A PARALLEL COMPUTING APPROACH FOR SCHEDULING MARKOVIAN PROJECT NETWORKS TO MAXIMIZE THE EXPECTED NET PRESENT VALUE

Xiaoming Wang, Guojian Zhou, Roel Leus, Stefan Creemers

REPRINT | 3359



UCLouvain

CORE | LIDAM Louvain Institute of Data Analysis and Modeling in economics and statistics

CORE

Voie du Roman Pays 34, L1.03.01

B-1348 Louvain-la-Neuve

Tel (32 10) 47 43 04

Email:lidam-library@uclouvain.be

https://uclouvain.be/en/research-institutes/lidam/core/core-reprints

A parallel computing approach for scheduling Markovian project networks to maximize the expected net present value

Xiaoming Wang[a,b]
Guojian Zhou[a,b]
Roel Leus[c]
Stefan Creemers[d,*]

[a] State Key Laboratory of Precision Electronic Manufacturing Technology and Equipment, Guangdong University of Technology, Guangzhou 510006, P.R. China

[b] Provincial Key Laboratory of Computer Integrated Manufacturing, Guangdong University of Technology, Guangzhou 510006, P.R. China

[c] ORSTAT, Faculty of Economics and Business, KU Leuven, 3000 Leuven, Belgium

[d] Université Catholique de Louvain, Center for Operations Research and Econometrics (CORE), Voie du Roman Pays 34, B-1348 Louvain-la-Neuve, Belgium, stefan.creemers@uclouvain.be

This paper proposes a multithreaded parallel stochastic dynamic programming (SDP) approach for solving resource-unconstrained project scheduling problems with exponentially distributed activity durations, aiming to maximize the expected net present value (eNPV). These problems are typically modeled as Markov decision processes, in which optimal policies can be derived through SDP. However, traditional SDP methods are computationally intractable for large-scale instances due to the exponential growth of state and action spaces. To overcome this limitation, we develop a producer-consumer parallelization scheme that separates state enumeration from eNPV evaluation. The producer thread enumerates feasible states, while multiple consumer threads concurrently compute their optimal NPVs. Several optimized implementation strategies are proposed to improve cache efficiency, reduce synchronization overhead, and balance workload among threads. We further provide a theoretical analysis of the asymptotic properties of the parallel SDP, establishing the conditions under which the load-balancing efficiency and computational proportion of state enumeration converge. Comprehensive computational experiments confirm the validity of the theoretical analysis and demonstrate the scalability and robustness of the proposed approach. The parallel SDP achieves a maximum speedup factor of over 16 on a 24-cores processor for benchmark instances, and enables the exact solution of more complex instances that were previously intractable using serial SDP. The proposed framework is general and can easily be extended to other Markovian optimization problems that are characterized by high-dimensional state spaces.

Keywords: Project scheduling; Markovian networks; dynamic programming; parallel computing.

*Corresponding author

1. Introduction

Projects play a fundamental role in engineering management, with applications in construction, new product development, and various other domains. A project typically comprises a set of activities that are subject to precedence constraints. Project scheduling involves determining the start time of each activity such that all precedence (and potentially resource) constraints are satisfied, while optimizing a specific performance criterion, such as minimizing project makespan or maximizing the NPV. Comprehensive overviews of deterministic project scheduling research can be found in Hartmann and Briskorn [2010, 2022], whereas Herroelen and Leus [2005] provide an extensive review of stochastic project scheduling.

Given the profit-oriented nature of enterprise operations, maximizing a project's NPV is often regarded as a more economically meaningful objective than minimizing its makespan. In practice, companies incur cash outflows when initiating activities and receive cash inflows upon project completion. To obtain the project NPV, these future cash flows are converted into present values using a specified discount rate, typically representing the cost of capital. A project with a positive NPV is deemed profitable; otherwise, it should be rejected. Consequently, project scheduling plays a pivotal role in assessing project profitability and determining the optimal timing of activities to maximize NPV. Notably, even in the absence of resource constraints, initiating all activities as early as possible does not necessarily lead to optimal outcomes [Buss and Rosenblatt, 1997]. For activities associated with cash outflows, delayed starts may be beneficial, whereas early initiation of activities generating substantial cash inflows is often advantageous. Thus, determining optimal activity start times requires carefully balancing these conflicting incentives.

The problem of NPV-based project scheduling under deterministic settings was first introduced by Russell [1970]. Since then, extensive research efforts have been devoted to this topic [Smith-Daniels and Aquilano, 1987, Elmaghraby and Herroelen, 1990, Gupta et al., 1992, Yang et al., 1993, 1995, Sung and Lim, 1994, Baroum and Patterson, 1996, Etgar et al., 1997, Etgar, 1999, Vanhoucke et al., 2001b,a, Selle and Zimmermann, 2003, Vanhoucke, 2010, Leyman and Vanhoucke, 2016, 2017]. However, it is well recognized that the intrinsic complexity of project scheduling primarily arises from the uncertainties encountered during project execution, such as random fluctuations in activity durations, resource availability, and cash flows [Herroelen and Leus, 2005, Hazır and Ulusoy, 2020]. These uncertainties pose major challenges for optimizing performance measures like expected makespan and eNPV. Compared to deterministic models, research addressing stochastic project scheduling for eNPV maximization remains relatively limited [Wiesemann and Kuhn, 2015], with most notable contributions emerging only in the past decade [Sobel et al., 2009, Wiesemann et al., 2010, Creemers et al., 2010, Hermans and Leus, 2018, Creemers, 2018b, Rostami et al., 2024].

A substantial body of work on stochastic project scheduling focuses on Markovian project networks, in which activity durations are modeled as independent exponential random variables. This class of problems was first studied by Kulkarni and Adlakha [1986], and subsequent works have adopted similar formulations [Buss and Rosenblatt, 1997, Sobel et al., 2009, Creemers et al.,

2010, Gutin et al., 2015, Hermans and Leus, 2018, Creemers, 2018b, 2019, Bertazzi et al., 2024]. In addition to exponential durations, several authors have also considered more general forms of uncertainties, such as discretely distributed durations and cash flows [Wiesemann et al., 2010, Rezaei et al., 2021, Rostami et al., 2024]. In contrast to deterministic scheduling, where a fixed start time is determined for each activity, the solution in stochastic environments is a scheduling policy, i.e., a set of state-dependent decision rules that guide project execution. SDP remains the most widely used method for deriving optimal policies that maximize the eNPV. However, it suffers from the well-known curse of dimensionality, i.e., the exponential growth of the state and action spaces [Sobel et al., 2009, Creemers et al., 2010, Creemers, 2018b]. As a result, the computational cost becomes prohibitively high even for moderately sized instances, primarily due to the extensive recursive state evaluations required by SDP.

This paper tackles the problem of scheduling Markovian projects to maximize eNPV. To address the computational bottleneck of conventional SDP, we propose a parallel SDP algorithm based on multithreading to efficiently solve complex instances. The remainder of the paper is structured as follows: Section 2 reviews related literature. Section 3 formally defines the problem. Section 4 reviews the Markov decision process (MDP) formulation and the classical SDP algorithm. Section 5 details the proposed parallel SDP. Section 6 provides a theoretical analysis of its asymptotic properties. Computational results are reported in Section 7. Finally, Section 8 concludes the paper and outlines future research directions.

2. Literature review

Over the past decades, project scheduling for NPV maximization has been studied extensively. Early contributions typically assumed deterministic activity durations and cash flows, an assumption that was later relaxed. Most research on stochastic NPV-oriented project scheduling adopts exponentially distributed activity durations. Buss and Rosenblatt [1997] were among the first to investigate this setting. Building on the continuous-time Markov chain framework of Kulkarni and Adlakha [1986], they derived analytical expressions for the eNPV with and without activity delays. Their analysis revealed that, under certain conditions, delaying the start of an activity beyond its earliest feasible time may increase eNPV by postponing negative cash flows. However, the identification of optimal delays was shown to be computationally demanding due to matrix-exponential evaluations, motivating the development of heuristic procedures. Subsequently, Sobel et al. [2009] and Creemers et al. [2010] formulated the problem as an MDP and solved it via SDP. The key distinction between their approaches lies in the state-space partitioning strategy. Sobel et al. [2009] grouped states according to the number of completed and ongoing activities, whereas Creemers et al. [2010] introduced a finer decomposition based on uniformly directed cuts (UDCs), each representing a maximal set of activities that can be executed in parallel. Computational evidence shows that the UDC-based partition substantially improves tractability and enables the solution of larger instances. Wang et al. [2018] modeled the resource-constrained project scheduling problem

with exponential durations and random rework as an MDP and applied SDP without exploiting structural state partitioning. All states and actions are explicitly enumerated and stored prior to backward recursion, which restricts the approach to very small instances.

Subsequent studies further advanced the SDP methodology. Creemers [2015] enhanced the computational efficiency of the SDP proposed by Creemers et al. [2010] by replacing simultaneous multi-activity transitions with sequential transitions, thereby reducing the number of decisions per state. Hermans and Leus [2018] demonstrated that preemption is unnecessary when maximizing eNPV under exponential durations and proposed an alternative MDP model in which the system state is defined solely by the set of completed activities. Creemers [2018b] generalized this finding, proving that non-preemptive policies are globally optimal when cash flows are incurred at activity initiation, regardless of the duration distribution. Both works showed that the revised state definition drastically reduces memory requirements and computational time by several orders of magnitude relative to traditional SDP approaches. Extending this line of work, Creemers [2019] applied the revised model to minimize expected project makespan in stochastic preemptive scheduling and observed that the benefits of preemption increase with problem complexity. In order to overcome the limitation of the assumption that activity durations are exponentially distributed, one can also allow activity durations to follow a more generalized phase-type (PH) distribution, that is, a convolution or mixture of exponential distributions that can be used to approximate any positive-valued distribution with arbitrary precision. When using PH distributions to approximate activity durations, the original project network can be transformed into an equivalent Markovian network in which each activity is represented by one or more exponential phases, preserving the Markov property required for SDP. A summary of major contributions in Markovian project network scheduling is presented in Table 1.

Table 1
Summary of literature on scheduling Markovian project networks.

Reference	Objective	Resource Constraints	Activity Duration Distribution
Sobel et al. [2009]	maximize eNPV	×	Exponential
Creemers et al. [2010]	maximize eNPV	×	Exponential
Creemers [2015]	minimize expected makespan	✓	Phase-type
Gutin et al. [2015]	minimize expected makespan	×	Exponential
Wang et al. [2018]	minimize expected makespan	✓	Exponential
Creemers [2018b]	maximize eNPV	×	Phase-type
Hermans and Leus [2018]	maximize eNPV	×	Exponential
Creemers [2019]	minimize expected makespan	✓	Phase-type

Beyond the Markovian framework, several studies have considered more general forms of uncertainty. Wiesemann et al. [2010] proposed a scenario-based framework in which activity durations and cash flows follow discrete probability distributions, and developed a branch-and-bound algorithm to determine optimal activity target durations. Zheng et al. [2018] and Liang et al. [2019] addressed the resource-constrained max-NPV project scheduling problem with stochastic activity durations, and developed metaheuristic algorithms to generate a robust baseline schedule that maximizes the project NPV. De Reyck and Leus [2008] and Zhao et al. [2020] both address the problem of scheduling projects to maximize eNPV when activities have a probability of failure, and where activity failure leads to immediate project termination. The earlier work by De Reyck and

Leus [2008] develops a branch-and-bound algorithm to find optimal schedules with fixed success probabilities per activity, showing that the problem is NP-hard. In contrast, Zhao et al. [2020] explicitly model dynamic project risk reduction as activities are completed, offering a more granular representation of risk evolution during project execution. Rezaei et al. [2020, 2021, 2024] extended this line of research by incorporating a conditional value-at-risk (CVaR) criterion, and developed several heuristic algorithms to trade-off between eNPV and the risk of eNPV. For general duration distributions, Creemers [2018a] obtain closed-form formulas for the moments of the eNPV distribution of a serial project. These moments are used to approximate the eNPV distribution using a three-parameter lognormal distribution. For multi-mode resource-constrained project scheduling with stochastic durations and cash flows, Chen and Zhang [2012] employed Monte Carlo simulation to evaluate solutions generated by an ant colony optimization algorithm. Peymankar et al. [2021] formulated two mixed-integer programming models and corresponding Benders decomposition methods for eNPV maximization under uncertain cash flows. More recently, Rostami et al. [2024] proposed both static and heuristic dynamic scheduling strategies to address uncertainties in activity durations and cash flows, demonstrating robust performance across multiple stochastic scenarios. Similarly, for the problem of maximizing eNPV with uncertain activity durations and cash flows, Lamas et al. [2024] proposed a target-time-windows technique to simultaneously determine a baseline time window for each activity and a non-anticipative policy that respects precedence constraints. Bruni and Hazır [2024] developed a risk-averse distributionally robust project scheduling model to manage cash flow under uncertain payment delays, incorporating financing costs and utilizing CVaR for optimization, along with an efficient heuristic algorithm that demonstrates strong performance in both solution quality and computational efficiency.

