas the time-indexed algorithm is well-known [31]. The names of the algorithms were selected with respect to the indexed unit of a resource state array. According to a heuristic, the probable faster resources evaluation algorithm is selected in the schedule evaluation procedure. Having considered both precedence and resource constraints the final earliest start time is $es_i = \max(es_i^{prec}, es_i^{res})$.

3.7.1 Capacity-Indexed Resources Evaluation

Required Data Structures

The most difficult part of the project makespan evaluation is the computation of the activities' start times with respect to the resource capacities. In our approach, the evaluation of resources requires one array c_k with length R_k per resource k. Value $c_k[R_k - r_{i,k}]$ corresponds to the earliest resource start time of activity i with resource requirement $r_{i,k} > 0$ on resource k. At the start of the evaluation, all the resources arrays are set

to zeros. After that, activities are added one by one to a schedule according to W and arrays are updated with respect to the activity requirements and precedences. The resources arrays are ordered descendly, i.e., $c_k[R_k - l] \leq c_k[R_k - l - 1] \mid \forall l \in \{1, \ldots, R_k - 1\}$. The state of resources is represented as a set $C = \{c_0, \ldots, c_{M-1}\}$.

The Earliest Resources Start Time

Resource earliest start time $es_i^{res} \in \mathbb{N}$ of activity i with respect to an occupation of resources can be calculated using Equation (3.7). It is guaranteed that resources are not overloaded if activity i start time $s_i \geq es_i^{res}$. Final activity start time s_i can be more delayed due to the precedence relations.

$$es_i^{res} = \begin{cases} \max\limits_{k \in \{0,\ldots,M-1\}:\, r_{i,k} > 0} c_k[R_k - r_{i,k}] & \exists r_{i,k} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

Update of the Resources Arrays

If activity i is added to the schedule, resources arrays C have to be updated by Algorithm 4. Each resource array c_k is updated individually (line 1). Value *requiredEffort* $= r_{i,k} \cdot d_i$ will be called *Required Resource Effort*. Activity i can be added to the schedule if and only if each resource can provide its Required Resource Effort. In other words, variable *requiredEffort* has to be decremented to zero (lines 2, 13, 19) for each resource k.

It is performed by setting $r_{i,k}$ elements of c_k to the activity finish time $s_i + d_i$ after the last $c_k \geq s_i + d_i$. Until the variable *requiredEffort* is zero, the shifted (right shift about $r_{i,k}$) copy of the original resource array c_k (original values are stored in *copy* auxiliary variable) is made. The complexity of the algorithm is $\mathcal{O}(M \cdot R_{max})$.

The algorithm is illustrated on an example in Figure 3.6. There is one resource with maximal capacity 7. Added activity i requires 3 resource units (i.e. $r_{i,k} = 3$) and its duration d_i is 3. The activity was scheduled at $s_i = 5$. The solid line corresponds to the original resource state $c_k = \{7, 7, 5, 5, 5, 5, 4\}$ and the dotted line corresponds to the updated resource $c'_k = \{8, 8, 8, 7, 7, 5, 4\}$. The activity required effort is depicted by a square with a dashed border. The positive numbers between solid and dotted lines are effort contributions when old start time (solid line) will be changed to the new start time (dotted line). It should be noticed, that the sum of all contributions is the *requiredEffort* for a given activity.

Algorithm 4 Method updates state of resources after adding activity i.

Require: $r_{i,k}, d_i, C, \mathcal{R}$
Require: *copy* – Auxiliary array (the length of the array is R_{max}).
Require: s_i – Scheduled start time of activity i.
Ensure: Update C - activity i is added.

```
 1: for (k = 0; k < M; ++k) do
 2:   requiredEffort = r_{i,k} · d_i
 3:   if (requiredEffort > 0) then
 4:     resIdx = 0; copyIdx = 0
 5:     newTime = s_i + d_i
 6:     while (requiredEffort > 0 AND resIdx < R_k) do
 7:       if (c_k[resIdx] < newTime) then
 8:         if (copyIdx ≥ r_{i,k}) then
 9:           newTime = copy[copyIdx − r_{i,k}]
10:         end if
11:         timeDiff = newTime − max(c_k[resIdx], s_i)
12:         if (requiredEffort − timeDiff > 0) then
13:           requiredEffort -= timeDiff
14:           copy[copyIdx++] = c_k[resIdx]
15:           c_k[resIdx] = newTime
16:         else
17:           c_k[resIdx] = max(c_k[resIdx], s_i)
18:           c_k[resIdx] += requiredEffort
19:           requiredEffort = 0
20:         end if
21:       end if
22:       resIdx = resIdx + 1
23:     end while
24:   end if
25: end for
```

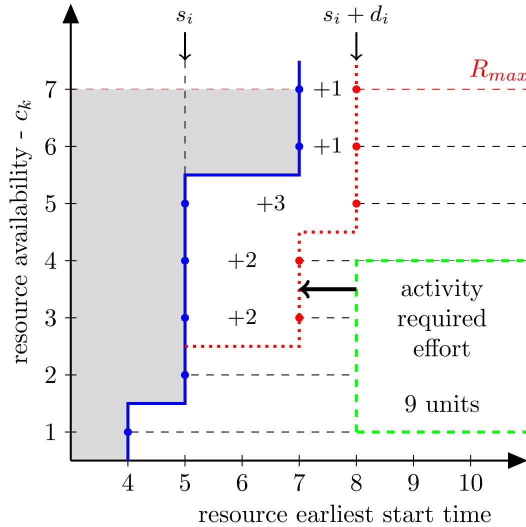


Figure 3.6: An example of the resource state update.

3.7.2 Time-Indexed Resources Evaluation

Required Data Structures

In the time-indexed evaluation algorithm, the state of each resource k is stored in array τ_k. Each element $\tau_k[t]$ corresponds to the number of available resource units that resource k can provide at time $t \in \{0, \ldots, \text{UB}_{C_{max}}\}$, where $\text{UB}_{C_{max}}$ is the upper bound of the makespan which can be calculated as, e.g., $\sum_{\forall i \in V} d_i$. Each τ_k array has initialized all its elements to the R_k value before the start of the evaluation. The state of all resources will be denoted as $T = \{\tau_0, \ldots, \tau_{M-1}\}$.

The Earliest Resources Start Time

The earliest start time of activity i can be calculated using Algorithm 5. In the algorithm, the *loadTime* variable corresponds with the number of consecutive time units in which resources are able to meet resource requirements of activity i. If $loadTime = d_i$ then a time interval into which activity i can be scheduled was found. Having considered variable t as a finish time of a candidate interval, the resulting interval is the first interval $[t - loadTime, t) \cap \mathbb{N}$ such that $loadTime = d_i$. The final earliest start time is the lower endpoint of the interval.

Algorithm 5 Algorithm calculates the earliest start time of activity i.

Require: $r_{i,k}, d_i, \mathcal{R}, \text{UB}_{C_{max}}, T$
Require: es_i^{prec} – The precedence earliest start time.
Ensure: Calculate the earliest start time es_i of activity i.

```
 1: loadTime = 0;
 2: for (t = es_i^prec; t < UB_Cmax AND loadTime < d_i; ++t) do
 3:    sufficientCapacity = true
 4:    for (k = 0; k < M; ++k) do
 5:       if (τ_k[t] < r_i,k) then
 6:          loadTime = 0
 7:          sufficientCapacity = false
 8:       end if
 9:    end for
10:    if (sufficientCapacity == true) then
11:       ++loadTime
12:    end if
13: end for
14: return t − loadTime
```

Update of the Resources Arrays

The state of resources is updated as is shown in Algorithm 6. Having scheduled activity i at s_i the τ_k arrays have to be updated in the $[s_i, s_i + d_i)$

interval. For each resource k, values in the interval are decreased by $r_{i,k}$ units.

Algorithm 6 Updating of resources after adding activity i.

Require: $d_i, T, r_{i,k}$
Require: s_i – Scheduled start time of activity i.
Ensure: It updates state of resources T.

```
for (k = 0; k < M; ++k) do
    for (t = s_i; t < s_i + d_i; ++t) do
        τ_k[t] -= r_{i,k}
    end for
end for
```

3.7.3 Schedule Evaluation Procedure

The schedule evaluation procedure is shown in Algorithm 7. Activities are read one by one from the activities order W. For each activity w_u, all its predecessors are found and the precedence relations are used to update activity w_u's precedence earliest start time $es^{prec}_{w_u} \in \mathbb{N}$ (see lines 3–6). Then the resources restrictions are checked and the start time is adjusted to $s_{w_u} = \max(es^{prec}_{w_u}, es^{res}_{w_u})$. Project makespan C_{max} is the finish time of activity $N-1$. As only feasible moves are allowed, an infeasibility test of the resulting schedules is not required.

Algorithm 7 Complete schedule evaluation.

