Optimal Task Scheduling on Parallel Systems

Sarad Venugopalan

A thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in the Faculty of
Electrical and Computer Engineering,
The University of Auckland, 2015.

Supervisor

Dr. Oliver Sinnen

Co-supervisor

Prof. Matthias Ehrgott

“Everything is simple, we are stupid” - Paul Irofti

Abstract

To fully benefit from a multiprocessor system, the tasks of a program are to be carefully assigned and scheduled on the processors of the system such that the overall execution time is minimal.

Due to the reached physical limits of processor technology the improvements are fading out and manufacturers have moved to multi(core)processors. With multiple processors, however, the performance growth is not automatic anymore and can only be achieved when the processors are efficiently employed in parallel. For the performance and efficiency of a parallel program the scheduling of its (sub)tasks is crucial. Unfortunately, scheduling is a fundamental unsolved problem (an NP-hard optimisation problem), as the time needed to solve it optimally grows exponentially with the number of tasks. I.e. the associated task scheduling problem with communication delays, $P|prec, c_{ij}|C_{max}$, is a well known NP-hard problem.

The tasks that are to be scheduled may or may not be dependent on each other and are represented as an acyclic directed graph. The nodes in the graph represent the tasks and the edges between the nodes, the communications. The node cost is the time required for the task to complete and the edge cost is the communication time between two tasks on different processors. We assume a connected network of processors with identical communication links. Further, there is no multitasking or parallelism within a task. Each processor may execute several tasks but no concurrent execution of tasks is permitted. The tasks are to be assigned in such a way as to minimise the overall schedule length.

Existing scheduling algorithms are mostly heuristics (for e.g. the list scheduling algorithm) as they try to produce good rather than optimal schedules. Having optimal schedules can make a fundamental difference, e.g. for time critical systems such as flight control, industrial automation, automotive applications, telecommunication systems, consumer electronics, robotics and multimedia systems. Multiprocessor

systems are also popular in small portable devices such as cellphones or navigators to large systems such as industrial robots or aircraft. An optimal schedule may also be used as a benchmark to enable the precise evaluation of scheduling heuristics. Moreover, once an optimal schedule is found, it may be reused when a parallel program is rerun.

This work has two main contributions 1) investigate and propose novel Mixed Integer Linear Programming (MILP) solutions to this scheduling problem, despite the fact that scheduling problems are often difficult to handle by MILP solvers. 2) Propose memory limited versions of the A* scheduling algorithm since A* for larger problem instances may run out of memory before finding an optimal schedule. The applicability of existing pruning techniques used on A* are re-investigated and new pruning techniques to further speedup the memory limited A*algorithms are proposed.

The first part of the work proposes MILP solutions that use problem specific knowledge to eliminate the need to linearise the bi-linear equations arising out of communication delays. The classic ILP formulation for the scheduling problem and its improved formulations PACKING-USUAL and PACKING-COMPACT are studied. Two new ILP formulations, namely ILP-RBL and ILP-TC are proposed to further speedup the solution. This is attained by re-formulating the constraints and variables needed to linearise the bi-linear forms arising from communication delays between tasks running on different processors. Next, the work in the thesis proposes the formulation ILP-DELTA by introducing new overlap constraints to ensure that no two tasks running on the same processor overlap in time and space. The purpose is to test if re-formulating these constraints would help speedup the formulation.

Further, the work in the thesis proposes the MILP formulations SHD-BASIC, SHD-RELAXED and SHD-REDUCED wherein the size of the proposed formulations in terms of variables are independent of the number of processors. We analyse and discuss the influence of the different MILP components in respect to characteristics of the task graph such as structure and communication to computation ratio. The proposed MILP formulations are experimentally compared with previous MILP formulations used to solve this scheduling problem. The proposed formulations displays a drastic improvement in performance, which allows to solve larger problems optimally. The work also observe strengths and weaknesses of the formulations related to the input characteristics.

The second part of the work proposes two memory limited optimal scheduling al-

gorithms: Iterative Deepening A* (IDA*) and Depth-First Branch and Bound A* (BBA*). When finding a guaranteed near optimal schedule length is sufficient, the proposed algorithms can be combined, reporting the gap while they run. A novel method to find a good initial lower bound close to the optimal schedule length in order to speed-up the execution of IDA* is proposed. Problem specific pruning techniques, which are crucial for good performance, are studied and new pruning techniques proposed for the two memory limited algorithms. Duplicate avoidance without memory and processor normalisation without memory are the pruning techniques proposed. Extensive experiments are conducted to evaluate and compare the proposed algorithms with previous optimal algorithms.

Acknowledgments

I thank my aunt Suma Verma, uncle Jairam, Mikhil and and Mythili for their love and hospitality, the very first of my extended family in Auckland. I thank Biju Balakrishnan and Priyanka Biju for being there for me. I thank my good friends Ruchi Chowdhary, Pankti Kansara, Hyewon Oh, Leandro Rasmussen, Sizhu (Carol) Zong, Gerry Geneva, Matt Ozball, Urie Bezuidenhout, Aisha Abubakar, Chandan Datta, Jeyavel Jawahar, Arulmozhivarman Parthiban, Anisha, Tomomi Fujikawa, Osberta Li, Dhruv Gaur, Rumi Yokoyama, Misaki Iwamoto, Zong Chen, Shey Mahendran, Helen He, Shradha Khiani, Shin Yee Teh, Dinoy Jose, Yosra, Emma Kang, Dominic Dagbanja, Latha Murugesan, Enoch D, Tony Zhao and Winyee Phong.

I am grateful to my extended family in Auckland, Anna Zhang for her clear, crisp soul and beautiful singing voice, Le Luo for more reasons than I can think of, Asha Rana for her amazing kindness and in teaching me how to cook, Lucy Law for being vibrant and energetic, Alvin Chan for being a sincere friend, Zainab Zeezi for her very beautiful spirit, Rivindu Weerasekera for saying probably-maybe but gets all the work done, Rusiru Karunaratne for being the funny guy, Supriya Siddiqui for her loving personality, Sarah Lim for being a strongly spirited lady and a wonderful violinist, Mika Tokida for her dry humor and evil laughs we share, Sonny P Kumar for being cool as a cucumber, Janu Rangel for having a welcoming heart, Ben Chayanun for bringing out the best of me, Shen Song for being precise and showing me the wisdom of the sunset, Keezrawati Mujan for being kind, Chiara Mannoni Papakonstantinou for having an open heart and an open mind, Cindy Cheung for having deep meaningful conversations, Sohaib Ijaz Majeed for his kindness, Caroline Ding for her empathy, Mark Kong for being caring, Samuel Mandal for his critical thinking, Melinda Thomas for celebrating life, Mike Rosing for inspiring me to solve problems and to continue with my studies, Balamurali B for his humility and vast understanding of humanity, Lu Yu for being my all weather friend, no matter the circumstances. Sahil Chaplot, Parth Doriwala for being kind and caring for the time

I lived with them. Prem Kumar, Arijit Chakraborty for letting me stay with them until I completed writing this thesis.

I extend my gratitude to my other friends whom I an constantly in touch with: Aparna Anil, Vipin Thekkedam, Dr. Gopi Krishna Kolluru, Indumathi Ram, Sreedevi Chandrakumar, Kip Kwiatkowski, Dr. Sobha Devi, Dr. Srikanth S, Sujith Raman, Divya Bhaskar, Navin Parakkal, Dr. Sanjeev Saini, Abhinesh Narayanan, Jayasurya Nambiar, Vidya Menon, Dr. Manikandadas Menon, Sankar K, Gesly George, Prof. Bharat B Amberker.

I thank Dilmah for making such fine Ceylon tea and the flat white coffee in New Zealand. They are both fuel for research and responsible for some of the eureka moments.

I thank the Electrical and Computer engineering department staff: Aruna S, Hanlie Van Zyl and Christine Salter; the most student friendly and compassionate staff.

I gratefully acknowledge that this work is supported by the Marsden Fund Council from Government funding, Grant 9073-3624767, administered by the Royal Society of New Zealand and additional postgraduate support from the University of Auckland.

I thank my supervisor Dr. Oliver Sinnen for teaching me how to research, provide valuable and critical feedback, to clearly present work in written form as conference and journal publications and giving me space to wander around the applied mathematics realm and solve some important questions related to the research problem. I thank Prof. Matthias Ehrgott for his feedback and deep insight into linear programming solutions that are used to solve the research problem. I also thank my PHD advisors Dr. Michael Dinneen and Dr. Partha Roop for their valuable feedback.

I thank my dad for the love and the will to see that I get the very best they could give. I thank my brother for having simple solutions to most problems, to take it easy and learn from him. Getting here would not have been possible without my mother, this one is for you mom.

Abbreviations and Acronyms

BBA*	-	Branch and Bound A*
BIP	-	Binary Integer Programming
CBL	-	Computational Bottom Level
CCR	-	Communication to Computation Ratio
CTL	-	Computational Top Level
DAG	-	Directed Acylic Graph
DEC-DEV	-	Decision Destructive
DLB	-	Destructive Lower Bound
DRT	-	Data Ready Time
EST	-	Earliest Start Time
FSL	-	Feasible Schedule Length
FTO	-	Fixed Task Order
GB	-	Green Banana
GXL	-	Graph eXchange Language
IDA*	-	Iterative Deepening A*
ILP	-	Integer Linear Programming
ILP-RBL	-	ILP-Revised Boolean Logic
ILP-TC	-	ILP-Transitivity Clause
LB	-	Lower Bound

MILP	-	Mixed Integer Linear Program
MINLP	-	Mixed Integer Non Linear Program
MSPCD	-	Multi-processor Scheduling Problem with Communication Delays
OSL	-	Optimal Schedule Length
SL	-	Schedule Length

Contents

List of Figures

List of Tables

1 Introduction

In the past, engineering and science have strongly benefited from an exponential growth in processor performance. Yet, due to the reached physical limits of processor technology the improvements are fading out [36] and manufacturers have moved to multi(core)processors. With multiple processors, however, the performance growth is not automatic anymore and can only be achieved when the processors are efficiently employed in parallel [21]. For the performance and efficiency of a parallel program the scheduling of its (sub)tasks is crucial. Unfortunately, scheduling is a fundamental unsolved problem (an NP-hard optimisation problem [44]), as the time needed to solve it optimally grows exponentially with the number of tasks. Existing scheduling algorithms are therefore heuristics that try to produce good rather than optimal schedules, e.g. [31], [37], [21], [41], [48], [55], [58], [7], [23]. However, having optimal schedules can make a fundamental difference, e.g. for time critical systems or to enable the precise evaluation of scheduling heuristics. The focus of this work is to find fundamental theoretical results that reduce the solution space, making it possible to efficiently search for an optimal solution.

The tasks that are to be scheduled are represented as a weighted directed acyclic graph. The nodes in the graph represent tasks while directed edges represent data precedence relationships. Precedence relationships (if any) have to be respected at all times. The node cost is the time required for the task to complete its execution on a processor and the edge cost is the communication time between two tasks on different processors. If two tasks with data dependence are mapped onto the same processor, the communication between them is implemented by data sharing in local memory and no communication delay is incurred. The model assumes a fully connected network of homogeneous multiprocessors $P = \{1, \ldots, |P|\}$ with identical communication links. Each processor may execute several tasks, but each task has to be assigned to exactly one processor, in which it is entirely executed without pre-emption. Further, no multi-tasking or parallelism is permitted within a task.

The execution time for each task on each processor and the data transfer times (or communication delays) between tasks with data dependence are given in advance as part of the task graph.

1.1 Objective

The objective of this work is to present fast solutions for the classic problem of scheduling task graphs on parallel systems with communication delay, which is $P|prec, c_{ij}|C_{max}$. I.e. to allocate and schedule the tasks onto the processors such that the overall completion time W (makespan) is minimised [11], [40]. Here P specifies the identical processor environment used, $prec, c_{ij}$ are the precedence characteristics and with communication costs c_{ij}; C_{max} is the maximum completion time of all tasks, which is the objective function to be minimised. Many heuristics have been proposed for task scheduling on parallel systems [27]. While they often provide good results and tend towards the optimal schedule there is no guarantee that the solutions are optimal, especially for task graphs with high communication costs [47], [46]. A number of approximation algorithms have been proposed for the scheduling problem [9], [17]. For the here addressed scheduling problem, $P|prec, c_{ij}|C_{max}$, no α-approximation is known [15]. The only known guaranteed approximation algorithm in [24] has an approximation factor depending on communication costs of the longest path in the schedule.

Given the NP-hardness, finding an optimal solution requires an exhaustive search of the entire solution space. For scheduling, this solution space is spawned by all possible processor assignments combined with all possible task orderings. Clearly this search space grows exponentially with the number of tasks, thus it becomes impractical already for very small task graphs. Hence, few attempts have been made to solve $P|prec, c_{ij}|C_{max}$ optimally. With the increase in processor power in computers, it is now feasible to find optimal solutions to larger instances of the scheduling problem. An example of a task graph wherein tasks are scheduled onto two processors is shown in Figure 1.1.

1.2 Problem Statement

To develop novel methods to optimally solve the task scheduling problem onto a

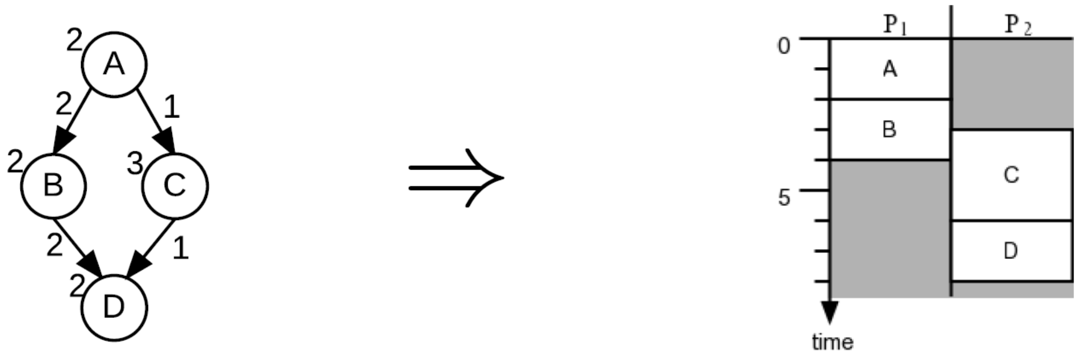


Figure 1.1: Tasks scheduled onto two processors

homogeneous multi-processor system for small to medium sized task graphs. I.e. the classic problem of scheduling task graphs on parallel systems with communication delay, which is $P|prec, c_{ij}|C_{max}$ in the $\alpha|\beta|\gamma$ notation [18], [51]. This will make the efficient parallelisation of more applications viable, hence allowing performance growth that can satisfy the future needs of engineering and science.

1.3 Task scheduling model

Formally, the tasks to be scheduled are represented by a directed acyclic graph (DAG) defined by a 4-tuple G=(V, E, C, L) where V denotes the set of tasks and E represents the set of edges. Each edge $(i, j) \in E$ defines a precedence relation between the tasks $i, j \in V$. A task cannot be executed unless all of its predecessors (parents) have completed their execution and all relevant data is available. The set $C = \{c_{ij} : (i, j) \in E\}$ denotes the set of edge communication times. If tasks i and j are executed on different processors $h, k \in P, h \neq k$, they incur a communication time penalty c_{ij}. If both tasks are scheduled to the same processor the communication time is zero. For a graph with $|V| = n$ tasks, the set $L = \{L_1 \ldots, L_n\}$ represents the task computation times (execution time length). Let $\delta^-(j)$ be the set of precedents of task j, that is $\delta^-(j) = \{i \in V|(i, j) \in E, j \in V\}$. The variables t_i and p_i are the main variables that describe a schedule for the problem to be solved. The start time of task i is t_i and the processor on which task i executes is p_i. The objective of this task scheduling problem is to allocate and schedule the tasks onto the processors such that the overall completion time W (makespan) is minimised [11], [40].

When the communication links between homogeneous processors are non-identical, the communication delay is modeled differently. If tasks i and j are executed on different processors $h, k \in P, h \neq k$, they incur a communication cost penalty γ_{ij}^{hk} dependent on the distance d_{hk} between the processors and on the amount of exchanged data c_{ij} between tasks ($\gamma_{ij}^{hk} = \Gamma c_{ij} d_{hk}$, where Γ is a known constant). For a fully connected processor network, γ_{ij}^{hk} is equivalent to γ_{ij} since the distance d_{hk} is unity. i.e. $\gamma_{ij} = \Gamma c_{ij}$.

1.4 Methodology

The proposal is to investigate novel search methods for optimal task scheduling that incorporate new problem specific knowledge. This is significant because generalised solution methodologies (as seen later in the thesis) may not be the best approach to solve the problem. Recent success in other NP-hard optimisation problems shows, e.g. Knapsack [38] or Traveling Salesman Problem [4], that an efficient algorithm for small to medium sized problems might be obtained by pruning the search space with problem specific knowledge.

The two approaches employed are 1) formulating the scheduling problem as a Mixed Integer Linear Program (MILP) 2) Use memory limited derivatives of the popular artificial intelligence A* algorithm.

1.5 Introduction to MILP

Integer Linear Programming are those linear programming problems which have the additional constraint that some or all the variables have to be integers. In contrast to linear programming, which can be solved efficiently in the worst case, integer programming problems are in many practical situations NP-hard. 0-1 integer programming or Binary Integer Programming (BIP) is the special case of integer programming where variables are required to be 0 or 1 (rather than arbitrary integers). This problem is also classified as NP-hard, and in fact the decision version was one of Karp's 21 NP-complete problems [25]. If only some of the unknown variables are required to be integers, then the problem is called a Mixed Integer Linear Program problem. These are generally also NP-hard. Many practical problems in operations research can be expressed as linear programming problems. Certain special cases of linear programming, such as network flow problems and multicommodity flow problems are considered important enough to have generated much research on specialised algorithms for their solution. Some of the popular problems solved using linear programming are the diet problem, portfolio optimisation, crew scheduling, machine and job scheduling, manufacturing and transportation, vehicle routing, call routing, capacity design in networks, traveling salesman problem and VLSI chip board manufacturing [34].

For the task scheduling problem, the MILP formulations can be broadly classified as discrete time and continuous time approaches [10], [16]. The discrete time approach introduces a new variable for each instant of time on each processor [1]. The number of time variables introduced in this approach explode when diverse execution times are present in the formulation. The continuous time approach, on the other hand, can handle diverse execution times, but its efficiency depends on how well the constraints and variables are formulated. The continuous time approach is further subdivided into three lines - sequencing, slots and overlaps. In sequencing, the formulation involves invoking new variables to determine if one task is executed after another task on the same processor [8], [5]. The number of constraints required to enforce the schedule requirements on each processor are known to grow quickly. In slots, each task is assigned to a space-time vacancy on a processor. The slot defines an order of tasks running on a processor [13], [32]. The start time and end time of tasks entering the slot are not fixed a priori. Since the exact number of slots required on each processor is not known a priori, a conservative number of slots (the number of tasks) has to be reserved and it suffers from a variable blow-up if the number of tasks to be scheduled is large. In overlap, variables are defined to prevent overlap of tasks scheduled on the same processor. Unlike other approaches, the number of variables and constraints in the formulation scales well as the number of tasks to be scheduled increases [10], [11].

1.6 Related Work using Mixed Integer Linear Programming

The classic formulation (as it is called in [11]) for a homogeneous multiprocessor system (with non-identical communication links) employs a set of binary variables which control both assignment of tasks to processors and positions in the order of tasks executed by the given processor and a set of continuous variables indicating the start time for each task. Let

$$\forall i \in V, k \in P, s \in tasknum_k;\; y_{ik}^{s} = \begin{cases} 1 & \text{task } i \text{ is the } s^{th} \text{ task on processor } k \\ 0 & \text{otherwise} \end{cases}$$

and $t_i \in \mathbf{R}$ be the starting time of task i for all $i{\in}V$. Note that $s \in tasknum_k$ gives the index of the task run on the k^{th} processor.

The Mixed Integer Non-Linear Program (MINLP) for the scheduling problem is as shown

$$\begin{array}{rr}
min & Z \quad \text{(A01)} \\
\forall i \in V & t_i + L_i - Z \leq 0 \quad \text{(A02)} \\
\forall i \in V & \sum_{k=1}^{p}\sum_{s=1}^{n} y_{ik}^{s} = 1 \quad \text{(A03)} \\
\forall k \in P & \sum_{i=1}^{n} y_{ik}^{1} \leq 1 \quad \text{(A04)} \\
\forall k \in P, s \in V\backslash\{1\} & \sum_{i=1}^{n} y_{ik}^{s} \leq \sum_{i=1}^{n} y_{ik}^{s-1} \quad \text{(A05)} \\
\forall j \in V, i \in \delta^{-}(j) & t_j \geq t_i + L_i + \sum_{h=1}^{p}\sum_{s=1}^{n}\sum_{k=1}^{p}\sum_{r=1}^{n} \Upsilon_{ij}^{hk} y_{ih}^{s} y_{jk}^{r} \quad \text{(A06)} \\
\forall k \in P, s \in V\backslash\{n\}, i,j \in V & t_j \geq t_i + L_i - M(2 - (y_{ik}^{s} + \sum_{r=s+1}^{n} y_{jk}^{r})) \quad \text{(A07)} \\
\forall i, s \in V, k \in P & y_{ik}^{s} \in 0,1 \quad \text{(A08)} \\
\forall i \in V & t_i \geq 0 \quad \text{(A09)}
\end{array}$$

A03 ensures that each task is assigned exactly to one processor. A04 - A05 state that each processor cannot be simultaneously used by more than one task. A04 means that at most one task will be on the first one at k , while A05 ensures that if some task is the s^{th} one (s $\geq$2) scheduled to the processor k then there must be another task assigned as $(s-1)^{th}$ to the same processor. A06 express precedence constraints together with communication time required for tasks assigned to different processors. A07 define the sequence of the starting times for the set of tasks assigned to the same processor. They express the fact that task j must start at least L_i time units after the beginning of task i whenever j is executed after i on the same processor k; the M parameter (set to a large positive integer constant) must be large enough so that A07 is active only if i and j are executed on the same processor k and $r > s$.

The mathematical formulation of the Multi-processor Scheduling Problem with Communication Delays (MSPCD) given by A02 - A09 contains linear terms in the continuous t variables and linear and bi-linear terms in the binary y variables. It therefore belongs to the class of mixed integer bi-linear programs. It is possible to linearise this model by introducing a new set of variables $\zeta_{ijhk}^{sr} \in$ [0,1] which replaces the bi-linear terms y_{ih}^{s} y_{jk}^{r} in A06.

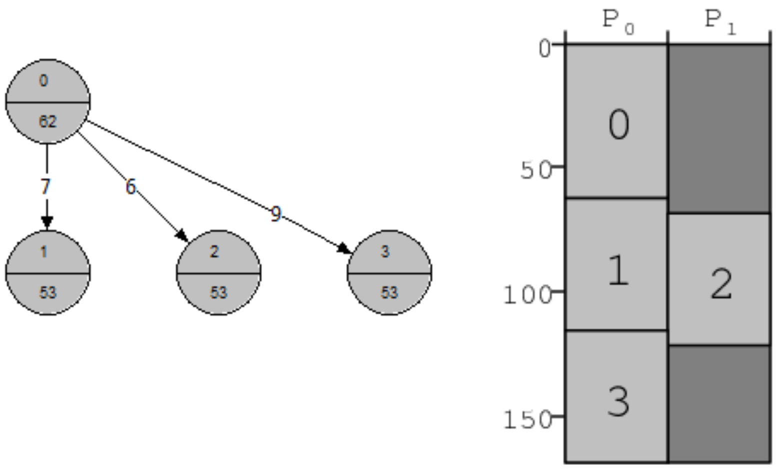


Figure 1.2: A fork graph scheduled onto two processors

An example fork graph with 4 tasks (task 0, task 1, task 2 and task 3) and its optimal schedule on 2 processors is shown in Figure 1.2. The optimal schedule length is $Z = 168$. The Boolean variables σ_{01}, σ_{02} and σ_{03} are set to 1 to indicate the precedence constraints. From the optimal schedule, the start times are seen to be $t_0 = 0$, $t_1 = 62$, $t_3 = 115$ and $t_2 = 62 + 6 = 68$ because $\gamma_{02}^{01} = 6$. Also, $L_0 = 62$, $L_1 = 53$, $L_2 = 53$ and $L_3 = 53$. From the optimal schedule, it is also seen that y_{00}^{1}, y_{10}^{2}, y_{30}^{3} and y_{21}^{1} are set to 1.

1.6.1 Linearisation of the classic formula

The above classic formulation has all but linear constraints in A06. The work in [11] linearises A06 by the introduction of a new set of continuous variables $\zeta_{ijhk}^{sr} \in$ [0,1] to replace the bi-linear term y_{ih}^{s} y_{jk}^{r} in A06.

The linearisation variable ζ_{ijhk}^{sr} satisfies the constraints

$$y_{ih}^{s} \geq \zeta_{ijhk}^{sr} \tag{A10}$$
$$y_{jk}^{r} \geq \zeta_{ijhk}^{sr} \tag{A11}$$
$$y_{ih}^{s} + y_{jk}^{r} - 1 \leq \zeta_{ijhk}^{sr} \tag{A12}$$

$\forall i,j,s,r \in V$, $\forall h,k \in P$, which guarantees that $\zeta_{ijhk}^{sr}= y_{ih}^{s}\ y_{jk}^{r}$. This approach is called the **usual linearisation**.

The number of variables and constraints in the linearised model is $O(|V|^4|P|^2)$. Since the constraint size is due to A10 - A12, a compact linearisation approach is used.

Compact Linearisation of the Classic Formula: The idea is to reduce the number of constraints by elimination A10 to A12. This is done by multiplying each assignment constraint in A03 by y_{jk}^{r}; the resulting bi-linear terms are successively linearised by substituting with the appropriate ζ variable: giving us a relationship between ζ and y variables. The compact linearisation also assumes $\zeta_{ijhk}^{sr} = \zeta_{jikh}^{rs}$ by commutativity of the product. Thus, the compact linearisation for the scheduling problem with communication delays is as follows:

$$\forall i,j,r \in V, k \in P \qquad \sum_{h=1}^{p}\sum_{s=1}^{n}\zeta_{ijhk}^{sr} = y_{jk}^{r} \tag{A13}$$
$$\forall i,j,s,r \in V, \forall h,k \in P \qquad \zeta_{ijhk}^{sr} = \zeta_{jikh}^{rs} \tag{A14}$$

Although, there are still $O(|V^4||P^2|)$ constraints in A14, these can be used to eliminate half of the linearisation variables and deleted from the formula. Hence, there are only $O(|V^3||P|)$ in the compact linearisation (see A13).

1.6.1.1 Packing Formulation

The idea on which the model is based is to liken task i to a rectangle of length L_i and height 1, and to pack all the rectangles representing the tasks into a larger rectangle of height $|P|$ and length W, where W is the total makespan to be minimised [11], [19].