Despite substantial progress in SDP-based Markovian project scheduling, most implementations remain serial and computationally intensive. To the best of our knowledge, Wang et al. [2018] is the only study that explicitly incorporates parallel computing within SDP for project scheduling. Their approach dynamically groups states during enumeration and assigns each group a separate thread. Although their computational results demonstrate a reduction in computation time, substantial redundancy arises because overlapping successor states are repeatedly generated and stored across threads. From a methodological perspective, SDP for Markovian project networks proceeds through stage-wise Bellman recursions over a structured state space, where the value of each state is computed by evaluating a finite set of actions and successor states. Importantly, the state-space partitioning schemes commonly adopted in existing literature naturally expose opportunities for parallelization, as they ensure that no state transitions occur between certain state subspaces within the same stage. For instance, the UDC-based partitioning schemes adopted by Creemers et al. [2010] and Creemers [2015] group UDCs according to their rank (i.e., the number of predecessor activities of the activities it contains), allowing UDCs within the same rank to be processed independently. Similarly, the completion-based formulations proposed by Creemers [2018b] and Creemers [2019] partition states by the number of completed activities, so that states within the same stage are mutually independent. Despite these favorable structural properties, parallel SDP

has not yet been systematically exploited in stochastic project scheduling.

Parallel SDP has, however, been explored in other domains. Early work by Hanson [1991] formulated SDP as the numerical solution of Hamilton-Jacobi-Bellman equations on a discretized state space and showed that Bellman updates across state points at a given time step are independent and can therefore be evaluated concurrently. The author implemented this scheme on a vector multiprocessor, exploiting state-wise parallelism to accelerate value-function updates for continuous-time stochastic control problems. Chung et al. [1992] extend this idea to SDP problems discretized via finite-element methods, demonstrating that Bellman equations associated with distinct finite elements can be processed in parallel. Eschenbach et al. [1995] further generalized earlier approaches by shifting the focus from problem-specific discretizations or hardware-oriented implementations to an explicit algorithmic decomposition of Bellman recursions. In this framework, Bellman updates are organized as collections of independent computational tasks whose scheduling and assignment to processors can be flexibly designed. By comparing alternative task-partitioning strategies for discrete-time SDP with continuous state and control spaces, the authors showed that finer-grained decomposition of state-update tasks can substantially improve load balance and parallel efficiency when computational effort varies across states. Subsequent studies applied parallel SDP to large-scale practical problems. For example, Dias et al. [2013] developed a parallel SDP framework for long-term hydrothermal scheduling, in which the most computationally intensive Bellman updates, formulated as independent linear programs for each discretized state and scenario, are evaluated concurrently. Their comparison of static and dynamic task assignment strategies demonstrated that adaptive scheduling improves load balance and parallel efficiency when state-wise solution times are heterogeneous. Aldasoro et al. [2015] proposed a parallel implementation of a serial SDP algorithm for multistage stochastic mixed 0-1 optimization problems. The authors distinguished two conceptually different parallelization strategies. The first parallelizes the solution of independent mixed-integer subproblems associated with different scenario clusters within the same stage, thereby accelerating value-function approximation through concurrent subproblem evaluation. The second executes multiple independent SDP solution paths in parallel, which periodically exchange incumbent solutions to enhance search diversification and solution quality. While the former primarily targets computational speedup, the latter emphasizes improved solution quality through parallel exploration of the decision space. Overall, the existing literature on parallel SDP remains relatively scarce, particularly when compared to research on other parallel optimization methods such as parallel branch and bound. Moreover, most available studies on parallel SDP report only sublinear speedups. For a more detailed discussion, the reader is referred to the comprehensive review by Schryen [2020].

A related but methodologically distinct stream concerns stochastic dual dynamic programming (SDDP), which approximates cost-to-go functions through iterative Benders-type cuts rather than enumerating discrete states. Parallelization in SDDP primarily exploits scenario-wise and stage-wise subproblem separability within forward-backward iterations. In contrast, the present study focuses on exact state-space Bellman recursion, where parallelism arises from independence among

states within the same stage. A comprehensive discussion of methodological developments and parallelization strategies in SDDP can be found in the survey by Füllner and Rebennack [2025].

From the above literature, two main insights emerge. First, SDP combined with appropriate state-space decomposition provides an exact and effective framework for Markovian project scheduling, yet computational scalability remains the primary limitation, especially for large-scale projects where the number of states grows exponentially. Second, although parallel SDP has been developed in other domains, a dedicated and structurally grounded parallel SDP framework for stochastic project scheduling is still lacking. To fill this gap, the present study makes the following contributions: (1) we propose, to the best of our knowledge, the first parallel SDP specifically designed for stochastic project scheduling, which exploits state-space decomposition and multi-threaded computation to accelerate value iteration without loss of optimality; (2) we study the asymptotic performance of the proposed parallel SDP through theoretical analysis, identifying conditions under which parallelization yields near-linear speedup; and (3) we demonstrate the effectiveness and scalability of the proposed parallel SDP using comprehensive computational experiments, highlighting scenarios where parallelization is most beneficial.

3. Problem description

A project is represented by an activity-on-node (AoN) acyclic graph $G = (V, E)$, where $V = \{0, 1, \cdots, n, n+1\}$ denotes the set of activities and $E \subset V \times V$ holds the precedence relations. A precedence constraint $(i, j) \in E$ implies that activity j cannot start until activity i has finished. An activity becomes eligible once all its predecessors are completed and if it has not yet been scheduled. Following prior research on stochastic project scheduling for maximizing the eNPV [Sobel et al., 2009, Creemers et al., 2010, Hermans and Leus, 2018, Creemers, 2018b], we assume unlimited resource availability. Activities 0 and $n+1$ are dummy activities representing the project's start and finish, respectively. Each activity $i \in V$ has a stochastic duration that follows an exponential distribution with mean d_i, and generates a deterministic cash flow c_i at its start time. As aforementioned, the exponential distribution is the fundamental building block of the PH family, and any nonnegative random variable can be approximated arbitrarily well by a PH distribution. Therefore, the methodologies developed under exponential durations can be naturally extended to more general PH-distributed durations. To calculate the eNPV of a project, future cash flows are discounted to their present values using a continuous discount rate γ. Thus, the present value of a cash flow c_i incurred at time t is $c_i e^{-\gamma t}$, where the continuous-time exponential discount factor $e^{-\gamma t}$ is a standard tool in finance for capturing the time value of money. Although the dummy activities have zero duration, their associated cash flows may be non-zero. For instance, the dummy-start activity typically corresponds to a negative cash flow, representing project startup costs, while the dummy-end activity corresponds to a large positive cash flow, representing revenue obtained from project completion.

Due to the stochastic durations of activities, their start times cannot be fixed in advance. In-

stead, a scheduling policy dynamically determines which eligible activities to start at each decision epoch. Since an activity may start at any time after becoming eligible, the policy space can grow very large, making the computation of an optimal policy computationally intractable. Fortunately, Sobel et al. [2009] showed that, when activity durations are exponentially distributed, restricting decision epochs to the project start and activity completion times preserves optimality. This result drastically reduces the policy space. Moreover, Hermans and Leus [2018] and Creemers [2018b] demonstrate that, under these assumptions, non-preemptive policies remain optimal, further improving the tractability and structural clarity of an optimal solution. Initiating an activity with a negative cash flow prematurely may be suboptimal, as it increases the present value of the incurred cost. Formally, a scheduling policy π is a real-valued function mapping realized activity durations to their corresponding start times, $\pi : \mathbb{R}_{\geq 0}^{n+2} \rightarrow \mathbb{R}_{\geq 0}^{n+2}$. For a given realization $\vec{d} = (\bar{d}_0, \ldots, \bar{d}_{n+1})$, the policy returns a vector of start times $\vec{s} = (s_1, \ldots, s_n)$, where $s_i \geq 0$ for all $i \in V$, and for which the precedence constraints $s_i + \bar{d}_i \leq s_j$ are satisfied for all $(i, j) \in E$. Let Π denote the set of all feasible policies. The problem can then be formulated as

$$\max_{\pi \in \Pi} \sum_{i=0}^{n+1} c_i e^{-\gamma s_i} \tag{1}$$

subject to

$$s_i + \bar{d}_i \leq s_j, \quad \forall (i, j) \in E \tag{2}$$

$$\vec{s} \; = \; \pi(\vec{d}), \quad \pi \in \Pi. \tag{3}$$

Note that the above model is a conceptual model based on duration samples, primarily intended to illustrate the logic of the optimization problem, rather than to actually solve for an optimal policy. In the next section, we will describe in detail how to formulate this problem as an MDP and use SDP to solve for an optimal policy.

4. MDP model and SDP recursion

As mentioned earlier, several scholars have developed different MDP formulations and corresponding SDP methods for the stochastic project scheduling problem considered in this study [Sobel et al., 2009, Creemers et al., 2010, Creemers, 2015, Hermans and Leus, 2018, Creemers, 2018b]. Among these, the model proposed by Creemers [2018b] represents the most advanced and computationally efficient formulation to date. In this section, we introduce this model in detail and later integrate our parallel computing approach with it.

Traditional models for Markovian project networks, such as those proposed by Sobel et al. [2009] and Creemers et al. [2010], are primarily based on the continuous-time Markov chain framework introduced by Kulkarni and Adlakha [1986]. In these models, each activity can exist in one of three states at any time: idle, ongoing, or completed. Consequently, the system state is expressed as a triplet $x = (W_x, P_x, F_x)$, where W_x denotes the set of idle activities, P_x the set of ongoing activities,

and F_x the set of completed activities.

Building upon the optimality of non-preemptive policies and the memoryless property of the exponential distribution, Creemers [2018b] proposed a simplified MDP formulation in which a state defined solely by the set of completed activities. This abstraction drastically reduces the dimensionality of the state space. Conceptually, it can be viewed as a higher-level aggregation of the MDP formulated by Creemers et al. [2010], which consolidates detailed system configurations into compact state-action representations. In this reformulation, the state no longer distinguishes between idle and ongoing activities, and the decision variable (action) is redefined as the set of post-decision ongoing activities. The latter consists of all activities that were started previously but are still ongoing, along with any newly initiated activities in the current decision epoch.

Let ζ_x denote the eligible activity set in state $x = F_x$. To determine an optimal action in state F_x, all non-empty subsets of ζ_x (i.e., all feasible actions) must be evaluated. An optimal post-decision ongoing activity set for any given state maximizes the remaining eNPV of the project upon entry into that state. Let $O_x \subseteq \zeta_x$ denote the selected post-decision ongoing activity set in state x, and ω_x denote the time until the first completion among activities in O_x. Clearly, ω_x is exponentially distributed with parameter $\lambda_x = \sum_{j \in O_x} \mu_j$, where $\mu_j = 1/d_j$ denotes the duration distribution rate parameter of activity j. The probability that activity $i \in O_x$ completes first is μ_i/λ_x, and its completion causes the system to transition from the current state x to the next state $x_i = F_x \cup \{i\}$. Note that $O_{x_i} \setminus O_x$ represents the set of newly starting activities in next state x_i. Let $G_x(O_x)$ denote the remaining eNPV after selecting O_x, which is given by

$$G_x(O_x) = \sum_{i \in O_x} \mathbb{E}\left[e^{-\gamma \omega_x} \frac{\mu_i}{\lambda_x} \left(G_{x_i}(O_{x_i}) + \sum_{j \in O_{x_i} \setminus O_x} c_j \right) \right]. \tag{4}$$

For exponentially distributed activity durations, the memoryless property implies that the identity of the activity completing first is independent of the realized length of the minimum completion time. Therefore, Equation (4) can be rewritten as

$$\begin{aligned} G_x(O_x) &= \mathbb{E}[e^{-\gamma \omega_x}] \sum_{i \in O_x} \frac{\mu_i}{\lambda_x} \left(G_{x_i}(O_{x_i}) + \sum_{j \in O_{x_i} \setminus O_x} c_j \right) \\ &= \int_0^\infty e^{-\gamma t} \, \lambda_x e^{-\lambda_x t} \, \mathrm{d}t \sum_{i \in O_x} \frac{\mu_i}{\lambda_x} \left(G_{x_i}(O_{x_i}) + \sum_{j \in O_{x_i} \setminus O_x} c_j \right) \\ &= \frac{\lambda_x}{\gamma + \lambda_x} \sum_{i \in O_x} \frac{\mu_i}{\lambda_x} \left(G_{x_i}(O_{x_i}) + \sum_{j \in O_{x_i} \setminus O_x} c_j \right). \end{aligned} \tag{5}$$

The remaining eNPV of a project after entering state x consists of two parts: the cost of new activities started in that state and the remaining eNPV after the activities have started. From the perspective of the SDP recursion, an optimal post-decision ongoing activity set for subsequent states

has been determined, but those for preceding states have not yet been determined. Consequently, $G_x(O_x)$ can be calculated accurately using Equation (5), while the cost of newly started activities cannot be calculated accurately because it depends not only on O_x but also on the post-decision ongoing activity set of the previous state. Nevertheless, the following proposition shows that we can obtain an optimal O_x through a surrogate evaluation function.