Require: W, C, T, E
Ensure: Calculate C_{max} and the activities' start times.

```
 1: C_max = 0
 2: for (u = 0; u < N; ++u) do
 3:     es^prec_{w_u} = 0
 4:     for all ((j, w_u) ∈ E) do
 5:         es^prec_{w_u} = max(es^prec_{w_u}, s_j + d_j)
 6:     end for
 7:     es^res_{w_u} = getEarliestResourcesTime(activity w_u, es^prec_{w_u})
 8:     s_{w_u} = max(es^prec_{w_u}, es^res_{w_u})
 9:     updateResources(activity w_u, s_{w_u})
10:     Mark current activity w_u as scheduled.
11:     C_max = max(C_max, s_{w_u} + d_{w_u})
12: end for
13: return C_max
```

3.7.4 Heuristic Selection of Resources Evaluation Algorithms

Before the search is started on the GPU, the probable faster resources evaluation algorithm is heuristically selected by decision rules and the required

resources arrays are allocated. To create the rules, the JRip classifier from the Weka data-mining tool [27] was learned using pre-calculated attributes shown in Table 3.2.

Min. resource capacity:	$\min_{\forall k \in \{0,\dots,M-1\}} R_k$
Avg. resource capacity:	$\frac{1}{M} \sum_{\forall k \in \{0,\dots,M-1\}} R_k$
Max. resource capacity:	$\max_{\forall k \in \{0,\dots,M-1\}} R_k$
Avg. activity duration:	$\frac{1}{N} \sum_{\forall i \in V} d_i$
Avg. branch factor:	$\frac{\lvert E \rvert}{N}$
Critical path length:	see Section 3.4
Evaluation algorithm:	CAPACITY/TIME

Table 3.2: Attributes used for learning.

Attribute "Evaluation algorithm" determines the class, i.e., the time-indexed or capacity-indexed evaluation algorithm, to which the classifier should classify. As the final class depends on the hardware and instance parameters, it is necessary to determine the probable correct class by measuring — each evaluation algorithm was selected for a small number of

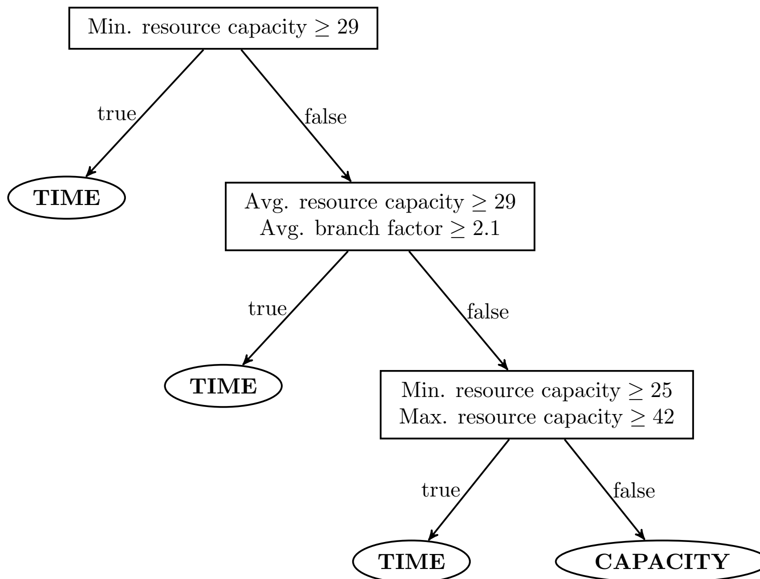


Figure 3.7: Example of the decision tree.

iterations and the faster one was selected as the desired one. The resulting rules heuristically decide which of the two algorithms should be more effective for a given instance. The rules can be transformed into a decision tree as is shown in Figure 3.7. Once the rules are created they can be applied to other similar instances without any measurable overhead as well. To show the effectivity and usefulness of the heuristic the experiments were performed in Section 3.9.

3.8 Parallel Tabu Search for the CUDA platform

Our Parallel Tabu Search for GPU (PTSG) is proposed with respect to the maximal degree of parallelization since thousands of CUDA threads are required to be fully loaded to exploit the graphics card power. In our approach, the parallelization is carried out in two ways. The first one is a parallelization performed within the scope of a block, for example, the parallel filter (see Section 3.6.4), the parallel neighborhood evaluation, and other parallel reductions. The second one is a parallelization introduced by launching many blocks on the multi-processors simultaneously.

The basic steps of the PTSG are described in Figure 3.8. First of all, an instance is read and the initial solutions are created in accordance with Section 3.6.1. After that, every second solution is improved by using the forward-backward improvement method. The method is iteratively shaking a schedule from left to right in order to make a resource profile straight as long as the schedule is getting shorter. To get more details about the method, refer to the original article by Li and Willis [35]. The created solutions are copied into a *working set*, i.e. a set of shared solutions. The best solution in the working set will be called the *global best solution* and its makespan will be denoted as C^*_{max}. Furthermore, the block's Tabu Lists and Tabu Caches are initialized and auxiliary arrays such as τ_k, and c_k are allocated. Having had prepared required data structures, the host is ready to launch the kernel.

In the GPU part, every block is an independent Tabu Search instance communicating with the others through the global memory (see Section 3.8.1). At the beginning, every block reads an initial solution from the working set. After that, the search is started for a specified number of iterations of the main loop. In the main loop, the neighborhood is generated, evaluated and the best move m^* is selected, applied and added into the STL. Move m^* leads to the best criterion improvement or to the smallest criterion deterioration. This move cannot be in the STL with one exception — the move leads to the global best solution. At the end of an iteration, the solutions are exchanged through the working set if the communication conditions are

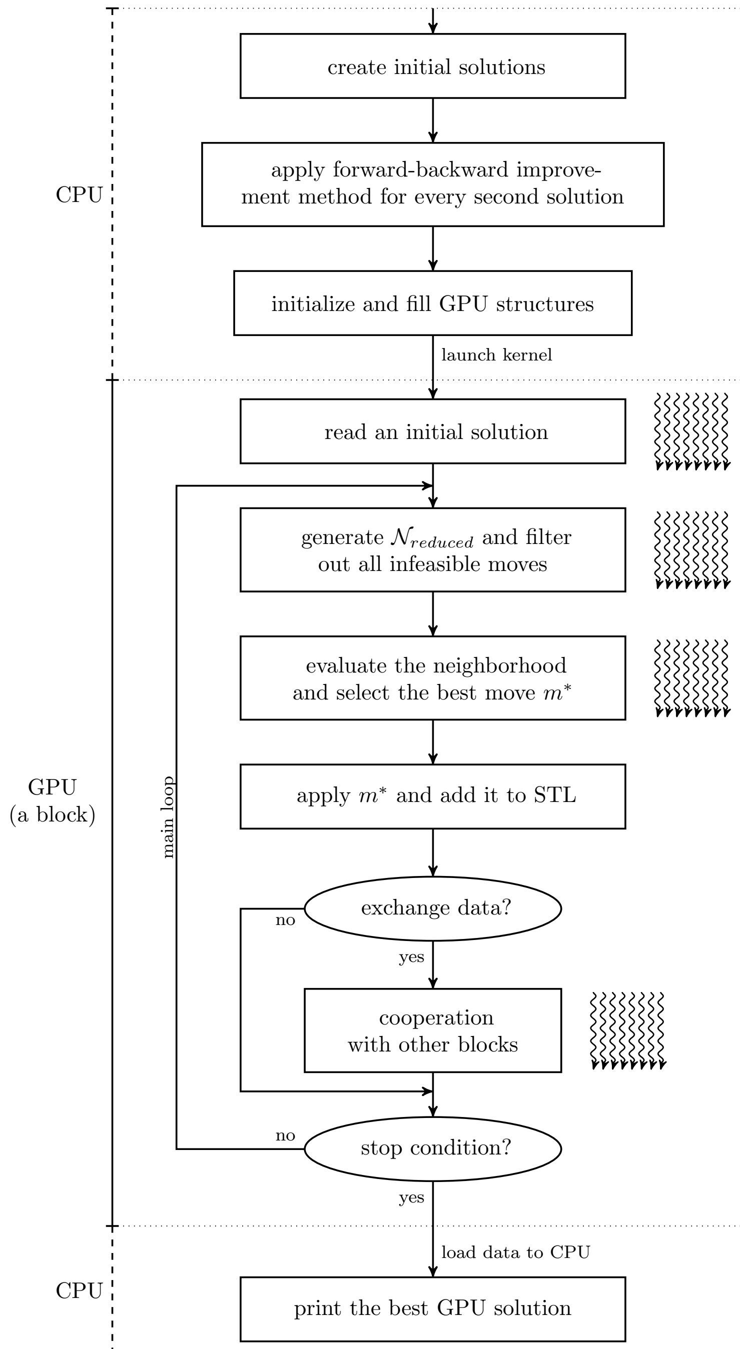


Figure 3.8: Parallel Tabu Search for the CUDA platform.

satisfied (see Section 3.8.1). The search is stopped if the specified number of iterations was achieved or C^*_{max} is equal to the length of a critical path.

After the termination of the kernel, the best global solution is copied from the global memory to the host memory. The solution is printed and all allocated data structures are freed.