For each task $i \in V$ let $t_i \in \mathbf{R}$ be the start execution time and $p_i \in \mathbf{N}$ be the ID of the processor where task i is to be executed. Let W be the total makespan. Let

x_{ik} be 1 if task i is assigned to processor k, and zero otherwise. In order to enforce non-overlapping constraints, define two sets of binary variables:

$$\forall i,\, j \in V \sigma_{ij} = \begin{cases} 1 & \text{task } i \text{ finishes before task } j \text{ starts} \\ 0 & \text{otherwise} \end{cases}$$

$$\forall i,\, j \in V \epsilon_{ij} = \begin{cases} 1 & \text{the processor index of task } i \text{ is strictly less than that of task } j \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned}
&min && W && \text{(A15)}\\
&\forall i \in V && t_i + L_i \leq W && \text{(A16)}\\
&\forall i \neq j \in V && t_j - t_i - L_i - (\sigma_{ij} - 1)W_{max} \geq 0 && \text{(A17)}\\
&\forall i \neq j \in V && p_j - p_i - 1 - (\epsilon_{ij} - 1)|P| \geq 0 && \text{(A18)}\\
&\forall i \neq j \in V && \sigma_{ij} + \sigma_{ji} + \epsilon_{ij} + \epsilon_{ji} \geq 1 && \text{(A19)}\\
&\forall i \neq j \in V && \sigma_{ij} + \sigma_{ji} \leq 1 && \text{(A20)}\\
&\forall i \neq j \in V && \epsilon_{ij} + \epsilon_{ji} \leq 1 && \text{(A21)}\\
&\forall j \in V \,:\, i \epsilon \delta^{-}(j) && \sigma_{ij} = 1 && \text{(A22)}\\
&\forall j \in V \,:\, i \epsilon \delta^{-}(j) && t_i + L_i + \sum_{h,k \in P} \gamma_{ij}^{hk} x_{ih} x_{jk} \leq t_j && \text{(A23)}\\
&\forall i \in V && \sum_{k \in P} k x_{ik} = p_i && \text{(A24)}\\
&\forall i \in V && \sum_{k \in P} x_{ik} = 1 && \text{(A25)}\\
& && W \geq 0 && \text{(A26)}\\
&\forall i \in V && t_i \geq 0 && \text{(A27)}\\
&\forall i \in V && p_i \in \{1, \ldots, |P|\} && \text{(A28)}\\
&\forall i \in V, k \in P && x_{ik} \in \{0, 1\} && \text{(A29)}\\
&\forall i, j \in V && \sigma_{ij}, \epsilon_{ij} \in \{0, 1\} && \text{(A30)}
\end{aligned}$$

where W_{max} is an upper bound on the makespan W, e.g.

$$W_{max} = \sum_{i \in V} L_i + \sum_{i,j \in V} c_{ij} \qquad \text{(A31)}$$

The makespan is minimised in A15 and A16, the time order on the tasks in terms of the σ variables in A17, similarly the CPU ID order on the tasks in terms of the ϵ variables in A18. By A19, the rectangles must have at least one relative positioning on the plane. By A20 a task cannot both be before and after another task; similarly, by A21 a task cannot be placed both on a higher and lower CPU ID than another task. A22 enforce the task precedents; A23 model the communication delays. A24 link the assignment variable x with the CPU ID variables p and A25 are the assignment constraints.

A15 - A30 is also a bi-linear MINLP, where the bi-linearities arise because of the communication delay constraints. This formulation has variables only in i, j, h and k where as the classic formulation and its linearisation had variables in i, j, h, k, s and r. It appears clear that the packing formulation is smaller than the classic one.

Linearisation of the packing formula: A15 - A30 are linearised by introducing variables 0≤z ≤1 as follows:

$$\forall j \in V\ :\ i \in \delta^{-}(j), h, k \in P \quad (z_{ij}^{hk} = x_{ih}.x_{jk})$$

Integrality on the linearisation variable z follows from the **usual linearisation** constraints

$$\forall j \in V, i \in \delta^{-}(j), h, k \in P \qquad (x_{ih} \geq z_{ij}^{hk} \wedge x_{jk} \geq z_{ij}^{hk} \wedge x_{ih} + x_{jk} - 1 \leq z_{ij}^{hk}) \tag{A32}$$

The above linearisation constraints are equivalent to the following **compact linearisation**:

$$\forall i \neq j \in V, k \in P \qquad \sum_{h \epsilon P} z_{ij}^{hk} = x_{jk} \tag{A33}$$

$$\forall i \neq j \in V, h, k \in P \qquad z_{ij}^{hk} = z_{ji}^{kh} \tag{A34}$$

Although the set of compact linearisation constraints A33 - A34 are smaller than the set of usual linearisation constraints in A32, the difference is not as dramatic as the classic formulation O($|V|^2|P|$) against $O(|V|^2||P|^2)$.

The work in Chapters 2 to 4 use the PACKING formulation as a basis for proposing and deploying faster ILP formulations to speedup the solution time for the optimal task scheduling problem.

1.7 Introduction to A*

For many difficult problems, algorithms require an exponential amount of time or memory or both. Brute force searches are prohibitively slow and the runtime of the algorithm is only fast enough for very small instances of the problem that may not be of sufficient for practical use. The result is a search that is too slow or never completes. The solution, for many problems, is to use pruning techniques to eliminate choices that are impossible to reach the goal state (called pruning the search tree). Pruning techniques may also supply the program with a best guess for the path on which the solution lies. For many problems, it is possible to begin the search with some form of a guess (heuristic) and then refine the guess incrementally until no more refinements can be made and leads to the goal (optimal) state much faster than without the guesses. It is important to note that the nature of the guesses are referred to as heuristics. The algorithm itself is not a heuristic since it finds an optimal solution (subject to its memory constraints).

The basic search strategy will be based on A*, a best-first search technique [33], [14], [42]. A* is guided by a problem specific cost function $f(s)$ for each solution state s, which underestimates the final cost of any solution based on s. In each step the state with the best $f(s)$ is expanded (hence best-first), with the aim that a goal state (i.e. a complete optimal schedule) is found without considering all other states with worse cost values. In general, the better the cost function, the less states have to be considered. Crucial for a successful search method is the investigation of scheduling theory that gives precise cost functions and allows to reduce the search space, as the limited success of the approach in [28] demonstrated. For example, proving that certain task to processor assignments can never lead to optimal solutions significantly reduces the space that has to be searched. This has not been fully investigated before due to the focus on heuristic algorithms that gives a solution close to the optimal solution (but whose quality of solution cannot be guaranteed). However, our results demonstrate that this is very feasible.

Depth first - Iterative Deepening A* (IDA*) [26] is a memory limited version of A* which tries to reduce the number of iterations in iterative deepening by setting an initial threshold that is below the cost of the goal node and then successively increases this threshold. It sets the initial threshold to a cost that is below the cost of the goal node and it is then successively increased. When the goal node is found, the IDA* algorithm terminates. The IDA* algorithm for any given iteration regenerates all the states for the previous iteration along with the new states generated for the present iteration. The main motivation in using IDA* is its very limited memory requirement. This gives IDA* an extremely small memory footprint when compared to A*. On the downside, IDA* generates more states than A* due to iterative deepening. Depth first - Branch and Bound A* (BBA*) is another memory bounded algorithm that searches the state space in depth first order. The initial state (feasible solution) is an overestimate on the goal state (optimal solution) [39], [57]. As such it approaches the optimal solution from above, successively improving the best found solution. Both IDA* and BBA* are employed for the proposed memory limited task scheduling algorithms in Chapter 5.

1.8 Related Work using A*

A* is a best-first search algorithm that searches through a graph, visiting one node at a time. It does so by looking in each step at the most promising node, according to a cost function f, hence the best-first search. The nodes adjacent to the currently best one are added to the candidate nodes for the next step.

Scheduling the task graph G on processors P is the assignment of a **processor allocation** p_i and a **start time** $t_s(i) = t_i$ to each $i \in V$. The task's **finish time** is given by $t_f(i) = t_s(i) + L_i$, i.e. the task's start time plus its computation costs. Let $tf(p) = \max_{i \in \mathbf{V}:p_i=p}\{t_f(i)\}$ be the **processor finish time** of $p \in P$ and let $sl(\mathcal{S}) = \max_{i \in V}\{t_f(i)\}$ be the **schedule length** (or makespan) of schedule $\mathcal{S}$, assuming $\min_{i \in V}\{t_s(i)\} = 0$. A fully connected network of homogeneous multi-processor is considered [49].

For such a schedule to be feasible, the following two conditions must be fulfilled for all tasks in G. The **Processor Constraint** in A35 enforces that only one task is executed by a processor at any point in time, which means for any two tasks $i,j \in V$

$$i = j \Rightarrow \begin{cases} \quad t_f(i) \leq t_s(j) \\ \text{or} \quad t_f(j) \leq t_s(i) \end{cases} \tag{A35}$$

A task cannot be executed unless all of its predecessors (parents) have completed their execution and all relevant data is available. The **Precedence Constraint** given in A36 enforces that for every edge $(i, j) \in E$, $i, j \in V$, the destination task j can only start after the communication associated with (i, j) has arrived at p_j.

$$t_s(j) \geq t_f(i) + \begin{cases} 0 & \text{if } p_i = p_j \\ \gamma_{ij} & \text{otherwise} \end{cases} \tag{A36}$$

For task $j \in V$, its start time on processor p is constrained by the **Data Ready Time (DRT)** and is represented as $t_{dr}(j, p)$. The DRT for task j is the time when all communications from task j's predecessors have arrived at p as shown in A37.

$$t_{dr}(j, p) = \max_{i \in \mathbf{parents}(j)} \left\{ t_f(i) + \begin{cases} 0 & \text{if } p_i = p \\ \gamma_{ij} & \text{otherwise} \end{cases} \right\} \tag{A37}$$

If $j \in V$ is a source task, then $t_{dr}(j, p) = 0$. The task may however not be able to immediately start its execution since the processor assigned may be occupied with the execution of another task. The Earliest Start Time (EST) of task $i \in V$ on processor p is given by (A38).

$$t_{EST}(i, p) = \max\{tf(p), t_{dr}(i, p)\} \tag{A38}$$

The computation bottom level of a task $i \in V$ is the length of the longest path starting in i, denoted by cbl_i. Recursively it is defined in (A39) as

$$cbl_i = L_i + \max_{j \in \mathbf{children}(i)} \{cbl_j\} \tag{A39}$$

Given the start time of any task i, the schedule length sl is bounded by $t_s(i) + cbl_i$. In other words, after the task i has started execution, it still takes (at least) the time to sequentially execute all the tasks on the longest path starting in i.

Let the computation top level of a task $j \in V$ be defined as the length of the longest path (sum of computational task weights) starting in j, denoted by ctl_j to the longest length of its parent, excluding its own weight. Recursively it is defined as

$$ctl_j = \max_{i \in \mathbf{parent}(j)} \{ctl_i + L_i\} \tag{A40}$$

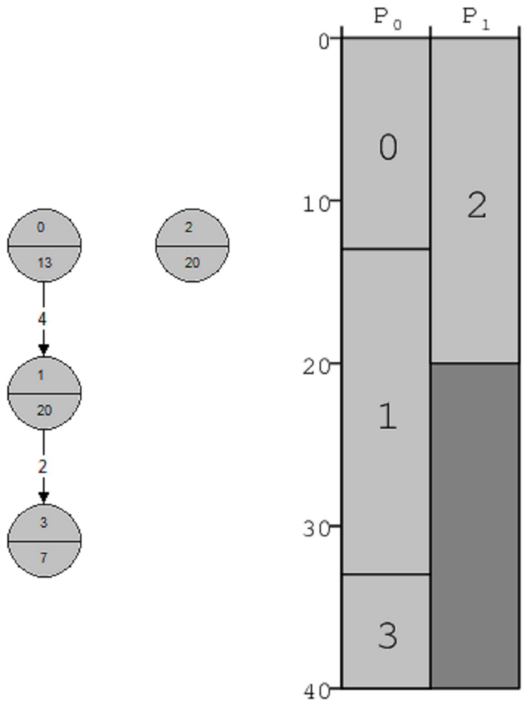


Figure 1.3: A random graph scheduled onto two processors

In Figure 1.2 we note that task 2 runs on P_1. Since there are no other tasks running on P_1, $t_f(P_1) = 0$. The data ready time of task 2 on P_1, $t_{dr}(2, P_1)$ is the sum of processing time of task 1 on P_0 and the communication cost between task 0 and task 2. Hence, $t_{dr}(2, P_1)$ = 62 + 6 =68. Then by (A38) , Earliest Start Time of task 2 on P_1, denoted as $EST(2, P_1) = max(0, 68) = 68$. From Figure 1.3, it is seen that tasks 1 and 3 have precedence constraints. Then, by the computational bottom length definition in (A39), $cbl_0 = 13 + 20 + 7 = 40$. Since, task 2 has no successors, $cbl_2 = 20$. I.e. its own computation time on P_1.

1.8.1 A* Scheduling Algorithm

When A* is used for combinatorial optimisation, as in our case, the search space has the form of a tree. Each node (usually called state) of the tree corresponds to a partial solution of the problem to be optimised, which becomes more complete the deeper we get in the tree. The root of the tree is the initial, empty state s_{init}. While a depth-first search or a breadth-first search only looks at a limited number of possible next states in each step, A* considers all currently unexplored states and chooses the state s with the best cost value $f(s)$. To achieve that it uses a priority queue, called `OPEN`, into which all discovered but unexplored states are inserted, ordered according to their cost f. Taking the best state s from `OPEN` in each step, s is expanded creating new states. This is done by extending the partial solution represented by state s, making it more complete. For each of these newly created states the cost is computed and they are inserted into the `OPEN` queue. When a state has been fully expanded, it is removed from `OPEN` and placed into the `CLOSED` list. The purpose of `CLOSED` is to be able to detect duplicates, i.e. identical states, before states are inserted into `OPEN`. The algorithm continues with this process until the state with the lowest cost value is also a **goal state**, i.e. a complete solution. This state is the optimal solution, if certain requirements on the cost function f hold for all states (see Section 1.8.1.1). Algorithm 1.1 gives the pseudo-code for the basic A* algorithm [49].

Algorithm 1.1 Pseudo-code for A* algorithm

```
Input: OPEN, a priority queue, ordered by ascending $f$-value of elements
Output: Optimal solution $s$
Method: Algorithm_Astar()
01: OPEN ← $s_{init}$
02: **While** OPEN $\neq \phi$ **do**
03:    $s$ ← headOf(OPEN)
04:        **If** $s$ complete state **then**
05:            return optimal solution $s$
06:        Expand $s$ to new states **NEW**
07:        **For all** $s_i \in$ **NEW do**
08:            Calculate $f(s_i)$
09:            Insert $s_i$ into OPEN, unless duplicate in CLOSED or OPEN
10:        CLOSED←CLOSED + $s$; OPEN ← OPEN - $s$
```

To apply this A* algorithm to solve our scheduling problem, we formulate the following components:

- **State** s: A partial schedule where some tasks have been allocated and scheduled on the processors.
- **Initial state** s_{init}: The initial state represents an empty schedule, where no task has been scheduled yet.
- **Expansion operator**: Based on state s, a new state is created by scheduling one more task, hence growing the partial solution represented by s. A task that can be scheduled must be *free*, which means it must be either independent or all its predecessors have already been scheduled in the partial schedule of s. The set of all those tasks are denoted as $\mathbf{free}(s)$. The number of new states expanded from s is then the product of all free tasks times the number of processors, $|\mathbf{NEW}| = |\mathbf{free}(s)| \cdot |P|$. Each task of $\mathbf{free}(s)$ is scheduled on every processor in P as early as possible after all other tasks on the same processor according to A38.
- **Cost function** f: The cost function $f(s)$ is an *underestimate* of the length of a complete schedule based on the partial schedule represented by s.

1.8.1.1 f function

According to the A* principle [50], the $f(s)$ function is an *underestimate* of the exact minimum cost $f^*(s)$ of any goal state that is based on the state s. Applied to our scheduling problem, $f^*(s)$ is the minimum schedule length of all possible schedules that can be constructed using the partial schedule represented by s. If the function $f(s)$ fulfils $f(s) \leq f^*(s)$ for every state s, it is called *admissible*. With an admissible $f(s)$, A* is guaranteed to find an optimal solution. The number of examined states depends on how accurate $f(s)$ is, i.e. how close it is to $f^*(s)$. In general, the more accurate, the fewer states have to be examined and the faster is the algorithm.

1.9 Thesis Framework

The rest of the thesis is organised as follows. Chapter 2 proposes two Integer Linear Programming formulations for the Multiprocessor Scheduling Problem with Communication Delays: ILP - Revised Boolean Logic and ILP - Transitivity Clause to speedup the time to find an optimal schedule. ILP - Revised Boolean Logic is designed for a fast solution when the number of processors the tasks are scheduled on is small and ILP - Transitivity Clause is designed to be efficient when tasks are scheduled on a larger number of processors. Each formulation uses a different linearisation of the Integer Bilinear Programming formulation and is tested on CPLEX using known benchmark graphs for task scheduling.

Chapter 3 proposes an optimal Integer Linear Programming formulation for the Multiprocessor Scheduling Problem with Communication Delays (MSPCD) using a variation to the definition of the task overlap variable and tests if this may help to speedup the formulation. The formulation uses an effective method to linearise the bi-linear forms arising out of communication delays and introduces new overlap constraints to ensure that no two tasks running on the same processor overlap in time and space. The proposed formulation is compared with the previously discussed formulation ILP - Transitivity Clause.

Chapter 4 proposes new MILP formulations for the task scheduling problem with communication delay. The main goal is to learn from and improve upon the formulations proposed in Chapter 2. The three proposed ILP formulations in this chapter are : SHD-BASIC, SHD-RELAXED and SHD-REDUCED. Each part of the formulation is motivated and discussed in detail, explaining its role and importance. A major feature of this formulation is the reduction of the number of variables and constraints by the effective linearisation of the bi-linear equation arising out of communication delays. Further, all variable indices in the MILP formulation are independent of the number of processors. As a result, the constraint complexity of the proposed MILP formulation was reduced to $O(|V|^2)$, which is significantly less than previous formulations as analysed in detail.

Chapter 5 proposes two memory limited algorithms, the depth first Iterative Deepening A* (IDA*) and the Branch and Bound A* (BBA*) algorithm. The applicability of known pruning techniques for the A* scheduling algorithm is investigated and new pruning techniques for the memory limited algorithms are proposed. The IDA* and BBA* algorithms are combined to give solutions with a guaranteed quality when finding an optimal solution is not essential.

1.10 Contributions

The main contributions of this thesis are:

- Chapter 2: Proposed two new formulations: ILP - RBL and ILP - TC. The number of constraints formed by ILP-RBL is $O(|E||P|^2)$, where E is the number of edges in the task graph, V the number of vertices and P the number of processors. When scheduled over a fewer number of processors, the effect of $|P^2|$ towards the constraint complexity diminishes. The number of constraints formed by ILP - TC is $O(|V|^3)$. The constraint complexity is independent of the number of processors and is thus suitable where there is a larger availability of processors for scheduling. Experimental results validate that their performance is consistent with their constraint complexity and runs faster than known ILP formulations that solve the task scheduling problem.
- Chapter 3: Proposed the formulation ILP - DELTA. The definition of the task overlap variable is modified to test its impact on its runtime. By reformulating the ILP to accommodate the new overlap variable, experimental results indicates that ILP-DELTA has a performance similar to ILP-TC but is faster than other previously known ILP formulations.
- Chapter 4: Proposed the ILP formulations SHD-BASIC, SHD-RELAXED and SHD-REDUCED. For all the three proposed formulations, all variable indices in the MILP formulation are independent of the number of processors. As a result, the constraint complexity of the proposed MILP formulation is reduced to $O(|V|^2)$. The experimental results indicate a significant speedup over previously known ILP formulations.
- Chapter 5: Proposed memory limited task scheduling algorithms to overcome the memory shortcomings of the A* scheduling algorithm. Additional pruning techniques are proposed to further speedup the algorithm(s). Experimental results indicates that the memory limited algorithms are able to return an optimal solutions for problem instances where A* runs out of memory.

2 ILP formulations for Task Scheduling Solutions for MSPCD

This chapter is a precursor to the work in **Chapter 4**. It is common to use heuristics to find solutions for task scheduling problem instances. However, there is no guarantee that the heuristic solution is close to the optimal solution.

The **contributions** of the work in this chapter is to provide

- Optimal solutions for small and medium sized instances of the task scheduling problem.
- Propose two optimal scheduling formulations using Integer Linear Programming (ILP) for the Multiprocessor Scheduling Problem with Communication Delays: ILP-RevisedBoolean Logic (ILP-RBL) and ILP-Transitivity Clause (ILP-TC).
- The number of constraints formed by ILP-RBL is $O(|E||P|^2)$, where E is the number of edges in the task graph, V the number of vertices and P the number of processors. When scheduled over a fewer number of processors, the effect of $|P^2|$ towards the constraint complexity diminishes. The number of constraints formed by ILP - TC is $O(|V|^3)$. The constraint complexity is made independent of the number of processors and is thus suitable where there is a larger availability of processors for scheduling. Experimental results validate that their performance is consistent with their constraint complexity and runs faster than known ILP formulations that solve the task scheduling problem.
- Simplified linearisation of the bi-linear forms in comparison to previously known methods.

Functionality

- ILP-RBL is designed to work efficiently when the number of processors available to be scheduled on are few.
- ILP-TC is efficient when the number of processors are available to be scheduled on are many.

Methodology

- Each formulation uses a different linearisation of the Integer Bilinear Programming formulation and is tested on CPLEX using known benchmark graphs for task scheduling.

Outcome

- Faster than previous ILP formulations used to solve this scheduling problem.
- Inline with the functionality expected.

Publication

Sarad Venugopalan and Oliver Sinnen. Optimal linear programming solutions for multiprocessor scheduling with communication delays. 12^{th} International Conference, ICA3PP 2012. In Yang Xiang, Ivan Stojmenovic, BernadyO. Apduhan, Guojun Wang, Koji Nakano, and Albert Zomaya, editors, Algorithms and Architectures for Parallel Processing, volume 7439 of Lecture Notes in Computer Science, pages 129–138. Springer Berlin Heidelberg, 2012.

2.1 Abstract

Task parallelism does not automatically scale with the use of parallel processors. Optimised scheduling of tasks is necessary to maximise the utilisation of each available processor. It is common to use heuristics to find solutions for task scheduling problem instances. However, there is no guarantee that the heuristic solution is close to the optimal solution. The outcome of this work is to provide optimal solutions for small and medium sized instances of the task scheduling problem. Two optimal scheduling formulations using Integer Linear Programming (ILP) are proposed for the Multiprocessor Scheduling Problem with Communication Delays: ILP-RevisedBoolean Logic and ILP-TransitivityClause. ILP-RevisedBooleanLogic is designed to work efficiently when the number of processors available to be scheduled on is small. ILP-TransitivityClause is efficient

when a larger number of processors are available to be scheduled on. Each formulation uses a different linearisation of the Integer Bilinear Programming formulation and is tested on CPLEX using known benchmark graphs for task scheduling.

2.2 Introduction

For the performance and efficiency of a parallel program, the scheduling of its (sub)tasks is crucial. Unfortunately, scheduling is a fundamental hard problem (an NP-hard optimisation problem[44]), as the time needed to solve it optimally grows exponentially with the number of tasks. Existing scheduling algorithms are therefore heuristics that try to produce good rather than optimal schedules, e.g.[31], [37], [21], [41], [48], [55], [58], [7], [23]. However, having optimal schedules can make a fundamental difference, e.g. for time critical systems or to enable the precise evaluation of scheduling heuristics. Optimal scheduling is central in minimising the task schedule length. An efficient parallelisation permits scheduling of a large number of tasks onto a large number of dedicated parallel processors to find solutions to generic and specialised problems. It is hence of enormous practical significance to be able to schedule small and medium sized task graphs optimally on parallel processors.

Many heuristics have been proposed for scheduling. While heuristics often provide good results, there is no guarantee that the solutions are close to optimal, especially for task graphs with high communication costs [47], [46]. Given the NP-hardness, finding an optimal solution requires an exhaustive search of the entire solution space. For scheduling, this solution space is spawned by all possible processor assignments combined with all possible task orderings. Clearly this search space grows exponentially with the number of tasks, thus it becomes impractical already for very small task graphs.

The objective is to develop a method that solves the scheduling problem optimally for small to medium sized problem instances using Integer Linear Programming. This will make the efficient parallelisation of more applications viable. To achieve this, two formulations for the Multiprocessor Scheduling Problem with Communication Delays are proposed: ILP-RevisedBooleanLogic (ILP-RBL) and ILP-TransitivityClause (ILP-TC).

The rest of the chapter is organised as follows. Section 2.3 describes the task scheduling model. Section 2.4 discusses the related work in solving the task scheduling problem optimally. Section 2.5 details the proposed formulations and compares their complexities in terms of number of constraints generated, with previous approaches. Section 2.6 compares the computational results of the proposed formulation with the packing formulation. Section 2.7 concludes the chapter.

2.3 Task scheduling model

The tasks that are to be scheduled may or may not be dependent on each other and are represented as an acyclic directed graph. The nodes in the graph represent the tasks and the edges between the nodes, the communications. The node cost is the time required for the task to complete and the edge cost is the communication time between two tasks on different processors. We assume a connected network of processors with identical communication links. Further, there is no multitasking or parallelism within a task. Each processor may execute several tasks but no concurrent execution of tasks is permitted. The tasks are to be assigned in such a way as to minimise the makespan [11], [40]. This model fits the definition of the Multiprocessor Scheduling Problem with Communication Delays (MSPCD) defined as follows: tasks (or jobs) have to be executed on several processors; we have to find where and when each task will be executed, such that the total completion time is minimal. The duration of each task is known as well as precedence relations among tasks, i.e., which tasks should be completed before some others can begin. In addition, if dependent tasks are executed on different processors, data transfer times (or communication delays) that are given in advance are also considered.

More formally, the tasks to be scheduled are represented by a directed acyclic graph (DAG) defined by a 4-tuple G $= (V,\ E,\ C,\ L)$ where $i \in V$ denotes the set of tasks; $(i, j) \in E$ represents the set of communications; $C = \{c_{ij} : i, j \in V\}$ denotes the set of edge communication costs; and $L = \{L_1, \ . \ . \ . \ , L_n\}$ represents the set of task computation times (execution times length). The communication cost $c_{ij} \in C$ denotes the amount of data transferred between tasks i and j if they are executed on different processors. If both tasks are scheduled to the same processor the communication cost equals zero. The set E defines precedence relation between tasks. A task cannot be executed unless all of its predecessors have completed their execution and all relevant data is available. If tasks i and j are executed on different processors $h, k \in P, h \neq k$, they incur a communication cost penalty γ_{ij}^{hk} dependent on the distance d_{hk} between the processors and on the amount of exchanged data c_{ij} between tasks ($\gamma_{ij}^{hk} = \Gamma c_{ij} d_{hk}$, where Γ is a known constant). Let $\delta^{-}(j)$ be the set of precedents of task j, that is $\delta^{-}(j) = \{i \in V | (i, j) \in E\}$. For a fully connected processor network γ_{ij}^{hk} is equivalent to γ_{ij} since the distance d_{hk} is unity. i.e. $\gamma_{ij} = \Gamma c_{ij}$.