Proposition 1. *Let $O^- \subseteq O_x$ denote the (unobserved) set of activities that were started in a predecessor state and remain unfinished upon entering state x. An optimal post-decision ongoing set O_x^* that maximizes $\sum_{i\in O_x\setminus O^-} c_i + G_x(O_x)$ for all admissible inherited sets O^- compatible with state x can equivalently be obtained by maximizing $\sum_{i\in O_x} c_i + G_x(O_x)$.*

Proof. Fix a state x and consider any two feasible post-decision ongoing sets $O_x^{(1)}$ and $O_x^{(2)}$. For a given inherited set O^-, the difference between their true remaining eNPV values is

$$\begin{aligned}
\Delta(O^-) &= \left(\sum_{i\in O_x^{(1)}\setminus O^-} c_i + G_x(O_x^{(1)})\right) - \left(\sum_{i\in O_x^{(2)}\setminus O^-} c_i + G_x(O_x^{(2)})\right) \\
&= \left(\sum_{i\in O_x^{(1)}} c_i - \sum_{i\in O_x^{(2)}} c_i\right) - \left(\sum_{i\in O_x^{(1)}\cap O^-} c_i - \sum_{i\in O_x^{(2)}\cap O^-} c_i\right) + \left(G_x(O_x^{(1)}) - G_x(O_x^{(2)})\right) \qquad (6)\\
&= \left(\sum_{i\in O_x^{(1)}} c_i - \sum_{i\in O_x^{(2)}} c_i\right) + \left(G_x(O_x^{(1)}) - G_x(O_x^{(2)})\right).
\end{aligned}$$

This indicated that the sign of $\Delta(O^-)$ is identical for all admissible inherited sets O^-, i.e., an optimal post-decision ongoing set at state x is history-independent. Specifically, the action that maximizes $\sum_{i\in O_x} c_i + G_x(O_x)$ also maximizes the true remaining eNPV. □

Therefore, an optimal set of post-decision ongoing activities in state x is given by

$$O_x^* = \arg\max_{O_x\subseteq\zeta_x} \left(\sum_{i\in O_x} c_i + \frac{\lambda_x}{\gamma+\lambda_x}\sum_{i\in O_x}\frac{\mu_i}{\lambda_x}\left(G_{x_i}(O_{x_i}^*) + \sum_{j\in O_{x_i}\setminus O_x} c_j\right)\right). \qquad (7)$$

Note that, when SDP recursively calculates to the initial state $x_0 = \emptyset$, there is only one optional action $O_\emptyset = \{0\}$. Since the duration of the dummy-start activity is zero, all activities in the post-decision ongoing set of state x_1 must be newly started in this state. Therefore, we can derive $G_\emptyset(\{0\}) = G_{x_1}(O_{x_1}) + \sum_{j\in O_{x_1}} c_j$ from Equation (5). Furthermore, the dummy-start activity must be started in the initial state, and the eNPV of the project is $c_0 + G_\emptyset(\{0\})$.

Because SDP suffers from the curse of dimensionality, an effective state partitioning strategy is essential for solving large-scale instances. Owing to its simplified state representation, the MDP model proposed by Creemers [2018b] not only substantially reduces the size of the state space but also facilitates a natural partitioning of states by the number of completed activities. Specifically,

Creemers [2018b] adopts a backward SDP where the system always transitions from a state with $k-1$ completed activities to one with k, so the SDP recursion only needs to maintain two consecutive state subspaces during computation. This property enables efficient implementation and supports scalable parallelization, as will be demonstrated in subsequent sections.

To more clearly explain the MDP model and SDP recursion, we use a simple project with six activities as shown in Figure 1, to illustrate how to calculate the optimal action for each state. The complete state transition network for this project is shown in Figure 2. In the diagram, the arrows represent state transition relationships. If there is only one post-decision ongoing activity, the corresponding state transition occurs with a probability of 1 (indicated by a solid line). Otherwise, state transition branches will appear (indicated by small circles), and the dashed lines represent the state transitions resulting from the completion of each ongoing activity.

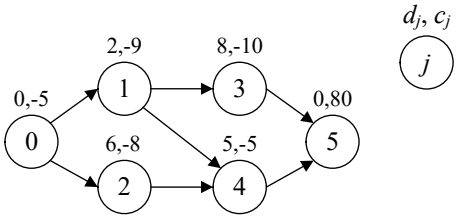


Fig. 1. An example project with six activities.

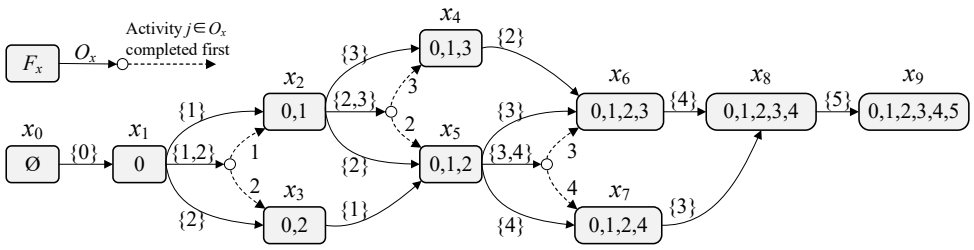


Fig. 2. State transition network of the example project.

As presented in Figure 2, the example project has a total of ten feasible states, with the initial state being $x_0 = \{\emptyset\}$ and the final state being $x_9 = \{0, 1, 2, 3, 4, 5\}$. Except for the final state, each state corresponds to at least one set of optional post-decision ongoing activities. The SDP backward recursion starts from the final state x_9, with an initial value of $G_{x_9}(\emptyset) = 0$. Since states x_8, x_7, and x_6 each have only one possible action, we just need to update $G_{x_8}(\{5\})$, $G_{x_7}(\{3\})$, and $G_{x_6}(\{4\})$ using Equation (5). Specifically, because $G_{x_9}(\emptyset) = 0$ and no new activities are started at x_9, we have $G_{x_8}(\{5\}) = 0$. When calculating $G_{x_7}(\{3\})$ and $G_{x_6}(\{4\})$ based on $G_{x_8}(\{5\})$, it can be observed that both states x_7 and x_6 have only one post-decision ongoing activity, and both involve activity 5 starting in state x_8. Substituting the relevant information into Equation (5), we can obtain $G_{x_7}(\{3\}) = \mu_3(G_{x_8}(\{5\}) + c_5)/(\gamma + \mu_3) = 60.61$ and $G_{x_6}(\{4\}) = \mu_4(G_{x_8}(\{5\}) + c_5)/(\gamma + \mu_4) = 66.67$, respectively. On this basis, we further evaluate the three optional post-decision ongoing activity sets for state x_5. Since $c_3 + G_{x_5}(\{3\}) = 36.72$, $c_3 + c_4 + G_{x_5}(\{3, 4\}) = 41.04$ and

$c_4 + G_{x_5}(\{4\}) = 37.17$, the optimal post-decision ongoing activity set for state x_5 is identified to be $\{3, 4\}$. We continue the backward recursive calculations until state x_1 is reached, resulting in $c_1 + G_{x_1}(\{1\}) = 15.44$, $c_1 + c_2 + G_{x_1}(\{1, 2\}) = 16.3$, and $c_2 + G_{x_1}(\{2\}) = 15.39$. This indicates that the optimal post-decision ongoing set for state x_1 is $\{1, 2\}$. Therefore, we have $G_\emptyset(\{0\}) = 16.3$ and the optimal eNPV for the project is $c_0 + G_\emptyset(\{0\}) = -5 + 16.3 = 11.3$.

5. Parallelized SDP

5.1. Multithreading based on producer-consumer pattern

Although the SDP framework of Creemers [2018b] substantially reduces the state space compared with classical formulations, solving large instances remains computationally intractable. The main bottleneck originates from the rapid growth of both the state space and the feasible action space when many activities become concurrently executable. Consequently, each stage requires evaluating a large number of states, and for each state, a combinatorially large set of admissible decisions must be examined.

From the perspective of algorithmic parallelization, our parallelization relies on the decomposable stage-wise structure of the SDP. Specifically, all value-function evaluations within the same stage are independent, since each state can only transition to the next stage with exactly one additional completed activity. This property naturally enables a parallel evaluation scheme in which the Bellman updates for different states are carried out concurrently.

We adopt a producer-consumer strategy [Dijkstra, 2002], which is widely used in parallel search and dynamic programming [Dias et al., 2013]. In this scheme, one producer thread is responsible for generating feasible states of the current stage, while multiple consumer threads perform the computationally intensive value-function updates. The communication topology follows a shared-memory queue, providing dynamic load balancing: whenever a consumer becomes idle, it immediately retrieves the next available state and evaluates its optimal decision.

In principle, the producer-consumer paradigm allows multiple producer threads. In our case, however, we fix the number of producer threads to one for two reasons. First, each state x is assigned a unique numerical index $\sum_{i \in F_x} 2^i$, which induces a deterministic ascending order within each stage. Maintaining this order is essential, as it enables efficient binary search when locating successor states during Bellman updates. When enumerating states, the states from the previous stage are processed sequentially, giving priority to those that leave activities with larger indices unfinished, thus ensuring that the states enumerated first correspond to smaller index values. If multiple producer threads are used to enumerate states in parallel, it may lead to states with larger index values being enumerated first, resulting in additional synchronization costs. Second, state enumeration is typically much cheaper than optimal action evaluation. Therefore, a single producer thread is sufficient, while the computational effort is concentrated on the parallel evaluation of $G_x(\cdot)$ over feasible actions. The overall parallel workflow is illustrated in Figure 3.

The above algorithmic scheme is implemented through a shared-memory multithreading en-

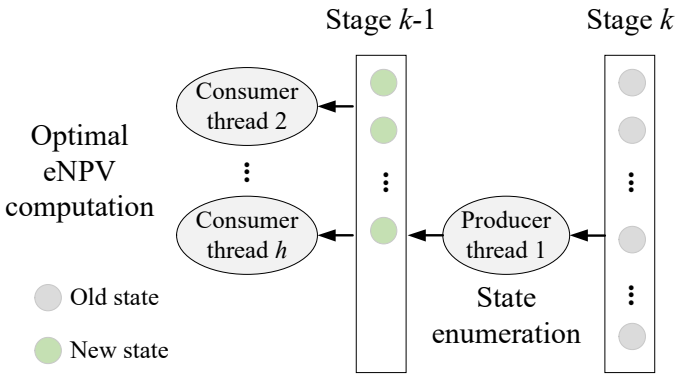


Fig. 3. Parallel computation at each stage of the SDP using a multithreading technique.

vironment on multi-core CPUs. Let Q be the shared queue storing enumerated states awaiting evaluation. At each stage, the producer thread sequentially enumerates feasible states and pushes them into the queue Q. Consumer threads repeatedly extract states from Q and compute the corresponding optimal action O_x^* by evaluating Eq. (7). Once enumeration is completed, the producer thread switches to a consumer role to further improve hardware utilization. Algorithm 1 summarizes the resulting parallel SDP procedure, where X_k denote the state subspace at stage $k = 1, 2, \ldots, n$. This design ensures efficient workload distribution across threads and achieves high utilization of available computational resources.

Algorithm 1 Parallel SDP

Require: $V, E, (c_i, d_i)_{i \in V}, \gamma, h$
$G_V(\emptyset) \leftarrow 0$, $X_n \leftarrow \{V\}, Q \leftarrow \emptyset$
for stage $k \leftarrow n$ **down to** 1 **do**
  initialize a shared flag *EnumDone* $\leftarrow$ **false**
  for all threads $l = 1, \ldots, h$ **in parallel do**
    if $l = 1$ **then** ▷ producer thread
      for all feasible states $x \in X_{k-1}$ based on X_k **do**
        push state x into Q
      end for
      EnumDone $\leftarrow$ **true**
    end if ▷ consumer threads
    while not (*EnumDone* = **true and** Q **is empty**) **do**
      if Q not empty **then**
        retrieve a state x from Q
        compute O_x^* and update $G_x(O_x^*)$
      end if
    end while
  end for
  free memory occupied by X_k
end for
return $c_0 + G_\emptyset(\{0\})$

5.2. Classical speedup laws as baseline references

To assess the scalability of the proposed parallel SDP algorithm, we employ two classical speedup laws, namely Amdahl's law and Gustafson's law, as baseline references. These models provide standard theoretical bounds that help interpret the observed acceleration and clarify the role of unavoidable serial overhead.

Amdahl's law [Amdahl, 1967] considers a fixed problem size and quantifies the maximum speedup achievable when only part of the computation can be parallelized. Let $\alpha \in [0, 1]$ denote the fraction of execution time that can be distributed across processors, and let $1 - \alpha$ represent the inherently serial portion. The speedup obtained with h processors is given by

$$S_a(h) = \frac{1}{(1-\alpha) + \frac{\alpha}{h}}. \tag{8}$$

This expression shows that the serial fraction imposes a strict upper bound on acceleration. Even with a large number of processors, the achievable speedup remains limited when $1 - \alpha$ is non-negligible.

Gustafson's law [Gustafson, 1988] adopts a different viewpoint by allowing the workload to scale with the number of processors. Under this assumption, larger instances can be solved within the same time budget, and the corresponding scaled speedup is

$$S_g(h) = \alpha h + (1 - \alpha). \tag{9}$$

Compared with Amdahl's model, Gustafson's law suggests that near-linear speedup may still be attainable for large-scale computations, provided that the parallel workload dominates the overall runtime. When $\alpha = 0.9$ and $h = 24$, Amdahl's law yields a speedup of approximately 7.27, whereas Gustafson's law predicts 21.7. The difference reflects the distinct assumptions regarding workload scaling and the impact of serial components.

Overall, these two laws provide useful baseline references for understanding the scalability limits of the parallel SDP. It should be noted that they are derived under the assumption of homogeneous processing units and therefore are not intended as quantitatively accurate predictors on hybrid-core architectures. In addition, both models neglect several algorithm-dependent overheads that become critical in practice, including synchronization costs, memory access latency, and workload imbalance. Such effects are particularly relevant for the proposed parallel SDP, as also emphasized in modern parallel performance studies [Roucairol, 1996, Sun and Chen, 2010, Schryen, 2024]. Accordingly, rather than relying on classical laws alone, we first identify the dominant sources of serial overhead specific to our implementation in the subsequent section. These sources then form the basis of a refined asymptotic speedup model for the parallel SDP.