3.8.1 Block Cooperation and Distribution of Iterations

To assure the high-quality solutions, the cooperation among Tabu Search instances is accomplished by exchanging solutions through the working set F that has a fixed number of solutions $|F|$. Each solution $k \in F$ consists of the order of activities W^k, makespan C^k_{max}, the tabu list, and iterations counter IC^k. The solution exchange takes place if the last read solution has not been improved for more than $\mathcal{I}_{assigned}$ iterations or the block found an improvement of the last read solution. The block writes the best found solution to F if it improves the last read solution and reads the next solution from F. Since the working set could be accessed by many blocks at the same time, it is necessary to use read/write locks in order to maintain data integrity. The cooperation among Tabu Search instances was inspired by James et al. [29].

After the block has read a solution from the working set, it is checked whether the solution was not read more than Φ_{max} times without being improved. If it is the case, a small number Φ_{steps} of random feasible swaps is applied to randomize the read solution — diversification. After that, the read solution $k \in F$ has assigned the number of iterations $\mathcal{I}_{assigned}$ according to the following equation.

$$\mathcal{I}_{assigned} = \left\lfloor \overbrace{\frac{1}{5}\frac{\mathcal{I}_{block}}{\mathcal{I}_{total}}}^{quantity} \left(\overbrace{0.8e^{-100\left(\frac{C^k_{max}}{C^*_{max}}-1\right)}}^{quality} + \overbrace{0.2e^{-4\left(\frac{IC^k}{\mathcal{I}_{block}}\right)}}^{intactness} \right) \right\rfloor \tag{3.8}$$

$\mathcal{I}_{block}$ is the number of iterations assigned to each Tabu Search instance and $\mathcal{I}_{total}$ is the total number of iterations calculated as $\mathcal{I}_{block}B$, where B is the number of launched blocks. The part denoted as *quantity* corresponds to the maximal number of iterations which can be assigned to read solution k. It is ensured, that at least 5 solutions are read from the working set by each block. The *quality* part takes into account the quality of read solution k. It is obvious that the high-quality solutions are preferred to poor ones — intensification. And the last part *intactness* guarantees that each solution $k \in F$ has been given some iterations to prove the quality.

3.8.2 Memory Model

The placement of the data structures is a crucial task highly influencing the effectiveness of the GPU program, therefore, each decision should be considered thoroughly with respect to the access pattern, required bandwidth and data visibility (local or shared data). In the shared memory, current block order W_{block}, durations of activities D, and auxiliary arrays are stored. Although D is a read-only array which could be located in the constants memory, it was moved to the shared memory due to the higher bandwidth. The texture memory is used for storing read-only data as $r_{i,k}$ values and predecessors of the activities. In the local memory, private data structures of each thread are located, i.e., resources arrays c_k, τ_k, and start times of activities S. The long latency of the memory is compensated by using a partial coalescing since the arrays are often accessed at the same relative indices as the majority of threads evaluate similar schedules (W_{block} + swap move). Finally, the global memory is employed to store the working set F.

3.9 Experimental Results

Experiments were performed on the AMD Phenom(tm) II X4 945 server (4 cores, 8 GB memory) equipped with a mid-range Nvidia Geforce GTX 650 Ti (1 GB, 768 CUDA cores, 4 multiprocessors) graphics card. The testing environment was the Windows Server 2008 with an installed CUDA toolkit (version 5.0.35) and Microsoft Visual Studio 2010.

The sequential CPU version of the algorithm corresponds to one Tabu Search instance with the exception that solutions are not interchanged ($|F| = 1$ and $B = 1$). Instead of using the selection heuristic (see Section 3.7.4) the faster evaluation algorithm was selected dynamically by periodic measuring (once per 1000 iterations). The parallel CPU version differs from the sequential version in the neighborhood evaluation. The feasible schedules in the neighborhood are divided among CPU threads to reduce

	J30	**J60**	**J90**	**J120**
N	30+2	60+2	90+2	120+2
M	4			
$\|dataSet\|$	480	480	480	600
δ	30	60	60	60
$\|tabuList\|$	60	250	600	800
Φ_{steps}	20			
Φ_{max}	3			
$\|F\|$	16			

Table 3.3: PTSG parameters and datasets information.

evaluation time. Both the CPU and GPU versions were fully optimized with respect to memory access patterns and hardware architecture (cache sizes). To fully saturate the GPU the maximal number of available registers per CUDA thread was limited to 32 due to the possibility to launch 4 blocks on a multiprocessor at once (altogether 16 blocks on the GPU), where each block has 512 CUDA threads.

To evaluate the performance and the quality of resulting solutions, the benchmark using the well-known J30, J60, J90 and J120 datasets was performed. The number of instances in a dataset will be denoted as $|dataSet|$. The selected PTSG parameters and datasets information are stated in Table 3.3.

	CPU			**GPU**		
$\mathcal{I}_{total}$	*CPM dev*	*OPT dev*	*Best_sol*	*CPM dev*	*OPT dev*	*Best_sol*
10000	13.43	0.04	471	13.41	0.02	473
20000	–	–	–	13.38	0.01	478

Table 3.4: Quality of solutions — J30.

	$\mathcal{I}_{total}$	*Comp_time*	*Sched_sec*	*Speedup*
CPU seq.	10000	1255	126400	1.00
CPU par.	10000	343	478300	3.65
GPU	10000	176	985543	7.12
GPU	20000	306	1120900	–

Table 3.5: Performance comparison — J30.

The results for the J30 dataset are shown in Tables 3.4 and 3.5. The *CPM dev* and *OPT dev* values are the average percentage distance from the critical path length and the average percentage distance from the optimal makespan respectively. *Best_sol* states the number of optimal solutions which have been proved to be optimal according to the results in the PSPLIB homepage — `http://www.om-db.wi.tum.de/psplib/`. The *Comp_time* is the total runtime stated in seconds and *Sched_sec* is the number of evaluated schedules per second. It is obvious that the GPU version is able to achieve a similar quality of solutions in terms of *CPM dev* as the CPU version. Having had $\mathcal{I}_{total}$ doubled, the GPU version found 478 optimal solutions from the 480 solutions in the dataset. From the performance point of view, Table 3.5 reveals a significant improvement in computational time if parallelization is performed. For example, the parallel CPU version is 3.65 times faster than the sequential CPU version and the GPU is almost 2 times faster than the parallel CPU version. If $\mathcal{I}_{total}$ is increased to 20000, the GPU is still slightly faster and achieves better quality solutions.

	CPU			GPU		
$\mathcal{I}_{total}$	*CPM dev*	*UB dev*	*Best_sol*	*CPM dev*	*UB dev*	*Best_sol*
10000	11.13	0.51	380	11.22	0.57	375
20000	–	–	–	11.08	0.47	388
30000	–	–	–	10.99	0.41	394
50000	–	–	–	10.91	0.36	394

Table 3.6: Quality of solutions — J60.

	$\mathcal{I}_{total}$	*Comp_time*	*Sched_sec*	*Speedup*
CPU seq.	10000	7094	59700	1.00
CPU par.	10000	1732	248700	4.10
GPU	10000	257	1733800	27.60
GPU	20000	485	1818600	–
GPU	30000	709	1861900	–
GPU	50000	1164	1879400	–

Table 3.7: Performance comparison — J60.

For the J60 dataset, the results are shown in Tables 3.6 and 3.7, where *UB dev* is the average percentage distance from the best currently known upper bounds. The CPU version gives slightly better solutions for 10000 iterations, but on the other hand, if the GPU is given 20000 iterations the quality of solutions is comparable with the CPU version and the GPU is still 3.56 times faster than the parallel CPU version. The lower quality of GPU solutions for the same $\mathcal{I}_{total}$ is probably caused by wasting work when many parallel Tabu Search instances have read the same solution from the working set and only one writes the best improvement. It can be noted, that the parallel CPU version is more than 4 times faster than the sequential one. The reason of that is either better cache utilization or the AMD True Core Scalability technology.

The results in Tables 3.8 and 3.9 for the J90 dataset show that the GPU is better utilized for bigger instances and the GPU is more than 10 times faster than the parallel CPU version for the same number of iterations. The same quality of solutions is achieved 5.4 times faster on the GPU.

Results for the J120 dataset are shown in Tables 3.10 and 3.11. It can be noted that the quality of GPU solutions is substantially lower for 10000 iterations. The GPU requires about 50000 iterations to achieve the quality of the CPU solutions. On the other hand, the GPU can compete with the CPU since 50000 iterations are performed 2.5 times faster than 10000 iterations for the parallel CPU version. The GPU evaluates more than one million schedules per second, whereas the CPU evaluates one hundred thousand.

	CPU			**GPU**		
$\mathcal{I}_{total}$	*CPM dev*	*UB dev*	*Best_sol*	*CPM dev*	*UB dev*	*Best_sol*
10000	10.81	0.92	367	11.04	1.09	365
20000	–	–	–	10.82	0.93	371
30000	–	–	–	10.73	0.86	373
50000	–	–	–	10.56	0.74	375

Table 3.8: Quality of solutions — J90.

	$\mathcal{I}_{total}$	*Comp_time*	*Sched_sec*	*Speedup*
CPU seq.	10000	20294	36000	1.00
CPU par.	10000	5001	148300	4.06
GPU	10000	475	1599600	42.70
GPU	20000	923	1632000	–
GPU	30000	1348	1660700	–
GPU	50000	2221	1674000	–

Table 3.9: Performance comparison — J90.