2.4 Related work

Very few attempts have been made to solve the MSPCD optimally. There are two different approaches, one is based on an exhaustive search of the solution space and the other on an Integer Linear Programming formulation. For many problems, heuristics provide a best effort solution of the scheduling problem. It is possible to begin the search with a best guess and then refine it incrementally until it reaches the solution state. The A* algorithm is one such search algorithm used to solve the MSPCD [28], [45]. A* is a best-first search technique [14], [42] and also a popular Artificial Intelligence algorithm guided by a problem specific cost function $f(s)$ for each solution state s, which underestimates the final cost of any solution based on s. The main drawback of A* is that it keeps all the nodes in memory and it usually runs out of memory long before it runs out of time making it unusable for a medium and large sized problem instances.

We propose an optimal scheduling alternative for the solution of the MSPCD that makes use of Linear Programming [11]. It involves linearisation of the bilinear forms resulting from communication delays. The work in [11] discusses a classic formulation and a packing formulation of the MSPCD. Their results indicate that the packing formulation is about 5000 times faster than the classic formulation. In this chapter we propose two significantly improved Linear Programming formulations of the MSPCD and compare them with the packing formulation in [11].

2.5 Proposed formulations

The performance of the ILP formulations in [11] suffer from the need to linearise bilinear equations. Two formulations to solve the MSPCD are proposed here: ILP-RBL and ILP-TC. ILP-RBL uses a new technique to linearise the bilinear forms of the packing formulation in [11] resulting from communication delays by readjusting the Boolean logic. ILP-TC reworks the linearisation of the bilinear forms in the packing formulation using a transitivity clause in a manner that aids the elimination of over defined linear equations in ILP-RBL. The runtime complexity of each ILP formulation depends on the number of constraints generated and the number of variables per constraint. The packing formulation and its linearisations in [11] is briefly discussed and the proposed ILP formulations are compared with the packing formulation in terms of constraints generated and number of variables per constraint.

2.5.1 ILP-RevisedBooleanLogic

For each task $i \in V$ let $t_i \in \mathbf{R}$ be the start execution time and $p_i \in \mathbf{N}$ be the ID of the processor on which task i is to be executed. Let W be the total makespan and $|P|$ the number of processors available. Let x_{ik} be 1 if task i is assigned to processor k, and zero otherwise. In order to enforce non-overlapping constraints, define two sets of binary variables as in [11]:

$$\forall i, j \in V \quad \sigma_{ij} = \begin{cases} 1 & \text{task } i \text{ finishes before task } j \text{ starts} \\ 0 & \text{otherwise} \end{cases}$$

$$\forall i, j \in V \quad \epsilon_{ij} = \begin{cases} 1 & \text{the processor index of task } i \text{ is strictly less than task } j \\ 0 & \text{otherwise} \end{cases}$$

$$
\begin{array}{rr}
min & W \quad \text{(A01)} \\
\forall i \in V & t_i + L_i \leq W \quad \text{(A02)} \\
\forall i \neq j \in V & t_j - t_i - L_i - (\sigma_{ij} - 1)W_{max} \geq 0 \quad \text{(A03)} \\
\forall i \neq j \in V & p_j - p_i - 1 - (\epsilon_{ij} - 1)|P| \geq 0 \quad \text{(A04)} \\
\forall i \neq j \in V & \sigma_{ij} + \sigma_{ji} + \epsilon_{ij} + \epsilon_{ji} \geq 1 \quad \text{(A05)} \\
\forall i \neq j \in V & \sigma_{ij} + \sigma_{ji} \leq 1 \quad \text{(A06)} \\
\forall i \neq j \in V & \epsilon_{ij} + \epsilon_{ji} \leq 1 \quad \text{(A07)} \\
\forall j \in V \,:\, i \epsilon \delta^{-}(j) & \sigma_{ij} = 1 \quad \text{(A08)} \\
\forall j \in V \,:\, i \in \delta^{-}(j), \forall h, k \in P & t_i + L_i + \gamma_{ij}^{hk}(x_{ih} + x_{jk} - 1) \leq t_j \quad \text{(A09)} \\
\forall j \in V \,:\, i \in \delta^{-}(j) & t_i + L_i \leq t_j \quad \text{(A10)} \\
\forall i \in V & \sum_{k \in P} k x_{ik} = p_i \quad \text{(A11)} \\
\forall i \in V & \sum_{k \in P} x_{ik} = 1 \quad \text{(A12)} \\
 & W \geq 0 \quad \text{(A13)} \\
\forall i \in V & t_i \geq 0 \quad \text{(A14)} \\
\forall i \in V & p_i \in \{1, \ldots, |P|\} \quad \text{(A15)} \\
\forall i \in V, k \in P & x_{ik} \in \{0, 1\} \quad \text{(A16)} \\
\forall i, j \in V & \sigma_{ij}, \epsilon_{ij} \in \{0, 1\} \quad \text{(A17)}
\end{array}
$$

where W_{max} is an upper bound on the makespan W.

$$W_{max} = \sum_{i \in V} L_i + \sum_{i,j \in V} c_{ij} \tag{A18}$$

The formulation is a min-max problem which involves minimising the maximum start execution times. This is achieved by minimising the makespan W and introducing A02. A03 defines the time order on the tasks in terms of the σ variables, i.e.

ensure $t_i + L_i \leq t_j$ if σ_{ij} defines an order. The CPU ID order on the tasks in terms of the ϵ variables in defined in A04. If the ϵ_{ij} variable is set, it implies $p_j > p_i$. By A05, at least one or both of the following conditions must be true: task i must finish before task j starts and the processor index of task i must be strictly less than that of task j. By A06, a task cannot both be before and after another task; similarly, by A07 a task cannot be placed both on a higher and lower CPU ID than another task. A08 enforce the task precedences defined by the edges of the graph; A09 and A10 model the communication delays between task i on processor h and task j on processor k. Since x_{ih} and x_{jk} are both binary variables and to simulate a Boolean multiplication $x_{ih} \cdot x_{jk}$, we use $x_{ih} + x_{jk} - 1$ in constraint A09. To compensate for the subtraction by 1 in A09 for the case that x_{ih} and x_{jk} both are 0, A10 is introduced, which must always be true (for local as well as remote communication). It is also clear from constraint A09 that the processor network need not be fully connected and can take up any connection configuration. A11 link the assignment variable x with the CPU ID variables p and A12 ensures that any given task runs only on one processor.

The complexity of this ILP formulation, in terms of constraints and variables, is dominated by A09. For the entire graph, A09 generates $|P|(|P| - 1)$ inequalities in terms of processor combinations for each edge of E and the number of variables per constraint is $O(1)$. Therefore the number of constraints formed by ILP-RBL is $O(|E||P|^2)$. In the worst case there are $|E| = |V|(|V| - 1)/2$ edges, hence in terms on number of nodes ILP-RBL's complexity is $O(|V|^2|P|^2)$. However, for task graphs representing real applications, we usually have $O(|E|) = O(|V|)$.

2.5.2 ILP-TransitivityClause

The focus of this ILP formulation is to eliminate the x variables from the formulation, as the ILP is over defined in terms of variables. If we can reformulate A09 without x, we can drop A11, A12 and A16. We replace the x variables in A09 with ϵ variables that enforce partial ordering of the processor indices with the help of an additional transitivity clause. A09 and A10 are replaced with A19 and A20. All other equations are retained.

$$\forall j \in V : i \in \delta^{-}(j) \qquad t_i + L_i + \gamma_{ij}(\epsilon_{ij} + \epsilon_{ji}) \leq t_j \tag{A19}$$

$$\forall i \neq j \neq k \in V \qquad \epsilon_{ij} + \epsilon_{jk} \geq \epsilon_{ik} \tag{A20}$$

For the entire graph, A19 produces $|E|$ constraints and A20 produces $|V|^3$ additional constraints but are independent of the number of processors unlike in A09. The number of variables in A19 is $O(1)$. So the complexity of this linearisation is $O(|V|^3)$, which can be better than $O(|E||P|^2)$ of the ILP-RBL formulation for graphs with many edges, i.e. E is large, and relatively higher number of processors in comparison with V.

2.5.3 Packing formulation

The packing formulation in [11] introduces a binary variable z to aid the linearisation of the bilinear equation. A09 and A10 of ILP-RBL are replaced by A21. Depending on the linearisation used either A22 or (A23-A24) is also used in the packing formulation. All others from A01 to A18 are retained. A21 uses a linearisation variable z_{ij}^{hk}

$$\forall j \in V : i \in \delta^{-}(j) \qquad t_i + L_i + \sum_{h,k \in P} \gamma_{ij}^{hk} z_{ij}^{hk} \leq t_j \tag{A21}$$

where $\forall j \in V \ : \ i \in \delta^{-}(j), h, k \in P \quad (z_{ij}^{hk} = x_{ih} x_{jk})$.

The packing formulation uses two linearisation approaches. The first linearisation uses A22

$$\forall j \in V, i \in \delta^{-}(j), h, k \in P \qquad (x_{ih} \geq z_{ij}^{hk} \wedge x_{jk} \geq z_{ij}^{hk} \wedge x_{ih} + x_{jk} - 1 \leq z_{ij}^{hk}) \tag{A22}$$

The second linearisation uses A23 and A24

$$\forall i \neq j \in V, k \in P \qquad \sum_{h \in P} z_{ij}^{hk} = x_{jk} \tag{A23}$$

$$\forall i \neq j \in V, h, k \in P \qquad z_{ij}^{hk} = z_{ji}^{kh} \tag{A24}$$

Both linearisation approaches require A21 to model communication between tasks running on different processors. A21 produces $|E|$ constraints and $O(|P|^2)$ variables in terms of the processor combinations over z_{ij}^{hk}. A22 produces $|E||P|^2$ constraints and $O(1)$ variables per constraint. Hence, the complexity of the first linearisation by A21 and A22 in terms of number of constraints is $O(|E||P|^2)$. A23 produces $O(|V|^2|P|)$ constraints and $O(|P|)$ variables per constraint. So, the complexity of the second linearisation by A21 and A23 in terms of number of constraints is $O(|E| + |V|^2|P|) = O(|V|^2|P|)$. However, through the linearisation of A21, there are always $|V|^2|P|^2$ z variables for both linearisations. Experimental results from [11] indicate that the first linearisation is beneficial for sparse graphs and the second linearisation is better for dense graphs. When the first linearisation is run over sparse graphs, the $|E|$ to $|P|^2$ ratio decreases as the sparsity of the graph increases making it faster.

Comparing the number of constraints complexity, ILP-RBL ($O(|E||P|^2)$) has the same complexity as the first linearisation. The constraint complexity comparison between ILP-RBL and the second linearisation ($O(|V|^2|P|)$) will depend on the number of edges and processors. ILP-TC ($O(|V|^3)$) will have a competitive number of constraints if $|P|$ is high. The major advantage of the two proposed formulations is that we have only $O(|V||P| + |V|^2)$ variables (x_{ik} and $\sigma_{ij}, \epsilon_{ij}$) for ILP-RBL and only $O(|V|^2)$ variables ($\sigma_{ij}, \epsilon_{ij}$) for ILP-TC to assign a value to.

2.6 Computational results

In this section we compare the performance of the two new proposed ILP formulations with both linearisations of the packing formulation in [11]. The result table for the packing formulation displays the best solution time amongst its two linearisations and is compared with ILP-RBL or ILP-TC. The computations are carried out using CPLEX 11.0.0 [2] on an Intel Core i3 processor 330M, 2.13 GHZ CPU and 2 GB RAM running with no parallel mode and on a single thread on Windows 7. 2 GB RAM was found to be a reasonable amount of physical memory for executing the ILP's used in this experiment. Extra RAM does not improve the speed of the program execution, but delays CPLEX running out of memory with large problems.

2.6.1 Experimental setup

For comparability, all experiments are run for a fully connected processor network with identical bandwidth capacity. The input graphs for this comparison are taken from those proposed and used in [11, 12]. The graph files with a name starting with `ogra_` are suffixed with the number of tasks in that file followed by its edge density in terms of a percentage of the maximum possible number of edges (i.e. $|V|(|V|-1)$). According to [11], they have a special graph structure that makes it hard to find the task ordering which yields the optimal solution when the number of mutually independent tasks are large. The graph file with a name starting with `t_` were generated randomly and are suffixed with the number of tasks in that file followed by its edge density and the index used to distinguish graphs of the same characteristics. The experiments are run on small to medium sized instances of the graphs on 4 and 8 processors.

2.6.2 Result table

The computational results for the graphs are given in Table 2.1, Table 2.2 and Table 2.3 over 4 processors and 8 processors. Not all problem instances were solved with all ILP formulations due to the excessive runtimes of the experiments, but the shown results are representative. The h:m:s notation is the standard Hours:Minutes: Seconds taken by the ILP formulation to find the solution. If the formulation is unable to find the optimal solution within 24 hours, the program is terminated and the gap (the difference between the lower bound and the best solution at that time (SL*)) is recorded. If the optimal schedule length is found, its value is displayed in the column corresponding to SL. Columns p and n record the number of processors and the number of tasks in the graph, respectively.

Table 2.1: Solution Time Comparison of Packing with ILP-RBL

Graph	p	n	SL	Packing	Gap	ILP-RBL	Gap
t30_60_1	4	30	467	7m:32s	0%	16s	0%
t40_10_1	4	40	233	10h:31m:37s	0%	24m:45s	0%
t40_25_1	4	40	270	4h:54m	0%	1m:37s	0%
Ogra50_60	4	50		24h	26.33% SL*=826	24h	3.02% SL*=612

Table 2.2: Solution Time Comparison of Packing with ILP-TC

Graph	p	n	SL	Packing	Gap	ILP-TC	Gap
Ogra20_75	8	20	100	51m:28s	0%	2m:37s	0%
t20_90	8	20	242	2m:24s	0%	7s	0%
t30_30_2	8	30	262	24h	0.69% SL*=287	4h:36m:18s	0%
t30_60_1	8	30	467	7h:22m:19s	0%	2h:6m:54s	0%

Table 2.1 compares the solution time for ILP-RevisedBooleanLogic (ILP-RBL) with the best solution time for the Packing linearisations in [11] for a 4 processor configuration over 30 to 50 nodes with varying densities. For these instances the solution time was 20 times or upward faster than the best version of the packing linearisations over a fully connected processor network. As can be observed, the speedup was achieved across different densities (ranging from 10% to 60%). Hence, the influence of the number of edges on ILP-RBL's relative performance was not as pronounced as could have been expected from the number of constraints complexity $O(|E||P|^2)$. For the second linearisation of the packing formulation the constraint complexity is $O(|V|^2|P|)$, so the ILP-RBL also benefited from the low number of processors. Despite this it seems that the strong improvement by ILP-RBL is explained with the lower number of variables, namely $O(|V||P| + |V|^2)$ variables for ILP-RBL compared with $O(|V|^2|P|^2)$ for the packing formulation.

Table 2.2 compares the solution time of ILP-TransitivityClause (ILP-TC) with the best solution time for the packing linearisations in [11]. The results in Table 2.2 are for ILP-TC on an 8 processor configuration over 20 to 30 nodes of varying density. We see that the solution time was 3 to 20 times faster than the best version of the packing formulation. The complexity of ILP-TC in terms of number of constraints is $O(|V|^3)$. This is independent of the number of processors unlike the packing formulation. Hence, ILP-TC benefits from the larger number of processors. But again, a large performance advantage is likely to come from the even further reduced number of variables, which is $O(|V|^2)$, thus does not depend on the number of processors.

Table 2.3: Solution Time Comparison of ILP-RBL with ILP-TC

Graph	p	n	SL	ILP-RBL	Gap	ILP-TC	Gap
t30_60_1	4	30	467	16s	0%	2h:39m:31s	0%
t40_25_1	4	40	270	1m:37s	0%	24h	34.72%
t30_90_1	8	30	562	4m:34s	0%	1m:19s	0%
t20_90	8	20	242	2m:23s	0%	7s	0%

Table 2.3 directly compares the two proposed ILP formulations. ILP-RBL, as expected, has a better solution time than ILP-TC when the number of processors is low. This is clear from the task graphs with 30 and 40 nodes on 4 processors, which have a better solution using ILP-RBL. ILP-TC was found to run faster on 20 and 30 nodes over 8 processors and high graph densities. This is in line with the expectation based on the complexities. We have seen that the complexity of ILP-RBL in terms of number of constraints is $O(|E||P|^2)$ and that of ILP-TC in terms of number of constraints is $O(|V|^3)$. Clearly, a higher density graph increases the solution time of ILP-RBL. A combination of the edge density and the number of processors serves as an indicator to decide between ILP-RBL and ILP-TC.

2.7 Conclusions

This chapter proposed two Linear Programming formulations for the Multiprocessor Scheduling Problem with Communication Delays. The improvement was in reducing the number of variables and constraints by the effective linearisation of the bilinear equation arising out of communication delays in the MSPCD model. The first of the proposed formulation ILP-RBL reworked the logic for Boolean multiplication and eliminated variables used to achieve the same result as in the packing formulation. The second proposed formulation ILP-TC also eliminates variables used in the packing formulation by enforcing the partial ordering of the processor indices with the help of an additional transitivity clause. We performed an experimental evaluation comparing the two proposed ILP formulations with the best previously published results. The linearisation used in ILP-RBL resulted in the formulation running faster over a small number of processors and the linearisation in ILP-TC resulted in it running faster over a larger number of processors.

3 Bi-Linear Reductions for MSPCD Using Integer Linear Programming

This chapter is a successor to the work in **Chapter 2**.

The **contributions** of the work in this chapter is to provide

- Optimal solutions for small and medium sized instances of the task scheduling problem.
- Propose a modification of the formulation ILP-TC in Chapter 2, based on a modified definition for the task overlap variables (that ensures tasks are separated in time when they run on the same processor).

Functionality

- To test if the modified overlap variable will result in an improvement in the runtime of the MILP.

Methodology

- The formulation is suitably modified to incorporate the new overlap variable and to maintain the soundness of the ILP.

Outcome

- No significant performance improvements with respect to ILP-TC but faster than the PACKING formulation.

Publication

Sarad Venugopalan and Oliver Sinnen. Bi-Linear Reductions for the Multiprocessor Scheduling Problem With Communication Delays Using Integer Linear Programming., 46th Annual Conference of the Operational Research Society of New Zealand, December 2012, p 378, ISSN 0114-6602.

3.1 Abstract

With computer processors running at speeds closer to their theoretical limit, the recent focus has turned to the use of parallelism in hardware by the use of multi-core processors for speedup. However, duplicating processors do not automatically translate to faster task execution. The tasks are to be carefully assigned and scheduled so that their total execution time on the multiple processors is minimal. We propose an optimal Integer Linear Programming formulation for the Multiprocessor Scheduling Problem with Communication Delays (MSPCD). The formulations use an effective method to linearise the bi-linear forms arising out of communication delays and introduce new overlap constraints to ensure that no two tasks running on the same processor overlap in time and space. The proposed formulation is compared with the previously discussed formulation ILP-TC.

3.2 Introduction

In the past, engineering and science have strongly benefited from an exponential growth in processor performance. Yet, due to the reached physical limits of processor technology the improvements are fading out [35] and manufacturers have moved to multi(core)processors. With multiple processors, however, the performance growth is not automatic [3] and can only be achieved when the processors are efficiently employed in parallel. Existing scheduling algorithms are therefore heuristics that try to produce good rather than optimal schedules, e.g.[31], [37], [21], [41], [48], [55], [58], [7], [23]. However, having optimal schedules can make a fundamental difference, e.g. for time critical systems or to enable the precise evaluation of scheduling heuristics. Given the NP-hardness of the processor scheduling problem [44], finding an optimal solution requires an exhaustive search of the entire solution space. For scheduling, this solution space is spawned by all possible processor assignments combined with all possible task ordering. Clearly this search space grows exponentially with the number of tasks, thus it becomes difficult already for very small task graphs. However, recent improvements in the Integer Linear Programming (ILP) formulation for the Multiprocessor Scheduling Problem with Communication Delays (MSPCD) [11], [52] allows larger instances of task graphs to be solved in a shorter time. The contribution of this work is to model the ILP based on the task overlap variable defined in [10] and compare it with the results in [11] and [52].

3.3 Task Scheduling Model

A fully connected network of homogeneous multiprocessors $P = \{1, \ldots, |P|\}$ with identical communication links is assumed. Each processor may execute several tasks but each task has to be assigned to exactly one processor, in which it is entirely executed without pre-emption. Further, no multitasking or parallelism is permitted within a task. The tasks to be scheduled are represented by a directed acyclic graph (DAG) defined by a 4-tuple G = (V, E, C, L) where $i \in V$ denotes the set of tasks and $(i, j) \in E$ represents the set of edges. The set E defines precedence relation between tasks. A task cannot be executed unless all of its predecessors have completed their execution and all relevant data is available. The set $C = \{\gamma_{ij} : i, j \in V\}$ denotes the set of edge communication time. If tasks i and j are executed on different processors $h, k \in P, h \neq k$, they incur a communication time penalty γ_{ij}. If both tasks are scheduled to the same processor the communication time is zero. For a graph with n tasks, the set $L = \{L_1, . . . , L_n\}$ represents the task computation times (execution time length). Let $\delta^{-}(j)$ be the set of precedents of task j, that is $\delta^{-}(j) = \{i \in V | (i, j) \in E\}$.

3.4 Related Work

Different approaches have been proposed to solve the MSPCD. One popular approach to the MSPCD makes use of Linear Programming [11]. This involves linearisation of the bilinear forms resulting from communication delays. The work in [11] discusses a classic formulation and a packing formulation of the MSPCD. Their results indicate that the packing formulation is about 5000 times faster than the classic formulation. The work in [52] further improves the ILP formulations in [11] and introduces two ILP formulations, one which runs faster when scheduled over a smaller number of processors and the other when scheduled over a larger number of processors. Another popular approach to solve the MSPCD is to use the A* search algorithm [28], [45]. A* is a best-first search technique [14], [42] and also a popular Artificial Intelligence algorithm guided by a problem specific cost function $f(s)$ for each solution state s. The main drawback of A* is that it keeps all the nodes in memory and it usually runs out of memory long before it runs out of time making it unusable for medium and large sized problem instances.

3.5 Bi-Linear Reductions

Let t_i be the start time of task i and t_j the start time of task j. Let L_i be the execution time of task i and γ_{ij} be the communication time between tasks i and j. Further, let $|P|$ be the total number of processors available for scheduling. Define

$$x_{ih} = \begin{cases} 1 & \text{task } i \text{ runs on processsor } h \\ 0 & \text{otherwise.} \end{cases}$$

If any two tasks i and j incur a communication cost, then

$$\forall j \in V \; : \; i \in \delta^{-}(j) \qquad t_i + L_i + \sum_{h,k \in P} \gamma_{ij}(x_{ih}.x_{jk}) \leq t_j \tag{A01}$$

By definition, x_{ih} and x_{jk} are Boolean variables and their multiplication needs to be linearised. The previous best linearisation in [11] uses two linearisation approaches. All though, the communication model in [11] has no restriction on the number of communication links, the model presented here assumes a fully connected network. Their linearisation variable z_{ij}^{hk} wherein task i runs on processor h and task j runs on processor k, is defined as

$$\forall j \in V \; : \; i \in \delta^{-}(j), h, k \in P \quad (z_{ij}^{hk} = x_{ih}.x_{jk})$$

Using this definition, the multiplication of the Boolean variables $x_{ih}.x_{jk}$ is replaced by the linearisation variable z_{ij}^{hk} in A02.

$$\forall j \in V \; : \; i \in \delta^{-}(j) \qquad t_i + L_i + \sum_{h,k \in P} \gamma_{ij}.z_{ij}^{hk} \leq t_j \tag{A02}$$

By A02, the number of constraints produced is $|E|$ and the number of variables per constraint in terms of the processor combinations over z_{ij}^{hk} is $O(|P|^2)$. The first linearisation uses A02 and A03 - A05.

$$\forall j \in V, i \in \delta^-(j), h, k \in P \qquad x_{ih} \geq z_{ij}^{hk} \qquad \text{(A03)}$$
$$\forall j \in V, i \in \delta^-(j), h, k \in P \qquad x_{jk} \geq z_{ij}^{hk} \qquad \text{(A04)}$$
$$\forall j \in V, i \in \delta^-(j), h, k \in P \qquad x_{ih} + x_{jk} - 1 \leq z_{ij}^{hk} \qquad \text{(A05)}$$

By A03 - A05, the number of constraints produced is $|E||P|^2$ and the number of variables per constraint is $O(1)$. Hence, the complexity of the first linearisation by A02, A03 - A05 in terms of number of constraints is $O(|E||P|^2)$.

The second linearisation uses A02 and A06 - A07.

$$\forall i \neq j \in V, k \in P \qquad \sum_{h \in P} z_{ij}^{hk} = x_{jk} \qquad \text{(A06)}$$
$$\forall i \neq j \in V, h, k \in P \qquad z_{ij}^{hk} = z_{ji}^{kh} \qquad \text{(A07)}$$

By A06, the number of constraints generated is $O(|V|^2|P|)$ and the number of variables per constraint is $O(|P|)$. So, the complexity of the second linearisation by A02 and A06 in terms of number of constraints is $O(|E| + |V|^2|P|) = O(|V|^2|P|)$.

3.6 Proposed Formulation

In this section a new ILP formulation (ILP-DELTA) is proposed and compared with the Packing formulation in [11] and ILP-TC in [52]. The ILP formulations, ILP-DELTA AND ILP-TC differ in the definition of the task overlap variable. They eliminate the use of the variable z for the linearisation of the bi-linear forms. This frees up to $|V|^2|P|^2$, z variables in the ILP formulation and speeds up the solution time.