5.3. Computing efficiency improvement strategies

Both Amdahl's and Gustafson's laws indicate that parallel efficiency depends critically on reducing the effective serial overhead. In the proposed parallel SDP, such overhead is not only caused by strictly sequential code segments, but also by synchronization costs and workload imbalance across threads. This section identifies the dominant sources of inefficiency and summarizes the implementation strategies adopted to mitigate them.

The main contributors to non-parallelizable runtime can be classified as follows.

- **Inherently sequential components.** This category includes operations that are fundamentally non-parallelizable. Typical examples are initialization and finalization routines (e.g., memory allocation for data structures and resource deallocation), control logic (e.g., sequential execution of the main loop that manages parallel tasks), and thread management overhead (e.g.,thread creation, scheduling, and termination). These components are unavoidable but should be minimized through careful design.

- **Synchronization and contention overhead.** The parallel SDP implementation follows a producer-consumer structure, where one producer thread enumerates feasible states and multiple consumer threads perform Bellman value updates. Consumer threads may remain idle when state enumeration becomes a bottleneck. Moreover, concurrent access to shared task queues introduces lock contention and waiting costs, which can significantly reduce efficiency, especially for small instances.

- **Load imbalance across states.** The computational effort required to evaluate a state depends critically on the number of eligible activities. More specifically, the number of candidate actions grows as $2^{|\zeta_x|} - 1$, so even moderate differences in $|\zeta_x|$ can translate into substantial variations in per-state workload. In the $n = 40$ instances, for example, adjacent states often differ by more than 10 eligible activities, implying a disparity of over 512-fold in computational effort. Consequently, an uneven allocation of states across consumer threads may cause some threads to finish earlier and remain idle while others continue processing, thereby increasing the effective serial fraction of the overall runtime.

Based on this analysis, the following implementation strategies are incorporated to improve parallel efficiency.

- **Thread pooling and reuse.** A thread-pool mechanism is adopted to avoid repeated thread creation and termination across stages, thereby reducing management overhead and improving performance stability.

- **Reducing waiting time and memory interference.** The optimized implementation addresses two main types of synchronization overhead. The first involves waiting time for state enumeration, where the producer thread is assigned to a performance core to accelerate

enumeration and reduce idle time for consumer threads. The second type pertains to inter-thread interference caused by competition for computing resources and shared memory. To mitigate this, threads are pinned to separate physical cores, and their private data caches are aligned to avoid false sharing, ensuring more efficient execution.

- **Dynamic load balancing.** Two complementary load-balancing strategies are employed to promote near-synchronous thread completion. First, a dynamic task allocation scheme is adopted: upon finishing its current computation, each consumer thread immediately retrieves the next available task from the shared queue. This approach mitigates the inefficiencies of static assignment schemes, such as round-robin scheduling or workload-partitioning methods that rely on prior cost assessment, by adaptively redistributing tasks at runtime. Second, after completing state enumeration, the producer thread joins the consumer pool and participates in the remaining Bellman evaluations, thereby further improving overall utilization.

The effectiveness of these implementation strategies is assessed in the subsequent computational experiments. Importantly, the parallel overheads discussed above are not necessarily constant as instance complexity increases. Therefore, the next section provides a more in-depth theoretical analysis of the asymptotic behavior of the dominant overhead components that limit the computational efficiency of parallel SDP. In this sense, these factors are not merely implementation-level concerns, but also form the core elements of the performance model developed thereafter.

6. Asymptotic properties of parallelized SDP

Based on the decomposition of the serial component in Section 5.3, we identify three dominant overhead sources that limit the computational efficiency of the parallelized SDP: (1) the delay caused by waiting for state enumeration, (2) workload imbalance across threads, and (3) contention for shared computing resources. Since the primary motivation for parallelizing SDP is to improve its ability to solve increasingly complex instances, our emphasis is on the asymptotic evolution of these overhead components as instance complexity grows. Accordingly, this section introduces quantitative metrics to capture these effects, investigates their behavior as the state space expands, and derives an explicit overhead-aware speedup model for the proposed algorithm. In contrast to the classical baseline laws reviewed in Section 5.2, the resulting characterization incorporates algorithm-specific overhead terms, thereby providing a more accurate explanation and prediction of the observed parallel performance.

The theoretical analysis below assumes that parallel SDP is executed on a shared-memory system (e.g., a single machine), so communication costs can be neglected. This assumption is well justified, as the proposed implementation is primarily intended for single-node execution. Extending the approach to a distributed-memory cluster would likely introduce communication overheads that outweigh the benefits of parallelism. In particular, because state enumeration is performed by a single producer thread to preserve an ordered state structure, the resulting state information must either be centralized on one node or distributed across nodes. In the former case, consumer threads

on other nodes would require frequent remote memory access, whereas in the latter case additional data transfers would be needed to dispatch enumerated states to the appropriate nodes. Both schemes would therefore incur substantial communication costs. For analytical convenience, we further assume homogeneous computational capabilities across threads. Although modern CPUs often adopt hybrid architectures with performance and efficiency cores, resulting in non-uniform per-thread speeds, this heterogeneity does not affect the asymptotic conclusions of the analysis.

6.1. Proportion of state enumeration

A higher proportion of state enumeration time in the total parallel computation time indicates longer waiting periods for consumer threads. Hence, we measure the overhead caused by enumeration waiting through the ratio of enumeration time to total computation time.

Definition 1 (Proportion of state enumeration)**.** *Let T_{enum} denote the state enumeration time, t_{kl} denote the cumulative computation time (excluding idle time) of thread $l = 1, \ldots, h$ in stage $k = 1, \ldots, n$, and $T_{cum}(h) = \sum_{k=1}^{n} \sum_{l=1}^{h} t_{kl}$ the cumulative computation time of all threads. The proportion of state enumeration is defined as*

$$\phi(h) = \frac{T_{enum}}{T_{cum}(h)}, \tag{10}$$

with $0 < \phi(h) \leq 1$.

Although $\phi(h)$ in Eq. (10) is defined based on actual computation time, its asymptotic behavior can be analyzed from the perspective of computational complexity. Enumerating a new state requires changing one completed activity in the previous stage to an unfinished one, provided all its successors remain unfinished; this incurs a constant computational cost $c_e > 0$. For a given state $x = F_x$, suppose the cardinality of the set of eligible activities ζ_x is g (bounded uniformly for all relevant states). Computing an optimal value of F_x requires evaluating all non-empty subsets of ζ_x, i.e., $2^g - 1$ subsets. Let $c_v > 0$ be the amortized cost per subset in the value computation step, independent of g. Additionally, each activity completion triggers a binary search in the next-stage state space $X_{|F_x|+1}$, with total complexity $g\mathcal{O}(\log |X_{|F_x|+1}|)$ per state. Let $X = \bigcup_{k=1}^{n} X_k$ be the complete state space with total size $|X|$. Typically, $|X|$ grows polynomially or subexponentially with g. The total enumeration time is thus $T_{\text{enum}} = c_e|X|$, and a lower bound on the total value-computation time that excluding binary search overhead is $\tilde{T}_{\text{val}} = c_v|X|\big(2^g - 1\big)$. Hence, a lower bound on the total computation time is $T_{\text{LB}} = T_{\text{enum}} + \tilde{T}_{\text{val}}$. Since the total workload of serial and parallel computations is the same for a given instance, T_{LB} is effectively a lower bound for the total time spent in both serial and parallel computations. Furthermore, considering that parallel computation inevitably incurs a parallel overhead related to h due to thread contention for computational resources, $T_{\text{cum}}(h)$ is usually significantly greater than T_{LB}.

Theorem 1. *The proportion of enumeration time in the parallelized SDP with h threads satisfies*

$$\lim_{g\to\infty} \phi(h) = 0. \tag{11}$$

Proof. Substituting the lower bound T_{LB} into Eq. (11) yields the inequality

$$\phi(h) \le \frac{T_{\mathrm{enum}}}{T_{\mathrm{LB}}} = \frac{c_e\,|X|}{c_e\,|X| + c_v\,|X|\,\left(2^g - 1\right)} = \frac{c_e}{c_e + c_v\,(2^g - 1)}. \tag{12}$$

Since $c_e, c_v > 0$ and $g \to \infty$ implies $2^g \to \infty$, the denominator tends to infinity and the right-hand side tends to 0. □

Theorem 1 shows that in complex instances (i.e., with a larger g), the waiting time for state enumeration has negligible influence on the computational efficiency of the parallelized SDP, even with a single producer thread.

6.2. Load-balancing efficiency

Load balance among threads is assessed using their actual computation times per stage. When the maximum thread computation time approximates the average, the load is well balanced. We define the load-balancing efficiency as follows:

Definition 2 (Load-balancing efficiency)**.** *Define the ideal and actual computation times per stage as $a_k = T_{\mathrm{ideal},k} = \frac{1}{h}\sum_{l=1}^{h} t_{kl}$ and $b_k = T_{\mathrm{actual},k} = \max_{1\le l\le h} t_{kl}$, respectively. The global ideal and actual times are $T_{\mathrm{ideal}} = \sum_{k=1}^{n} a_k$ and $T_{\mathrm{actual}} = \sum_{k=1}^{n} b_k$. Then the global load-balancing efficiency is*

$$\beta(h) = \frac{T_{ideal}}{T_{actual}} \tag{13}$$

with $0 < \beta(h) \le 1$.

Let w_{kq} denote the computational load of q-th state in stage k. Since most states at the same stage have similar numbers of eligible activities, we model w_{kq} as independent random variables, and threads as sampling without replacement from this set. Two mild assumptions are introduced:

(A1) (Moment bounds) For each stage k, the computational load of each state satisfies $\mathbb{E}[w_{kq}] = \rho_{kq}$, $\mathrm{Var}(w_{kq}) \le \sigma_k^2$, and $w_{kq} \le B_k$ almost surely. Define the stage mean $\rho_k = \frac{1}{|X_k|}\sum_q \rho_{kq}$.

(A2) (Dense regime) Stage k is dense if $|X_k| \gg h$, so that with high probability each thread processes about $m_k \approx |X_k|/h$ states; otherwise, it is sparse.

Assumption (A1) is reasonable, as the magnitude of $|\zeta_x|$ is typically moderate. In most solvable instances, $|\zeta_x|$ rarely exceeds 30, and even in the worst case it remains bounded by n. According to the well-known Popoviciu's inequality on variances [Popoviciu, 1935], for any bounded random variable $z \in [a, b]$, the variance is limited by $\mathrm{Var}(z) \le (b-a)^2/4$. This ensures that the variability of w_{kq} is inherently constrained. Assumption (A2) captures the structural imbalance of the state

space, where the initial and terminal project stages contain only a few feasible states (sparse), while the intermediate stages exhibit exponentially larger state spaces (dense).

The following lemma bounds the maximum load deviation per dense stage, leading to an asymptotic convergence of $\beta(h)$.

Lemma 1. *Fix a dense stage k with $m_k \approx |X_k|/h$ states per thread. Under (A1)–(A2), for any $\delta \in (0,1)$, with probability at least $1-\delta$,*

$$\max_{1\le l\le h}\big(t_{kl}-\mathbb{E}[t_{kl}]\big) \le \frac{2}{3}B_k \ln\frac{h}{\delta} + \sigma_k\sqrt{2m_k\ln\frac{h}{\delta}}. \tag{14}$$

Consequently, defining

$$\varepsilon_k(\delta) = \frac{2}{3}\frac{B_k}{m_k\rho_k}\ln\frac{h}{\delta} + \frac{\sigma_k}{\rho_k}\sqrt{\frac{2\ln(h/\delta)}{m_k}}, \tag{15}$$

we have, with probability at least $1-\delta$,

$$b_k \le a_k\big(1+\varepsilon_k(\delta)\big). \tag{16}$$

Proof. For a fixed thread l consider the (approximately) m_k state loads assigned to it, $w_{l,1},\dots,w_{l,m_k}$. By (A1) the sum $t_{kl}=\sum_{r=1}^{m_k} w_{l,r}$ satisfies Bernstein's inequality [Boucheron et al., 2013]: for any $\tau>0$,

$$\Pr\big(t_{kl}-\mathbb{E}[t_{kl}]\ge t\big) \le \exp\Big(-\frac{\tau^2}{2U_l+\frac{2}{3}B_k\tau}\Big), \tag{17}$$

where $U_l=\sum_{r=1}^{m_k}\mathrm{Var}(w_{l,r}) \le m_k\sigma_k^2$. Applying the union bound over h threads,

$$\Pr\Big(\exists\, l:\ t_{kl}-\mathbb{E}[t_{kl}]\ge t\Big) \le h\exp\Big(-\frac{\tau^2}{2m_k\sigma_k^2+\frac{2}{3}B_k\tau}\Big). \tag{18}$$

Set the right-hand side equal to δ and write $L=\ln(h/\delta)>0$, we have

$$\frac{\tau^2}{2m_k\sigma_k^2+\frac{2}{3}B_k\tau} = L. \tag{19}$$

The threshold $\tau>0$ is obtained by solving Eq. (19), which is

$$\tau = \frac{\frac{2}{3}B_kL+\sqrt{\frac{4}{9}B_k^2L^2+8m_k\sigma_k^2L}}{2}. \tag{20}$$

Using $\sqrt{A^2+B}\le A+\sqrt{B}$ with $A=\frac{2}{3}B_kL$ and $B=8m_k\sigma_k^2L$ yields

$$\tau \le \frac{2}{3}B_kL+\sigma_k\sqrt{2m_kL}, \tag{21}$$

which gives (14). The sampling-without-replacement nature of dynamic state pulling admits Serfling-type bounds [Serfling, 1974] that preserve this scaling when $|X_k|\gg h$. Since $\mathbb{E}[t_{kl}]=$

$\sum_{r=1}^{m_k} \mathbb{E}[w_{l,r}] \approx m_k \rho_k$, dividing both sides of the inequality (14) by $a_k \approx m_k \rho_k$ yields

$$\frac{b_k - a_k}{a_k} \leq \frac{2}{3} \frac{B_k}{m_k \rho_k} \ln \frac{h}{\delta} + \frac{\sigma_k}{\rho_k} \sqrt{\frac{2 \ln(h/\delta)}{m_k}}, \tag{22}$$

and define $\varepsilon_k(\delta)$ as the right-hand side yields (15) and (16). □

Let $\mathcal{S}_{\text{dg}}$ and $\mathcal{S}_{\text{sp}}$ denote the sets of dense and sparse stages, respectively. The following theorem shows that since the load imbalance of the dense stage tends to 0 and the dense stage plays a dominant role in the computation, the global load-balancing efficiency converges to 1.