	CPU			**GPU**		
$\mathcal{I}_{total}$	*CPM dev*	*UB dev*	*Best_sol*	*CPM dev*	*UB dev*	*Best_sol*
10000	33.41	2.70	215	34.67	3.50	194
20000	–	–	–	34.04	3.11	208
30000	–	–	–	33.66	2.85	213
50000	–	–	–	33.54	2.76	222

Table 3.10: Quality of solutions — J120.

	$\mathcal{I}_{total}$	*Comp_time*	*Sched_sec*	*Speedup*
CPU seq.	10000	148170	25700	1.00
CPU par.	10000	35812	107200	4.14
GPU	10000	2938	1340400	50.40
GPU	20000	5742	1351900	–
GPU	30000	8513	1353300	–
GPU	50000	14160	1347800	–

Table 3.11: Performance comparison — J120.

The quality of the solutions is compared with the existing solutions for the RCPSP in Table 3.12. Our proposed PTSG outperforms other Tabu Search implementations with respect to the quality of solutions. For example, Artigues' Tabu Search [3] has been given at least 11000 iterations for the J120 dataset and achieves 36.16 % CPM dev. Having had 10000 iterations, the proposed PTSG reaches 33.41 % and 34.67 % for the CPU and GPU respectively. In addition, the proposed PTSG can be just as good as other heuristic approaches like Ant Colony Optimization and Simulated Annealing. On the other hand, the state-of-the-art random-key genetic algorithms give even better solutions than the PTSG.

	CPM dev			
Algorithm and reference	**J30**	**J60**	**J90**	**J120**
Genetic Algorithm - Gonçalves et al. [25]	13.38	10.49	-	30.08
This work - Nvidia Geforce GTX 650 Ti	13.38	10.91	10.56	33.54
This work - AMD Phenom(tm) II X4 945	13.43	11.13	10.81	33.41
CARA algorithm - Valls et al. [52]	13.46	11.45	11.12	34.53
Ant Colony Optimization - Zhou et al. [59]	-	11.42	-	35.11
Tabu Search - Artigues et al. [3]	-	12.05	-	36.16
Simulated Annealing - Bouleimen and Lecocq [7]	-	11.90	-	37.68

Table 3.12: Comparison with other heuristics.

From the performance point of view, it is difficult to compare since the different algorithms and hardware architectures were used for experiments. For example, Artigues's Tabu Search requires 67 s per J120 instance on average. The testing configuration was not stated. The PTSG requires 4.9 s (10000 iterations) on the mid-range GPU with the substantially higher quality of solutions. The genetic algorithm by Gonçalves et al. [25] takes 180 s per J120 instance on average on the Intel Core 2 Duo 2.4 GHz processor.

3.9.1 Evaluation of the Selection Heuristic

The heuristic (see Section 3.7.4) is using the JRip classifier from the Weka data mining tool [27] to decide which resources evaluation algorithm should be faster. To get data for the learning, the Progen generator [33] was used to generate 4 datasets with 30, 60, 90, and 120 activities respectively. The parameters of the generated datasets were set the same as for J30, J60, J90, and J120 datasets with the exception that different random seeds were used. For each generated dataset the classifier was learned using `weka.classifiers.rules.JRip -F 3 -N 2.0 -O 10 -S 0` command and tested on the corresponding standard dataset with the same number of activities. The achieved results in Table 3.13 reveal that the accuracy is

decreasing with the number of activities. The reason for this behavior can be the smaller ratio of the resources evaluation time to the total runtime.

J30	**J60**	**J90**	**J120**
72.3 %	85.8 %	91.9 %	96.3 %

Table 3.13: Percentage of correctly classified problems for each dataset.

To prove that the proposed heuristic also improves the PTSG performance the runtime was measured for each evaluation algorithm and normalized with respect to the referenced runtime, i.e., the runtime achieved by using the heuristic. The results in Table 3.14 show that the heuristic accelerates the PTSG up to 2 times and its effect is decreasing as the evaluation of schedules becomes a less time-consuming part of the PTSG. The time-indexed algorithm seems to be faster than the capacity-indexed algorithm on the standard datasets. On the other hand, the achieved speedup is dependent on the characteristics of instances and it cannot generally be determined which evaluation algorithm is faster. The capacity-indexed evaluation algorithm is usually faster for long schedules with low resource capacities in contrast to the time-indexed algorithm which usually performs better for short schedules with high resource capacities.

	J30	**J60**	**J90**	**J120**
time-indexed	1.02	1.10	1.09	1.23
capacity-indexed	1.27	1.34	1.96	1.73
heuristic	1	1	1	1

Table 3.14: Effect of the selection heuristic on the PTSG performance.

3.9.2 Demonstration of Convergence

To demonstrate that cooperation among blocks is beneficial the graph of convergency (in Figure 3.9) was created for `j1206_4.sm` instance. It can be seen that the quality of solutions is getting better with the increasing number of launched blocks, therefore, it is obvious that cooperation leads to the better solutions. To ensure the smoothness of the graph each point was averaged over 50 measurements.

3.10 Conclusion

The first known GPU algorithm dealing with the Resource Constrained Project Scheduling Problem has been proposed. The performed experiments on the standard benchmark instances reveal the merits of the proposed solution. The achieved quality of solutions is very good and outperforms the

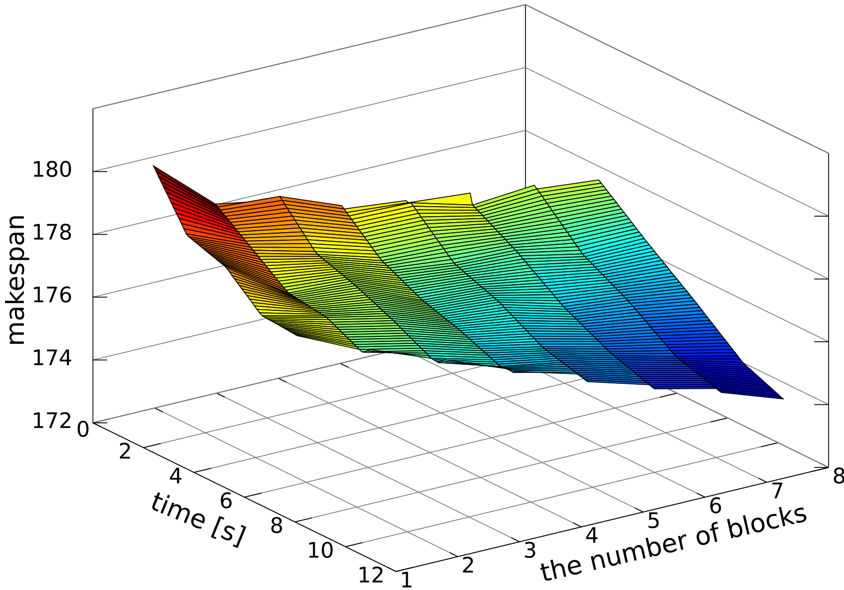


Figure 3.9: Graph of convergence for the GPU version.

other Tabu Search implementations to the best of our knowledge. In addition to this, the GPU algorithm design has proved to be very effective since the mid-range GPU was substantially faster than the optimized parallel CPU version. The Nvidia Geforce GTX 650 Ti GPU is able to evaluate more than one million schedules per second for the J120 dataset on average. The achieved performance boost could not be reached without effective structures and auxiliary algorithms. The Simple Tabu List implementation is adapted to the features of the GPU, the capacity-indexed evaluation algorithm was proposed, and many parallel reductions were applied. In addition to this, the homogeneous model reduces the required communication bandwidth between the CPU and GPU.

In spite of the fact that GPUs are not primarily designed for solving combinatorial problems the rising interest in these solutions can be seen [12]. The reason for this is the high computational power of graphics cards and the relatively user-friendly programming API that the CUDA offers. So it can be expected that GPUs will be more and more used in operations research in the future.

Chapter 4

Conclusion and Future Work

Thesis is dedicated to the design of novel optimization algorithms for production systems. Two problems, i.e., the energy optimization of robotic cells and RCPSP, were addressed and their fundamental properties were taken into account during the design of efficient algorithms. The parallel algorithms are scalable, memory-friendly, and utilize advanced data structures.

Besides the algorithms and related publications, the robotic cell in Škoda Auto was optimized with the result of 20 % energy saving. The industrial cooperation continues under the eRobot project, which main goal is to integrate the optimization algorithms into the digital factory software. In the future, we believe that the optimization will be a part of the designing process, and the increased efficiency of production will pay off time invested in the optimization.

4.1 Fulfillment of Goals

1. Study the existing literature related to the energy optimization of robotic cells and identify possible improvements.

   ⇒ The investigation of the literature in Section 2.1 revealed that only a little research had been conducted regarding the energy optimization of robotic cells. In the closest work of Wigström and Lennartson [55], the non-linear optimization model was proposed to optimize the robotic cells as a whole by changing speeds of robots and order of operations. This pioneering work, however, does not consider other optimization aspects such as alternative robotic paths or power-saving modes, and the work cycle time makes parallel processing of more workpieces impossible. Therefore, we decided to design novel optimization algorithms that overcome these limitations.