3.6.1 ILP-DELTA

For each task $i \in V$ let $t_i \in \mathbf{R}$ be the start execution time and $p_i \in \mathbf{N}$ be the ID of the processor on which task i is to be executed. Let W be the total makespan and $|P|$ the number of processors available. The task overlap variable $\triangle_{ij}$ is modeled similar to the definition of the task overlap variable o_{ij} in [10]. If any two tasks i and j have a serial ordering in time, one of $\triangle_{ij}$ or $\triangle_{ji}$ is set to 1. Both $\triangle_{ij}$ and $\triangle_{ji}$ are set to 1 if the two tasks overlap in time. The variables $\triangle_{ij}$ and ϵ_{ij} are defined as follows:

$$\forall i, j \in V \quad \triangle_{ij} = \begin{cases} 1 & \text{task } j \text{ finishes after task } i \text{ starts} \\ 0 & \text{otherwise} \end{cases}$$

$$\forall i, j \in V \quad \epsilon_{ij} = \begin{cases} 1 & \text{PI of task } i \text{ is less than of task } \ j \\ 0 & \text{otherwise} \end{cases}$$

where PI is the Processor Index.

$$
\begin{aligned}
&min && W && \text{(A11)}\\
&\forall i \in V && t_i + L_i \leq W && \text{(A12)}\\
&\forall i \neq j \in V && \triangle_{ij} + \triangle_{ji} \geq 1 && \text{(A13)}\\
&\forall i \neq j \in V && \epsilon_{ij} + \epsilon_{ji} \leq 1 && \text{(A14)}\\
&\forall i \neq j \neq k \in V && \epsilon_{ij} + \epsilon_{jk} \geq \epsilon_{ik} && \text{(A15)}\\
&\forall i \neq j \in V && \triangle_{ij} + \triangle_{ji} + \epsilon_{ij} + \epsilon_{ji} \geq 1 && \text{(A16)}\\
&\forall i \neq j \in V && \triangle_{ij} + \triangle_{ji} - 1 \leq \epsilon_{ij} + \epsilon_{ji} && \text{(A17)}\\
&\forall i \neq j \in V && p_j - p_i - 1 - (\epsilon_{ij} - 1)|P| \geq 0 && \text{(A18)}\\
&\forall j \in V \;:\; i \in \delta^{-}(j) && t_i + L_i + \gamma_{ij}(\epsilon_{ij} + \epsilon_{ji}) \leq t_j && \text{(A19)}\\
&\forall i \neq j \in V && t_j - t_i - L_i - (\triangle_{ij} - \triangle_{ji} - 1)W_{max} \geq 0 && \text{(A20)}\\
&\forall j \in V \;:\; i \in \delta^{-}(j) && \triangle_{ij} = 1 && \text{(A21)}\\
& && W \geq 0 && \text{(A22)}\\
&\forall i \in V && t_i \geq 0 && \text{(A23)}\\
&\forall i, j \in V && \triangle_{ij}, \epsilon_{ij} \in \{0, 1\} && \text{(A24)}\\
&\forall i \in V && p_i \in \{1, \ldots, |P|\} && \text{(A25)}
\end{aligned}
$$

The upper bound on the makespan W is given by W_{max}

$$W_{max} = \sum_{i \in V} L_i + \sum_{i,j \in V} \gamma_{ij} \qquad \text{(A26)}$$

A11 and A12 are min-max constraints and minimises the maximum start task execution times. A12 specifies that the sum of task start time and its execution time is to be less than or equal to the makespan W. A13 - A17 are overlap constraints. Together, they ensure that no two tasks overlap in time and space. I.e. if two tasks have overlapping execution times, then they must run on different processors. The variable $\triangle$ defines a serial ordering of tasks in time if $\triangle_{ij}$ or $\triangle_{ji}$ is set to 1. If the execution of two tasks i and j overlap in time, both $\triangle_{ij}$ and $\triangle_{ji}$ are simultaneously set to 1. By A13, at least one of $\triangle_{ij}$ or $\triangle_{ji}$ is set to 1. A14 mandates that the sum of ϵ_{ij} and ϵ_{ji} be less than or equal to 1. If two tasks i and j run on the same processor, then both ϵ_{ij} and ϵ_{ji} are set to 0. If the two tasks run on different processors, then one of ϵ_{ij} or ϵ_{ji} is set to 1, depending on which of the two task is assigned to a higher processor index. Both ϵ_{ij} and ϵ_{ij} cannot be simultaneously set to 1, as it is not possible to assign a task to a higher and lower processor index at the same time. In A15, the ϵ variables enforces a partial ordering of the processor indices with the help of an additional transitivity clause. A16 prevents task executions from overlapping in time on the same processor by setting either one of the $\triangle$ or ϵ variables to 1. By A17, if any two tasks i and j overlap in time (i.e. both $\triangle_{ij}$ and $\triangle_{ji}$ are set to 1), then either ϵ_{ij} or ϵ_{ji} is set to 1. A18 sets the processor constraint. It is used to enforces the condition, $p_j > p_i + 1$, if $\epsilon_{ij} = 1$. This ensures that if $\epsilon_{ij} = 1$, then the processor index of task j is higher than task i. A19 and A20 are timing constraints. A19 models the communication between tasks with edges. If any two tasks i and j run on different processors, then ϵ_{ij} or ϵ_{ji} is set to 1. This implies that a communication cost is incurred. If tasks i and j run on the same processor, then ϵ_{ij} and ϵ_{ji} are both set to 0. In this case, A19 reduces to $t_i + L_i \leq t_j$. For all tasks, A20 enforces the condition $t_j \geq t_i + L_i$ if and only if $\triangle_{ij} = 1$ and $\triangle_{ji} = 0$. I.e. only when the tasks have a serial ordering in time. A21 is an edge constraint. All variables $\triangle_{ij}$ for which there is an edge from task i to task j in the graph is set to 1. A22 to A26 are the bounds on the ILP formulation. In A26, the upper bound on the makespan is computed as the sum of all task execution costs and edge communication costs in the graph.

3.6.2 ILP-TC

In this formulation [52] the variable σ defines a serial task execution order in time if one of σ_{ij} or σ_{ji} is set to 1. If the tasks overlap in time, both σ_{ij} and σ_{ji} are set to 0. The variables σ_{ij} and ϵ_{ij} are defined as follows:

$$\forall i,j \in V \;\; \sigma_{ij} \;\; = \begin{cases} 1 & \text{task } i \text{ finishes before task } j \text{ starts} \\ 0 & \text{otherwise} \end{cases}$$

$$\forall i,j \in V \quad \epsilon_{ij} = \begin{cases} 1 & \text{PI of task } i \text{ is less than of task } j \\ 0 & \text{otherwise} \end{cases}$$

In ILP-TC, A17 is removed since it is no longer required by the definition of σ_{ij}. Constraints A13, A16 and A20 are replaced by A31, A32 and A33 respectively. All other constraints remain unchanged.

$$\forall i \neq j \in V \qquad \sigma_{ij} + \sigma_{ji} \leq 1 \qquad \text{(A31)}$$

$$\forall i \neq j \in V \qquad \sigma_{ij} + \sigma_{ji} + \epsilon_{ij} + \epsilon_{ji} \geq 1 \qquad \text{(A32)}$$

$$\forall i \neq j \in V \qquad t_j \geq t_i + L_i + (\sigma_{ij} - 1)W_{max} \qquad \text{(A33)}$$

By A31, the sum of σ_{ij} and σ_{ji} is utmost one. If there is a serial ordering of the tasks executed, either σ_{ij} or σ_{ji} is set to 1 depending on which task finishes its execution before the other starts. If the two tasks overlap in time, both σ_{ij} and σ_{ji} are set to 0. By A32, at least one of the 4 variables ϵ_{ij}, ϵ_{ji}, σ_{ij} or σ_{ji} must be set to 1. A32 ensures that no two tasks i and j run on the same processor if their execution overlaps in time. By A33, if $\sigma_{ij} = 1$ then $t_j \geq t_i + L_i$.

Both ILP-DELTA and ILP-TC differ by 3 constraints, namely A31, A32 and A33. A simpler definition of the task overlap variable in ILP-TC allows A17 to be dropped. ILP-DELTA and ILP-TC have a constraint complexity of $O(|V|^3)$ due to A15. However, both these formulations have only $O(|V|^2)$ variables $(\sigma_{ij}, \epsilon_{ij})$ to assign a value to. Computational results indicate that though these formulations are faster than the Packing formulation, there is no clear winner between ILP-DELTA and ILP-TC as they exhibit a similar run time.

3.7 Computational Results

In this section, we compare the run times of the proposed formulation (ILP-DELTA) with the Packing formulation [11] and ILP-TC [52]. The computations are carried out using CPLEX 11.0.0 [2] on an Intel Core i3 processor 330M, 2.13 GHZ CPU and 2 GB RAM running with no parallel mode and on a single thread on Windows 7.

3.7.1 Experimental Setup and Result Table

All experiments are run for a fully connected processor network with identical bandwidth capacity. The input graphs for this comparison are taken from those proposed and used in [11, 12]. The graph files with a name starting with `ogra_` are suffixed with the number of tasks in that file followed by its edge density in terms of a percentage of the maximum possible number of edges (I.e. $|V|(|V|-1)/2$). According to [11], they have a special graph structure that makes it hard to find the task ordering which yields the optimal solution when the number of mutually independent tasks is large. The graph file with a name starting with `t_` were generated randomly and are suffixed with the number of tasks in that file followed by its edge density and the index used to distinguish graphs of the same characteristics. The Stencil graph is suffixed by the number of tasks followed by the Computational cost to Communication Ratio (CCR) value.

The experiments are run on small to medium sized instances of the graphs on 8 processors. ILP-TC [11] is designed to work well for tasks scheduled on to a larger number of processors. Since ILP-DELTA is modeled similar to ILP-TC, the proposed ILP also works well for tasks assigned to a larger number of processors. For tasks assigned to a smaller number of processors (e.g. 2 or 4), ILP-RBL [11] is well tailored and suitable for the purpose. A 24 hour time out is set for all the task graphs solved. If the execution exceeds 24 hours, the execution is terminated and the *gap* recorded. The *gap* gives a guaranteed lower bound on the optimal schedule length. For e.g. if the gap is 0.69% at 24 hours, it implies the optimal schedule length is within 0.69% of the schedule length returned by the ILP solver at 24 hours. The usual timing convention h:m:s is used to denote hours:minutes:seconds.

Table 3.1: Solution Time Comparison of ILP-DELTA with Packing and ILP-TC

Graph	p	n	Packing	ILP-DELTA	ILP-TC
Ogra20_75	8	20	51m:28s	3m:30s	2m:37s
t20_90	8	20	2m:24s	2s	7s
t30_30_2	8	30	24h, 0.69% gap	5h:15m:11s	4h:36m:18s
t30_60_1	8	30	7h:22m:19s	1h:39m:10s	2h:6m:54s
Stencil15_CCR_1	8	15	14m	35s	16s

Table 3.1 compares the solution time of ILP-DELTA with the Packing Formulation and ILP-TC. For these instances the solution time of ILP-DELTA or ILP-TC is found to be 5 to 20 times or upward faster than the best version of the Packing formulation. The result table indicates that changing the definition of the task overlap variable does not result in a significant difference in the run time between ILP-DELTA and ILP-TC.

3.8 Conclusions

An ILP formulation for the MSPCD was proposed and modeled based on the task overlap variable defined in [10] and compared with known ILP formulations in [11] and [52]. The ILP formulations in ILP-DELTA and ILP-TC eliminated the use of the variable z for the linearisation of the bi-linear forms, hence speeding up the solution time. It was found that the proposed formulation easily outperforms the packing formulation in [11] but had a similar run time as compared to ILP-TC, when scheduled on a larger number of processors. Although ILP-DELTA and ILP-TC use different task overlap variables, their run time were similar when formulated in a concise form.

4 ILP formulations for Task Scheduling with Communication Delays

This chapter is a successor to the work in **Chapter 2**.

The **contributions** of the work in this chapter are to provide

- Optimal solutions for small and medium sized instances of the task scheduling problem.
- Propose a novel mixed integer linear programming (MILP) solution to this scheduling problem, to further speedup the solution time.
- For the proposed formulations, all variable indices in the MILP formulation are independent of the number of processors. As a result, the constraint complexity of the proposed MILP formulation is reduced to $O(|V|^2)$.

- Analyse and discuss the influence of the different MILP components with respect to characteristics of the task graph such as structure and communication to computation ratio.
- Compare experimentally the proposed MILP formulation with previous MILP formulations used to solve this scheduling problem.
- Observe strengths and weaknesses of the formulation related to the input characteristics.

Functionality

- Use problem specific knowledge to fully eliminate the need to linearise the bi-linear equations arising out of communication delays.

- The size of the proposed formulation in terms of variables is made independent of the number of processors.

Methodology

- Modify the formulations ILP-RBL and ILP-TC to add the above required functionality.

Outcome

- Proposed formulation displays a drastic improvement in performance, which allows to solve larger problems.

Publication

Sarad Venugopalan and Oliver Sinnen. ILP formulations for Optimal Task Scheduling with Communication Delays on Parallel Systems, IEEE Transactions on Parallel and Distributed Systems, 26(1): 142-151 (2015). DOI: 10.1109/TPDS.2014.2308175.

4.1 Abstract

To fully benefit from a multiprocessor system, the tasks of a program are to be carefully assigned and scheduled on the processors of the system such that the overall execution time is minimal. The associated task scheduling problem with communication delays, $P|prec, c_{ij}|C_{max}$, is a well known NP-hard problem. We propose a novel Mixed Integer Linear Programming (MILP) solution to this scheduling problem, despite the fact that scheduling problems are often difficult to handle by MILP solvers. The proposed MILP solution uses problem specific knowledge to eliminate the need to linearise the bi-linear equations arising out of communication delays. Further, the size of the proposed formulation in terms of variables is independent of the number of processors. We analyse and discuss the influence of the different MILP components in respect to characteristics of the task graph such as structure and communication to computation ratio. The proposed MILP formulation is experimentally compared with previous MILP formulations used to solve this scheduling problem. The proposed formulation displays a drastic improvement in performance, which allows to solve larger problems optimally. We also observe strengths and weaknesses of the formulation related to the input characteristics.

4.2 Introduction

For the performance and efficiency of a parallel program, the scheduling of its (sub)tasks is crucial. Unfortunately, scheduling is a fundamentally hard problem (an NP-hard optimisation problem [44]), as the time needed to solve it optimally grows exponentially with the number of tasks (unless $P = NP$). Existing scheduling algorithms are mostly heuristics as they try to produce good rather than optimal schedules, e.g. [31], [37], [21], [41], [48], [55], [58], [7], [23]. Having optimal schedules can make a fundamental difference, e.g. for time critical systems such as flight control, industrial automation, automotive applications, telecommunication systems, consumer electronics, robotics and multimedia systems. Multiprocessor systems are also popular in small portable devices such as cellphones or navigators to large systems such a industrial robots or aircraft. An optimal schedule may also be used as a benchmark to enable the precise evaluation of scheduling heuristics. Moreover, once an optimal schedule is found, it may be reused when a parallel program is rerun. Due to today's widespread use of parallel systems, an efficient parallelisation is fundamental to take advantage of the computational power available. It is hence of enormous practical significance to be able to schedule small and medium sized task graphs optimally on parallel processors. The objective of this work is to present a fast Mixed Integer Linear Programming (MILP) formulation for the classic problem of scheduling task graphs on parallel systems with communication delay, which is $P|prec, c_{ij}|C_{max}$ in the $\alpha|\beta|\gamma$ notation [18], [51]. Many heuristics have been proposed for task scheduling on parallel systems [27]. While they often provide good results and tend towards the optimal schedule there is no guarantee that the solutions are optimal, especially for task graphs with high communication costs [47], [46]. A number of approximation algorithms have been proposed for the scheduling problem [9], [17]. For the here addressed scheduling problem, $P|prec, c_{ij}|C_{max}$, no α-approximation is known [15]. The only known guaranteed approximation algorithm in [24] has an approximation factor depending on communication costs of the longest path in the schedule.

Given the NP-hardness, finding an optimal solution requires an exhaustive search of the entire solution space. For scheduling, this solution space is spawned by all possible processor assignments combined with all possible task orderings. Clearly this search space grows exponentially with the number of tasks, thus it becomes impractical already for very small task graphs. Hence, few attempts have been made to solve $P|prec, c_{ij}|C_{max}$ optimally. With the increase in processor power in computers, it is now feasible to find optimal solutions to larger instances of the scheduling problem. The A* algorithm is one such popular optimal search algorithm used to solve this scheduling problem [28], [45]. It begins the search with an empty solution space and is then incrementally grown. A* employs a best-first search technique [14], [42] and is guided by a problem specific cost function. The main drawback of A* is that it keeps all the nodes in memory and it usually runs out of memory long before it runs out of time making it unusable for medium and large sized problem instances.

In this chapter, a MILP formulation is proposed for the task scheduling problem on parallel systems with communication delays. It uses an overlap approach [10] to set variables and constraints to ensure that no two tasks executing on the same processor overlap in time. The proposed formulation eliminates the need for the linearisation of bi-linear equations arising out of communication delays. Further, the number of variables is not a function of the number of processors, which is beneficial for the complexity of the formulation. The different components of the MILP formulation are analysed and their significance discussed. This helps to gain insights into the relevance of task graph characteristics for the efficient formulations of MILPs. An extensive experimental evaluation consisting of over 7400 schedules is carried out and compared with other existing MILP formulations that solve this scheduling problem. The proposed formulation displays a drastic improvement in performance. The proposed formulation is found to outperform other MILPs especially when the Communication to Computation Ratio (CCR) of the task graph is high. It is also seen from the experiments that a larger number of processors do not necessarily mean a slow down in the runtime of the MILP.

The rest of the chapter is organised as follows: Section 4.3 discusses the general use of MILP formulations for scheduling problems. Section 4.4 then describes the task scheduling model. Section 4.5 discusses bi-linear forms arising out of communication delays and its linearisation for the studied scheduling problem. Section 4.6 discusses the proposed mathematical formulation, its relaxation and reduction using MILP. The constraint complexity of the proposed formulation is compared with other known formulations. Section 4.7 details the experimental results wherein the runtime of the proposed formulation is compared with other known formulations in literature.

4.3 Mixed Integer Linear Programming

MILP may be used to solve optimisation problems, including scheduling problems. The MILP formulations can be broadly classified as discrete time and continuous time approaches [10], [16]. The discrete time approach introduces a new variable for each instant of time on each processor [1]. The number of time variables introduced in this approach explode when diverse execution times are present in the formulation. The continuous time approach, on the other hand, can handle diverse execution times, but its efficiency depends on how well the constraints and variables are formulated. The continuous time approach is further subdivided into three lines - sequencing, slots and overlaps. In sequencing, the formulation involves invoking new variables to determine if one task is executed after another task on the same processor [8], [5]. The number of constraints required to enforce the schedule requirements on each processor are known to grow quickly. In slots, each task is assigned to a space-time vacancy on a processor. The slot defines an order of tasks running on a processor [13], [32]. The start time and end time of tasks entering the slot are not fixed a priori. Since the exact number of slots required on each processor is not known a priori, a conservative number of slots (the number of tasks) has to be reserved and it suffers from a variable blow-up if the number of tasks to be scheduled is large. In overlap, variables are defined to prevent overlap of tasks scheduled on the same processor. Unlike other approaches, the number of variables and constraints in the formulation scales well as the number of tasks to be scheduled increases [10], [11], [52].

4.4 Task scheduling model

The tasks that are to be scheduled are represented as a weighted directed acyclic graph. The nodes in the graph represent tasks while directed edges represent data precedence relationships. Precedence relationships (if any) have to be respected at all times. The node cost is the time required for the task to complete its execution on a processor and the edge cost is the communication time between two tasks on different processors. If two tasks with data dependence are mapped onto the same processor, the communication between them is implemented by data sharing in local memory and no communication delay is incurred. The model assumes a fully connected network of homogeneous multiprocessors $P = \{1, \ldots, |P|\}$ with identical communication links. Each processor may execute several tasks, but each task has to be assigned to exactly one processor, in which it is entirely executed without pre-emption. Further, no multitasking or parallelism is permitted within a task. The execution time for each task on each processor and the data transfer times (or communication delays) between tasks with data dependence are given in advance as part of the task graph.

Formally, the tasks to be scheduled are represented by a directed acyclic graph (DAG) defined by a 4-tuple G=(V, E, C, L) where V denotes the set of tasks and E represents the set of edges. Each edge $(i, j) \in E$ defines a precedence relation between the tasks $i, j \in V$. A task cannot be executed unless all of its predecessors (parents) have completed their execution and all relevant data is available. The set $C = \{\gamma_{ij} : (i, j) \in E\}$ denotes the set of edge communication times. If tasks i and j are executed on different processors $h, k \in P, h \neq k$, they incur a communication time penalty γ_{ij}. If both tasks are scheduled to the same processor the communication time is zero. For a graph with $|V| = n$ tasks, the set $L = \{L_1 \ldots, L_n\}$ represents the task computation times (execution time length). Let $\delta^-(j)$ be the set of precedents of task j, that is $\delta^-(j) = \{i \in V|(i, j) \in E, j \in V\}$. The variables t_i and p_i are the main variables that describe a schedule for the problem to be solved. The start time of task i is t_i and the processor on which task i executes is p_i. The objective of this task scheduling problem is to allocate and schedule the tasks onto the processors such that the overall completion time W (makespan) is minimised [11], [40].

4.5 Bi-linear reductions

Communication between tasks executing on different processors results in bi-linear constraints and needs to be linearised. A fast linearisation is crucial in developing an efficient MILP formulation for the task scheduling problem. A commonly used linearisation of the bi-linear forms arising out of communication delays for the task scheduling problem called the usual linearisation and compact linearisation are discussed in [30], [29]. The MILP formulations in [11] use the linearisation in [30] to solve the task scheduling problem and is categorised under the overlap approach discussed in Section 4.3.

We are now looking at that linearisation. Let t_i be the start time of task i and t_j

the start time of task j. Define the following variable for the MILP of the scheduling problem

$$x_{ih} = \begin{cases} 1 & \text{task } i \text{ runs on processor } h \in P \\ 0 & \text{otherwise} \end{cases}$$

When using this variable in the formulation, the precedence constraint created by an edge between two tasks i and j incurring a communication cost, is then

$$\forall j \in V : i \in \delta^{-}(j) \qquad t_j \geq t_i + L_i + \sum_{h,k \in P, h \neq k} \gamma_{ij}(x_{ih}.x_{jk}) \tag{A01}$$

So task j can only start after task i has finished (t_i+L_i) plus the communication time. Remember, the communication cost is only incurred if the tasks are on different processors, otherwise it is zero. The task scheduling model presented in Section 4.4 assumes a fully connected network with identical communication links. Hence, the communication time between any two tasks i and j running on different processors is given by γ_{ij}. By definition, x_{ih} and x_{jk} are Boolean variables and their multiplication needs to be linearised.

The linearisation in [11] uses two different approaches. The linearisation variable z_{ij}^{hk}, where task i runs on processor h and task j runs on processor k, is defined as

$$\forall j \in V : i \in \delta^{-}(j), h, k \in P \qquad z_{ij}^{hk} = x_{ih}.x_{jk} \tag{A02}$$

Using this definition, the multiplication of the Boolean variables $x_{ih}.x_{jk}$ of A01 is replaced by the linearisation variable z_{ij}^{hk} resulting in

$$\forall j \in V : i \in \delta^{-}(j) \qquad t_j \geq t_i + L_i + \sum_{h,k \in P, h \neq k} \gamma_{ij}.z_{ij}^{hk} \tag{A03}$$

By A03, the number of constraints produced is $|E|$ and the number of variables per constraint in terms of the processor combinations over z_{ij}^{hk} is $O(|P|^2)$.

The first linearisation method called **PACKING-USUAL** makes use of A03 and needs the additional A04 - A06 for the complete linearisation.

$$\forall j \in V, i \in \delta^-(j), h, k \in P \qquad x_{ih} \geq z_{ij}^{hk} \tag{A04}$$

$$\forall j \in V, i \in \delta^-(j), h, k \in P \qquad x_{jk} \geq z_{ij}^{hk} \tag{A05}$$

$$\forall j \in V, i \in \delta^-(j), h, k \in P \qquad z_{ij}^{hk} \geq x_{ih} + x_{jk} - 1 \tag{A06}$$

A04 - A06 are used to simulate the logic of a Boolean multiplication using linear inequalities. By A04 - A06, the number of constraints produced is $|E||P|^2$ and the number of variables per constraint is $O(1)$. Hence, the total complexity of the PACKING-USUAL linearisation in terms of number of constraints is $O(|E||P|^2)$.

The second linearisation method called **PACKING-COMPACT** uses A03 plus A07 - A08, instead of A04 - A06.

$$\forall i \neq j \in V, k \in P \qquad \sum_{h \in P} z_{ij}^{hk} = x_{jk} \tag{A07}$$

$$\forall i \neq j \in V, h, k \in P \qquad z_{ij}^{hk} = z_{ji}^{kh} \tag{A08}$$

A07 is obtained by multiplying both sides of the equality A09 with x_{jk}; $\forall i \neq j \in V$, $k \in P$. A09 implies that any given task can run on exactly one processor.

$$\forall i \in V \qquad \sum_{h \in P} x_{ih} = 1 \tag{A09}$$

A08 indicates that the multiplication $z_{ij}^{hk} = x_{ih}.x_{jk}$ is commutative. By A07, the number of constraints generated is $O(|V|^2|P|)$ and the number of variables per constraint is $O(|P|)$. So, the total complexity of PACKING-COMPACT in terms of number of constraints is $O(|E| + |V|^2|P|) = O(|V|^2|P|)$.

4.6 Proposed formulation

The literature surveyed indicates that amongst all MILP formulations, an overlap approach is best suited to tackle the task scheduling problem. All the MILP formulations discussed in this section are based on the overlap approach. A new MILP formulation for scheduling (SHD-BASIC), its relaxation (SHD-RELAXED) and reduction (SHD-REDUCED) are proposed and compared with the PACKING formulation in [11] as well as ILP-RBL and ILP-TC in [52]. The formulation in [11] utilise overlap variables adopted from [20] along with a technique for the linearisation discussed in the previous section. The formulation in [11] is improved in [52] by reworking the Boolean logic for dependent tasks and by defining a partial order using a transitivity clause.

The first contribution of this work is to use problem specific knowledge to eliminate the bi-linear forms [11] arising out of communication delays. The proposed formulation eliminates the use of the z variable (in Section 4.5) for the linearisation of the bi-linear forms. This frees up to $|V|^2|P|^2$ z variables and its associated constraint complexity in the MILP formulation and speeds up the runtime of the solver. The second contribution is to run all variable indices in the proposed MILP formulation independent of the number of processors. As a result, the constraint complexity of the proposed MILP reduces to $O(|V|^2)$.

Next, we propose the basic formulation SHD-BASIC in Section 4.6.1, its relaxation SHD-RELAXED in Section 4.6.2 and its reduction SHD-REDUCED in Section 4.6.3. SHD-RELAXED and SHD-REDUCED formulations are compared with PACKING formulation, ILP-RBL and ILP-TC in Section 4.6.4 to bring out the advantages in using the proposed formulation.

4.6.1 BASIC formulation (SHD-BASIC)

The MILP is formulated as a min-max problem that involves minimising the maximum task finish time. The variables t and p gives the allocation and schedule of tasks on processors. The σ and ϵ variables model the relative position of tasks respectively along time and on the processors. They together ensures that tasks running on the same processor do not overlap in time. The constraints relate these variables to allow a fast ILP without the need for linearisation. Let W be the total makespan and $|P|$ the number of processors available. For each task $i \in V$ let $t_i \in \mathbf{R}$

be the start execution time and $p_i \in \mathbf{N}$ be the ID of the processor on which task i is to be executed. In order to enforce non-overlapping constraints, define two sets of binary variables

$$\forall i, j \in V \quad \sigma_{ij} = \begin{cases} 1 & \text{task } i \text{ finishes before task } j \text{ starts} \\ 0 & \text{otherwise} \end{cases}$$

$$\forall i, j \in V \quad \epsilon_{ij} = \begin{cases} 1 & \text{PI of task } i \text{ is less than of task } j \\ 0 & \text{otherwise} \end{cases}$$

where PI is the Processor Index.