Theorem 2. *Under the conditions of Lemma 1, if $m_k \to \infty$ for all dense stages and the total ideal work of sparse stages is negligible, then for any fixed confidence $\delta \in (0, 1)$, the global load-balancing efficiency $\beta(h) \to 1$ as the instance size grows.*

Proof. From Lemma 1 we have $b_k \leq a_k(1 + \varepsilon_k(\delta))$ for dense stages with a probability greater than or equal to $1 - \delta$. Hence

$$\beta(h) = \frac{\sum_k a_k}{\sum_k b_k} \geq \frac{1}{1 + \dfrac{\sum_k a_k \varepsilon_k(\delta)}{\sum_k a_k}} = \frac{1}{1 + \dfrac{\sum_{k \in \mathcal{S}_{\text{dg}}} a_k \varepsilon_k(\delta) + \sum_{k \in \mathcal{S}_{\text{sp}}} a_k \varepsilon_k(\delta)}{\sum_k a_k}}. \tag{23}$$

For dense stages $\varepsilon_k(\delta) \to 0$ and $\sum_{k \in \mathcal{S}_{\text{dg}}} a_k / \sum_k a_k \to 1$, so the dense contribution vanishes. For sparse stages use the trivial bound $\varepsilon_s(\delta) \leq C$ (finite), hence their weighted contribution is at most $C \sum_{k \in \mathcal{S}_{\text{sp}}} a_k / \sum_k a_k \to 0$. Therefore the weighted deficiency tends to zero, which implies $\beta(h) \to 1$ in probability. □

Similarly to Theorem 1, which establishes that $\phi(h)$ eventually converges to 0, Theorem 2 shows that, for sufficiently complex instances and any fixed number of threads, $\beta(h)$ converges to 1. In contrast, for small instances, $\beta(h)$ may decline as the number of threads increases. In this regime, the fixed costs associated with parallel execution, such as thread management and synchronization, represent a non-negligible fraction of the total runtime, and even minor stochastic delays across threads can lead to disproportionate load imbalance. When the computational workload per thread is small, the marginal gains from additional parallelism are therefore insufficient to offset these effects, resulting in reduced load balance.

6.3. Computation time increment

The third metric quantifies the delay caused by competition for computing resources among threads. For a fixed instance, the total computational workload of the parallel and serial versions is identical. If threads execute independently and maintain serial efficiency, the cumulative computation time of all threads (excluding idle time) should equal the serial time. In practice, it is typically larger due to heterogeneous core performance, frequency throttling under high load, and bandwidth contention.

Definition 3 (Incremental percentage of computation time)**.** *Let T_h denote the total computation time with h threads (e.g., T_1 is the serial computation time). The incremental percentage of total parallel computation time relative to serial computation time is defined as*

$$\Delta T(h) = \frac{T_{cum}(h) - T_1}{T_1} \times 100\%. \tag{24}$$

Let the total computational work W consist of arithmetic workload W_c and memory-access workload W_m. The serial computation time can be approximated as

$$T_1 \approx \frac{W_c}{f} + \frac{W_m}{\theta}, \tag{25}$$

where f is the single-core computational throughput and θ is the available memory bandwidth.

Under parallelization, the arithmetic workload is evenly divided, but total bandwidth is shared. Let f_l and θ_l denote the per-thread throughput and bandwidth. Since frequency reduction is limited, $f_l \approx f$, whereas $\theta_l \approx \theta/h$. Thus, for each thread,

$$t_l \approx \frac{W_c}{hf} + \frac{W_m}{\theta}, \tag{26}$$

and the cumulative time becomes

$$T_{\text{cum}}(h) = ht_l \approx \frac{W_c}{f} + h\frac{W_m}{\theta}. \tag{27}$$

As $W \to \infty$, the ratio $T_{\text{cum}}(h)/T_1$ converges to

$$\lim_{W\to\infty} \frac{T_{\text{cum}}(h)}{T_1} \approx \frac{\eta + h(1-\eta)}{\eta + (1-\eta)}, \tag{28}$$

where $\eta = \frac{W_c/f}{W_c/f + W_m/\theta}$ represents the normalized compute intensity. This ratio approaches h (i.e., $\Delta T(h) \to (h-1) \times 100\%$) as $\eta \to 0$ (memory-bound regime), and approaches 1 (i.e., $\Delta T(h) \to 0$) as $\eta \to 1$ (compute-bound regime).

Unlike the first two metrics (i.e., $\phi(h)$ and $\beta(h)$), $\Delta T(h)$ is relatively difficult to quantify and is more sensitive to h. Intuitively, $\Delta T(h)$ may decrease for simple instances due to improved data locality, but tends to increase as instance complexity grows and memory effects become dominant. Moreover, increasing the number of threads generally amplifies this effect, as additional threads place greater pressure on shared resources and introduce extra synchronization overhead. These aspects will be addressed in detail through numerical analysis in the computational experiments.

6.4. Asymptotic speedup model

Theorems 1 and 2 indicate that for large-scale instances, the effects of state-enumeration waiting and load imbalance on speedup become negligible. Therefore the computation time increment ΔT

becomes the dominant factor limiting parallel efficiency. Let the actual speedup be

$$S(h) = \frac{T_1}{T_h}. \tag{29}$$

When all threads are fully utilized and workloads are balanced, $T_h \approx T_{\text{cum}}(h)/h = T_1(1 + \Delta T(h))/h$. Substituting into Eq. (29) yields the asymptotic speedup model

$$S'(h) \approx \frac{h}{1 + \Delta T(h)}. \tag{30}$$

This expression provides an accurate estimate of the achievable speedup of the parallelized SDP under large-scale instances.

7. Computational study

To evaluate the effectiveness of the proposed parallel computing approach, we conducted extensive computational experiments using benchmark instances. The computational study is organized to progressively examine (1) the impact of implementation-level improvement strategies, (2) the observed speedup under increasing thread counts, (3) the empirical behavior of the dominant overhead factors analyzed in Section 6, and (4) the capability of the parallelized SDP to solve substantially more complex instances that require extremely long runtimes for the serial SDP.

All algorithms were implemented in C++ and executed on a personal computer equipped with an Intel Core i9-13900K processor and 128 GB of RAM. The Intel Core i9-13900K features a hybrid architecture consisting of 8 performance cores and 16 efficiency cores. The base frequencies of the performance cores and efficiency cores are 3.00 GHz and 2.20 GHz, respectively. Each software thread is pinned to a distinct physical core via processor affinity, and the number of active threads never exceeds the 24 physical cores, avoiding oversubscription.

7.1. Experimental setup

We employed the benchmark dataset introduced by Creemers et al. [2010], generated using the RanGen generator [Demeulemeester et al., 2003]. The dataset contains 1080 instances covering 12 project sizes ($n \in \{10, 20, \ldots, 120\}$) and three levels of order strength ($OS \in \{0.4, 0.6, 0.8\}$), with 30 instances for each parameter combination. Order strength (OS) measures the structural density of the project network and is defined as

$$OS = \frac{2M}{n(n-1)}. \tag{31}$$

where M denotes the total number of precedence-related activity pairs (both direct and indirect) among non-dummy activities.

In each instance, the dummy end activity has a relatively large positive cash flow representing

the revenue obtained upon project completion, whereas all other activities are associated with negative cash flows reflecting initiation or execution costs. As noted by Creemers et al. [2010], in this setting the cash flow values of individual activities have little influence on the algorithmic performance, since they do not affect the MDP state space nor the set of actions evaluated in each state. However, if the cash flow of a non-dummy activity is changed from negative to positive, it becomes optimal under resource-unconstrained conditions to start the activity as early as possible. This reduces the number of relevant candidate actions and, consequently, the computational effort required for evaluation.

Two classic metrics are adopted to assess the performance of the parallelized SDP. The first is the actual speedup $S(h)$, as defined in Eq. (29). The second is the parallel efficiency, which defined as $\epsilon(h) = S(h)/h$. Parallel efficiency is used to measure the scalability of parallel algorithms. A parallel efficiency close to 1 indicates that the computational resources are utilized effectively as the number of threads increases.

The experiments are organized into three groups. The first group evaluates the effectiveness of the proposed implementation improvements. Here, the parallelized SDP with 24 threads is applied to both simple instances ($n = 30$, $OS = 0.4$) and complex instances ($n = 60$, $OS = 0.4$). The second group investigates the actual speedup and scalability across various instance sizes. Experiments are conducted on small- and medium-scale instances that can be solved by the serial SDP within a reasonable time, specifically $n \in \{10, \ldots, 60\}$ for $OS = 0.4$, $n \in \{10, \ldots, 90\}$ for $OS = 0.6$, and $n \in \{10, \ldots, 120\}$ for $OS = 0.8$. Both the serial SDP and the parallelized SDP with 2 to 24 threads are tested for each configuration. The third group focuses on more complex instances that are computationally prohibitive for the serial SDP, including $n \in \{70, 80\}$ with $OS = 0.4$, and $n \in \{100, 110, 120\}$ with $OS = 0.6$, all solved using the parallelized SDP with 24 threads.

The maximum number of stored states at any stage for the SDP is set to 1.5×10^8, Under this configuration, the total memory consumption is approximately 23 GB. It is noteworthy that the parallelized and serial versions of the SDP consume identical amounts of memory, since both require storing only the states corresponding to two adjacent stages at any given time.

7.2. Effectiveness of the improvement strategies

We first assess the contribution of the improvement strategies proposed in Section 5.3, which are designed to reduce the serial overheads. Specifically, we employed a thread pool to avoid frequent thread creation and destruction, assigned the producer role to a thread running on a performance core to reduce waiting time for state enumeration, aligned thread-private cache data to prevent delays caused by false sharing, and implemented dynamic load balancing. To verify the effectiveness of these strategies, we refer to the SDP incorporating all improvements as the baseline method. Four variants were constructed by disabling one strategy at a time: (1) creating and destroying threads at each stage ($\text{SDP}_{\text{stage}}$); (2) selecting an efficiency-core thread as the producer (SDP_{ecp}); (3) disabling cache alignment (SDP_{dca}); and (4) employing round-robin task assignment ($\text{SDP}_{\text{round}}$).

These variants, each utilizing 24 threads, were tested on two sets of instances: ($n = 30, OS =$

0.4) and ($n = 60, OS = 0.4$). Table 2 reports the computation time (in seconds) of each variant and its percentage deviation from the baseline, defined as

$$\text{Dev.}(\%) = \frac{(T_{\text{given}} - T_{\text{base}})}{T_{\text{base}}} \times 100\%, \tag{32}$$

where T_{base} and T_{given} denote the computation time of the baseline and the given variant, respectively.

Table 2
Comparison of different variants of the parallelized SDP with 24 threads.

SDP variants	$n = 30$, $OS = 0.4$		$n = 60$, $OS = 0.4$	
	T_{24}	Dev.(%)	T_{24}	Dev.(%)
Baseline	0.007	0	173.694	0
$\text{SDP}_{\text{stage}}$	0.026	243.86	173.807	0.07
SDP_{ecp}	0.008	9.37	173.475	-0.13
SDP_{dca}	0.009	21.35	222.151	27.90
$\text{SDP}_{\text{round}}$	0.009	22.91	255.25	46.95

The results in Table 2 confirm that the baseline method achieves the highest efficiency among all tested variants. Specifically, creating and destroying threads at each stage ($\text{SDP}_{\text{stage}}$) increases the solution time by 243.86% for simple instances, although its impact becomes negligible for complex instances. Similarly, using an efficiency core as the producer (SDP_{ecp}) significantly increases the waiting time for state enumeration in simple instances but has little effect for complex ones. In contrast, disabling cache alignment (SDP_{dca}) and employing round-robin task assignment ($\text{SDP}_{\text{round}}$) both lead to substantial increases in computation time for both simple and complex instances. Overall, the results validate the effectiveness of the efficiency-improvement strategies, and the baseline SDP is therefore adopted in all subsequent experiments.