2. Devise a mathematical model that considers important optimization aspects to minimize the energy consumption of industrial robots.

   ⇒ Based on measurements of a small KUKA robot (described in [15]), we identified important aspects influencing the energy consumption of industrial robots. These aspects were used to formulate the mathematical model (see Sections 2.3 and 2.4) that minimizes the energy consumption for a fixed robot cycle time. Compared to the state-of-the-art works, more optimization aspects are considered and the robot cycle time enables higher parallelism during the production.

3. Propose heuristic and exact algorithms to solve industrial-sized robotic cells. Both the algorithms should utilize the problem structure and multi-core processors.

   ⇒ The parallel Hybrid Heuristic and Branch & Bound algorithm, which optimize robotic cells with up to 12 robots, were proposed in Chapter 2. The heuristic consists of subheuristics where each of them optimizes a problem specific aspect and estimates the energy saving of modifications. The exact algorithm, on the other hand, utilizes the tight lower bound based on convex envelopes, and efficient propagation and pruning techniques.

4. Design and implement a parallel Tabu Search algorithm to solve the RCPSP on graphics cards.

   ⇒ Section 3 describes the design and implementation of the first Tabu Search algorithm that solves the RCPSP on graphics cards. The high performance of this algorithm lies in the effective schedule evaluation, optimized memory access, and homogeneous model. Moreover, the faster schedule evaluation algorithm, i.e., the time-indexed or capacity-indexed evaluation algorithm, is dynamically selected based on the properties of the current problem.

5. Verify the proposed algorithms on benchmark instances and compare them with the existing works.

   ⇒ Experiments in Sections 2.12 and 3.9 confirmed that the proposed algorithms are highly scalable and the resulting quality of solutions is better or comparable with the existing works. To be more particular, the algorithms that optimize the energy consumption of robotic cells were tested on benchmark instances with 3 to 12 robots, compared to the literature where 4 robots were considered at maximum without consideration of additional optimization aspects. Moreover, the existing robotic cell from Škoda Auto was optimized and the achieved savings of 20 % met the expectations.

   The Tabu Search algorithm, which solves the universal RCPSP, was compared to the existing works in Section 3.9 (see Table 3.12). The results on J30, J60, J90, and J120 datasets show that the algorithm is competitive and its performance is superior to other Tabu Search implementations.

Bibliography

[1] Energy Efficiency – European Commision. `http://ec.europa.eu/energy/en/topics/energy-efficiency`. Accessed on: 9.11.2017.

[2] PROFIenergy homepage. `http://www.profibus.com/technology/profienergy/`. Accessed on: 9.11.2017.

[3] Christian Artigues, Philippe Michelon, and Stéphane Reusser. Insertion techniques for static and dynamic resource-constrained project scheduling. *European Journal of Operational Research*, 149(2): 249–267, 2003.

[4] Christian Artigues, Sophie Demassey, and Emmanuel Néron. *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*. ISTE, 2007. ISBN 190520972X.

[5] Maria Ayala, Abir Benabid, Christian Artigues, and Claire Hanen. The resource-constrained modulo scheduling problem: an experimental study. *Computational Optimization and Applications*, 54(3): 645–673, 2013. ISSN 0926-6003. DOI: 10.1007/s10589-012-9499-2.

[6] J. Blazewicz, J. K. Lenstra, and A.H.G. Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1): 11–24, 1983.

[7] K. Bouleimen and H. Lecocq. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. *European Journal of Operational Research*, 149(2): 268–281, 2003.

[8] V. Boyer, D. El Baz, and M. Elkihel. Solving knapsack problems on GPU. *Computers & Operations Research*, 39(1): 42 – 47, 2012. ISSN 0305-0548. DOI: 10.1016/j.cor.2011.03.014. Special Issue on Knapsack Problems and Applications.

[9] V. Boyer, D. El Baz, and M. A. Salazar-Aguilar. Chapter 10 - GPU computing applied to linear and mixed-integer programming. In Hamid Sarbazi-Azad, editor, *Advances in GPU Research and Practice*, Emerging Trends in Computer Science and Applied Computing, pages 247 – 271. Morgan Kaufmann, Boston, 2017. ISBN 978-0-12-803738-6. DOI: 10.1016/B978-0-12-803738-6.00010-0.

[10] Wojciech Bożejko, Zdzisław Hejducki, Mariusz Uchroński, and Mieczysław Wodecki. Solving the Flexible Job Shop Problem on Multi-GPU. *Proceedings of the International Conference on Computational Science, ICCS 2012*, 9(0): 2020–2023, 2012.

[11] Pavol Božek. Robot Path Optimization for Spot Welding Applications in Automotive Industry. *Tehnicki vjesnik / Technical Gazette*, 20(5): 913 – 917, 2013. ISSN 13303651.

[12] André R. Brodtkorb, Trond R. Hagen, and Martin L. Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1): 4–13, 2013. ISSN 0743-7315.

[13] Peter Brucker, Andreas Drexl, Rolf Möhring, Klaus Neumann, and Erwin Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1): 3–41, 1999. ISSN 0377-2217.

[14] L. Bukata, P. Šůcha, Z. Hanzálek, and P. Burget. Energy optimization of robotic cells. *IEEE Transactions on Industrial Informatics*, 13(1): 92–102, Feb 2017. ISSN 1551-3203. DOI: 10.1109/TII.2016.2626472.

[15] Pavel Burget, Libor Bukata, Přemysl Šůcha, Martin Ron, and Zdeněk Hanzálek. Optimisation of Power Consumption for Robotic Lines in Automotive Industry. In L. Ghezzi, D. Hömberg, and C. Landry, editors, *Math for the Digital Factory. Mathematics in Industry*, volume 27, chapter 7, pages 135–161. Springer, Cham, 2017. ISBN 978-3-319-63955-0. DOI: 10.1007/978-3-319-63957-4_7.

[16] Ali Shirzadeh Chaleshtarti and Shahram Shadrokh. Branch and Bound Algorithms for Resource Constrained Project Scheduling Problem Subject to Cumulative Resources. In *Proceedings of the 2011 International Conference on Information Management, Innovation Management and Industrial Engineering - Volume 01*, ICIII '11, pages 147–152, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4523-3.

[17] Milind W. Dawande, H. Neil Geismar, Suresh P. Sethi, and Chelliah Sriskandarajah. *Throughput Optimization in Robotic Cells*, volume 101 of *International Series in Operations Research & Management Science*. Springer US, 1 edition, 2007. ISBN 978-0-387-70987-1. DOI: 10.1007/0-387-70988-6.

[18] Audrey Delévacq, Pierre Delisle, Marc Gravel, and Michaël Krajecki. Parallel Ant Colony Optimization on Graphics Processing Units. *Journal of Parallel and Distributed Computing*, 73(1): 52–61, 2013.

[19] Mokhtar Essaid, Lhassane Idoumghar, Julien Lepagnot, and Mathieu Brévilliers. Gpu parallelization strategies for metaheuristics: a survey. *International Journal of Parallel, Emergent and Distributed Systems*, pages 1–26, 2018. DOI: 10.1080/17445760.2018.1428969.

[20] Alan P. French and John M. Wilson. An LP-Based Hybrid Heuristic Procedure for the Generalized Assignment Problem with Special Ordered Sets. In María J. Blesa, Christian Blum, Andrea Roli, and Michael Sampels, editors, *Hybrid Metaheuristics*, volume 3636 of *Lecture Notes in Computer Science*, pages 12–20. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-28535-9. DOI: 10.1007/11546245_2.

[21] Na Fu, Hoong Chuin Lau, Pradeep Varakantham, and Fei Xiao. Robust Local Search for Solving RCPSP/max with Durational Uncertainty. *Journal of Artificial Intelligence Research*, 43(1): 43–86, January 2012. ISSN 1076-9757.

[22] Michele Gadaleta, Giovanni Berselli, and Marcello Pellicciari. Energy-optimal layout design of robotic work cells: Potential assessment on an industrial case study. *Robotics and Computer-Integrated Manufacturing*, 47: 102 – 111, 2017. ISSN 0736-5845. DOI: 10.1016/j.rcim.2016.10.002.

[23] Michel Gendreau. An Introduction to Tabu Search. In Fred Glover and Gary Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, pages 37–54. Springer New York, 2003. ISBN 978-0-306-48056-0.

[24] Fred Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers & Operations Research*, 13(5): 533–549, May 1986. ISSN 0305-0548.

[25] José Fernando Gonçalves, Mauricio G.C. Resende, and Jorge J.M. Mendes. A Biased Random-Key Genetic Algorithm with Forward-Backward Improvement for the Resource Constrained Project Scheduling Problem. *Journal of Heuristics*, 17(5): 467–486, 2011.

[26] Boukthir Haddar, Mahdi Khemakhem, Saïd Hanafi, and Christophe Wilbaut. A hybrid heuristic for the 0-1 Knapsack Sharing Problem. *Expert Systems with Applications*, 42(10): 4653 – 4666, 2015. ISSN 0957-4174. DOI: 10.1016/j.eswa.2015.01.049.

[27] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.*, 11(1): 10–18, November 2009. ISSN 1931-0145.