Based on these two types of binary variables the SHD-BASIC formulation is proposed next, followed by a detailed explanation of the role of each constraint. A10 is the objective, A11 - A19 are the main constraints and A20 - A24 are the bounds.

$$\begin{aligned}
&min && W && \text{(A10)}\\
&\forall i \in V && t_i + L_i \leq W && \text{(A11)}\\
&\forall i \neq j \in V && \sigma_{ij} + \sigma_{ji} \leq 1 && \text{(A12)}\\
&\forall i \neq j \in V && \epsilon_{ij} + \epsilon_{ji} \leq 1 && \text{(A13)}\\
&\forall i \neq j \in V && \sigma_{ij} + \sigma_{ji} + \epsilon_{ij} + \epsilon_{ji} \geq 1 && \text{(A14)}\\
&\forall i \neq j \in V && p_j - p_i - 1 - (\epsilon_{ij} - 1)|P| \geq 0 && \text{(A15)}\\
&\forall i \neq j \in V && p_j - p_i - \epsilon_{ij}|P| \leq 0 && \text{(A16)}\\
&\forall i \neq j \in V && t_i + L_i + (\sigma_{ij} - 1)W_{max} \leq t_j && \text{(A17)}\\
&\forall j \in V : i \in \delta^-(j) && t_i + L_i + \gamma_{ij}(\epsilon_{ij} + \epsilon_{ji}) \leq t_j && \text{(A18)}\\
&\forall j \in V : i \in \delta^-(j) && \sigma_{ij} = 1 && \text{(A19)}\\
&\forall i, j \in V && \sigma_{ij}, \epsilon_{ij} \in \{0, 1\} && \text{(A20)}\\
&\forall i \in V && p_i \in \{1, .., |P|\} && \text{(A21)}\\
&\forall i \in V && t_i \geq 0 && \text{(A22)}\\
& && W \geq 0 && \text{(A23)}\\
& && \sum_{i \in V} L_i + \sum_{i,j \in V} \gamma_{ij} - W_{max} = 0 && \text{(A24)}
\end{aligned}$$

The constraints can be roughly categorised in the min-max objective (A10 - A11), overlap constraints (A12 - A14), processor constraints (A15 - A16) and precedence constraints (A17 - A19). To discuss the completeness and correctness of the formulation let us start by considering the case where there are no precedence constraints and hence no communication delays between tasks. Then, the minimisation objective A10 along with A11 - A15, A17, A20 - A24 are sufficient to give a valid formulation. For each task, A11 specifies that the sum of the task start time and its execution time (hence the task's finish time) is less than or equal W. By A12, the sum of σ_{ij} and σ_{ji} is at most 1 and A13 enforces the same for the sum of ϵ_{ij} and ϵ_{ji}. When tasks are assigned to processors, the tasks that run on the same processor are to be ordered by defining their start time. Let us consider the set V_x of tasks that are assigned to some processor x (I.e. all tasks $i \in V$ such that $p_i = x$). A15 enforces that for any two tasks $i, j \in V_x$, $\epsilon_{ij} = \epsilon_{ji} = 0$. Consequently, the non-overlapping constraint (A14) will imply an order on these two tasks by raising exactly one of the two variables σ_{ij} or σ_{ji} to 1 (by A12). This set of variables σ_{ij} for all pairs of tasks $i, j \in V_x$ will give a total order of the tasks on processor x (transitivity being enforced by A14 and A17). Then A17 will affect valid starting times to the tasks while A10 will define the objective to be minimised.

For this independent tasks case, an ordering on a pair of tasks executing on different processors is not required as there are no precedence constraints and do not require ϵ_{ij} or ϵ_{ji} to be set to 1 whenever $p_i \neq p_j$. To model the precedence and communication delays, A16, A18 and A19 are added to the independent tasks case. A19 pre-sets an ordering for all tasks that are connected by an edge and A17 affects the start times according to this order. A18 will ensure that the communication delays are taken into account, but to meet this aim, exactly one of ϵ_{ij} or ϵ_{ji} must be equal to 1 when $p_i \neq p_j$. This is guaranteed by the addition of A16. Note that A15 and A16 are both required to enforce that exactly one of ϵ_{ij} or ϵ_{ji} is set to 1.

Both σ_{ij} and ϵ_{ij} are Boolean variables due to A20. A21 gives the bound on processor allocation for each task $i \in V$. By A22, all tasks have a start time greater than or equal to zero. By A23, the makespan W is greater than or equal to zero. By A24, W_{max} gives an upper bound on W and is defined as the sum of all task execution times and edge communication times. This is a worst case value which ensures that A17 is correct when $\sigma_{ij} = 0$. Note that there is no strict requirement that σ values have to correspond to a transitive closure for the task graph. E.g. If $\sigma_{13} = 1$ and $\sigma_{35} = 1$, then the value of σ_{15} is not relevant with respect to time ordering.

4.6.2 RELAXED formulation (SHD-RELAXED)

The RELAXED formulation is introduced to speed up the BASIC formulation. In the BASIC formulation (SHD-BASIC), A16 is created for all edge pairs $i \neq j$. In the RELAXED formulation, these constraints are defined only for tasks with a direct edge between them. I.e. A25A and A25B are used instead of A16. The formulation using A25A and A25B instead of A16 is named SHD-RELAXED.

$$\forall j \in V : i \in \delta^{-}(j) \qquad p_j - p_i - \epsilon_{ij}|P| \leq 0 \qquad \text{(A25A)}$$

$$\forall j \in V : i \in \delta^{-}(j) \qquad p_i - p_j - \epsilon_{ji}|P| \leq 0 \qquad \text{(A25B)}$$

From Section 4.6.1, it is seen that A16 is included to model communication delays. However, since communication delay exists only between tasks that have an edge between them, A16 are needed only for such pairs of tasks.

4.6.3 REDUCED formulation (SHD-REDUCED)

The REDUCED formulation is introduced to remove redundant constraints that appear in SHD-RELAXED. The logic in A12 is redundant in A17 and the logic in A13 is redundant in A15. For A17: if $\sigma_{ij} = 1$, then $t_j \geq t_i + L_i$ and if $\sigma_{ji} = 1$, then $t_i \geq t_j + L_j$. If $\sigma_{ji} = \sigma_{ji} = 1$, then $t_j \geq t_i + L_i$ and $t_i \geq t_j + L_j$ should be simultaneously valid constraints. This is true only if $t_i = t_j$ and $L_i = L_j = 0$. If non-zero task execution times are considered, then by A08, at most one of the σ variable may be set to 1.

For A15: if $\epsilon_{ij} = 1$, then $p_j \geq p_i + 1$ and if $\epsilon_{ji} = 1$, then $p_i \geq p_j + 1$. If $\epsilon_{ij} = \epsilon_{ji} = 1$, then $p_j \geq p_i + 1$ and $p_i \geq p_j + 1$ should be simultaneously valid constraints. However, both the constraints cannot be simultaneously true as it is not possible to place a task on a higher and lower processor index at the same time. Hence, A15 may set at most one of the ϵ variables to 1. This allows A12 and A13 to be eliminated from SHD-RELAXED. The reduced formulation eliminating A12 and A13 from SHD-RELAXED is named SHD-REDUCED and the performance of SHD-REDUCED is compared with SHD-RELAXED. Note that A12 and A13 are redundant in the formulation but not for their linear relaxation.

Table 4.1: Comparison between formulations tested

	PACKING-USUAL	PACKING-COMPACT	ILP-RBL	ILP-TC	SHD-BASIC, RELAXED, REDUCED
VARIABLES	$\lvert E\rvert.\lvert P\rvert^2$, z variables	$\lvert V\rvert^2.\lvert P\rvert^2$ z variables	free of z	free of z	free of z
CONSTRAINTS	$O(\lvert V\rvert^2 + \lvert E\rvert\lvert P\rvert^2)$	$O(\lvert V\rvert^2.\lvert P\rvert)$	$O(\lvert V\rvert^2 + \lvert E\rvert\lvert P\rvert^2)$	$O(\lvert V\rvert^3)$	$O(\lvert V\rvert^2)$

4.6.4 Comparison of SHD-RELAXED and SHD-REDUCED with PACKING formulation, ILP-RBL and ILP-TC

SHD-RELAXED and its reduction SHD-REDUCED are used instead of SHD-BASIC in the following analysis. SHD-RELAXED generates $O(|E|)$ constraints using A25A - A25B whereas SHD-BASIC generates $O(|V|^2)$ constraints using A16. The PACKING formulation in [11] uses linearisation variables to linearise the bi-linear inequalities arising from communicating edges. These linearisation variables are eliminated in ILP-RBL [52] by reworking the Boolean logic of the communicating edges. ILP-TC [52] reworks the linearisation of the bi-linear forms in the PACKING formulation using a transitivity clause in a manner that aids the elimination of over-defined inequalities in ILP-RBL.

For uniformity across comparisons, it is noted that the task scheduling model for the PACKING formulation unlike SHD-BASIC and its variants do not mandate a fully connected processor network. Table 4.1 compares the variable and constraint complexities of the formulations tested.

When ILP-RBL is used to schedule a large number of tasks on a small number of processors, the contribution of $|P|^2$ towards the constraint complexity of ILP-RBL diminishes.

The proposed formulation SHD-RELAXED has a constraint complexity of $O(|V|^2)$ as all variable indices are free of the number of processors. The bound on the number of processors available for scheduling is given by A21. SHD-RELAXED is also free of the linearisation variable z, making it faster than the PACKING formulation. SHD-RELAXED and ILP-RBL have a similar constraint complexity over a small number of processors but is much faster than ILP-RBL over a larger number of processors. SHD-RELAXED also runs faster than ILP-TC which have a higher constraint complexity, as observed from Table 4.1. The reduced formulation SHD-REDUCED has fewer inequalities than SHD-RELAXED through the elimination of A12 and A13. However, as noted in Section 4.6.3, these inequalities are redundant only for the formulation but not their linear relaxation. The experiments carried out in Section 4.7 confirm that the complexity comparisons in this section are in agreement with the experimental results.

4.7 Experimental results

The main goals of this section are: (a) performance comparisons of the proposed MILP formulation SHD-RELAXED with both linearisation of the PACKING formulation in [11] (PACKING-USUAL and PACKING-COMPACT), the MILP formulations in [52], namely ILP-RBL and ILP-TC and the reduced formulation SHD-REDUCED from Section 4.6.3 (b) to analyse the behaviour of the MILP formulations with respect to graph structures (c) study the effect of Communication cost to Computation cost Ratio (CCR) on MILP formulation runtime.

The computations are carried out using CPLEX 11.0.0 [2] on an Intel Core i3 processor 330M, 2.13 GHZ CPU and 2 GB RAM running with no parallel mode and on a single thread on Windows 7.

All experiments are run for the task scheduling model discussed in Section 4.4. I.e. a fully connected processor network with identical bandwidth capacity is assumed. The input graphs used for experiments in Section 4.7.1 are from [45] and the input graphs used for benchmarking in Section 4.7.2 are from [11], [12]. The following two definitions of graph densities are used:

$$\Omega = |E|/|V| \tag{A26}$$

$$\omega = (|E|/\gamma)100 \tag{A27}$$

with the maximum possible number of edges in the graph as $\gamma = |V|(|V|-1)/2$. The two density equations A26 and A27 arise due the difference in density definitions of the task graph databases used. It is also worth noting that very high densities are not realistic for most real software applications.

Two types of performance comparison experiments are carried out: a 1 minute timeout in Section 4.7.1 and 12 hour timeout in Section 4.7.2. The 1 minute timeout experiments carried out in Section 4.7.1.1 are to get many results to learn about the performance behaviour and on which input characteristics it depends. Section 4.7.1.2 compares the relation between characteristics of the task graph structure with respect to the MILP formulations. Section 4.7.1.3 studies the effect of CCR on MILP formulations. The 12 hour timeout experiments in Section 4.7.2 are longer experiments for a direct comparison between runtime of the MILP formulations, especially in regards to previous work. Larger input graphs are tackled in these experiments. A Java implementation of all the MILP formulations compared and some of the optimal results that it returned can be found in the Green Banana (GB) scheduler suite [54]. They use the Graph eXchange Language (GXL) format [22] to represent input task graphs.

4.7.1 MILP comparisons with 1 minute timeout

A database of 207 task graphs, summarised in Table 4.2, comprising of 10, 21 and 30 tasks of the following structures: fork, join, fork-join, in-tree, out-tree, series-parallel, pipeline, random, stencil and independent tasks is chosen [45]. The densities for the graphs in the database conform to the definition in A26. The experiments are carried out for 2, 4, 8 and 16 processors. A one minute timeout is set for each graph.

Table 4.2: Detailed Structure of the 207 Graph Database

Graph Structure	$n = 10$	$n = 21$	$n = 30$	Total
Fork-Join	4	4	4	12
Fork	4	4	4	12
Independent	1	1	1	3
InTree	8	8	8	24
Join	4	4	4	12
OutTree	8	8	8	24
Pipeline	4	4	4	12
Random	16	16	16	48
Series-Parallel	16	16	16	48
Stencil	4	4	4	12

4.7.1.1 Overall performance evaluation

In this section, the overall performance of the graphs in the database are compared for the MILP formulations discussed. The number of graphs for which an optimal schedule was returned by CPLEX within one minute for the 207 graph set is tallied and plotted in Figure 4.1. The comparison criteria used here is completed schedules within one minute.

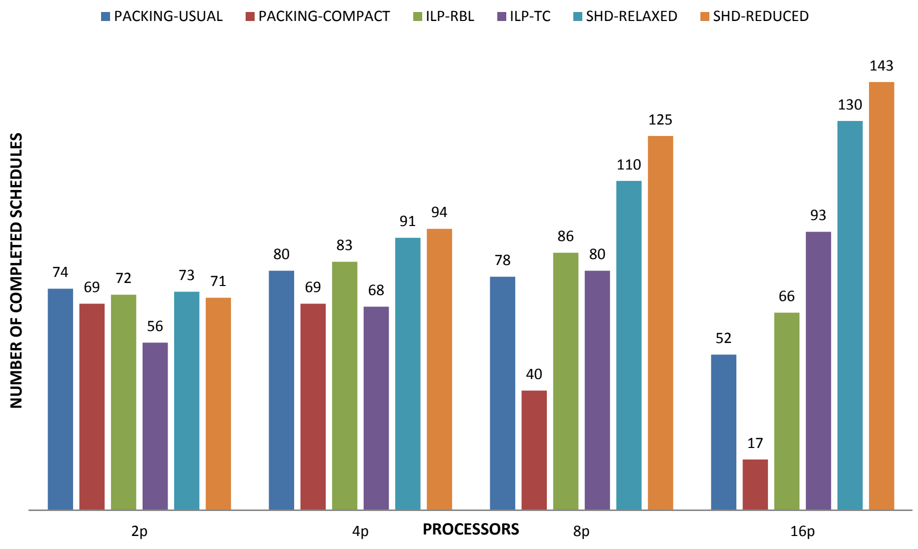


Figure 4.1: Completed schedules of different formulations over number of processors

It is seen from Figure 4.1 that SHD-RELAXED and SHD-REDUCED outperform all the other ILP's on 4, 8 and 16 processors. PACKING-USUAL marginally outperforms SHD-RELAXED and SHD-REDUCED over 2 processors, with 74 optimal solutions found in the 207 graph dataset. However, the performance of PACKING-USUAL and PACKING-COMPACT quickly degrades over 4, 8 and 16 processors in comparison to SHD-RELAXED and SHD-REDUCED. ILP-TC has a steady performance improvement over 2, 4, 8 and 16 processors unlike PACKING-USUAL, PACKING-COMPACT and ILP-RBL. This performance increase is attributed to the constraint complexity of ILP-TC that is independent of the number of processors. It is found that an increase in the number of processors does not necessarily imply a decrease in performance. Other than ILP-TC, only SHD-RELAXED and SHD-REDUCED have a steady performance increase over 2, 4, 8 and 16 processors. SHD-RELAXED and SHD-REDUCED also have a constraint complexity independent of the number of processors. Unlike ILP-TC with a constraint complexity of $O(|V|^3)$, the constraint complexity of SHD-RELAXED and SHD-REDUCED is $O(|V|^2)$. This improved constraint complexity matches with the experimental comparisons as it is observed that SHD-RELAXED and SHD-REDUCED outperforms ILP-TC.

4.7.1.2 Structure based performance evaluation

In this section, we analyse the performance of the MILP formulations over the graph structures given in Table 4.2. Figure 4.2, Figure 4.3, Figure 4.4 and Figure 4.5 depict the previous results separated by the individual graph structures over 2, 4, 8 and 16 processors for the formulations SHD-RELAXED, SHD-REDUCED, PACKING-USUAL, PACKING-COMPACT, ILP-RBL and ILP-TC.

From the 2 processor experiments in Figure 4.2, it is seen that SHD-RELAXED and SHD-REDUCED display a relatively similar performance with respect to PACKING-USUAL or PACKING-COMPACT except for PIPELINE and RANDOM graphs where PACKING-USUAL exhibits a better performance. ILP-RBL is also seen to perform well in comparison with the other formulations (as expected to be the case over a smaller number of processors). For the 4 processor experiments in Figure 4.3, it is observed that either SHD-RELAXED or SHD-REDUCED displays a similar or better performance in relation to the other formulations and for the 8 and 16 processor experiments (in Figure 4.4 and Figure 4.5 respectively), their performance gains over other formulations are observed to be prominent.

The Fork-Join graphs have the best performance with SHD-RELAXED and ILP-TC whereas the Forks perform best on SHD-REDUCED. Independent graphs have a similar performance with SHD-RELAXED, SHD-REDUCED, ILP-RBL, ILP-TC and PACKING-USUAL. InTree graphs on 2, 4 processors are seen to best perform with SHD-RELAXED, PACKING-USUAL and on 8, 16 processors are seen to best perform with SHD-REDUCED. OutTree graphs are seen to perform best with SHD-REDUCED whereas Pipeline graphs are seen to perform best with SHD-RELAXED (except on 2 processors where PACKING-USAL and ILP-RBL are seen to perform better). Random graphs are seen to perform best on SHD-REDUCED (except on 2 processors were PACKING-USUAL and ILP-RBL perform better). Series-Parallel and Stencil graphs are both seen to perform best over SHD-REDUCED. Overall, it is observed that SHD-RELAXED or SHD-REDUCED have the best performance for each graph structure.

Table 4.3 gives the percentage of individual graph structures that passed the 1 minute timeout test averaged over 2, 4, 8 and 16 processors for SHD-RELAXED and SHD-REDUCED in Figure 4.2 to Figure 4.5. It is seen that the graph structures that perform the best over these two MILP formulations are Random and Pipeline and the worst are Join, Fork and Independent. These graphs have a very similar structure because a Fork or Join is a set of Independent tasks with an edge from or to another task. Independent tasks are in general harder to schedule since they have no precedence constraints and all possible task combinations on an allocated processor have to be tried out. Stencil graphs are seen to perform better over SHD-RELAXED as compared to SHD-REDUCED. The other graph structures are found to have a similar performance with SHD-RELAXED and SHD-REDUCED.

Table 4.3: Performance Statistics on Individual Graph Structures over SHD-RELAXED and SHD-REDUCED

RELAXED	% Passed	REDUCED	% Passed
Random	62.5	Random	70
Pipeline	62.5	Pipeline	60.25
Stencil	54	Series-Parallel	56
Series-Parallel	52.25	OutTree	47.75
InTree	44.75	InTree	46.75
OutTree	42.5	Stencil	39.5
Fork-Join	35.25	Fork-Join	33.25
Join	27	Join	29.25
Independent	25	Fork	27
Fork	22.75	Independent	25

4.7.1.3 Effect of CCR on MILP formulation runtime

In this section, the effect of CCR on the performance of MILP formulation is studied. Of the 207 graphs in the database, 51 each are of CCR 0.1, CCR 1.0, CCR 2.0 and CCR 10. The independent graphs in the database have no communication edges and are not considered for the analysis. The number of graphs for which an optimal schedule was returned by CPLEX within one minute is tallied and plotted in Figure 4.6 for 4 and 8 processors.

It is observed that an increase in CCR results in a decrease in the performance of all the MILP formulations in the experiment irrespective of the number of processors

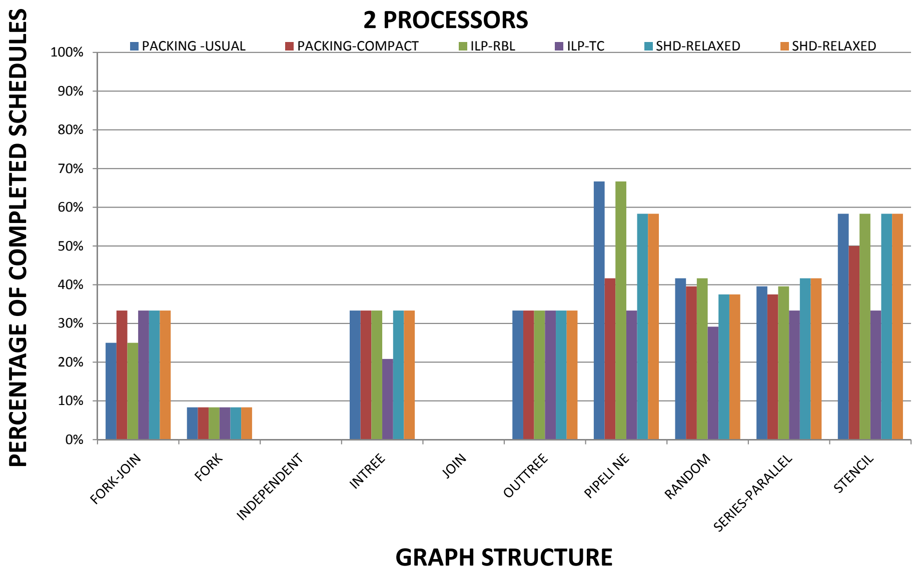


Figure 4.2: Completion percentage of formulations over graph structures 2 processors

they are scheduled on. A possible explanation for this behavior is that higher CCR values increase the schedule length variance between different schedules. A single task allocated to the wrong processor can imply a strong penalty on the schedule length due to large remote communication. The LP relaxation of the ILP formulations used by the solver can then be further away from the optimal solution of the ILP. This results in longer runtimes of the branch-and-bound part of the solver.

This CCR dependent behavior is very interesting as it shows that the ILP runtimes do not only related to the size of the input problem in terms of number of tasks, edges and processors, but also depends on the weight values.

4.7.2 MILP comparisons set for a 12 hour timeout

The 12 hour timeout experiments are for a direct comparison between runtimes of the MILP formulations. The input graphs used for benchmarking in this section are from [11], [12]. If the CPLEX solver is unable to find an optimal solution within 12 hours, the program is terminated and the results tabulated. The h:m:s notation is the standard hours:minutes:seconds taken by the MILP formulation to find an optimal solution. The graphs with a name starting with 't' were generated randomly and suffixed with the number of tasks in that graph followed by its edge density and the index used to distinguish graphs when they have the same n and ω

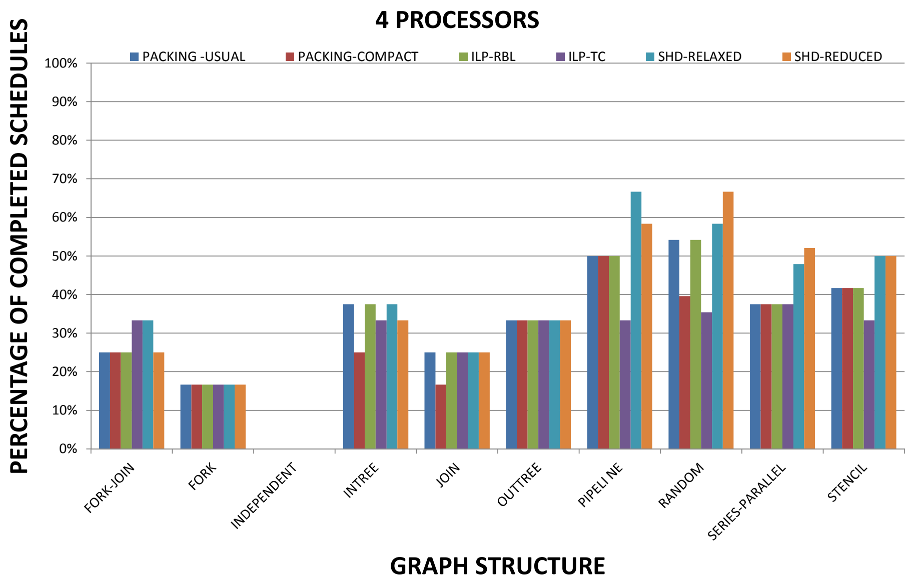


Figure 4.3: Completion percentage of formulations over graph structures on 4 processors

values. The graphs with a name starting with 'ogra' are suffixed with the number of tasks followed by its edge density. These edge densities conform to the percentage definition of density in (A27). According to [11], the optimal solution for 'ogra' graphs are obtained when the tasks are well packed (as in ideal schedule). This has similar characteristics with respect to a number of mutually independent tasks (that can be well packed as there are no communication delays), for which it is hard to find the task ordering which yields the optimal schedule. In Table 4.5, the first column gives the name of the graph, n records the number of tasks in the graph. The symbols Ω and ω conform to the density definition in A26 and A27 respectively. Column p refers to the number of processors. The rest of the columns record the solution time for the MILP formulations being compared. Where the comparison table refers to the MILP formulation as **PACKING**, it implies that both PACKING-USUAL and PACKING-COMPACT in [11] are used for the experiments and the shorter of the two MILP runtimes is recorded. When the CPLEX solver is unable to find an optimal solution within 12 hours, the program is terminated and the gap returned by CPLEX is recorded. Let the feasible schedule length returned when the program terminates be FSL and the optimal schedule length (to be found) be OSL. The gap (in percentage) is a guarantee that the the difference between the FSL and OSL is at most FSL times the *gap*. I.e. $FSL{-}OSL{\leq}FSL{\cdot}gap$. If a solution is found

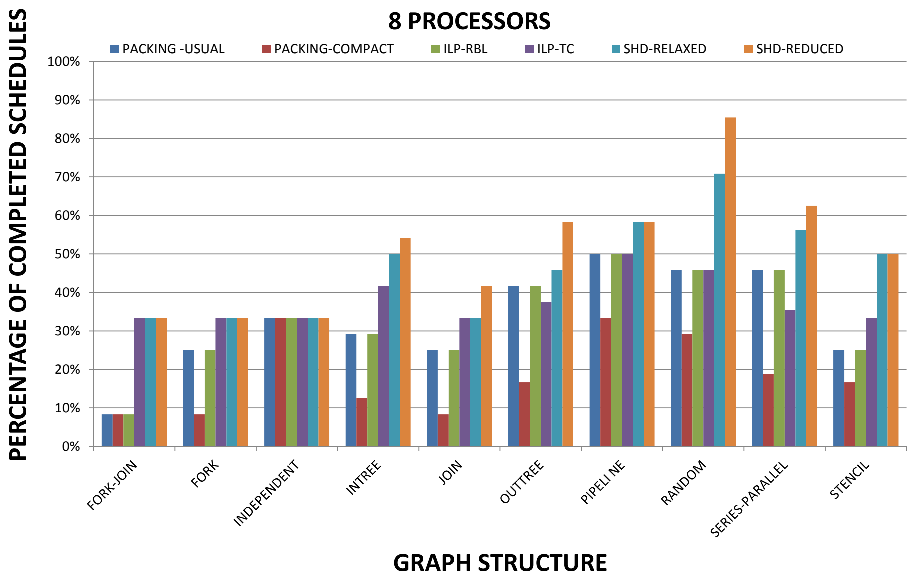


Figure 4.4: Completion percentage of different formulations over graph structures on 8 processors

within 12 hours, the gap is 0% and not recorded. If no feasible solution could be found within 12 hours, the gap is infinite and recorded as inf. For example, if the schedule length returned by the program when it terminates at the end of 12 hours is 200 units and the gap is 10%, then the optimal schedule length is guaranteed to be greater than or equal to 180 units.