7.3. Observed speedup of the parallelized SDP

We next report the observed speedup and parallel efficiency of the parallelized SDP as the number of threads increases. Table 3 summarizes the results of the serial SDP and the parallelized SDP with 24 threads, including the number of successfully solved instances, the average computation time in seconds, the observed speedup using 24 threads, and the average state-space size (denoted as #States). In addition, the average state-space sizes of solved instances across all parameter combinations are illustrated in Figure 4.

As shown in Table 3, although the SDP can solve all instances up to $n = 60$, the serial computation time grows sharply with increasing instance complexity (i.e., larger n or smaller OS). The root cause of this computational explosion lies in the exponential growth of both the state and action spaces, as illustrated in Figure 4. It can be observed that, for a fixed OS, the state-space size increases exponentially with n, although it remains several orders of magnitude smaller than that of the model proposed by Creemers et al. [2010]. Conversely, for a fixed n, the state-space size increases substantially as OS decreases. These observations align with theoretical expectations: each state corresponds to a unique combination of completed activities, and thus the total number

Table 3
Computational results of the serial SDP and the parallelized SDP with 24 threads.

n	OS	Solved	T_1	T_{24}	$S(24)$	#States
10	0.8	30	0.001	0.001	0.97	2.18×10
10	0.6	30	0.001	0.001	1.16	4.08×10
10	0.4	30	0.001	0.001	0.97	8.42×10
20	0.8	30	0.001	0.002	0.84	8.77×10
20	0.6	30	0.001	0.002	0.89	3.30×10^2
20	0.4	30	0.003	0.002	1.67	1.62×10^3
30	0.8	30	0.002	0.002	0.79	2.54×10^2
30	0.6	30	0.004	0.003	1.33	1.90×10^3
30	0.4	30	0.060	0.007	8.04	1.71×10^4
40	0.8	30	0.003	0.003	0.91	6.62×10^2
40	0.6	30	0.023	0.005	4.35	9.48×10^3
40	0.4	30	3.906	0.273	14.28	1.94×10^5
50	0.8	30	0.004	0.004	1.01	1.54×10^3
50	0.6	30	0.201	0.019	10.33	4.04×10^4
50	0.4	30	89.154	5.867	15.20	1.66×10^6
60	0.8	30	0.007	0.005	1.41	3.56×10^3
60	0.6	30	2.232	0.150	14.84	1.75×10^5
60	0.4	30	2909.452	173.694	16.75	1.38×10^7
70	0.8	30	0.016	0.007	2.23	7.75×10^3
70	0.6	30	19.748	1.313	15.04	7.28×10^5
80	0.8	30	0.043	0.010	4.14	1.64×10^4
80	0.6	30	103.447	6.774	15.27	2.26×10^6
90	0.8	30	0.130	0.018	7.26	3.41×10^4
90	0.6	30	1203.441	76.998	15.63	9.85×10^6
100	0.8	30	0.367	0.036	10.19	6.67×10^4
110	0.8	30	1.309	0.103	12.75	1.47×10^5
120	0.8	30	8.277	0.577	14.35	5.16×10^5

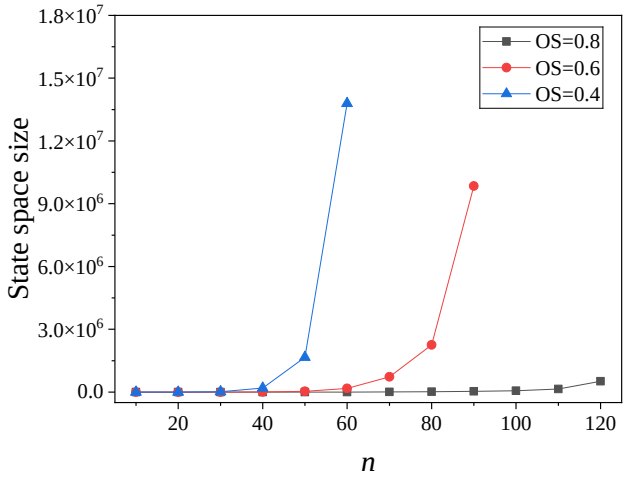


Fig. 4. Average state-space size of solved instances for each parameter combination.

of feasible states increases exponentially with the number of eligible activities. A larger OS implies a denser project network with more precedence constraints, reducing the number of concurrently executable activities, whereas a smaller OS yields a sparser network with more eligible activities, thereby enlarging the feasible state space.

The parallelized SDP with 24 threads exhibits a dramatic reduction in computational time compared with the serial SDP, and its observed speedup increases with problem complexity. Specifically, it achieves a maximum observed speedup of up to 16 for complex instances (e.g., $n = 60$ and

$OS = 0.4$), whereas the speedup for simple instances remains modest due to the relative dominance of parallel overhead. Figures 5 and 6 further depict the observed speedup and parallel efficiency under varying thread counts. Several key findings can be drawn, as follows.

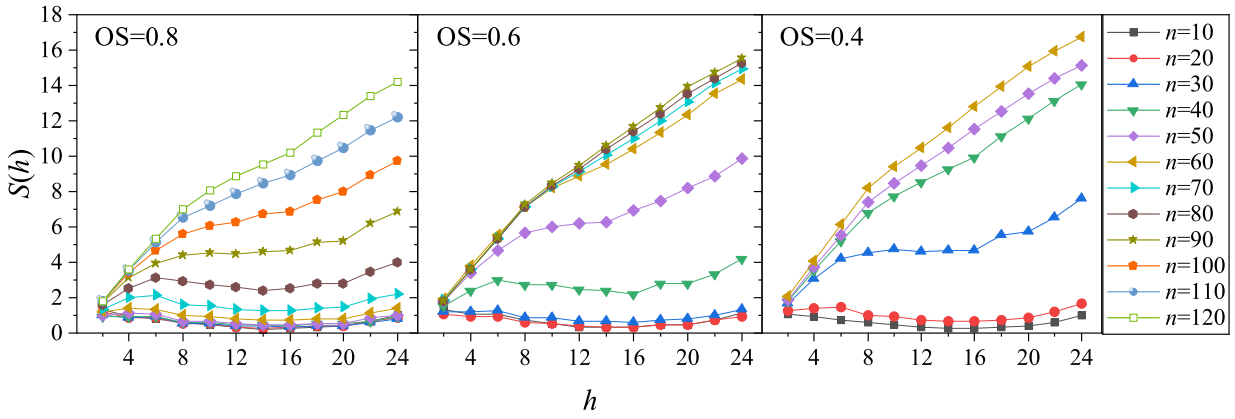


Fig. 5. Observed speedup of the parallelized SDP for different thread counts.

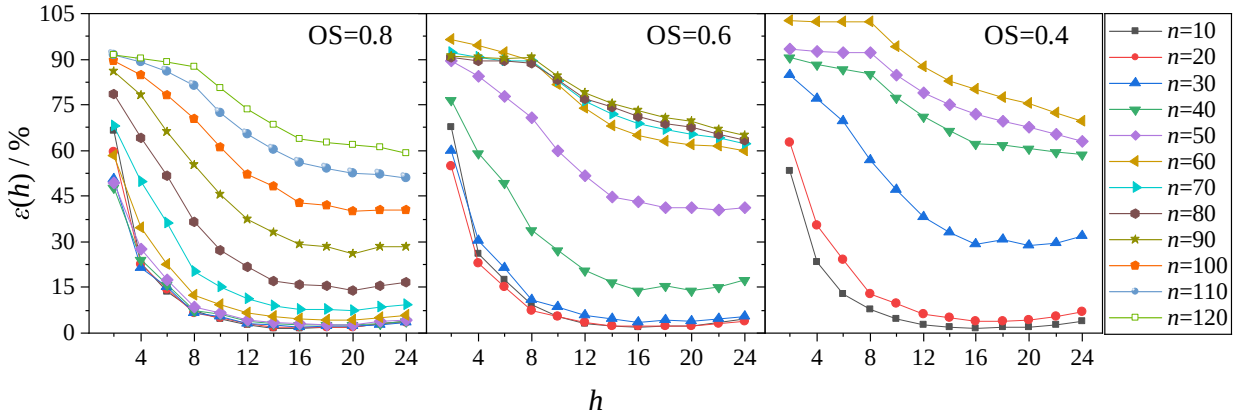


Fig. 6. Observed parallel efficiency of the parallelized SDP for different thread counts.

First, for complex instances, the parallelized SDP achieves an approximately linear speedup, as illustrated in Figure 5. Speedup grows rapidly with the first eight threads and then more gradually with the remaining 16, reflecting the processor's heterogeneous architecture: the initial threads are mapped to performance cores with higher computational throughput, whereas the additional threads run on efficiency cores. These trends are consistent with the asymptotic speedup model developed in Section 6, which predicts that, for sufficiently complex instances, the computational burden is dominated by the eNPV calculations and the serial fraction becomes negligible. As a result, the speedup approaches the theoretical upper bound given in Eq. (30). Moreover, Eq. (30) implies that an approximately linear speedup can only be observed when $\Delta T(h)$ remains constant or varies only marginally with thread count. As noted by Schryen [2020], existing parallel dynamic programming approaches typically exhibit sublinear speedups. In contrast, our parallel SDP is able to attain (approximately) linear speedup, at least over a substantial range of thread counts.

Second, as instance complexity decreases, the observed speedup declines sharply and becomes nonlinear with respect to the number of threads. This is explained by the same asymptotic model and the three primary factors analyzed in Section 6. In small instances, state enumeration and synchronization delays constitutes a substantial portion of total computation, and load balancing among threads is also less efficient. Together, these factors result in a relatively large serial fraction, which directly limits the achievable speedup.

Third, the observed parallel efficiency, shown in Figure 6, declines with increasing thread count across all settings, which is a common observation in parallel computing. Since the marginal contribution of additional threads decreases while parallel overheads and shared-resource contention increase with the number of threads, the parallel efficiency $\epsilon(h)$ naturally declines as h grows, even when speedup continues to increase. For small instances, low efficiency is mainly caused by the high proportion of serial state enumeration and uneven load balancing. For large instances, in contrast, the overhead fraction becomes negligible. As a result, efficiency decreases more slowly. According to Eq. (30), the parallel efficiency under complex instances is $\epsilon(h) \approx 1/(1+\Delta T(h))$. Figure 6 shows that in complex instances such as $n = 60$ and $OS = 0.4$, the parallel efficiency of the first eight threads remains nearly constant at close to 100%, implying that $\Delta T(h)$ stays close to 0. When threads are confined to homogeneous performance cores, contention for shared resources such as last-level cache and memory bandwidth remains symmetric and does not become a performance bottleneck. This enables the parallel SDP to sustain a near-ideal linear speedup. However, once the thread count exceeds the number of performance cores (i.e., $h > 8$), efficiency-core threads are engaged, creating asymmetric access patterns and execution speeds. This asymmetry amplifies contention for shared resources and increases synchronization delays. As a result, parallel efficiency declines markedly beyond this point, corresponding to a positive and growing $\Delta T(h)$ as h surpasses the number of dedicated performance cores.

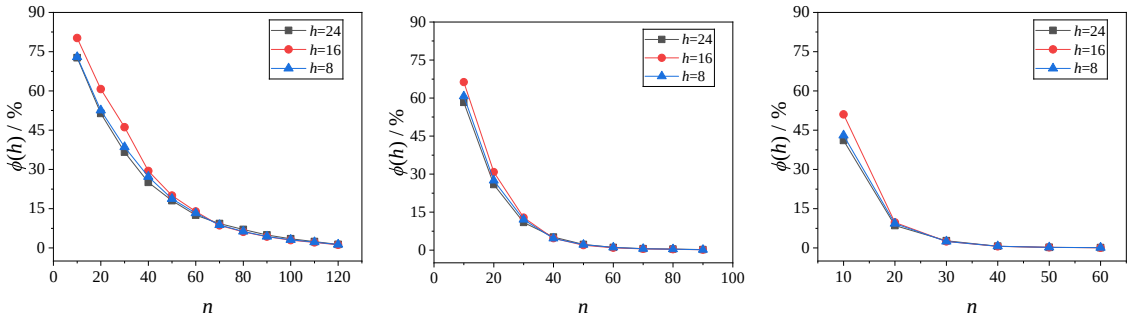
7.4. Observed overheads of parallelization

Although the proposed parallel SDP has been shown in the previous section to be effective and to exhibit favorable scalability, a non-negligible gap remains between the observed speedup and the ideal upper bound, given by the number of threads, especially for simpler instances. To better understand the sources of this deviation and to assess the validity of the theoretical framework developed in Section 6, we therefore conduct a detailed analysis of the parallel overheads inherent in the proposed approach.

All instances from the first group of experiments were re-solved using the parallelized SDP, during which we recorded, at each stage, the state enumeration time and the per-thread eNPV evaluation time. In addition to varying instance complexity, we also examined the effect of the number of threads on the three principal sources of parallel overhead. To this end, three thread configurations were considered, namely $h = 8$, 16, and 24. The resulting values of the three overhead metrics, observed across different instance complexities and thread counts, are reported in Figures 7, 8 and 9, respectively. Several important observations emerge from these results.