[28] Johannes Hofmann, Steffen Limmer, and Dietmar Fey. Performance investigations of genetic algorithms on graphics cards. *Swarm and Evolutionary Computation (2013)*. ISSN 2210-6502.

[29] Tabitha James, Cesar Rego, and Fred Glover. A cooperative parallel tabu search algorithm for the quadratic assignment problem. *European Journal of Operational Research*, 195(3): 810–826, 2009.

[30] Jianyong Jin, Teodor Gabriel Crainic, and Arne Løkketangen. A cooperative parallel metaheuristic for the capacitated vehicle routing problem. *Computers & Operations Research*, 44: 33 – 41, 2014. ISSN 0305-0548. DOI: 10.1016/j.cor.2013.10.004.

[31] J. E. Kelley. 1963. The critical-path method: Resources planning and scheduling. In: Muth, J. F., Thompson, G. L. (Eds.), Industrial Scheduling. Prentice-Hall, Englewood Cliffs, NJ, pp. 347–365.

[32] Daecheol Kim and Hyun Joon Shin. A hybrid heuristic approach for production planning in supply chain networks. *The International Journal of Advanced Manufacturing Technology*, 78(1-4): 395–406, 2015. ISSN 0268-3768. DOI: 10.1007/s00170-014-6599-4.

[33] Rainer Kolisch, Christoph Schwindt, and Arno Sprecher. Benchmark Instances for Project Scheduling Problems. In *Handbook on Recent Advances in Project Scheduling*, pages 197–212. Kluwer, 1998.

[34] M. E. Lalami and D. El-Baz. GPU Implementation of the Branch and Bound Method for Knapsack Problems. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1769–1777, 2012.

[35] K. Y. Li and R. J. Willis. An iterative scheduling technique for resource-constrained project scheduling. *European Journal of Operational Research*, 56(3): 370–379, 1992. ISSN 0377-2217.

[36] M. Mashaei and B. Lennartson. Energy Reduction in a Pallet-Constrained Flow Shop Through On–Off Control of Idle Machines. *IEEE Transactions on Automation Science and Engineering*, 10(1): 45–56, Jan 2013. ISSN 1545-5955.

[37] D. Meike and L. Ribickis. Energy Efficient Use of Robotics in the Automobile Industry. In *2011 15th International Conference on Advanced Robotics (ICAR)*, pages 507–511, June 2011. DOI: 10.1109/ICAR.2011.6088567.

[38] D. Meike, M. Pellicciari, G. Berselli, A. Vergnano, and L. Ribickis. Increasing the energy efficiency of multi-robot production lines in the automotive industry. In *2012 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 700–705, Aug 2012. DOI: 10.1109/CoASE.2012.6386391.

[39] D. Meike, M. Pellicciari, and G. Berselli. Energy Efficient Use of Multi-robot Production Lines in the Automotive Industry: Detailed System Modeling and Optimization. *IEEE Transactions on Automation Science and Engineering*, 11(3): 798–809, July 2014. ISSN 1545-5955. DOI: 10.1109/TASE.2013.2285813.

[40] Michał Czapiński. An effective Parallel Multistart Tabu Search for Quadratic Assignment Problem on CUDA platform. *Journal of Parallel and Distributed Computing (2012).* ISSN 0743-7315.

[41] Michał Czapiński and Stuart Barnes. Tabu Search with two approaches to parallel flowshop evaluation on CUDA platform. *Journal of Parallel and Distributed Computing*, 71: 802–811, June 2011.

[42] A. Migdalas, Panos M. Pardalos, and Sverre Story. *Parallel Computing in Optimization.* Springer Publishing Company, Incorporated, 1st edition, 2012. ISBN 1461334020, 9781461334026.

[43] T. Miyamoto, K. Mori, S. Kitamura, and Y. Izui. A Study of Resource Constraint Project Scheduling Problem for Energy Saving. In *2014 IEEE International Conference on System Science and Engineering (ICSSE)*, pages 23–26, July 2014. DOI: 10.1109/ICSSE.2014.6887897.

[44] J. Mohammadi, K. Mirzaie, and V. Derhami. Parallel genetic algorithm based on GPU for solving quadratic assignment problem. In *2015 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, pages 569–572, Nov 2015. DOI: 10.1109/KBEI.2015.7436107.

[45] NVIDIA Corporation. NVIDIA CUDA C Programming Guide, 2012. `https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`. Accessed on: 9.11.2017.

[46] Koen Paes, Wim Dewulf, Karel Vander Elst, Karel Kellens, and Peter Slaets. Energy Efficient Trajectories for an Industrial ABB Robot. *Procedia CIRP*, 15(0): 105 – 110, 2014. ISSN 2212-8271. DOI: 10.1016/j.procir.2014.06.043. 21st CIRP Conference on Life Cycle Engineering.

[47] M. Pellicciari, G. Berselli, F. Leali, and A. Vergnano. A Minimal Touch Approach for Optimizing Energy Efficiency in Pick-and-Place Manipulators. In *2011 15th International Conference on Advanced Robotics (ICAR)*, pages 100–105, June 2011. DOI: 10.1109/ICAR.2011.6088620.

[48] Tetsuo Samukawa and Haruhiko Suwa. An Optimization of Energy-Efficiency in Machining Manufacturing Systems Based on a Framework of Multi-Mode RCPSP. *International Journal of Automation Technology*, 10(6): 985–992, nov 2016. ISSN 1881-7629.

[49] Ursula Schnabl. Auch Roboter können jetzt abschalten. `https://www.bbheute.de/nachrichten/auch-roboter-koennen-jetzt-abschalten-29-10-2010/`. Accessed on: 13.3.2018.

[50] A. Subramanian, L.M.A. Drummond, C. Bentes, L.S. Ochi, and R. Farias. A parallel heuristic for the Vehicle Routing Problem with Simultaneous Pickup and Delivery. *Computers & Operations Research*, 37(11): 1899 – 1911, 2010. ISSN 0305-0548. DOI: 10.1016/j.cor.2009.10.011. Metaheuristics for Logistics and Vehicle Routing.

[51] J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[52] Vicente Valls, Sacramento Quintanilla, and Francisco Ballestín. Resource-constrained project scheduling: A critical activity reordering heuristic. *European Journal of Operational Research*, 149(2): 282–301, 2003.

[53] A. Vergnano, C. Thorstensson, B. Lennartson, P. Falkman, M. Pellicciari, Chengyin Yuan, S. Biller, and F. Leali. Embedding detailed robot energy optimization into high-level scheduling. In *2010 IEEE Conference on Automation Science and Engineering (CASE)*, pages 386–392, Aug 2010. DOI: 10.1109/COASE.2010.5584686.

[54] Přemysl Šůcha and Zdeněk Hanzálek. A cyclic scheduling problem with an undetermined number of parallel identical processors. *Com-*

putational Optimization and Applications, 48(1): 71–90, 2011. ISSN 0926-6003. DOI: 10.1007/s10589-009-9239-4.

[55] O. Wigstrom and B. Lennartson. Integrated OR/CP optimization for Discrete Event Systems with nonlinear cost. In *2013 IEEE 52nd Annual Conference on Decision and Control (CDC)*, pages 7627–7633, Dec 2013.

[56] O. Wigstrom and B. Lennartson. Sustainable Production Automation - Energy Optimization of Robot Cells. In *2013 IEEE International Conference on Robotics and Automation (ICRA)*, pages 252–257, May 2013. DOI: 10.1109/ICRA.2013.6630584.

[57] O. Wigstrom, B. Lennartson, A. Vergnano, and C. Breitholtz. High-Level Scheduling of Energy Optimal Trajectories. *IEEE Transactions on Automation Science and Engineering*, 10(1): 57–64, Jan 2013. ISSN 1545-5955.

[58] T. Zajíček and P. Šůcha. Accelerating a Flow Shop Scheduling Algorithm on the GPU. *Workshop on Models and Algorithms for Planning and Scheduling Problems (MAPSP)*, 2011.

[59] Li Zhou, Dong Wang, and Wuliang Peng. An ACO for Solving RCPSP. In *Computer Science and Computational Technology, 2008. ISCSCT '08. International Symposium on*, volume 2, pages 250–253, dec. 2008.

Appendix A

Nomenclature – Chapter 2

Problem Statement Symbols

a	Static, dynamic, or composite activity.
v	Static activity (robot operation).
e	Dynamic activity (set of trajectories).
v_h^r	Home activity of robot r that closes the cycle.
l	Location of static activity.
t	Trajectory of dynamic activity.
r	Industrial robot from set $\mathcal{R}$.
m	Power-saving mode of a robot.
A	All activities of a problem instance, i.e., $A = A_{\mathcal{S}} \cup A_{\mathcal{D}}$.
$A_{\mathcal{S}}$	Set of static activities.
L_v	Set of locations of activity $v \in A_{\mathcal{S}}$.
$A_{\mathcal{D}}$	Set of dynamic activities.
T_e	Set of trajectories of activity $e \in A_{\mathcal{D}}$.
$A_{\mathcal{O}}$	Set of optional dynamic activities.
S_a	Set of successors of activity a.
P_a	Set of predecessors of activity a.
R	Set of industrial robots.
G	Set of robot graphs.
$\mathcal{HC}_{\text{act}}^r$	Hamiltonian circuit through activities in a graph of robot r.
$\mathcal{HC}_{\text{loc}}^r$	Hamiltonian circuit through locations in a graph of robot r.
s_a	Start time of activity a.
d_a	Duration of activity a.
$\underline{d}_a$	Minimal duration of activity a.
$\overline{d}_a$	Maximal duration of activity a.
$\underline{d}_e^t$	Duration of the fastest robot movement for trajectory $t \in T_e$.