Table 4.5 compares the proposed formulations SHD-BASIC, SHD-RELAXED and SHD-REDUCED with PACKING, ILP-RBL and ILP-TC over 2, 4, 8 and 16 processors in Table 4.4. The MILP formulation that yields the fastest result (or the least gap) is highlighted. It is seen that for most cases SHD-RELAXED or SHD-REDUCED runs faster than SHD-BASIC. Despite 'ogra' graphs having a special structure making it harder to solve optimally, it is seen that the proposed formulations have a significantly improved performance and outdo other formulations when more processors are available for scheduling. For the random 't' graphs, ILP-RBL runs fast over a smaller number of processors. ILP-TC for most cases displays an improved runtime with an increase in number of processors, but are found to run slower with an increase in the number of tasks in the graph.

It is seen from Table 4.4 that 10 out of 16 column entries for the PACKING formulation timeout at 12 hours, indicating a much longer runtime over the proposed formulations. For most instances, SHD-RELAXED and SHD-REDUCED are

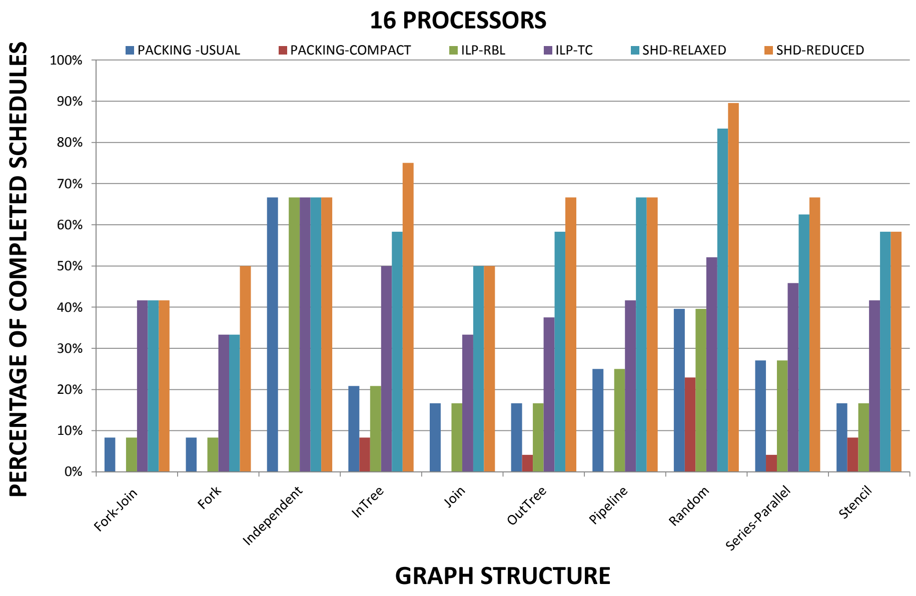


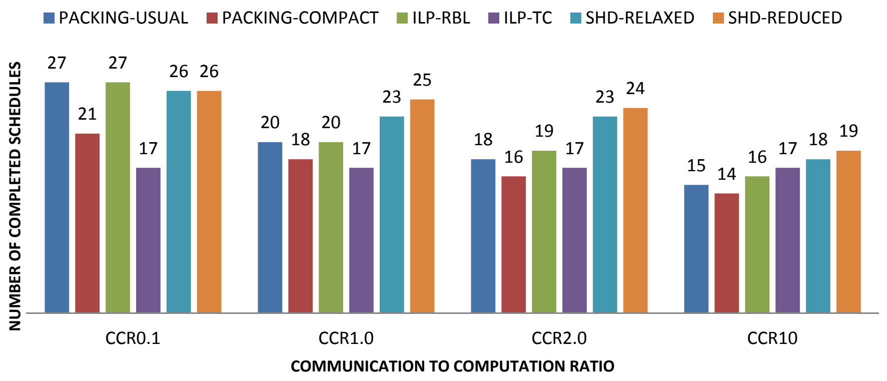
Figure 4.5: Completion percentage of formulations over graph structures on 16 processors

many orders of magnitude faster than the PACKING formulation and ILP-TC. In fact, for many instances the runtime of the proposed formulations are seen to decrease when more processors are available for scheduling.

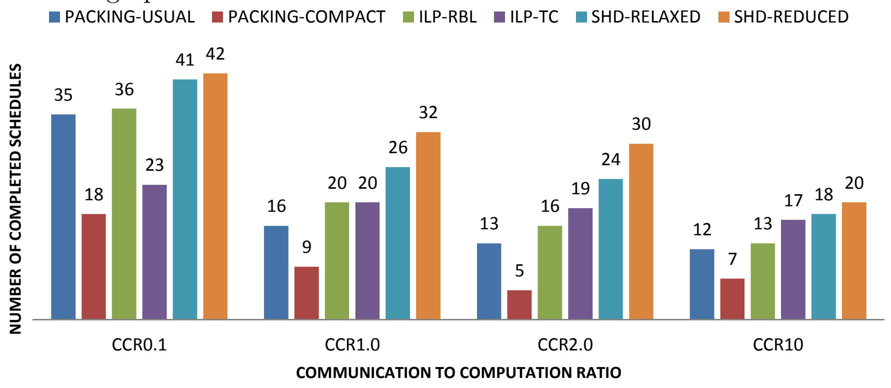
It is observed from Table 4.5 that SHD-REDUCED has the overall best performance in terms of the number of fastest results found over all the MILP formulations compared. Absolute stability cannot be expected, this being the nature of ILP's. From the experiments carried out, it is seen that SHD-RELAXED and SHD-REDUCED are often an order of magnitude faster than the other formulations and hence a significant improvement of previous work. The experiments indicate that ILP performance is structure dependent and their performance degrades with high CCR.

4.8 Conclusions

A MILP formulation for the task scheduling problem with communication delay was proposed. Each part of the formulation was motivated and discussed in detail, explaining its role and importance. A major feature of this formulation is the reduction of the number of variables and constraints by the effective linearisation



(a) Completed schedules of different formulations over CCR on 4 processors for 51 graph set for a 1 minute timeout



(b) Completed schedules of different formulations over CCR on 8 processors for 51 graph set for a 1 minute timeout

Figure 4.6: CCR Comparisons on 4 and 8 Processors Set for a 1 minute Timeout

of the bi-linear equation arising out of communication delays. Further, all variable indices in the MILP formulation are independent of the number of processors. As a result, the constraint complexity of the proposed MILP formulation was reduced to $O(|V|^2)$, which is significantly less than previous formulation as analysed in detail. An optimisation of the proposed SHD-RELAXED formulation was developed through the investigation of redundant terms in its formulation. The proposed formulations were experimentally compared with several existing MILP formulations. The experimental results indicate that the proposed formulation (SHD-RELAXED) and its reduction (SHD-REDUCED) provide a drastic improvement in runtime over other MILP formulations. The analysis of the behaviour of the MILP formulations with respect to graph structures indicate that some structures perform better than the others. It was seen that an increase in CCR deteriorates the performance of all the MILP formulations and to conclude that MILP formulations do not scale well for high CCR graphs. It was also seen that a larger number of processors is not necessarily bad for the performance of the MILP formulation.

Table 4.4: 12 hour Timeout Comparisons on ILP Formulations - Set 1

Graph	p	PACKING	ILP-RBL	ILP-TC
ogra20 _55, n=20 Ω=5.2 ω=55	2	*12h* *32.09%*	*12h* *26.15%*	*12h* *35.40%*
	4	8m:49s	18s	21m:9s
	8	35m:56s	6m:11s	11m:32s
	16	*12h* *3.92%*	1h:38m:5s	11m:46s
t30_56 _1, n=30 Ω=8.1 ω=56	2	9s	**2s**	46m:26s
	4	7m:32s	**17s**	5h:17m:51s
	8	7h:22m:19s	*12h* *0.21%*	2h:6m:54s
	16	*12h* *8.94%*	*12h* *4.07%*	1h:11s
t40_30 _1, n=40 Ω=5.8 ω=30	2	6m	**46s**	*12h* *22.06%*
	4	*12h* *7.18%*	29m:33s	*12h* *16.32%*
	8	*12h* *9.83%*	*12h* *5.07%*	*12h* *11.56%*
	16	*12h* *23.66%*	*12h* *9.43%*	*12h* *23.12%*
Ogra50 _ 53, n=50 Ω=12.9 ω=53	2	*12h* *46.33%*	***12h*** ***46.15%***	*12h* *inf*
	4	*12h* *19.78%*	***12h*** ***5.26%***	*12h* *inf*
	8	*12h* *inf*	*12h* *2.46%*	*12h* *inf*
	16	*12h* *inf*	*12h* *5.58%*	*12h* *inf*

Table 4.5: 12 hour Timeout Comparisons on ILP Formulations - Set 2

Graph	p	BASIC	RELAXED	REDUCED
ogra20 _55, n=20 Ω=5.2 ω=55	2	***12h 25.78%***	*12h 30.01%*	*12h 26.58%*
	4	**14s**	20s	27s
	8	22s	11s	**6s**
	16	12s	15s	**10s**
t30_56 _1, n=30 Ω=8.1 ω=56	2	16s	5s	22s
	4	32s	21s	23s
	8	7m:38s	**48s**	2m:6s
	16	33s	19s	**15s**
t40_30 _1, n=40 Ω=5.8 ω=30	2	3h:49m:40s	3m:29s	5m:2s
	4	2h:50m:14s	10m:40s	**4m:25s**
	8	9h:36m:14s	23m:24s	**3m:34s**
	16	2h:12m:11	7m:49s	**4m:3s**
Ogra50 _ 53, n=50 Ω=12.9 ω=53	2	*12h 48.50%*	*12h 48.27%*	*12h 48.22%*
	4	*12h 12.94%*	*12h 15.76%*	*12h 12.02%*
	8	2h:0m:1s	33m:34s	**17m:17s**
	16	1h:23m:4s	**58m:2s**	2h:16m:48s

5 Memory Limited Algorithms for Task Scheduling on Parallel Systems

The **contributions** of the work in this chapter are to

- Provide optimal solutions and near optimal solutions whose quality is guaranteed for small and medium sized instances of the task scheduling problem.
- Propose two memory limited optimal scheduling algorithms: Iterative Deepening A* (IDA*) and Depth-First Branch and Bound A* (BBA*) for the task scheduling problem to overcome the shortcomings of memory unbounded A* task scheduling algorithm and to see how they fare with respect to the proposed ILP formulations.
- Propose specific pruning techniques for the memory limited algorithms IDA* and BBA*.
- Use existing pruning techniques that may be used on IDA* and BBA* without any loss of generality.
- Conduct extensive experiments to evaluate and compare the proposed algorithms with previous optimal algorithms.

Functionality

- IDA* and BBA* are sped-up with the aid of a heuristic which is an underestimate on the optimal schedule length and with the aid of pruning techniques to cut down the number of nodes searched in the algorithmic state space.
- When finding a guaranteed near optimal schedule length is sufficient, the proposed algorithms can be combined, reporting the gap while they run.

Methodology

- The functionality is achieved by running IDA* and BBA* in combination. IDA* approaches the optimal solution from the top whereas BBA* approaches the optimal solution from the bottom. Together, they give a guaranteed bound on the quality of the solution obtained.

Outcome

- Proposed formulation displays a good improvement in performance, particularly considering that IDA* and BBA* are still able to continue its execution and find an optimal solution for cases where the A* algorithm runs out of memory. They are also competitive with the ILP formulations despite these formulations being experimented on a state of the art solver such as CPLEX.

Publication

Sarad Venugopalan and Oliver Sinnen. Memory Limited Algorithms for Optimal Task Scheduling on Parallel Systems, submitted to Journal of Parallel and Distributed Computing, Elsevier.

5.1 Abstract

To fully benefit from a multi-processor system, tasks need to be scheduled optimally. Given that the task scheduling problem with communication delays, $P|prec, c_{ij}|C_{max}$, is a well known strong NP-hard problem, exhaustive approaches are necessary. The previously proposed A* based algorithm retains its entire state space in memory and often runs out of memory before it finds an optimal solution. This chapter investigates and proposes two memory limited optimal scheduling algorithms: Iterative Deepening A* (IDA*) and Depth-First Branch and Bound A* (BBA*). When finding a guaranteed near optimal schedule length is sufficient, the proposed algorithms can be combined, reporting the gap while they run. Problem specific pruning techniques, which are crucial for good performance, are studied for the two proposed algorithms. Extensive experiments are conducted to evaluate and compare the proposed algorithms with previous optimal algorithms.

5.2 Introduction

The problem of scheduling tasks with precedence constraints and communication delays onto a set of homogeneous multi-processor system with the objective of minimising the overall finish time is essential and fundamental to speed up task execution on a multiprocessor system. The problem addressed is the classic problem of scheduling task graphs on parallel systems with communication delay, which is $P|prec, c_{ij}|C_{max}$ in the $\alpha|\beta|\gamma$ notation [18], [51]. Optimal scheduling is a well known hard problem (an NP-hard optimisation problem[44]), as the time needed to solve it optimally grows exponentially with the number of tasks. A number of heuristics have been proposed for this classical problem, but they try to produce good rather than optimal schedules, e.g. [31], [37], [21], [41], [48], [55], [58], [7], [23]. For the classical scheduling problem, no α-approximation is known [15]. The only known guaranteed approximation algorithm in [23] has an approximation factor depending on communication costs of the longest path in the schedule. While heuristics often provide good results, there is no guarantee that the solutions are close to optimal, especially for task graphs with high communication costs [47], [46].

Optimal schedules make a significant difference where the schedule is reused multiple times and in time critical systems with applications to flight control, industrial automation, telecommunication systems, consumer electronics, robotics and multimedia systems. The lack of good guaranteed approximation algorithms makes this very relevant. Moreover, having optimal solutions for scheduling instances allows to better judge the quality of heuristics and thereby to gain insights into their behaviour. The work presented in this chapter addresses the optimal solution of the $P|prec, c_{ij}|C_{max}$ scheduling problem.

Previous approaches to optimally solve this scheduling problem are based on Mixed Integer Linear Programming (MILP) formulations [11], [52] and smart state space enumerations using A* [28]. Both approaches showed strengths and weaknesses. The MILP formulations are less efficient for small numbers of processors and when communication costs are high [53]. They also require powerful solvers, which are like black boxes and make performance predictions difficult. The compact A* scheduling algorithm is very well suited to inject problem specific knowledge to prune the search space, which is crucial for efficient searches [49]. However, the approach suffers from the well known drawback of A*: it runs out of memory very quickly.

The main contributions of this chapter are 1) to overcome the memory limitations by proposing two new algorithms: Iterative Deepening A* (IDA*) scheduling algorithm which employs iterative deepening to limit the memory utilisation of the algorithm and the Depth-First Branch and Bound A* (BBA*) which improves the solution as the search traverses through the state space of the $P|prec, c_{ij}|C_{max}$ scheduling problem; 2) to propose an exhaustive search approach based on the two algorithms that finds a feasible schedule whose quality with respect to the schedule length is guaranteed, updating the current quality while running; 3) improve the initial lower bound on the schedule length to reduce the number of states revisited by IDA* during iterative deepening. 4) to investigate and propose new pruning techniques for generic and structure specific graphs to significantly reduce the state (solution) space.

The rest of the chapter is organised as follows. Section 5.3 gives the related work on other approaches used to optimally solve this scheduling problem. Section 5.4 discusses the task scheduling model and Section 5.5 details the proposed IDA* and BBA* scheduling algorithms and supplies different methods to improve the $f-$function calculations that guides the algorithms. The section also proposes the gap calculation method to find a feasible schedule with a guaranteed quality on the schedule length. Section 5.6 proposes a method to find a good initial lower bound close to the optimal schedule length in order to speed-up the execution of IDA*. Section 5.7 analyses existing and investigates novel state space pruning techniques that are essential to speed-up the runtime of the algorithm. Duplicate avoidance without memory and processor normalisation without memory are the pruning techniques proposed in this chapter. Section 5.8 evaluates and compares the performance of the proposed algorithms with previous approaches. Section 5.9 concludes by highlighting the main results of the chapter and the significance in using memory limited algorithms to solve the task scheduling problem.

5.3 Related work

Given the NP-hardness of the problem, few attempts have been made to solve it optimally. The solution space for the scheduling problem is spawned by all possible processor assignments combined with all possible task orderings. The search space grows exponentially making it impractical already for small task graphs. In this section we discuss the Mixed Integer Linear Programming (MILP) formulations and A* algorithm for the task scheduling problem. We observe the strengths and weaknesses of each approach and explain the motive for the proposed IDA* and BBA* scheduling algorithms.

One approach used to solve this problem optimally is by using Mixed Integer Linear Programming. Examples of MILP formulations are [1], [5], [32], [13], [16] and more recently [11], [52] and [53]. The MILP formulations are generally good but they are not efficient for high CCR graphs. They exhibit poorer performance when there are fewer processors available for scheduling [53]. Another method to solve the scheduling problem is to use the A* algorithm, an optimal search algorithm [28], [45], [49]. It begins the search with an empty solution space and is then incrementally grown. A* employs a best-first search technique [33], [14], [42] and is guided by a problem specific cost function. A* keeps all the nodes in memory and it usually runs out of memory before it runs out of time making it unusable for medium and large sized problem instances. This memory problem of the A* search approach has been recognised earlier and alternative approaches overcoming this problem have been proposed making trade-offs between memory requirement and performance. A good comparison between the A* and IDA* techniques (not the task scheduling problem) is given in [43]. Iterative deepening [57] is a memory limited algorithm whose search space frontier is first limited to an under-estimate on the optimal solution. In every successive iteration, this frontier is expanded until an optimal solution is found. The Depth-First Branch and Bound is another memory bound algorithm that searches the state space in depth first order. The initial feasible solution is an overestimate on the optimal solution [39], [57]. These two memory limited techniques are complementary in their operation, approaching the optimal solution from below and above, respectively. The optimal scheduling algorithms proposed in this chapter are based on these two techniques.

5.4 Task scheduling model

Formally, the tasks to be scheduled are represented by a directed acyclic graph (DAG) defined by a 4-tuple G=(V, E, C, L) where V denotes the set of tasks and E represents the set of edges. Each edge $(i, j) \in E$ defines a precedence relation between the tasks $i, j \in V$. The model assumes a fully connected network of homogeneous multiprocessors $P = \{1, \ldots, |P|\}$ with identical communication links. Each processor may execute several tasks, but each task has to be assigned to exactly one processor, in which it is entirely executed without pre-emption. Further, no multitasking or parallelism is permitted within a task. The execution time for each task on each processor and the data transfer times (or communication delays) between tasks with data dependence are given in advance as part of the task graph. A task cannot be executed unless all of its predecessors (parents) have completed their execution and all relevant data is available. The set $C = \{\gamma_{ij} : (i, j) \in E\}$

denotes the set of edge communication times. If tasks i and j are executed on different processors $h, k \in P, h \neq k$, they incur a communication time penalty γ_{ij}. If both tasks are scheduled to the same processor the communication between them is implemented by data sharing and the communication time is assumed to be zero. For a graph with $|V| = n$ tasks, the set $L = \{L_1 \dots, L_n\}$ represents the task computation times (execution time length). Let $\delta^-(j)$ be the set of predecessors of task i, that is $\delta^-(j) = \{i \in V | (i,j) \in E, j \in V\}$.

Scheduling this task graph G on processors P is the assignment of a **processor allocation** p_i and a **start time** $t_s(i)$ to each $i \in V$. The task's **finish time** is given by $t_f(i) = t_s(i) + L_i$, i.e. the task's start time plus its computation costs. Let $tf(p) = \max_{i \in \mathbf{V}: p_i = p}\{t_f(i)\}$ be the **processor finish time** of $p \in P$ and let $sl(\mathcal{S}) = \max_{i \in V}\{t_f(i)\}$ be the **schedule length** (or makespan) of schedule $\mathcal{S}$, assuming $\min_{i \in V}\{t_s(i)\} = 0$.

For such a schedule to be feasible, the following two conditions must be fulfilled for all tasks in G. The **Processor Constraint** in (5.1) enforces that only one task is executed by a processor at any point in time, which means for any two tasks i,$j \in V$

$$i = j \Rightarrow \begin{cases} \quad t_f(i) \leq t_s(j) \\ \text{or} \quad t_f(j) \leq t_s(i) \end{cases} \tag{5.1}$$

A task cannot be executed unless all of its predecessors (parents) have completed their execution and all relevant data is available. The **Precedence Constraint** given in (5.2) enforces that for every edge $(i,j) \in E$, $i,j \in V$, the destination task j can only start after the communication associated with (i,j) has arrived at p_j.

$$t_s(j) \geq t_f(i) + \begin{cases} 0 & \text{if } p_i = p_j \\ \gamma_{ij} & \text{otherwise} \end{cases} \tag{5.2}$$

For task $j \in V$, its start time on processor p is constrained by the **Data Ready Time (DRT)** and is represented as $t_{dr}(j,p)$. The DRT for task j is the time when all communications from task j's predecessors have arrived at p as shown in (5.3).

$$t_{dr}(j,p) = \max_{i \in \mathbf{parents}(j)} \left\{ t_f(i) + \begin{cases} 0 & \text{if } p_i = p \\ \gamma_{ij} & \text{otherwise} \end{cases} \right\} \tag{5.3}$$

If $j \in V$ is a source task, then $t_{dr}(j,p) = 0$. The task may however not be able to immediately start its execution since the processor assigned may be occupied with the execution of another task. The Earliest Start Time (EST) of task $i \in V$ on processor p is given by (5.4).

$$t_{EST}(i,p) = \max \{tf(p), t_{dr}(i,p)\} \tag{5.4}$$

The computation bottom level of a task $i \in V$ is the length of the longest path starting in i, denoted by cbl_i. Recursively it is defined in (5.5) as

$$cbl_i = L_i + \max_{j \in \mathbf{children}(i)} \{cbl_j\} \tag{5.5}$$

Given the start time of any task i, the schedule length sl is bounded by $t_s(i) + cbl_i$. In other words, after the task i has started execution, it still takes (at least) the time to sequentially execute all the tasks on the longest path starting in i.

Let the computation top level of a task $j \in V$ be defined as the length of the longest path (sum of computational task weights) starting in j, denoted by ctl_j to the longest length of its parent, excluding its own weight. Recursively it is defined as

$$ctl_j = \max_{i \in \mathbf{parent}(j)} \{ctl_i + L_i\} \tag{5.6}$$

The objective of this task scheduling problem is to allocate and schedule the tasks onto processors such that the overall completion time (makespan $C_{max} = sl$) is minimised [49], [45], [11], [40].

5.5 Memory limited optimal scheduling algorithms

In order to employ IDA* and BBA* to optimally solve the scheduling problem defined in the previous section we formulate it as a combinatorial problem. Essentially, the solution space to be searched is created by generating all possible processor allocations with all possible task orders. The latter are constrained by the precedence relations of the tasks expressed through the edges of the task graph. Starting with an empty schedule, a new state (i.e. partial schedule) is created by selecting an unscheduled free task (i.e. a task whose predecessors have already been scheduled) and allocating it to a processor. The start time of the task is the earliest possible as defined in (5.4). This way, we create $|free| \cdot |P|$ new states from a given state, with $free$ being the set of free tasks (ready to be scheduled) for the given state. This solution space has the form of a tree and its size grows exponentially. The complete schedules are the leaves of the tree at depth $|V|$, including the ones with optimal schedule length. This creation of the solution space can be considered an exhaustive list scheduling with all possible orders and processor allocations.

Given the fast exponential growth of this solution space, smart methods to search through it are necessary to go beyond graphs with a handful of tasks. A* was proposed to solve the scheduling problem, but clearly suffers from the high memory consumption of this best first search approach [28, 45]. The here proposed scheduling algorithms IDA* and BBA* overcome the memory problem as they both employ a depth-first search strategy. An essential instrument for both algorithms is to estimate the best possible schedule length achievable from a partial schedule, which will be discussed in Section 5.5.2.

Depth-first iterative deepening [26] approach tries to reduce the number of iterations in iterative deepening by setting an initial threshold that is tightly below the cost of the goal node and then successively increases this threshold. The proposed IDA* scheduling algorithm follows the depth-first iterative deepening methodology to traverse its search space. It limits the initial depth of the search tree to a number that is a lower bound on the schedule length, i.e. the depth is measured in terms of schedule length of the states and not in terms of scheduled tasks. If a feasible schedule is not found for the given search depth (i.e. schedule length) on the present iteration, the next iteration increments the bound (threshold) on its schedule length. The IDA* scheduling algorithm for any given iteration regenerates all the states for the previous iteration along with the new states generated for the present iteration. A tighter lower bound on the schedule length, which determines the initial threshold of the first iteration, significantly reduces the number of iterations required to find the optimal schedule length and results in overall fewer number of states generated.

BBA* is another memory bound algorithm that searches the state space in depth first order. The initial feasible solution is an overestimate on the optimal solution [39], [57]. As such it approaches the optimal solution from above, successively improving the best found solution.

The proposed IDA* and BBA* scheduling algorithms have a memory requirement of the order of number of tasks in the graph. This gives both of these algorithms an extremely small memory footprint when compared to the A* scheduling algorithm, however at the cost of exploring more states in general. Section 5.5.1 explains the proposed IDA* scheduling algorithm and Section 5.5.2 discusses the $f-$function calculations that are used to guide the IDA* scheduling algorithm towards the optimal solution. Section 5.5.3 explains the proposed BBA* scheduling algorithm and how it is differs from IDA*. Section 5.5.4 proposes a method to run both the IDA* and BBA* simultaneously on the same task graph to find a feasible schedule with a guaranteed quality. This is useful since finding an optimal solution may take a longer runtime for the scheduling algorithm and in this case, a feasible solution whose quality is guaranteed is faster to find.

5.5.1 IDA* Scheduling Algorithm

As stated above, when IDA* is used for combinatorial optimisation, its state space corresponds to a tree. Each node (usually called state) of the tree corresponds to a partial solution of the problem to be optimised, which becomes more complete the deeper the search progresses down the tree. The pseudo-code for the IDA* scheduling algorithm is given in Algorithm 5.1.

The main components of the IDA* scheduling algorithm are:

- **State** s: A partial schedule where some tasks have been allocated and scheduled on the processors.
- **Lower bound** SL_{LB}: The lower bound on the schedule length.
- **Threshold** T: The upper bound on the present iteration depth for the IDA* scheduling algorithm.
- **Initial state** $s_{\emptyset}$: The initial state representing an empty schedule, where no task has been scheduled yet.

- **Legal Expansion operator**: Given a state s, a new state is created by scheduling one more task, hence growing the partial solution represented by s. A task that can be scheduled must be *free*, which means it must be either independent or all its predecessors have already been scheduled in the partial schedule of s. We denote the set of all such tasks as $\mathbf{free}(s)$. The number of new states expanded from s is then the product of all free tasks times the number of processors, $|\mathbf{NEW}| = |\mathbf{free}(s)| \cdot |P|$. Each task of $\mathbf{free}(s)$ is scheduled on every processor $p \in P$ as early as possible after all other tasks on the same processor according to (5.4). They constitute the legal $(task, processor)$ combinations in lines 03 and 04 for Method IDA*_Recursion of Algorithm 5.1.

- **Cost function f**: The cost function $f(s)$ is an *underestimate* of the length of a complete schedule based on the partial schedule represented by s. The $f-$function calculation is discussed in detail in Section 5.5.2.