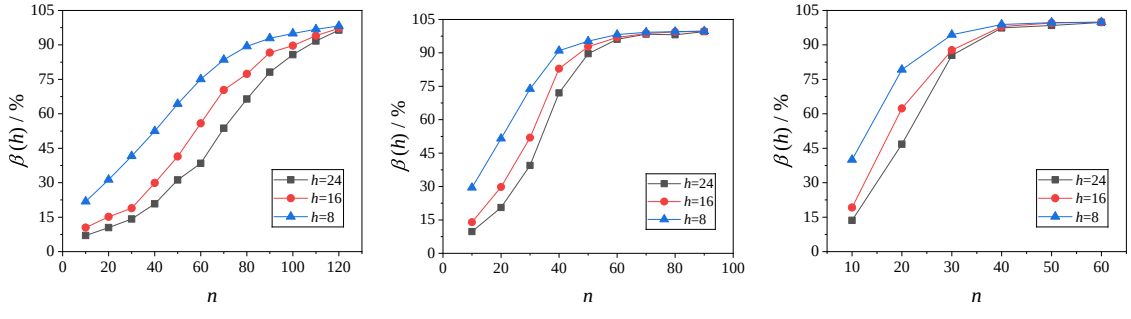
First, state enumeration delay becomes asymptotically negligible as instance complexity increases, regardless of the number of threads used. This is consistent with our analytical conclusion in Theorem 1. However, for simple instances, state enumeration accounts for more than 70% of the total runtime, thus severely limiting the speedup of parallel SDP. Furthermore, Figure 7 shows that the number of threads has almost no effect on $\phi(h)$. This indicates that the parallel overhead of increasing the number of threads affects both the state enumeration time and the optimal eNPV calculation time. In other words, the numerator and denominator of Eq. (10) increase proportionally, rendering $\phi(h)$ unaffected by h.

Second, under all thread settings, the load-balancing efficiency increases from a low value to around 100% as instance complexity increases. This validates Theorem 2. Furthermore, Figure 8 shows that for a simple instance, $\beta(h)$ decreases as h increases. The computational results are consistent with our analysis at the end of Section 6.2. In particular, our results empirically confirm that fixed parallelization overheads and inter-thread delays become dominant when the per-thread workload is limited. These findings suggest that load imbalance among threads substantially constrains the achievable speedup in small-scale instances, but its impact diminishes in large-scale instances.



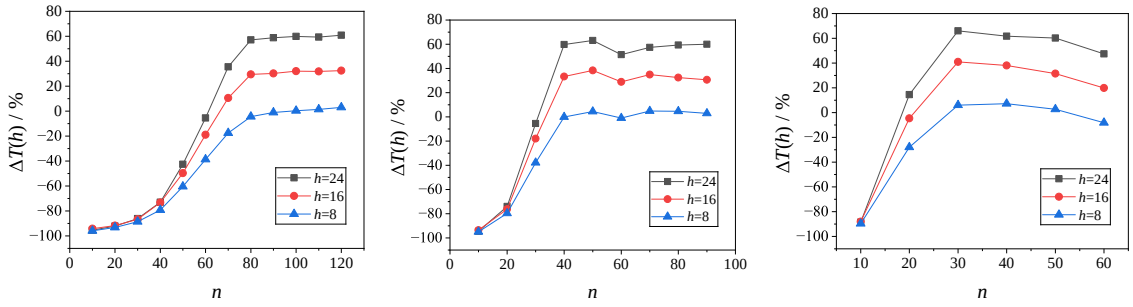
(a) $OS = 0.8$. (b) $OS = 0.6$. (c) $OS = 0.4$.

Fig. 7. Proportion of state enumeration of the parallelized SDP.



(a) $OS = 0.8$. (b) $OS = 0.6$. (c) $OS = 0.4$.

Fig. 8. Load-balancing efficiency of the parallelized SDP.



(a) $OS = 0.8$. (b) $OS = 0.6$. (c) $OS = 0.4$.

Fig. 9. Incremental computation time of the parallelized SDP.

Third, the incremental percentage of computation time grows with instance complexity, and this effect becomes more pronounced as the number of threads increases. As presented in Figure 9, when the number of threads is 8, 16, and 24, $\Delta T(h)$ increases from approximately -100% to about 0%, 40%, and 60%, respectively. For a given instance, the total amount of computation in the serial and parallel modes is theoretically equivalent; thus, their total computing times should be comparable. However, as analyzed in Section 6.3, when more threads are added, the competition for shared computing resources among threads becomes more intense, resulting in an increase in $\Delta T(h)$. Interestingly, the negative increment in simple instances appears counterintuitive but can be attributed to instruction-level overlap and concurrent cache utilization. In particular, when the instance is small, the two adjacent state subspaces can fully reside in the cache, resulting in highly overlapping memory accesses among threads. A data block loaded by one thread remains in the shared cache and can subsequently be reused by others, thereby reducing the effective average latency per memory access compared with sequential execution. Consequently, $\Delta T(h)$ can become strongly negative, approaching its theoretical lower bound of -100%. In addition, another interesting result is that $\Delta T(8)$ for the complex instance stabilizes around 0, which is consistent with the previous result, inferred based on parallel efficiency. The main reason is that the resource contention of threads running on the performance cores remains symmetric and does not dominate performance. Substituting the convergence value of $\Delta T(24)$ (i.e., 60%) into Eq. (30) yields an asymptotic speedup of about 15, which closely matches the observed speedup values reported in Table 3. This consistency provides strong evidence supporting the correctness of our theoretical asymptotic analysis.

Overall, these empirical measurements demonstrate that the deviations from ideal linear scalability are well explained by the combined effects of enumeration delay, workload imbalance, and resource contention. In particular, the observed convergence patterns of the three indicators are fully consistent with the asymptotic properties established in Section 6. Therefore, the computational evidence not only validates the refined overhead-aware speedup model, but also clarifies the dominant performance-limiting mechanisms governing the parallelized SDP in practice.

7.5. Results for more complex instances

In the final experiment group, the parallelized SDP with 24 threads is applied to solve more complex instances for which the serial SDP becomes computationally intractable due to excessive runtime. Solving these instances not only demonstrates the practical efficiency of the proposed approach, but also provides optimal benchmark solutions that can serve as valuable references for evaluating approximate algorithms. The detailed results are summarized in Table 4.

Table 4
Results of more complex instances obtained using the parallelized SDP with 24 threads.

n	OS	Solved	T_{24}		#States	
			avg	max	avg	max
70	0.4	30	3980.94	41809.80	1.03×10^8	5.03×10^8
80	0.4	30	70994.14	463977.19	7.09×10^8	1.77×10^9
100	0.6	30	424.99	1825.74	3.13×10^7	6.05×10^7
110	0.6	30	3031.66	35065.50	1.11×10^8	4.57×10^8
120	0.6	30	20828.23	74519.60	3.76×10^8	7.73×10^8

Creemers [2018b] reported that solving all instances with $n = 70$ and $OS = 0.4$ using the serial SDP required approximately 46 days. In contrast, the proposed parallelized SDP reduced the total computation time to about 1.4 days. The average and maximum computation times were improved by factors of 33.0 and 17.95, respectively, compared with those reported in Creemers [2018b]. It should be noted that this improvement also reflects differences in hardware performance. Given the prohibitive runtime of the serial SDP, we solved only the most complex instance to obtain a more accurate measure of actual speedup. The serial SDP required 186.63 hours for this instance, which is 1.12 times faster than the corresponding result in Creemers [2018b]. Based on this result, the actual speedup of the parallelized SDP for the most complex instance is 16.07. Assuming that the same 1.12 performance ratio applies to the remaining instances, the average actual speedup for $n = 70$ and $OS = 0.4$ is about 29.46. Considering that the state and action spaces of other instances are smaller, the lower degree of memory access contention would make the hardware advantage more pronounced, leading to an actual speedup below the theoretical upper bound of 24. Therefore, the overall average actual speedup is reasonably expected to fall between 16 and 24. These results indicate that the proposed parallelized SDP exhibits strong scalability and remains highly efficient even for more complex instances.

To further analyze the parallelization overheads for $n = 70$ and $OS = 0.4$, the 24-thread parallelized SDP was executed again, and the state enumeration and eNPV computation times of each thread were recorded at every stage. The results show that the proportion of state enumeration is only 0.03% (lower than 0.09% for $n = 60$ and $OS = 0.4$), while the load-balancing efficiency reaches 99.94% (higher than 99.79% for $n = 60$ and $OS = 0.4$). These findings not only confirm the validity of Theorems 1 and 2, but also demonstrate that the scalability of the parallelized SDP primarily arises from a stable computation time increment. Specifically, the most complex instance exhibits $\Delta T(24) = 54.38\%$, which is close to the convergence value of 60% shown in Figure 9. This observation suggests that computational intensity (e.g., the ratio of arithmetic time to memory access time) remains stable as instance complexity increases.

Moreover, the high computational efficiency of the proposed parallelized SDP allows us to solve, for the first time, all instances with $OS = 0.6$, as well as all instances with $n = 80$ and $OS = 0.4$. Despite parallelization, solving the most challenging instances (e.g., the case with $n = 120$ and $OS = 0.6$, and that with $n = 80$ and $OS = 0.4$) remains highly time-consuming. Since the exact speedup for these instances is difficult to measure directly, we estimate it to be 16 based on the results for $n = 70$ and $OS = 0.4$. Under this assumption, solving a single instance with $n = 120$ and $OS = 0.6$ using the serial SDP would take approximately 3.9 days, whereas the parallelized SDP completes the computation in about 5.8 hours. Similarly, using serial SDP and parallelized SDP to solve a single instance with $n = 80$ and $OS = 0.4$ can be estimated to take around 13.1 days and 19.7 hours, respectively.

8. Conclusions

This study developed a multithreaded parallel SDP for project scheduling problems with exponentially distributed activity durations. By employing a producer-consumer design, the proposed method effectively decouples state enumeration and eNPV evaluation, thereby exploiting fine-grained parallelism inherent in the SDP recursion. Theoretical analysis and extensive computational experiments jointly reveal several important findings. First, as instance complexity increases, the proportion of time devoted to state enumeration decreases rapidly, while the load-balancing efficiency approaches 100%, confirming the asymptotic properties proved in the theoretical analysis. Second, the experimental results indicate that the computational intensity remains stable in complex instances, demonstrating the excellent scalability of the proposed parallel SDP. Third, the approach achieves substantial acceleration over the serial SDP and enables the optimal solution of instances that were previously computationally intractable, thus providing reliable benchmark solutions for evaluating approximate algorithms.

Overall, the results presented in this paper demonstrate that parallel SDP constitutes a powerful and scalable framework for solving complex stochastic scheduling problems. Nevertheless, the main limitation of the parallel SDP proposed in this paper is that it is only suitable for running on a single computer. This demonstrates that achieving greater speedups comes at a significant cost, as modern CPUs tend to have a relatively small number of cores. For example, the Intel i9-13900K processor used in this study has only 24 physical cores. While some top-tier server CPUs boast hundreds of physical cores (e.g., the Intel Xeon 6980P with 128 cores), they are very expensive.

Future research can proceed in several directions. One promising avenue is to extend the proposed framework to more general optimization problems on Markovian networks, such as the stochastic resource-constrained project scheduling problem with makespan minimization [Creemers, 2015, 2019], interdiction games on Markovian project networks [Gutin et al., 2015], project makespan distribution estimation [Burgelman and Vanhoucke, 2019], and dynamic project expediting [Bertazzi et al., 2024]. Similar to the problems studied in this paper, these problems can also be formulated as MDPs and solved using SDP. The parallel SDP framework proposed in this paper is a general

method that does not impose any restrictions on the structural characteristics of the problem, and therefore, it can be easily extended to other problems solved using SDP. However, it should be noted that the corresponding theoretical analysis results may not necessarily apply to new problems, and the actual achievable speedup needs to be verified empirically. Another promising direction is to integrate the parallel SDP framework with approximate methods. For example, it may be possible to construct problem-specific dominance rules or heuristic methods, thereby effectively reducing the size of the MDP state space and action space. Combining this with the high computational efficiency of parallel SDP may pave the way toward solving more complex problem instances.

Acknowledgements

This work was supported by the National Natural Science Foundation of China [Grant No. 52575565, 71972053 and 51505090]. We thank the China Scholarship Council (CSC) for supporting the first author's visit to KU Leuven. The collaboration established during this visit played a key role in achieving the results presented in this paper.

References

U. Aldasoro, L. F. Escudero, M. Merino, J. F. Monge, and G. Pérez. On parallelization of a stochastic dynamic programming algorithm for solving large-scale mixed 0–1 problems under uncertainty. *Top*, 23: 703–742, 2015. doi: https://doi.org/10.1007/s11750-014-0359-3.

G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery. doi: https://doi.org/10.1145/1465482.1465560.

S. M. Baroum and J. H. Patterson. The development of cash flow weight procedures for maximizing the net present value of a project. *Journal of Operations Management*, 14(3):209–227, 1996. doi: https://doi.org/10.1016/0272-6963(96)00005-8.

L. Bertazzi, R. Mogre, and N. Trichakis. Dynamic project expediting: A stochastic shortest-path approach. *Management Science*, 70(6):3748–3768, 2024. doi: https://doi.org/10.1287/mnsc.2023.4876.

S. Boucheron, G. Lugosi, and P. Massart. *Concentration Inequalities: A Nonasymptotic Theory of Independence*. Oxford University Press, 2013. doi: https://doi.org/10.1093/acprof:oso/9780199535255.001.0001.

M. E. Bruni and Ö. Hazır. A risk-averse distributionally robust project scheduling model to address payment delays. *European Journal of Operational Research*, 318(2):398–407, 2024. doi: https://doi.org/10.1016/j.ejor.2024.05.037.

J. Burgelman and M. Vanhoucke. Computing project makespan distributions: Markovian PERT networks revisited. *Computers & Operations Research*, 103:123–133, 2019. doi: https://doi.org/10.1016/j.cor.2018.10.017.