$\overline{d}_e^t$	Duration of the slowest robot movement for trajectory $t \in T_e$.
$\overline{C_T}$	The desired cycle time of the robotic cell.
M^r	Set of available power-saving modes for robot $r \in R$.
$\underline{d}^m$	Minimal time required by stationary robot $r \in R$ to apply power-saving mode $m \in M^r$.
$f_{v,l}^m(d_v)$	Energy function of static activity.
$f_e^t(d_e)$	Energy function of dynamic activity.
$E_{\mathcal{TL}}$	Set of inter-robot time lags.
l_{a_i,a_j}	Length of the time lag from activity a_i to activity a_j.
h_{a_i,a_j}	Height of time lag, i.e., time offset in multiples of $\overline{C_T}$.
$t_{\max}$	The maximal execution time of the heuristic, Branch & Bound algorithm, or the Mixed Integer Linear Programming solver.

Symbols Related to the Heuristic

$\Phi_{\min}$	Minimal number of optimization iterations for a given tuple $\mathcal{T}$.
$\mathcal{T}$	Tuple, defined as triple $\mathcal{T} = (\mathscr{A}, \mathscr{P}, \alpha : A_{\mathcal{S}} \to M)$, is a partial solution which timing is determined by the Linear Programming solver, see Section 2.5 for more information.
$\mathscr{A}$	Defines the order of activities. Set $\mathscr{A}$ contains one $\mathcal{HC}_{\text{act}}^r$ for each robot $r \in R$.
$\mathscr{P}$	Set contains one closed robotic path (i.e., $\mathcal{HC}_{\text{loc}}^r$) through selected locations for each robot $r \in R$.
α	Function mapping each static activity $v \in A_{\mathcal{S}}$ to its assigned power-saving mode $m \in M$.
$A_{\mathcal{D}}(\mathcal{T})$	Dynamic activities selected in the partial solution given by tuple $\mathcal{T}$.
$\mathcal{F}_1(\mathcal{T})$	Set contains assigned trajectory $t \in T_e$ for each dynamic activity $e \in A_{\mathcal{D}}(\mathcal{T})$ in the form of pair (e, t).
$\mathcal{F}_2(\mathcal{T})$	Set contains assigned location $l \in L_v$ and power-saving mode $m \in M$ for each static activity $v \in A_{\mathcal{S}}$ in the form of triple (v, l, m).
$K(\mathcal{T})$	Possible collisions between robots that may occur in a partial solution given by tuple $\mathcal{T}$.
$\mathcal{D}_{\geq}, \mathcal{D}_{\leq}$	These sets are used to generate collision-avoidance constraints as described in Section 2.5.3.

Symbols Related to Branch & Bound Algorithm

c Composite activity, defined in Section 2.9.1.

$\boldsymbol{n}$ Node of the Branch & Bound tree that is defined as triple $(\mathcal{P}, \mathcal{L}, \mathcal{M})$. See Section 2.7 for more information.

p Subpath with the fixed order of activities for node $\boldsymbol{n}$.

$\mathcal{P}$ Set of subpaths with the fixed activity order for node $\boldsymbol{n}$.

$\mathcal{L}$ Maps each static activity v to selectable locations $L_v(\boldsymbol{n})$.

$L_v(\boldsymbol{n})$ Selectable locations of activity v for node $\boldsymbol{n}$.

$T_e(\boldsymbol{n})$ Selectable trajectories of activity e for node $\boldsymbol{n}$.

$\mathcal{M}$ Maps each static activity v to applicable power-saving modes $M_v^r(\boldsymbol{n})$.

$M_v^r(\boldsymbol{n})$ Power-saving modes of robot r applicable in activity v for node $\boldsymbol{n}$.

$next(p, \boldsymbol{n})$ Leaving edges ($\subseteq A_{\mathcal{O}}$) from subpath p for node $\boldsymbol{n}$.

$prev(p, \boldsymbol{n})$ Entering edges ($\subseteq A_{\mathcal{O}}$) to subpath p for node $\boldsymbol{n}$.

$next(l, \boldsymbol{n})$ Leaving trajectories from location l for node $\boldsymbol{n}$.

$prev(l, \boldsymbol{n})$ Entering trajectories to location l for node $\boldsymbol{n}$.

$A_{\mathcal{D}}(\boldsymbol{n})$ Set of viable dynamic activities for node $\boldsymbol{n}$ where $A_{\mathcal{D}}(\boldsymbol{n}) = A_{\mathcal{D}_f}(\boldsymbol{n}) \cup A_{\mathcal{D}_u}(\boldsymbol{n})$. See Section 2.9.2 for more information.

$A_{\mathcal{D}_f}(\boldsymbol{n})$ Set of selected dynamic activities for node $\boldsymbol{n}$.

$A_{\mathcal{D}_u}(\boldsymbol{n})$ Set of dynamic activities selectable in descendant nodes of node $\boldsymbol{n}$.

$A_{\mathcal{D}_u}(c)$ Set of optional dynamic activities encapsulated by composite activity c, see Section 2.9.1.

$A_{\mathcal{C}}(\boldsymbol{n})$ Set of composite activities for node $\boldsymbol{n}$, see Sections 2.9.1 and 2.9.2.

$A_1(\boldsymbol{n})$ Set of activities considered in node $\boldsymbol{n}$, i.e., $A_1(\boldsymbol{n}) = A_{\mathcal{S}} \cup A_{\mathcal{D}}(\boldsymbol{n})$. Refer to Section 2.9.2 for the definition.

$A_2(\boldsymbol{n})$ Set of activities considered in node $\boldsymbol{n}$, i.e., $A_2(\boldsymbol{n}) = A_{\mathcal{S}} \cup A_{\mathcal{D}_f}(\boldsymbol{n}) \cup A_{\mathcal{C}}(\boldsymbol{n})$. Compared to $A_1(\boldsymbol{n})$, selectable dynamic activities are encapsulated by composite ones, see Section 2.9.2 for more information.

$K(\boldsymbol{n})$ Set specifying pairs of time disjunctive activities for node $\boldsymbol{n}$. The set is used in the MILP formulation of the lower bound (see Section 2.9.4).

C_T Robot cycle time as a variable, compared to constant $\overline{C_T}$. Used in the node evaluator and Deep Jumping that are described in Sections 2.9.4 and 2.10, respectively.

Appendix B

Nomenclature – Chapter 3

Problem Statement Symbols

V Set of activities in the project.

E Precedences between activities, e.g., if $(i, j) \in E$, $i, j \in V$, then activity j starts after activity i, i.e., $s_j \geq s_i + d_i$.

G Directed Acyclic Graph $G(V, E)$ where nodes are activities and edges are precedences.

N Number of activities, i.e., $|V|$.

S Set of start time values, i.e., $S = \{s_0, \ldots, s_{N-1}\}$.

s_i Start time of activity $i \in V$.

D Set of activity durations, i.e., $D = \{d_0, \ldots, d_{N-1}\}$.

d_i Duration of activity $i \in V$.

W Order of activities for a feasible solution, i.e., $W = \{w_0, \ldots, w_{N-1}\}$ where w_u is u-th activity in the schedule.

$\mathcal{W}$ Set of all feasible solutions for a given instance.

M Number of resources considered in the project.

$\mathcal{R}$ Set of resources $\mathcal{R} = \{R_0, \ldots, R_{M-1}\}$ used for the activities.

R_k Capacity of the k-th resource in $\mathcal{R}$.

R_{max} Maximal capacity of a resource, i.e., $R_{max} = \max_{k=0}^{M-1} R_k$.

$r_{i,k}$ Activity $i \in V$ requires $r_{i,k}$ units of resource $k \in \mathcal{R}$.

C_{max} Project makespan, i.e., the time needed to execute the schedule.

Symbols Related to Tabu Search

l_k Denotes k-th level in graph $G(V, E)$. Level l_k contains all the activities which longest path from dummy activity 0 (edges weighted by 1) is equal to k. See Section 3.6.1 for more information.

l_{max} Last level in graph $G(V, E)$, i.e., level l_k with the highest k value.

$\mathcal{N}_{full}(W)$ Full neighborhood generated by applying all feasible swap moves.

$\mathcal{N}_{reduced}(W)$ Reduced neighborhood, i.e., a subset of $\mathcal{N}_{full}(W)$, where only the swap moves satisfying δ parameter are applied, see Section 3.6.3.

δ Maximal distance between two swapped activities in order W.

es_i^{prec} Earliest start time of activity $i \in V$ if the precedences are only considered.

es_i^{res} Earliest start time of activity $i \in V$ if the resources are only considered. The value is determined by a resource evaluation algorithm, see Section 3.7 for more information.

es_i Earliest start time of activity $i \in V$ if the precedences and resources are considered.