Algorithm 5.1 uses two main methods IDA*_Round() and IDA*_Recursion(). IDA*_ Round() is used to traverse the state space for all states whose $f(s)$ is less than or equal to the threshold T. This is done by recursively calling the method IDA*_Recursion(). This method recursively deepens the state space (lines 12 and 13) and exits with an optimal solution when the round threshold T is equal to $f(s)$ for the given state (lines 14 and 15). If the solution is not optimal ($f(s) > T$), the recursion backtracks (lines 16 and 17) and continues to traverse the search space (lines 03 to 10). Lines 18 to 21 ensure that the threshold T is incremented to max($T + 1$,(smallest $f(s) > T$, if it exists)). This is the new threshold for the next round when control is handed back to the method IDA*_Round().

5.5.2 f function calculation

For the given scheduling problem, $f^*(s)$ is the minimum schedule length of all possible schedules that can be constructed using the partial schedule represented by s. If the function $f(s)$ fulfils $f(s) \leq f^*(s)$ for every state s, it is called *admissible* [50]. With an admissible $f(s)$, IDA* and BBA* are guaranteed to find an optimal solution. The number of examined states depends on how close $f(s)$ is to $f^*(s)$. In general, the more accurate, the fewer states have to be examined and the faster is the algorithm.

With the requirement of admissibility, the quest of finding a good f function for the scheduling problem is to find tight lower bounds on the best schedule length achievable given a partial schedule. The generic f function used with IDA* on its state is determined from the idle time $f_{idle-time}(s)$, bottom level $f_{bl}(s)$ and data ready time $f_{DRT}(s)$ calculations in [45], [49]. The idle-time is a load balancing bound by taking into account the time lost in waiting for communication dependencies to be resolved and the bottom level is the critical path bound. Given the four components $f(s_{init}), f_{idle-time}(s), f_{bl}(s), f_{DRT}(s)$ in [49], the complete f function is

$$f(s) = \max\{f(s_{init}), f_{idle-time}(s), f_{bl}(s), f_{DRT}(s)\} \tag{5.7}$$

The f function defined in (5.7) is also *consistent*, which means for any state s it holds that $f(s) \leq f(t)$, where t is any descendant state of s. This is an important property for a fast A* algorithm [50] as well as for the memory limited A* algorithms.

5.5.3 Branch and Bound A* Scheduling Algorithm

BBA* is also a depth-first approach, but in contrast to IDA* not with multiple iterations. Instead it bounds its search with the currently best known solution, combined with the use of the admissible f-function of Section 5.5.2. The initial upper bound B on the schedule length is an overestimate of the optimal schedule length calculated using a list scheduling heuristic with tasks ordered by their computation bottom level [56], which is fast to compute but not necessarily close to the optimal schedule length. During the execution of BBA*, when a feasible schedule length lower than the upper bound B is found, the new solution is updated and the bound B is set to that schedule length. The algorithm then progressively searches for better solutions in the state space.

The main components of the BBA* are identical to the IDA* scheduling algorithm, including the f-function calculations of Section 5.5.2. The main difference is that the IDA* uses a moving threshold T that increments and eventually arrives at the optimal schedule length. This allows IDA* to immediately terminate once such a feasible solution is found. The BBA*, on the other hand decrements, the overestimate B, to arrive at the optimal schedule length. Hence, the algorithm is still required to traverse through the remaining unchecked nodes (i.e. partial schedules) in the state space (with an $f(s)$ value identical to B) to prove that the feasible solution is optimal. Once an optimal solution is found, BBA* searches for only those schedule lengths lower than the optimal schedule length. Then on, the number of states it searches is similar to the number of states A* would expand to find an optimal solution. The f-function for any given state helps to prune all the partial states whose cost is greater than the bound B.

The BBA* scheduling algorithm is given in Algorithm 5.2. The initial value of B is an overestimate on the schedule length and is provided by a list scheduler heuristic. The method BBA*_Start() transfers control to the recursive method BBA*_Recursion(). The threshold B is decremented when a complete schedule with a lower f-value $bestFState$ is found (as shown in lines 11 to 13 of method BBA*_Recursion). Once the search space is traversed, control is returned to the method BBA*_Start() and the optimal schedule is printed.

5.5.4 Gap Calculation

The complimentary nature of IDA* and BBA*, with a lower and upper bound, respectively, can be uniquely combined (run side by side in parallel) to find a solution with a guaranteed quality for the schedule length. It may arise that an optimal schedule length is not an absolute necessity given the longer runtime needed to find one. It may be sufficient to find a guaranteed feasible solution, saving on computation time. The proposed IDA* algorithm in Section 5.5.1 and BBA* algorithm in Section 5.5.3 are run in parallel to determine a guaranteed bound on the schedule length for the task scheduling problem. While the IDA* algorithm approaches the optimal schedule length (C^*_{max}) from the bottom ($T \leq C^*_{max}$), the BBA* approaches the optimal schedule from the top ($B \geq C^*_{max}$). When these two algorithms are run in parallel, the bounds close in and are improved from both ends. Given T and B at any instant of time, the worst case gap (in percentage) between the best currently found solution and the optimal solution is calculated as

$$gap = ((B - T)/B) \cdot 100 \tag{5.8}$$

The gap is updated and can be reported while the two algorithms are searching. Once the desired gap with (5.8) is attained, the feasible schedule satisfying the gap condition is output and the algorithm(s) terminate. The live-updating possibility also allows to manually terminate a search, knowing that the current solution has a certain quality. In contrast, an A* based search usually does not offer any solution until the optimal is found.

5.6 Lower Bound for IDA* Scheduling

A good lower bound close to the optimal schedule length is crucial to reduce the runtime of the IDA* scheduling algorithm. The tighter the lower bound the less iterations are necessary, hence fewer states need to be regenerated. In the input to Algorithm 5.1, $T \leftarrow SL_{LB}$ assigns the lower bound on the schedule length to the round threshold. This section discusses an improved method to determine a lower bound on the schedule length. The best (maximum of) all lower bounds calculated is assigned to SL_{LB}. First, Section 5.6.1 reviews analytical lower bounds on the scheduling lengths for generic graphs. Then, Section 5.6.2 proposes destructive lower bound calculations based on constraint formulations.

5.6.1 Generic Lower Bound

From the definition of the complete f function and its components it follows that the f-value of the initial state s_{init} (no task has been scheduled yet) is

$$f(s_{init}) = \max\{\frac{\sum_{i \in V} L_i}{|P|}, \max_{i \in V}\{cbl_i\}\} \tag{5.9}$$

which is the maximum of perfect load balancing (total computational weights divided by number of processors) and the maximum of the bottom level of each task (which is the length of the critical path of the graph).

5.6.2 Destructive Lower Bounds

Destructive lower bounds are found by starting with a best guess lower bound T and then look for contradictions on its feasibility. If some method gives us a lower bound that can be immediately found infeasible, then $T{+}1$ is the next valid lower bound. In general, finding a larger lower bound can be expedited by using binary search instead of incremental search. A number of strategies are discussed in [6] to improve the deduction of infeasibility on the lower bound. This involves strengthening the time window, deducing additional conjunctions, disjunctions or parallel relations and using transitive closure to deduce additional constraint relations. In this section, we use mixed integer linear programming to find destructive lower bounds. The MILP is here solely used to check infeasibility through constraint violations. If any one of the constraints is violated, the tested lower bound is infeasible and can be ruled out. In contrast to completely solving an MILP formulation [53] (which of course shares the NP-hardness of the scheduling problem), checking for constraint violations is very fast. In fact, in relation to the typical time to find the optimal solution it can be considered instantaneous. In the experimental evaluation in Section 5.8, the proposed algorithms IDA* and BBA* are compared against using the MILP to find the optimal solution.

5.6.2.1 Scheduling constraints on MILP DECISION-DESTRUCTIVE (Dec-Dev)

The IDA* scheduling algorithm in Section 5.5.1 takes as input an initial lower bound (SL_{LB}) returned by a decision problem and expands its search in accordance to this bound. In order to find a destructive lower bound, the scheduling problem is formulated as an MILP. Though any constraint programming methodology may be used to detect constraint violations, the formulation here extends previous work [53]. The full MILP is given below, where the constraints (A11-A13) and (A17) are added, which transform the minimisation problem into a decision problem: "is there a schedule with makespan less than or equal to some given bound". This decision problem is of course NP-complete, but infeasibility can often be shown very quickly, only using the pre-solver stage of the CPLEX optimisation solver to check for constraint violation. In the experiments, the CPLEX pre-solver detects these constraint violations (if any) in 100-300 milliseconds, which is negligible compared to typical solution times to find the optimal schedule. However, having tighter lower bounds, which we achieve with this method, significantly improves the performance of IDA*.

The process of improving the lower bound is repeatedly carried out using a binary search. The lower bounding procedure is however non-conclusive. If constraints are violated for a tested lower bound, the lower bound must be higher. However, if no constraints are immediately violated that does not mean that there is an optimal solution with a length equal to the lower bound. The decision problem may terminate with a lower bound that is less than the optimal schedule length. In general, very tight lower bounds close to the optimal solution may be difficult to find as it is not possible to determine beforehand which set of edges in the graph will be communicated remotely (hence induce costs). Nevertheless, we take advantage of the structure of the task graph to deduce tighter bounds and derive more constraints.

The MILP is formulated as a min-max problem that involves minimising the maximum task finish time. The variables t (here in the MILP we simply use t for the start time instead of t_s) and p give the schedule and allocation of tasks on processors, respectively. The σ and ϵ variables together ensure that tasks running on the same processor do not overlap in time. The constraints relate these variables to allow a fast ILP without the need for linearisation. For each task $i \in V$, let $t_i \in \mathcal{R}$ be the execution start time and $p_i \in \mathcal{N}$ be the ID of the processor on which task i is to be executed. In order to enforce non-overlapping constraints, define two sets of binary variables

$$\forall i, j \in V \quad \sigma_{ij} = \begin{cases} 1 & \text{task } i \text{ finishes before task } j \text{ starts} \\ 0 & \text{otherwise} \end{cases}$$
$$\forall i, j \in V \quad \epsilon_{ij} = \begin{cases} 1 & \text{PI of task } i \text{ is less than of task } j \\ 0 & \text{otherwise} \end{cases}$$

where PI is the Processor Index.

Based on the σ and ϵ binary variables the Decision-Destructive (DEC-Dev) formulation is proposed next, followed by a detailed explanation of the role of each constraint. The constraints (A01)-(A10) appear in the formulation SHD-BASIC in [53] and constraint (A11) appears in the formulation ILP-TransitivityClause [52]. They are combined with a strengthened time window (A12-A13) to give us a set of constraints that we then check for constraint violations on the best guess lower bound.

$$\begin{array}{lrr}
min & W & \text{(A01)} \\
\forall i \in V & t_i + L_i \leq W & \text{(A02)} \\
\forall i \neq j \in V & \sigma_{ij} + \sigma_{ji} \leq 1 & \text{(A03)} \\
\forall i \neq j \in V & \epsilon_{ij} + \epsilon_{ji} \leq 1 & \text{(A04)} \\
\forall i \neq j \in V & \sigma_{ij} + \sigma_{ji} + \epsilon_{ij} + \epsilon_{ji} \geq 1 & \text{(A05)} \\
\forall i \neq j \in V & p_j - p_i - 1 - (\epsilon_{ij} - 1)|P| \geq 0 & \text{(A06)} \\
\forall i \neq j \in V & p_j - p_i - \epsilon_{ij}|P| \leq 0 & \text{(A07)} \\
\forall i \neq j \in V & t_i + L_i + (\sigma_{ij} - 1)W_{max} \leq t_j & \text{(A08)} \\
\forall j \in V : i \in \delta^-(j) & t_i + L_i + \gamma_{ij}(\epsilon_{ij} + \epsilon_{ji}) \leq t_j & \text{(A09)} \\
\forall j \in V : i \in \delta^-(j) & \sigma_{ij} = 1 & \text{(A10)} \\
\forall i \neq j \neq k \in V & \epsilon_{ij} + \epsilon_{jk} \geq \epsilon_{ik} & \text{(A11)} \\
\forall i \in V & ctl_i \leq t_i & \text{(A12)} \\
\forall i \in V & W - cbl_i \geq t_i & \text{(A13)} \\
\forall i, j \in V & \sigma_{ij}, \epsilon_{ij} \in \{0, 1\} & \text{(A14)} \\
\forall i \in V & p_i \in \{1, .., |P|\} & \text{(A15)} \\
 & W \geq 0 & \text{(A16)} \\
 & W \leq dlb & \text{(A17)} \\
 & \sum_{i \in V} L_i + \sum_{i,j \in V} \gamma_{ij} = W_{max} & \text{(A18)}
\end{array}$$

Min-max objective (A01-A02)

These two lines specify the min-max objective. By (A01), W is the schedule length to be minimised. For each task, (A02) specifies that the sum of the task start time and its execution time (hence the task's finish time) is less than or equal W.

Overlap constraints (A03-A05)

These constraints ensure that no two tasks executing on the same processor overlap in time.

Processor constraints (A06-A07)

These constraints create the relation between the overlap variables ϵ and the processor indices p. (A06)-(A07) iterates over all values $i \neq j \in V$. (A06) and (A07) are used to enforce the definition condition of ϵ: if $\epsilon_{ij} = 1$ then $p_j > p_i$ and if $\epsilon_{ij} = \epsilon_{ji} = 0$ then $p_j = p_i$. A discussion on its correctness appears in [53].

Precedence constraints (A08-A10)

The inequalities (A08) to (A10) are enforcing the precedence constraints, including the consideration of remote communication costs created by the edges E of the task graph.

Transitivity constraints (A11)

The ϵ variables in the transitivity constraint enforces a partial ordering of the processor indices. The constraint $\epsilon_{ij} + \epsilon_{jk} \geq \epsilon_{ik}$ is defined for all $i \neq j \neq k$. If $\epsilon_{ij} = 1$ and $\epsilon_{jk} = 1$, ϵ_{ik} may still be 0. However, we also have the constraint $\epsilon_{ik} + \epsilon_{kj} \geq \epsilon_{ij}$. If $\epsilon_{jk} = 1$, then $\epsilon_{kj} = 0$ by (A04). This ensures that $\epsilon_{ik} = 1$. As a result, the transitivity is enforced in chains. Adding these constraints is not necessary in the MILP as they are implicitly contained in the other constraints. However, these additional constraints help to quicker detect a constraint violation as suggested in [6].

Strengthened time window (A12-A13)

By (A12), the start time of task i is greater than its computational top level (*ctl*) discussed in (5.6). By (A13), the start time of task i is less than the difference of the schedule length W (being minimised in (A01)) and its computational bottom level (*cbl*) discussed in (5.5). Again, these two constraints (A12) and (A13) are implicit in the other constraints, but they tighten the time window and hence help to find violations quicker [6].

Bounds (A14-A18)

The lines (A14)-(A18) define the bounds or domains of the variables used in the formulation. By (A14), both σ_{ij} and ϵ_{ij} are Boolean variables. (A15) gives the bound on processor allocation for each task $i \in V$. By (A16), all tasks have a start time greater than or equal to zero. By (A17), the destructive lower bound (*dlb*) on W is set, hence the schedule length is artificially constrained to test if a solution within this bound is feasible. By (A18), W_{max} gives an upper bound on W and is defined as the sum of all task execution times and edge communication times (which is simply an appropriate constant definition for (A08)).

5.6.2.2 Finding Destructive Lower bounds

To find a tight lower bound on the schedule length, we initially determine an analytical lower bound (LB) according to (5.9) and then improve on it. The Upper Bound (UB) on the schedule length is given by any feasible schedule obtained by a heuristic. Here a list scheduling algorithm is used to find a valid upper bound, where task priority is their decreasing computational bottom levels (i.e. the same algorithm as used for the bound of BBA* in Section 5.5.3). Given a valid lower and upper bound, a binary search is employed in the interval [LB,UB] and the pivot element in the interval is set as the destructive lower bound for that round. The *dlb* is then assigned to (A17) of DEC-Dev formulation in Section 5.6.2.1 and the MILP is checked for constraint violations. If there is a violation, then LB = *dlb*, else UB = *dlb*. The binary search continues and the above steps are repeated until no further constraint violation appear. The latest value of *dlb* is recorded as the new lower bound. The pseudo code for finding the destructive lower bound is given in Algorithm 5.3. It is important to understand that the UB only decreases for the purpose of this algorithm. If there is no violation that does not guarantee an optimal solution can be found for the schedule length equal to *dlb*. It just means we are not able to quickly demonstrate that it is infeasible.

5.7 State Space Pruning

In order to control the exponential explosion of states in the solution space, a number of pruning techniques are investigated in this section. Ideally, we want to employ previously proposed pruning techniques for state space search [28, 45]. However, these techniques often rely on a complete and reliable duplication detection as done in A* with the Open and Closed lists [49]. In A* newly created states are compared to all the states generated before and duplicates are dropped. IDA* and BBA* only keep a very limited number of states in memory at any point in time, hence pruning techniques need to be re-investigated regarding the appropriateness for these algorithms. We start by briefly revisiting two existing pruning techniques which are used with IDA* and BBA* and then continue by proposing how to (partially) avoid duplicates, given that we cannot detect them and end by proposing a new method for processor normalisation.

5.7.1 Fixed Task Order Pruning

Fixed Task Order (FTO) is a pruning technique developed in [49] due to the difficulty in scheduling simple graph (sub-)structures, such as independent, fork, join and fork-join. The main idea is that for certain sub-structures, the order of the tasks can be fixed. While their structures are very simple, they are often more difficult to be handled by optimal algorithms, compared to more complex structures, such as irregular series-parallel graphs [45]. The reason for the poor performance of optimal algorithms on such graphs is due to the higher degree of task ordering freedom. In all of the above mentioned structures, almost all tasks are independent of each other (except for the source and sink tasks), hence creating many valid task orderings. The FTO technique effectively reduces the number of states that need to be expanded by pruning them for certain graph (sub-)structures. When the task that are currently free for a given state s fulfil certain conditions (see [49]), they only need to be scheduled in one specific order (instead of considering all possible orders as normally) and it can still be guaranteed that an optimal schedule can be found. In terms of implementation, it suffices to have a flag associated with a state that signals a fixed order. Then only the first task of the free task queue will be considered in the expansion (on all available processors). When a new task becomes free, the fixed-order condition needs to be re-evaluated. The FTO pruning significantly reduces the size of the state space searched for independent graphs, fork, join and graphs in fork-join order and graphs that have these graphs as substructures. The FTO technique can be directly applied to IDA* and BBA* without any loss of generality.

When FTO is used, the duplication avoidance described in Section 5.7.3 cannot be employed. However, fixing the task order eliminates all duplicates, hence the duplicate avoidance is obsolete for the above mentioned sub-structures.

5.7.2 Equivalent Schedules

For a given partial schedule that is defined by its processor allocation and task start times, the order of some of the tasks may be irrelevant for the final schedule length. The two main considerations here are the finish times of the processors of the partial schedule and how the communications from the already scheduled tasks (if any) affect the start time of the yet unscheduled tasks. Together, these observations allow us to prune some of the equivalent orders (in this case permutations of tasks on a given processor). The main idea is to try to bring the tasks on each processor into ascending index order of the tasks by 'bubbling up' a new task scheduled onto a processor and to check if the finish time on that processor has not worsened and the start times of the yet unscheduled tasks is not negatively influenced. If that is the case, only one order needs to be considered and all other duplicate orderings may be pruned. A complete procedure for pruning equivalent schedules is given in [49] and is applied to IDA* and BBA* without any loss of generality.

5.7.3 Partial Duplicate Avoidance

The same partial schedule may reappear as the search space is traversed, resulting from the order in which tasks are assigned to processors. For example, on a 2 processor system with 3 tasks A, B, C: task A and task B may be scheduled in that order on processor p_1 and then task C on p_2. Alternately task A is scheduled on p_1 followed by task C on p_2 and task B on p_1. Both orderings result in the same partial schedule and it is sufficient to consider one of the two.

In the following an algorithm is proposed that avoids that some of the duplicates are created. The pseudo code for partial avoidance of duplicates without memory is given in Algorithm 5.4. If the processor index for the task scheduled in the present round is less than the processor index of the task scheduled in the previous round and if no new free tasks are added to free(s) for the present round, the node is pruned by returning a true $pruneFlag$.

The way in which the state space is created guarantees that all possible processor allocations and task permutations are generated. With this avoidance algorithm, we enforce that all free tasks of a state are scheduled to the processors in ascending order of the processor index. Other orders can be dropped as they will result in the same schedule, just created in a different order. All free tasks for a given state are independent of each other by definition, hence they can be scheduled in any order. All these orders are still created, but the processor allocation is done in order. The avoidance is restricted by the appearance of new free tasks (when a task is scheduled) in $free(s)$, because these new tasks need to have the chance to be scheduled on any processor.

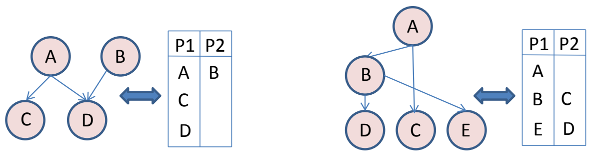


Figure 5.1: Graphs with corresponding schedules

Consider the following example graphs in Figure 5.1 scheduled on two processors p_1 and p_2. For the graph with 4 tasks, once tasks A and C are scheduled, task B needs to be scheduled before task D in order to satisfy the edge precedence condition. This implies that a schedule order $A \rightarrow C \rightarrow B \rightarrow D$ with A, C, D appearing on p_1 and B appearing on p_2 is possible, because the If condition on line 01 of Algorithm 5.4 is not executed for the corresponding schedule shown in Figure 5.1.This prevents the accidental pruning of states that otherwise may not be generated. Consider the second graph with 5 tasks scheduled on processors p_1 and p_2. Let the schedule order of the first four tasks be $A \rightarrow B \rightarrow C \rightarrow D$, with A, B on p_1 and C, D on p_2. Now if E was to be scheduled on p_1, the algorithm prunes this state together with the entire sub-tree rooted in it. However, the schedule order $A \rightarrow B \rightarrow E \rightarrow C \rightarrow D$ with A, B, E on p_1 and C, D on p_2 is a valid order permitted by the algorithm.

The algorithm employs two checks and a negligible amount of memory for the partial avoidance of duplicates on every state of the partial schedule. Though not all duplicates are detected, it is considered to be worthwhile taking into consideration the O(1) time complexity of the algorithm and the benefit when it avoids duplicates and prunes the entire resultant sub-tree.

5.7.4 Memory Limited Processor Normalisation

The previous section discussed a technique that helped avoiding duplicates that stem from the order in which tasks are scheduled. Since the processors are homogeneous and identical with a complete symmetric communication network, processors (or their names) are interchangeable. For example consider tasks A, B scheduled on processor p_1 and task C on processor p_2. Alternatively, task C can be scheduled on p_1 and tasks A, B on p_2. While these two schedules are different (they are no duplicates per se), they are identical for the addressed scheduling problem as all processors are the same. We can convert one schedule into the other by simply swapping the processor names. Only one of the two states needs to be considered and the other can be pruned. In order to do so, we normalise the processors using the proposed memory limited processor normalisation algorithm. Unlike the previous processor normalisation used for A* [45], that relies on duplication detection after normalisation, the here proposed technique uses a constant time and its memory requirement is of the order of the number of processors. It is not reliant on duplication detection, instead it avoids the creation of processor equivalent schedules.

The normalisation idea is that the processors are reordered (i.e. renamed) according to the index of the first task scheduled on each processor. Let α_k be the task id of the first task on processor p_k, whereby the first task is the one with the earliest start time on p_k, $\alpha_k = \{i \in V : proc(i) = p_k \wedge t_s(i) = \min_{j \in V, proc(j)=p_k}\{t_s(j)\}\}$. For example, there are three processors and their first tasks have task id 10, 5, 7, respectively. Then, the processor with 5 as its first task is called p_1, the processor with first task 7 is called p_2 and the processor with 10 as its first task is called p_3.

Now, instead of reordering or renaming processors (as done in [45]), we make sure that a new task, say j, can only be scheduled on an empty processor if it adheres to the ascending task id order. To check this, we compare j with $\alpha^{\max} = \max_{1 \leq k \leq |P|} \alpha_k$. If $j < \alpha^{\max}$, j cannot be scheduled on an empty processor, because that would violate our processor naming convention.

This is a valid pruning approach, as it does not exclude any possible (unique) processor allocations or task orderings. To see that, recall that the tasks are also a topological order. Hence a task with a higher id can only be scheduled after one with a lower id if they are independent (otherwise there would be a precedence violation). But if they are independent, then there will be a state with a different schedule order (our search space considers all possible orders) where the lower id task is scheduled before the higher id task. In that case the lower task id can be scheduled onto an empty processor. It follows that we can prune the state where this is done in the opposite order, as the schedules would be identical after normalisation.

5.7.4.1 Fixed-task order and processor normalisation

We now have to be careful that other pruning techniques do not conflict with this processor normalisation. The Fixed Task Order (FTO) pruning fixes the task order under certain conditions, hence violates our assumption that all possible orders are generated. For that reason, the processor normalisation is suspended while the task order is fixed (which is not a disadvantage, because while the tasks are in a fixed order, no duplicates are created anyway). In other words, each of the fixed tasks can be scheduled on an empty processor, irrespective of the task id.

When the fixing of the task order ends, and free tasks are scheduled without any fixed-order, these tasks are again restricted regarding their scheduling to an empty processor. We resume the initial behaviour and ignore what has happened during the time when the task order was fixed. With these measures, the processor restriction remains a valid pruning even when some of the tasks are scheduled in a fixed order.

First, the restriction is valid considering only the tasks that have been scheduled before a fixed-order period and after. These tasks are treated as before. Second, during the fixed-order period there is no restriction regarding empty processors. Third, tasks that are scheduled after the fixed-order period are not influenced by what has happened during the fix-order period, this is simply ignored and only what has happened before is considered.

The pseudo-code for the proposed processor normalisation algorithm is given in Algorithm 5.5. If TRUE, line 02 indicates that when N is not scheduled on the next available empty processor, then the state is pruned. Line 04 indicates that task N can always be scheduled on a non-empty processor. Line 06 indicates that N can be scheduled on the next available empty processor. When N is a Fixed Task Object (lines 07 to 09), the pruning is bypassed by using the previous value of $\alpha^{\max}$. If N is not a FTO (lines 10 to 14), the state is not pruned if $N > \alpha^{\max}$ and the values of $\alpha^{\max}$, p^{empty} are updated. The external calls FTO() and NonFTO() are not given for brevity and the name of the method suggests its functionality.

There are only a constant number of check conditions for every state of the schedule and the time complexity of this algorithm is $O(1)$. The memory complexity corresponding to the size of the variables is $O(1)$. The processor normalisation algorithm hence significantly reduces the time and memory footprint and speeds up execution.

It is also useful to note that the partial duplicate avoidance can simultaneously work with processor normalisation. Processor normalisation only has an impact on scheduling tasks on empty processors (forbidding it for some task ids). Partial duplication avoidance is only valid for a group of independent tasks (they are all free at the same time). These free tasks will be scheduled in all possible orders, including one order where first p_1 is filled, then p_2, then p_3 and so on, never going back (this forward order is an order that will not be pruned). At the same time this order can also have a partially increasing id order, so that the tasks that are first on an empty processor are in increasing id order. This does not contradict the previous order and it is not pruned by the partial duplication avoidance, hence both pruning work side by side. When a new free task comes along the situation might change, but this also stops the partial duplication avoidance.