A. H. Buss and M. J. Rosenblatt. Activity delay in stochastic project networks. *Operations Research*, 45 (1):126–139, 1997. doi: https://doi.org/10.1287/opre.45.1.126.

W.-N. Chen and J. Zhang. Scheduling multi-mode projects under uncertainty to optimize cash flows: A Monte Carlo ant colony system approach. *Journal of Computer Science and Technology*, 27(5):950–965, 2012. doi: https://doi.org/10.1007/s11390-012-1276-2.

S. Chung, F. Hanson, and H. Xu. Parallel stochastic dynamic programming: Finite element methods. *Linear Algebra and its Applications*, 172:197–218, 1992. doi: https://doi.org/10.1016/0024-3795(92)90026-7.

S. Creemers. Minimizing the expected makespan of a project with stochastic activity durations under resource constraints. *Journal of Scheduling*, 18:263–273, 2015. doi: http://dx.doi.org/10.1007/s10951-015-0421-5.

S. Creemers. Moments and distribution of the net present value of a serial project. *European Journal of Operational Research*, 267(3):835–848, 2018a. doi: https://doi.org/10.1016/j.ejor.2017.12.039.

S. Creemers. Maximizing the expected net present value of a project with phase-type distributed activity durations: An efficient globally optimal solution procedure. *European Journal of Operational Research*, 267(1):16–22, 2018b. doi: https://doi.org/10.1016/j.ejor.2024.03.029.

S. Creemers. The preemptive stochastic resource-constrained project scheduling problem. *European Journal of Operational Research*, 277(1):238–247, 2019. doi: http://dx.doi.org/10.1016/j.ejor.2019.02.030.

S. Creemers, R. Leus, and M. Lambrecht. Scheduling Markovian PERT networks to maximize the net present value. *Operations Research Letters*, 38(1):51–56, 2010. doi: http://dx.doi.org/10.1016/j.orl.2009.10.006.

B. De Reyck and R. Leus. R&D project scheduling when activities may fail. *IIE Transactions*, 40(4):367–384, 2008. doi: https://doi.org/10.1080/07408170701413944.

E. Demeulemeester, M. Vanhoucke, and W. Herroelen. Rangen: A random network generator for activity-on-the-node networks. *Journal of scheduling*, 6:17–38, 2003. doi: https://doi.org/10.1023/A:1022283403119.

B. H. Dias, M. A. Tomim, A. L. M. Marcato, T. P. Ramos, R. B. S. Brandi, I. C. da Silva Junior, and J. A. Passos Filho. Parallel computing applied to the stochastic dynamic programming for long term operation planning of hydrothermal power systems. *European Journal of Operational Research*, 229(1):212–222, 2013. doi: https://doi.org/10.1016/j.ejor.2013.02.024.

E. W. Dijkstra. *Cooperating Sequential Processes*, pages 65–138. Springer New York, New York, NY, 2002. doi: https://doi.org/10.1007/978-1-4757-3472-0_2.

S. E. Elmaghraby and W. Herroelen. The scheduling of activities to maximize the net present value of projects. *European Journal of Operational Research*, 49(1):35–49, 1990. doi: https://doi.org/10.1016/0377-2217(90)90118-U.

E. A. Eschenbach, C. A. Shoemaker, and H. M. Caffey. Parallel algorithms for stochastic dynamic programming with continuous state and control variables. *ORSA Journal on Computing*, 7(4):386–401, 1995. doi: https://doi.org/10.1287/ijoc.7.4.386.

R. Etgar. Scheduling project activities to maximize the net present value the case of linear time-dependent cash flows. *International Journal of Production Research*, 37(2):329–339, 1999. doi: https://doi.org/10.1080/002075499191797.

R. Etgar, A. Shtub, and L. J. LeBlanc. Scheduling projects to maximize net present value — the case of time-dependent, contingent cash flows. *European Journal of Operational Research*, 96(1):90–96, 1997. doi: https://doi.org/10.1016/0377-2217(95)00382-7.

C. Füllner and S. Rebennack. Stochastic dual dynamic programming and its variants: A review. *SIAM Review*, 67(3):415–539, 2025. doi: https://doi.org/10.1137/23M1575093.

S. K. Gupta, J. Kyparisis, and C.-M. Ip. Note—project selection and sequencing to maximize net present value of the total return. *Management Science*, 38(5):751–752, 1992. doi: https://doi.org/10.1287/mnsc.38.5.751.

J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988. doi: https://doi.org/10.1145/42411.42415.

E. Gutin, D. Kuhn, and W. Wiesemann. Interdiction games on Markovian PERT networks. *Management Science*, 61(5):999–1017, 2015. doi: https://pubsonline.informs.org/doi/10.1287/mnsc.2014.1973.

F. Hanson. Computational stochastic dynamic programming on a vector multiprocessor. *IEEE Transactions on Automatic Control*, 36(4):507–511, 1991. doi: https://doi.org/10.1109/9.75113.

S. Hartmann and D. Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 207(1):1–14, 2010. doi: https://doi.org/10.1016/j.ejor.2009.11.005.

S. Hartmann and D. Briskorn. An updated survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 297(1):1–14, 2022. doi: https://doi.org/10.1016/j.ejor.2021.05.004.

Ö. Hazır and G. Ulusoy. A classification and review of approaches and methods for modeling uncertainty in projects. *International Journal of Production Economics*, 223:107522, 2020. doi: https://doi.org/10.1016/j.ijpe.2019.107522.

B. Hermans and R. Leus. Scheduling Markovian PERT networks to maximize the net present value: New results. *Operations Research Letters*, 46(2):240–244, 2018. doi: https://doi.org/10.1016/j.orl.2018.01.010.

W. Herroelen and R. Leus. Project scheduling under uncertainty: Survey and research potentials. *European Journal of Operational Research*, 165(2):289–306, 2005. doi: https://doi.org/10.1016/j.ejor.2004.04.002.

V. G. Kulkarni and V. G. Adlakha. Markov and Markov-regenerative PERT networks. *Operations Research*, 34(5):769–781, 1986. doi: https://doi.org/10.1287/opre.34.5.769.

P. Lamas, M. Goycoolea, B. Pagnoncelli, and A. Newman. A target-time-windows technique for project scheduling under uncertainty. *European Journal of Operational Research*, 314(2):792–806, 2024. doi: https://doi.org/10.1016/j.ejor.2023.10.027.

P. Leyman and M. Vanhoucke. Payment models and net present value optimization for resource-constrained project scheduling. *Computers & Industrial Engineering*, 91:139–153, 2016. doi: https://doi.org/10.1016/j.cie.2015.11.008.

P. Leyman and M. Vanhoucke. Capital- and resource-constrained project scheduling with net present value optimization. *European Journal of Operational Research*, 256(3):757–776, 2017. doi: https://doi.org/10.1016/j.ejor.2016.07.019.

Y. Liang, N. Cui, T. Wang, and E. Demeulemeester. Robust resource-constrained max-npv project scheduling with stochastic activity duration. *OR Spectrum*, 41(1):219–254, 2019. doi: https://doi.org/10.1007/s00291-018-0533-3.

M. Peymankar, M. Davari, and M. Ranjbar. Maximizing the expected net present value in a project with uncertain cash flows. *European Journal of Operational Research*, 294(2):442–452, 2021. doi: https://doi.org/10.1016/j.ejor.2021.01.039.

T. Popoviciu. Sur les équations algébriques ayant toutes leurs racines réelles. *Mathematica*, 9:129–145, 1935. doi: https://scholar.google.com/scholar?cluster=16917578814869468986.

F. Rezaei, A. A. Najafi, and R. Ramezanian. Mean-conditional value at risk model for the stochastic project scheduling problem. *Computers & Industrial Engineering*, 142:106356, 2020. doi: https://doi.org/10.1016/j.cie.2020.106356.

F. Rezaei, A. A. Najafi, R. Ramezanian, and E. Demeulemeester. Simulation-based priority rules for the stochastic resource-constrained net present value and risk problem. *Computers & Industrial Engineering*, 160:107607, 2021. doi: https://doi.org/10.1016/j.cie.2021.107607.

F. Rezaei, A. A. Najafi, E. Demeulemeester, and R. Ramezanian. A stochastic bi-objective project scheduling model under failure of activities. *Annals of Operations Research*, 38(1):453–476, 2024. doi: https://doi.org/10.1007/s10479-023-05600-2.

S. Rostami, S. Creemers, and R. Leus. Maximizing the net present value of a project under uncertainty: Activity delays and dynamic policies. *European Journal of Operational Research*, 317(1):16–24, 2024. doi: https://doi.org/10.1016/j.ejor.2024.03.029.

C. Roucairol. Parallel processing for difficult combinatorial optimization problems. *European Journal of Operational Research*, 92(3):573–590, 1996. doi: https://doi.org/10.1016/0377-2217(96)00009-4.

A. Russell. Cash flows in networks. *Management Science*, 16(5):357–373, 1970. doi: https://doi.org/10.1287/mnsc.16.5.357.

G. Schryen. Parallel computational optimization in operations research: A new integrative framework, literature review and research directions. *European Journal of Operational Research*, 287(1):1–18, 2020. doi: https://doi.org/10.1016/j.ejor.2019.11.033.

G. Schryen. Speedup and efficiency of computational parallelization: A unifying approach and asymptotic analysis. *Journal of Parallel and Distributed Computing*, 187:104835, 2024. doi: https://doi.org/10.1016/j.jpdc.2023.104835.

T. Selle and J. Zimmermann. A bidirectional heuristic for maximizing the net present value of large-scale projects subject to limited resources. *Naval Research Logistics (NRL)*, 50(2):130–148, 2003. doi: https://doi.org/10.1002/nav.10052.

R. J. Serfling. Probability inequalities for the sum in sampling without replacement. *The Annals of Statistics*, 2(1):39–48, 1974. doi: https://doi.org/10.1214/aos/1176342611.

D. E. Smith-Daniels and N. J. Aquilano. Using a late-start resource-constrained project schedule to improve project net present value. *Decision Sciences*, 18(4):617–630, 1987. doi: https://doi.org/10.1111/j.1540-5915.1987.tb01550.x.

M. J. Sobel, J. G. Szmerekovsky, and V. Tilson. Scheduling projects with stochastic activity duration to maximize expected net present value. *European Journal of Operational Research*, 198(3):697–705, 2009. doi: https://doi.org/10.1016/j.ejor.2008.10.004.

X.-H. Sun and Y. Chen. Reevaluating Amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010. doi: https://doi.org/10.1016/j.jpdc.2009.05.002.

C. Sung and S. Lim. A project activity scheduling problem with net present value measure. *International Journal of Production Economics*, 37(2):177–187, 1994. doi: https://doi.org/10.1016/0925-5273(94)90169-4.

M. Vanhoucke. A scatter search heuristic for maximising the net present value of a resource-constrained project with fixed activity cash flows. *International Journal of Production Research*, 48(7):1983–2001, 2010. doi: https://doi.org/10.1080/00207540802010781.

M. Vanhoucke, E. Demeulemeester, and W. Herroelen. Maximizing the net present value of a project with linear time-dependent cash flows. *International Journal of Production Research*, 39(14):3159–3181, 2001a. doi: https://doi.org/10.1080/00207540110056919.

M. Vanhoucke, E. Demeulemeester, and W. Herroelen. On maximizing the net present value of a project under renewable resource constraints. *Management Science*, 47(8):1113–1121, 2001b. doi: https://doi.org/10.1287/mnsc.47.8.1113.10226.

X. Wang, R. Leus, S. Creemers, Q. Chen, and N. Mao. A CTMDP-based exact method for RCPSP with uncertain activity durations and rework. In *Operations Research Proceedings 2017: Selected Papers of the Annual International Conference of the German Operations Research Society (GOR), Freie Universiät Berlin, Germany, September 6-8, 2017*, pages 559–565. Springer, 2018. doi: https://doi.org/10.1007/978-3-319-89920-6_74.

W. Wiesemann and D. Kuhn. The stochastic time-constrained net present value problem. In C. Schwindt and J. Zimmermann, editors, *Handbook on Project Management and Scheduling Vol. 2*, pages 753–780. Springer International Publishing, Cham, 2015. doi: doi.org/10.1007/978-3-319-05915-0_5.

W. Wiesemann, D. Kuhn, and B. Rustem. Maximizing the net present value of a project under uncertainty. *European Journal of Operational Research*, 202(2):356–367, 2010. doi: https://doi.org/10.1016/j.ejor.2009.05.045.

K. K. Yang, F. Talbot, and J. H. Patterson. Scheduling a project to maximize its net present value: An integer programming approach. *European Journal of Operational Research*, 64(2):188–198, 1993. doi: https://doi.org/10.1016/0377-2217(93)90176-N.

K. K. Yang, L. C. Tay, and C. C. Sum. A comparison of stochastic scheduling rules for maximizing project net present value. *European Journal of Operational Research*, 85(2):327–339, 1995. doi: https://doi.org/10.1016/0377-2217(94)00039-F.

W. Zhao, N. G. Hall, and Z. Liu. Project evaluation and selection with task failures. *Production and Operations Management*, 29(2):428–446, 2020. doi: https://doi.org/10.1111/poms.13107.

W. Zheng, Z. He, N. Wang, and T. Jia. Proactive and reactive resource-constrained max-npv project scheduling with random activity duration. *Journal of the Operational Research Society*, 69(1):115–126, 2018. doi: https://doi.org/10.1057/s41274-017-0198-3.