C State of resources for the capacity-indexed evaluation algorithm, i.e., $C = \{c_0, \ldots, c_{M-1}\}$.

c_k State of resource R_k, i.e., an array where $c_k[R_k - r_{i,k}]$ is equal to the earliest time when resource R_k is capable of providing $r_{i,k}$ units for activity i.

T State of resources for the time-indexed evaluation algorithm, i.e., $T = \{\tau_0, \ldots, \tau_{M-1}\}$.

τ_k State of resource R_k, i.e., an array where $\tau_k[t]$ is equal to the number of available resource units of R_k at time t.

$\mathrm{UB}_{C_{max}}$ Upper bound on the project makespan.

F Set of solutions in the working set, see Section 3.8. Each solution $k \in F$ consists of the makespan C_{max}^k, order of activities W^k, Tabu List, and iteration counter IC^k.

B Number of CUDA blocks used for the execution of the parallel Tabu Search algorithm.

$\mathcal{I}_{block}$ Number of iterations assigned to each CUDA block, i.e., to each instance of the Tabu Search algorithm.

$\mathcal{I}_{total}$ Total number of Tabu Search iterations, i.e., $\mathcal{I}_{total} = \mathcal{I}_{block}B$.

$\mathcal{I}_{assigned}$ Number of Tabu Search iterations assigned to loaded solution $k \in F$. The value is determined by Equation (3.8) in Section 3.8.1.

Φ_{max} Value determines how many times a solution can be read from the working set without being improved. If the value is exceeded, the solution is diversified by random swap moves.

Φ_{steps} Number of random swap moves applied during the diversification of a solution.

Appendix C

Curriculum Vitae

Libor Bukata received his bachelor's degree in Cybernetics and measurements from the Czech Technical University in Prague in 2010. He continued at the same university to study Open Informatics programme with the branch specialization in Software Engineering, where he completed his master degree with honors in 2012. Subsequently, he started his Ph.D. career at the Department of Control Engineering on the topic of parallel optimization algorithms for production systems. During his research work, he has published two papers in impacted international journals (IEEE Transactions on Industrial Informatics and Journal of Parallel and Distributed Computing), and another one which is currently under the review in Computers & Operations Research. Besides the journal papers, he is a coauthor of a book chapter (Math for the Digital Factory) and the results of his research have been presented in more than three international conferences (e.g., PDP 2013, SCOR 2014, ECCO 2014, MISTA 2015).

With respect to teaching activities, he led lab exercises of Combinatorial Optimization and Parallel Algorithms courses and helped with the preparation of the educational material. He also supervised two master students, namely Tomaš Poledný and Jan Kůrka, who successfully defended their master theses. He was an opponent of two other students.

Besides the scientific value, many outcomes of his research are applicable in the manufacturing industry. For example, the energy consumption of a robotic cell in Škoda Auto was reduced by 20 % without any deterioration in the throughput after using the results of the proposed optimization algorithms, and these algorithms are being integrated into the Siemens Process Simulate software to enable an easy optimization and verification before the deployment. Škoda Auto and Blumenbecker showed interest in our solutions and assisted with the verification of the technology. In general, his research and engineering work reflects his passion for scheduling, combinatorial optimization, and parallel and distributed systems.

Libor Bukata
Prague, April 2018

Appendix D

List of Author's Publications

List of publications and technical reports related to this thesis is included in this appendix. Besides the scientific contributions, the energy optimization of robotic cells is a verified technology that was tested at Škoda Auto in cooperation with Pavel Burget's team.

Publications in Journals with Impact Factor

Libor Bukata, Přemysl Šůcha, and Zdeněk Hanzálek. Solving the Resource Constrained Project Scheduling Problem using the parallel Tabu Search designed for the CUDA platform. *Journal of Parallel and Distributed Computing*, 77(0): 58 – 68, 2015. ISSN 0743-7315. DOI: 10.1016/j.jpdc.2014.11.005. **Coauthorship 45 %, indexed in Web of Science, 12 citations (6 of them in WoS).**

Libor Bukata, Přemysl Šůcha, and Zdeněk Hanzálek. Optimizing Energy Consumption of Robotic Cells by a Branch & Bound Algorithm. *Computers & Operations Research*, November 2017. **Under the review**, the decision will be known soon.

Libor Bukata, Přemysl Šůcha, Zdeněk Hanzálek, and Pavel Burget. Energy optimization of robotic cells. *IEEE Transactions on Industrial Informatics*, 13(1): 92–102, Feb 2017. ISSN 1551-3203. DOI: 10.1109/TII.2016.2626472. **Coauthorship 40 %, indexed in Web of Science, 5 citations.**

Book Chapters

Pavel Burget, Libor Bukata, Přemysl Šůcha, Martin Ron, and Zdeněk Hanzálek. Optimisation of Power Consumption for Robotic Lines in Automotive Industry. In L. Ghezzi, D. Hömberg, and C. Landry, editors, *Math for the Digital Factory. Mathematics in Industry*, volume 27, chapter 7, pages 135–161. Springer, Cham, 2017. ISBN 978-3-319-63955-0. DOI: 10.1007/978-3-319-63957-4_7. **Coauthorship 20 %**.

International Conferences and Workshops

Libor Bukata and Přemysl Šůcha. A GPU Algorithm Design for Resource Constrained Project Scheduling Problem. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 367–374, Feb 2013. DOI: 10.1109/PDP.2013.59. **Coauthorship 50 %, indexed in Web of Science, 5 citations.**

Libor Bukata and Přemysl Šůcha. High-level Optimisation of Energy Consumption of the Robotic Line with Respect to Production Cycle Time using Integer Linear Programming and Lagrangian Relaxation. In *SCOR 2014 - 4th Student Conference on Operational Research. Nottingham: University of Nottingham, Northern Ireland*, page 35, May 2014. **Coauthorship 50 %**.

Libor Bukata and Přemysl Šůcha. High-level Optimisation of Robotic Lines with Respect to Power Consumption and Given Production Cycle Time. In *ECCO 2014 – the 27-th Conference of the European Chapter on Combinatorial Optimization, München: Technische Universität München, Germany*, page 49, May 2014. **Coauthorship 50 %**.

Libor Bukata, Přemysl Šůcha, and Zdeněk Hanzálek. A new lower bound for optimisation of energy consumption of robotic cells. In Z. Hanzálek, G. Kendall, B. McCollum, and P. Šůcha, editors, *Proceedings of the 7th Multidisciplinary International Conference on Scheduling : Theory and Applications (MISTA 2015), Prague, Czech Republic*, pages 662–665, August 2015. **Coauthorship 40 %**.

Libor Bukata, Přemysl Šůcha, and Zdeněk Hanzálek. A tight relaxation of the energy optimization problem. In *ISCO 2016 - 4th International Symposium on Combinatorial Optimization, Vietri sul Mare (Salerno), Italy*, pages 114–115, May 2016. **Coauthorship 33.3 %**.

Přemysl Šůcha, Libor Bukata, and Zdeněk Hanzálek. Algorithms for Optimizing Energy Consumption of Robotic Cells. In *Workshop Math for the Digital Factory, Limerick: University of Limerick, Ireland*, March 2018. **Coauthorship 33.3 %**.

Other Publications and Technical Reports

Libor Bukata. Preliminary doctoral thesis: Design and application of novel parallel algorithms addressing challenging combinatorial problems. Technical report, Department of Control Engineering, Czech Technical University in Prague, 2014. **Coauthorship 100 %**.

Pavel Burget, Martin Ron, Libor Bukata, Přemysl Šůcha, and Ondřej Fiala. Optimalizace spotřeby elektrické energie na lince Příčník zadní podlahy v hale m12. Research report, Department of Control Engineering, Czech Technical University in Prague, 2014. **Coauthorship 20 %**.

Přemysl Šůcha and Libor Bukata. Specifikace rozšíření nástroje Process Simulate o plugin pro optimalizaci robotických buněk. Software specification, Czech Institute of Informatics, Robotics, and Cybernetics, 2017. **Coauthorship 50 %**.

Přemysl Šůcha, Libor Bukata, and Zdeněk Hanzálek. Specifications extension tools Process Simulate on plugins to optimize robotic lines. Research report, Department of Control Engineering, Czech Technical University in Prague, 2016. **Coauthorship 33.3 %**.

Libor Bukata
Prague, April 2018

This thesis is focused on the optimization of manufacturing systems. Its main contributions are as follows:

1. Designed and implemented novel algorithms that optimize the energy consumption of robotic cells without deterioration in throughput.

2. A robotic cell in Škoda Auto was modified to verify the results of the algorithms. Measurements confirmed the energy saving of 20 % only by changing robot speeds.

3. Proposed algorithms can optimize robotic cells with up to 12 robots compared to the existing literature where only one to four robots were considered. A high efficiency of the algorithms, which are multithreaded and cache-friendly, is achieved by the utilization of the problem structure.

4. The Resource Constrained Project Scheduling problem, which is useful for the optimization of production, was solved on graphics cards by using Tabu Search meta-heuristic. The parallel heuristic outperformed existing Tabu Search implementations.