5.8 Experimental evaluation

This section performs an experimental evaluation of the two proposed algorithms, using two approaches. First, we conduct a performance comparison of the proposed IDA* and BBA* task scheduling algorithms against the A* scheduling algorithm in [49] and the MILP formulation in [53] for a 1 minute time-out, varying the number of target processors between 2, 4, 8 and 16 processors. Using relatively small graphs with a short time-out allows us to gather many results, which can better reveal strengths and weaknesses of the algorithms and behavioural tendencies. We also include in these experiments an assessment of the gap calculations defined in Section 5.5.4 by running IDA* and BBA* algorithms in parallel for gaps (defined in eq. 5.8) of 5%, 10% and 15% .

Second, we conduct longer performance comparisons between IDA* and BBA* against the A* scheduling algorithm and the MILP formulation using a 12 hour time-out. Given the long time-out, this can only be done for few graphs and processor numbers, but demonstrates the behaviour for difficult graphs.

The computations are carried out using a single-threaded Java implementation on an AMD Opteron 6272 @ 2.1 GHz and a Java heap of 1 GB memory on Linux Ubuntu (Java version 1.7.0_51 on Ubuntu 12.04.4 LTS) . All the input graphs used for the time-out experiments are from [45], using the Graph eXchange Language (GXL) format [22] to represent them. The following definition of graph density is used

$$\Omega = |E|/|V| \tag{5.10}$$

with the maximum possible number of edges in the graph as $\gamma = |V|(|V| - 1)/2$.

In the experiments we expect A* to perform better than the proposed IDA* and BBA* algorithms, given its best-first search and the heavy use of memory. However, when A* runs out of memory, the here proposed algorithms will still be able to produce results.

5.8.1 Comparisons with 1 minute time-out

A database of 207 task graphs, summarised in Table 4.2, comprising of 10, 21 and 30 tasks of the following structures: fork, join, fork-join, in-tree, out-tree, series-parallel, pipeline, random, stencil and independent tasks is chosen [45]. The densities for the graphs in the database conform to the definition in (5.10). The experiments are carried out for 2, 4, 8 and 16 processors. A one minute time-out is set for each graph.

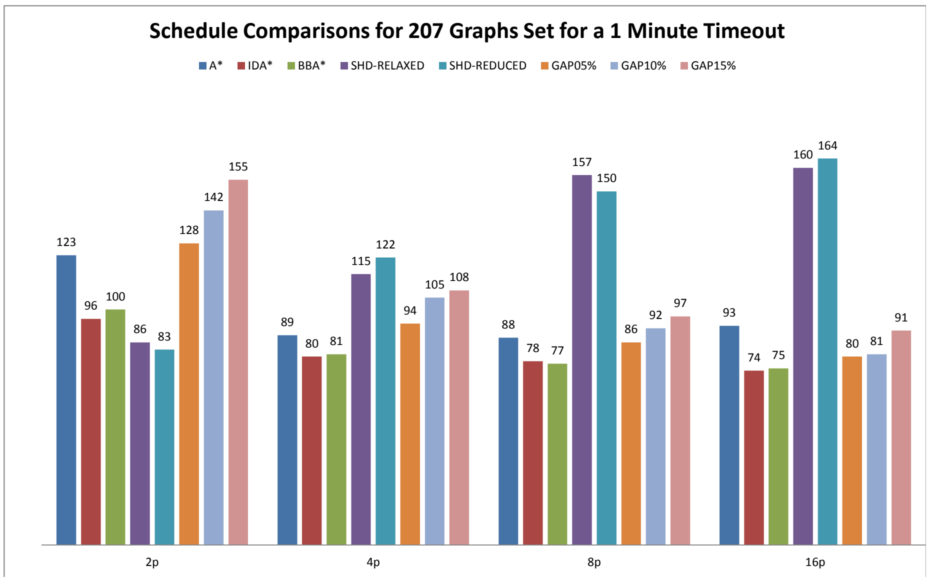


Figure 5.2: Completed schedules (out of 207) in 1 minute over algorithms for different number of processors

Table 5.1: Detailed Structure of the 207 Graph Database

Graph Structure	$n = 10$	$n = 21$	$n = 30$	Total
Fork-Join	4	4	4	12
Fork	4	4	4	12
Independent	1	1	1	3
InTree	8	8	8	24
Join	4	4	4	12
OutTree	8	8	8	24
Pipeline	4	4	4	12
Random	16	16	16	48
Series-Parallel	16	16	16	48
Stencil	4	4	4	12

A* returns either an optimal solution or no solution if it runs out of memory. It is seen from Figure 5.2 that A* as expected, outperforms both IDA* and BBA* by up to 22%. The gap experiments are run on 2 physical cores, each for 1 minute. With gaps of 5, 10 and 15 percent from the optimal schedule length, the number of completed schedules finding a feasible solution within one minute are steadily seen to rise. With a gap of 5%, IDA*+BBA* is already seen to be better than A* on 2 and 4 processors. A* is also seen to be better than IDA* and BBA* individually, but the improvements are less pronounced over more processors. This is remarkable given the less (memory) resources IDA* and BBA* use. On the other hand, the ILP's RELAXED and REDUCED [53] have a pronounced performance increase over more processors. In general, the ILP formulations are finding the most optimal schedules within the time limit. However, it needs to be considered in the assessment of these results that the ILP's are solved by a highly optimised CPLEX solver, whereas the state space search approaches (IDA*, BBA* and A*) are proof-of-concept implementations in Java. In that light it is remarkable, that they already outperform the ILPs for 2 processors and show significant potential in general.

5.8.2 Comparisons with 12 hour time-out

In this set of experiments, we used demanding graphs and ran the algorithms A*, IDA*, BBA*, the ILP's: RELAXED and REDUCED for up to 12 hours. For a number of graphs for which A* runs out of memory, IDA* and BBA* were also not able to find an optimal solution within 12 hours. Table 5.2 lists results for four selected graphs and helps to bring out the advantages in using memory limited algorithms.

The first column gives the name of the graph, the second column gives the number of tasks (n). The third column gives the graph density (Ω) and the fourth column gives the number of processors (p) the task graph is scheduled on. The rest of the columns give the names of the algorithms tested. It is seen for the Outtree graph that IDA* and BBA* are slower than A* and both the MILP formulations. For the Intree and Join graph, where A* runs out of memory, IDA* and BBA* benefit from fast runtime. For the Random graph, the MILP formulations are found to give the best results, whereas A* runs out of memory and both IDA*, BBA* times out at the end of 12 hours. The experiments presented here are for a 1 GB Java memory heap, but when the heap memory is reduced to 512 MB or 256 MB, A* was not surprisingly found to run out of memory sooner. In summary, IDA* and BBA* showed their potential to provide optimal or guaranteed solutions when other solution methods might fail.

5.9 Conclusions

This chapter proposed two new optimal scheduling algorithms named IDA* and BBA*. In contrast to previous algorithms, they use a limited memory and their implementations have a small memory footprint. Their approaches are complimentary, approaching the optimal solution from below and above. Using this nature they can be innovatively employed to compute non-optimal but quality guaranteed solutions. The destructive lower bound was introduced for the proposed algorithms along with other pruning techniques such as duplicate avoidance and processor normalisation. In an extensive experimental evaluation it was shown that IDA* and BBA* perform reasonably well in comparison with A*, given that they are memory limited. They can provide solutions when A* runs out of memory and are competitive when non-optimal, but quality guaranteed solutions are sufficient. IDA* and BBA* even perform better than strong ILP approaches on 2 processors, despite the employed highly optimised ILP solvers. As the experiments show, no single algorithm will always provide the fastest solution time. The proposed algorithms are a useful addition to the optimal scheduler spectrum.

In future work, it seems promising to investigate the parallelisation of IDA* and BBA* to speed-up their execution time. It may also be worth investigating an A*-IDA* hybrid algorithm wherein A* hands off execution to IDA* when it almost runs out of memory and then rely on IDA* to find an optimal solution.

Algorithm 5.1 Pseudo-code for IDA*

```
Input: Graph $G$, processors $|P|$, $T \leftarrow SL_{LB}$, $numTasks \leftarrow n$
$cTask \leftarrow \phi$, $cProc \leftarrow -1$//Current Task, Current Processor
$pTask \leftarrow$-1, $pProc \leftarrow$-1//Previous Task, Previous Processor
$\#freeTasks(s) \leftarrow |free(s)|$, $depth \leftarrow 0$, $s \leftarrow \phi$
Output: Optimal Schedule represented by state $s$
Method: IDA*_Round()
01: While( True) //Overall smallest $f(s)$ when all tasks are scheduled
02:    $T$ =IDA*_Recursion($cTask, cProc, pTask, pPproc, \#freeTasks(s)$,$depth$,$s$,$T$);
```

```
Method: IDA*_Recursion($cTask, cProc, pTask, pPproc, \#freeTasks(s)$,$depth$,$s$,$T$)
01: $done \leftarrow 0$
02: If($free(s) \neq \phi$)//if there exists free tasks for state $s$
03:    For each $i = 1$ to $\#freeTasks(s)$ do
04:       For each $j = 1$ to $|P|$ do
05:       $depth \leftarrow depth + 1$
06:        Sanitise_schedule();// Removes scheduled entries when backtracked
07:        $\#freeTasks(s) \leftarrow |free(s)|$
08:        Schedule a picked task $t$ from $free(s)$ onto proc $j$. Add it to state $s$.
09:        $pTask \leftarrow cTask, pProc \leftarrow cProc$
10:        $cTask \leftarrow t, cProc \leftarrow j$
11:        Update smallest $f(s) > T$ if exists
12:        If($f(s) \leq T$ AND $depth \leq numTasks$)
13:           $done$ =IDA*_Recursion($cTask, cProc, pTask, pPproc, \#freeTasks(s)$,$depth$,$s$,$T$);
14:        Else If($s$ is a complete solution AND $f(s) = T$)
15:              Print optimal solution $s$ and exit.
16:        If($done = 0$)//Recursion exits and backtrack search tree
17:              $depth \leftarrow depth - 1$
18: If(smallest $f(s) > T$ exists)
19:    return $T \leftarrow smallestf(s)$
20: Else
21:    return $T \leftarrow T + 1$
```

Algorithm 5.2 Pseudo-code for BBA*

```
Input: Graph $G$, processors $|P|$, $B \leftarrow$List Scheduling Heuristic, $numTasks \leftarrow n$
$cTask \leftarrow \phi$, $cProc \leftarrow -1$//Current Task, Current Processor
$pTask \leftarrow$-1, $pProc \leftarrow$-1//Previous Task, Previous Processor
$\#freeTasks(s) \leftarrow |free(s)|$, $depth \leftarrow 0$
$s \leftarrow \phi$, $bestFState \leftarrow \phi$
Output: Optimal Schedule represented by state $s$
BBA*_Start()
01: BBA*_Recursion($cTask, cProc, pTask, pPproc, \#freeTasks(s), depth, s, bestFState, B$);
02: Print optimal solution $bestFState$ and exit
```

```
Method: BBA*_Recursion($cTask, cProc, pTask, pPproc, \#freeTasks(s), depth, s, bestFState, B$)
01: $done \leftarrow 0$
02: **If**($free(s) \neq \phi$)//if there exists free tasks for state $s$
03:     **For each** $i = 1$ to $\#freeTasks(s)$ do
04:         **For each** $j = 1$ to $|P|$ do
05:         $depth \leftarrow depth + 1$
06:          Sanitise_schedule();// Removes scheduled entries when backtracked
07:          $\#freeTasks(s) \leftarrow |free(s)|$
08:          Schedule a picked task $t$ from $free(s)$ onto proc $j$. Add it to state $s$.
09:          $pTask \leftarrow cTask, pProc \leftarrow cProc$
10:          $cTask \leftarrow t, cProc \leftarrow j$
11:          **If**($f(s) \leq B$ AND $depth = numTasks$)
12:             $bestFState \leftarrow s$
13:             $B \leftarrow f(s)$
14:          **If**($f(s) \leq B$ **AND** $depth \leq numTasks$)
15:             $done$ =BBA*_Recursion($cTask, cProc, pTask, pPproc, \#freeTasks(s), depth, s, B$);
16:          **If**($done = 0$)//Recursion exits and backtrack search tree
17:                $depth \leftarrow depth - 1$
```

Algorithm 5.3 Pseudo code for finding destructive lower bound

```
Input: LB, UB
Output: dlb
Method: BinarySearch(LB, UB)
01: While (LB$\neq$ UB)
02:         $pivot \leftarrow \lceil(\text{LB}+\text{UB})/2\rceil$
03:         $dlb \leftarrow pivot$
04:         Assign dlb to (A17) of DEC-Dev formulation
05:         Check DEC-Dev for constraint violations
06:         If(any constraint is violated)
07:             LB$\leftarrow$ dlb
08:         Else
09:             UB$\leftarrow$ dlb
10: End While
11: Output dlb
```

Algorithm 5.4 Partial Duplicate Avoidance

```
Input:
$p_{this} \leftarrow p_{taskScheduledThisRound}$
$p_{prev} \leftarrow p_{taskScheduledPreviousRound}$
Partial state $s$
Output: $pruneFlag$
Method:FindDupNoMem($p_{this}, p_{prev}, s$)
01: If($isNewFreeNode(s)$ = FALSE)
02:     If($p_{this} < p_{prev}$)
03:         return TRUE
04: return FALSE
```

Algorithm 5.5 Pseudo-code for Processor Normalisation

```
Input: $N \leftarrow TaskToSchedule$,$p_N$ //processor for task $N$.
$p^{empty}$// index of first empty processor, i.e. number of non-empty processors+1
$\alpha^{\max} \leftarrow 0$
Output: $prune$
Method: Processor_Normal($N$,$p_N$,$\alpha^{\max}$, $p^{empty}$)
01: $prune \leftarrow$ FALSE
02: **If** ($p_N > p^{empty}$)
03:     return $prune \leftarrow$ TRUE
04: **If** ($p_N < p^{empty}$)
05:     return $prune \leftarrow$ FALSE
06: **If** ($p_N = p^{empty}$)
07:     **If**(FTO($N$,$p_N$,$\alpha^{\max}$))
08:         $p^{empty} \leftarrow p_{N+1}$//Update the next empty processor
09:         return $prune \leftarrow$ FALSE //remain with previous $\alpha^{max}$
10:     **Else If**(nonFTO($N$,$p_N$,$\alpha^{\max}$))
11:         **If**($N > \alpha^{\max}$)
12:             $\alpha^{\max} \leftarrow N$
13:             $p^{empty} \leftarrow p_{N+1}$
14:             return $prune \leftarrow$ FALSE
15.     **Else**
16:         return $prune \leftarrow$ TRUE
```

Table 5.2: 12 hour Time-out Comparisons on Algorithms Tested

Graph	n	Ω	p	A*	IDA*	BBA*	RELAXED	REDUCED
OutTree-Unbalanced-MaxBf-3_Nodes_10_CCR_10.01	10	0.90	2	1s	*3s*	5s	2s	2s
			4	1s	7s	8s	2s	2s
			8	1s	3m:32s	11s	2s	2s
			16	1s	13m:22s	16s	2s	2s
Intree_Unbal-anc ed_MAXBf-3_Nodes_21_CCR_ 0.10	21	0.95	2	$NoMem$	$> 12h$	$> 12h$	$> 12h$	$> 12h$
			4	$NoMem$	1s	1s	$> 12h$	$> 12h$
			8	$NoMem$	1s	1s	1s	1s
			16	1s	1s	1s	1s	1s
Join_Nodes_30_CCR_10.01	30	0.96	2	$NoMem$	1s	1s	$> 12h$	$> 12h$
			4	$NoMem$	1s	1s	$> 12h$	$> 12h$
			8	$NoMem$	2s	1s	$> 12h$	$> 12h$
			16	$NoMem$	3s	2s	$> 12h$	$> 12h$
Random_30 Den-sity_2.10_CCR_0.99	30	2.1	2	$NoMem$	1m:45s	$> 12h$	$> 12h$	$> 12h$
			4	$NoMem$	$> 12h$	$> 12h$	12s	7s
			8	$NoMem$	$> 12h$	$> 12h$	7s	3s
			16	$NoMem$	$> 12h$	$> 12h$	3s	1s

6 Conclusions and Future Work

This work investigated existing solutions to the task scheduling problem onto a system of multi-processors. The scheduling problem was formulated as 1) an Integer Linear Program and 2) as memory limited state space search algorithms (IDA* and BBA*). The ILP formulations in Chapter 1(Packing formulation) acted a framework for further improvements. The ILP formulations in Chapter 2, namely ILP-RBL and ILP-TC rework the linearisation of bi-linear forms arising out of communication delays. ILP-RBL was experimentally found to be faster than the Packing formulation when scheduled on a small number of processors and ILP-TC was found to be faster than the Packing formulation when scheduled a larger number of processors. In Chapter 3, the definition of the overlap variable (to ensures that tasks running on the same processor do not overlap in time) is modified and the ILP formulation reworked to check if this may result an speed improvement. The resulting formulation, ILP-DELTA was experimentally observed to have a speed performance similar to (but not better than) ILP-TC. Chapter 4 proposed three ILP formulations: SHD - BASIC. SHD - RELAXED and SHD - REDUCED. They have the property that size of the proposed formulations in terms of variables is made independent of the number of processors. Extensive experiments are carried out to test the speed efficiency of the proposed formulations. Results confirm a significant speed improvement in comparison to the previous ILP formulations discussed. Chapter 5 proposes the memory limited algorithms IDA* and BBA*. Their limited memory requirements allows them to continue execution and to find an optimal solution whereas the A* scheduling algorithm when it runs out of memory terminates without returning a solution. The IDA* and BBA* are combined to give a solution whose quality is guaranteed, when finding an optimal solution is not compulsory. Extensive experimental evaluation indicates that the memory limited algorithms are useful in finding an optimal solution when A* runs out of memory. They are also found to be competitive with the ILP formulations, inspite of them using a state of the art solver such as CPLEX.

In future work, it appears to be promising to investigate the parallelisation of IDA* and BBA* to speed-up their execution time. It may also be worth investigating an A*-IDA* hybrid algorithm wherein A* hands off execution to IDA* when it almost runs out of memory and then rely on IDA* to find an optimal solution.

References

[1] Chapter 3, Parallel machines - Linear programming and enumerative algorithms. *Annals of Operations Research*, 7:99–132, 1986.

[2] ILOG CPLEX 11.0 User's Manual. ILOG S.A., Gentilly, France, 2007.

[3] A. Gupta G. Karypis A. Grama and V. Kumar. *Introduction to Parallel Computing.* Pearson, Addison Wesley, second edition, 2003.

[4] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics).* Princeton University Press, Princeton, NJ, USA, 2007.

[5] Armin Bender. MILP Based Task Mapping for Heterogeneous Multiprocessor Systems. In *in Proceedings European Design Automation Conference*, pages 190–197. IEEE, 1996.

[6] Peter Brucker and Sigrid Knust. A linear programming and constraint propagation-based lower bound for the rcpsp. *European Journal of Operational Research*, 127:355–362, 1998.

[7] E. G. Coffman Jr. and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Informat.*, 1:200–213, 1972.

[8] Pablo E. Coll, Celso C. Ribeiro, and Cid C. de Souza. Multiprocessor scheduling under precedence constraints: Polyhedral results. *Discrete Applied Mathematics*, 154(5):770–801, 2006. IV ALIO/EURO Workshop on Applied Combinatorial Optimization.

[9] Pierluigi Crescenzi and Viggo Kann. Approximation on the Web: A Compendium of NP Optimization Problems. In *RANDOM*, pages 111–118, 1997.

[10] Abhijit Davare, Jike Chong, Qi Zhu, Douglas Michael Densmore, and Alberto L. Sangiovanni-Vincentelli. Classification, Customization, and Characterization: Using MILP for Task Allocation and Scheduling. Technical Report UCB/EECS-2006-166, EECS Department, University of California, Berkeley, Dec 2006.

[11] T. Davidović, L. Liberti, N. Maculan, and N. Mladenovic. Towards the Optimal solution of the Multiprocessor Scheduling Problem with Communication Delays. In *3rd Multidisciplinary International Conference on Scheduling: Theory and Application.*, pages 128–135, 2007.

[12] Tatjana Davidović and Teodor Gabriel Crainic. Benchmark-problem instances for static scheduling of task graphs with communication delays on homogeneous multiprocessor systems. *Comput. Oper. Res.*, 33:2155–2177, August 2006.

[13] Tatjana Davidović, Leo Liberti, Nelson Maculan, and Nenad Mladenović. Mathematical programming-based approach to scheduling of communicating tasks. Technical report, 2004.

[14] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *J. ACM*, 32(3):505–536, July 1985.

[15] Maciej Drozdowski. *Scheduling for Parallel Processing.* Springer Publishing Company, Incorporated, 1st edition, 2009.

[16] Christodoulos A. Floudas and Xiaoxia Lin. Mixed Integer Linear Programming in Process Scheduling: Modeling, Algorithms, and Applications. *Annals of Operations Research*, 139(1):131–162, 2005.

[17] Satoshi Fujita and Masafumi Yamashita. Approximation Algorithms for Multiprocessor Scheduling Problem. *IEICE Transactions on Information and Systems*, 83:503–509, 2000.

[18] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. 5:287–326, 1979.

[19] Yongpei Guan and RaymondK. Cheung. The berth allocation problem: models and solution methods. *OR Spectrum*, 26:75–92, 2004.

[20] Yongpei Guan and RaymondK. Cheung. The berth allocation problem: models and solution methods. In *Container Terminals and Automated Transport Systems*, pages 141–158. Springer Berlin Heidelberg, 2005.

[21] T. Hagras and J. Janecek. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. *Parallel Computing*, 31(7):653–670, 2005.

[22] R.C. Hull and A. Winter. A short introduction to the gxl software exchange format. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 299–301, 2000.

[23] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *SIAM J. Comput.*, 18(2):244–257, 1989.

[24] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257, 1989.

[25] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, pages 85–103, 1972.

[26] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, September 1985.

[27] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[28] Yu-Kwong Kwok and Ishfaq Ahmad. On multiprocessor task scheduling using efficient state space search approaches. *Journal of Parallel and Distributed Computing*, 65(12):1515–1532, 2005.

[29] L. Liberti. Compact linearization for bilinear mixed-integer problems. Technical report, 2005.

[30] Leo Liberti. Compact linearization for binary quadratic problems. *4OR*, 5:231–245, 2007.

[31] Welf Löwe and Wolf Zimmermann. Scheduling Iterative Programs onto LogP-Machine. 1685:332–339, 1999. Euro Par 99 Parallel Processing, Springer, Lecture Notes in Computer Science.

[32] Nelson Maculan, Stella C. S. Porto, Celso Carneiro Ribeiro, Cid Carvalho de Souza, Ribeiro Cid, and Carvalho Souza. A New Formulation for Scheduling Unrelated Processors under Precedence Constraints. *RAIRO Operations Research*, 33:87–91, 1997.

[33] Zbigniew Michalewicz and David B. Fogel. *Chapter 4: How to solve it - modern heuristics: second, revised and extended edition (2. ed.).* Springer, 2004.

[34] John E. Mitchell. Application to Linear Programming. www.rpi.edu/mitchj/handouts/lp/lp.pdf.

[35] K. Olukotun and L. Hammond. *The future of microprocessors.*, volume 3. Queue - Multiprocessors, September 2005.

[36] Kunle Olukotun and Lance Hammond. The Future of Microprocessors. *Queue*, 3(7):26–29, September 2005.

[37] A. Palmer and O. Sinnen. Scheduling Algorithm Based on Force Directed Clustering. pages 311–318, dec. 2008. Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008. Ninth International Conference on.

[38] David Pisinger. Where are the hard knapsack problems? *Computers and OR*, 32, 2005.

[39] David Poole and Alan K. Mackworth. *Chapter 3: Artificial Intelligence - Foundations of Computational Agents.* Cambridge University Press, 2010.

[40] Camille C. Price and Udo W. Pooch. Search techniques for a nonlinear multiprocessor scheduling problem. *Naval Research Logistics Quarterly*, 29(2):213–233, 1982.

[41] A. Radulescu and A.J.C. van Gemund. Low-cost task scheduling for distributed-memory machines. *Parallel and Distributed Systems, IEEE Transactions on*, vol 13(6):648–658, jun 2002.

[42] S.J. Russell and P. Norvig. *Artificial intelligence: a modern approach.* Prentice hall, 2010.

[43] U.K. Sarkar, P.P. Chakrabarti, S. Ghose, and S.C. De Sarkar. Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. *Artificial Intelligence*, 50(2):207 – 221, 1991.

[44] V. Sarkar. *Partitioning and scheduling parallel programs for multiprocessors.* MIT press, 1989.

[45] Ahmed Zaki Semar Shahul and Oliver Sinnen. Scheduling task graphs optimally with A*. *Journal of Supercomputing*, 51(3):310–332, March 2010.

[46] O. Sinnen and L. Sousa. Scheduling Task Graphs on Arbitrary Processor Architectures Considering Contention. In *High Performance Computing and Networking (HPCN'01)*, volume 2110 of *Lecture Notes in Computer Science*, pages 373–382. Springer-Verlag, 2001.

[47] O. Sinnen and L. Sousa. Experimental Evaluation of Task Scheduling Accuracy: Implications for the Scheduling Model. *IEICE Transactions on Information and Systems*, E86-D(9):1620–1627, September 2003.

[48] Oliver Sinnen. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing).* Wiley-Interscience, 2007.

[49] Oliver Sinnen. Reducing the solution space of optimal task scheduling. *Comput. Oper. Res.*, 43:201–214, March 2014.

[50] Peter Norvig Stuart J. Russell. *Artificial Intelligence - A Modern Approach.* Prentice Hall. ISBN 0-13-103805-2.

[51] B. Veltman, B. J. Lageweg, and J. K. Lenstra. Multiprocessor Scheduling with Communication Delays. 16(2-3):173–182, 1990.

[52] Sarad Venugopalan and Oliver Sinnen. Optimal Linear Programming Solutions for Multiprocessor Scheduling with Communication Delays. In Yang Xiang, Ivan Stojmenovic, BernadyO. Apduhan, Guojun Wang, Koji Nakano, and Albert Zomaya, editors, *Algorithms and Architectures for Parallel Processing*, volume 7439 of *Lecture Notes in Computer Science*, pages 129–138. Springer Berlin Heidelberg, 2012.

[53] Sarad Venugopalan and Oliver Sinnen. ILP formulations for optimal task scheduling with communication delays on parallel systems, IEEE transactions on parallel and distributed systems. volume 26, pages 142–151, 2015.

[54] Sarad Venugopalan and Oliver Sinnen. Green banana task scheduler for multiprocessor systems., URL (Case Sensitive): http:// homepages.engineering.auckland.ac.nz/ ~parallel/ OptimalTaskScheduling/.

[55] Tao Yang and Apostolos Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993.

[56] Tao Yang and Apostolos Gerasoulis. List scheduling with and without communication delays. *Parallel Comput.*, 19(12):1321–1344, December 1993.

[57] Weixiong Zhang. *Chapter 2: State-space search - algorithms, complexity, extensions, and applications.* Springer, 1999.

[58] A.Y. Zomaya, C. Ward, and B. Macey. Genetic scheduling for parallel processor systems: comparative studies and performance issues. *Parallel and Distributed Systems, IEEE Transactions on*, 10(8):795–812, aug 1999.