




RYO KUROIWA¹, YUJI SHINANO², J. CHRISTOPHER
BECK³

Massively Parallel and Distributed Solvers for Domain-Independent Dynamic Programming

¹  0000-0002-3753-1644

²  0000-0002-2902-882X

³  0000-0002-4656-8908

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30 84185-0
Telefax: +49 30 84185-125

E-mail: bibliothek@zib.de
URL: <http://www.zib.de>

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

Massively Parallel and Distributed Solvers for Domain-Independent Dynamic Programming

Ryo Kuroiwa

Principles of Informatics Division, National Institute of Informatics

Department of Advanced Studies, The Graduate University of Advanced Studies, SOKENDAI

kuroiwa@nii.ac.jp

Yuji Shinano

Zuse Institute of Berlin

shinano@zib.de

J. Christopher Beck

Department of Mechanical and Industrial Engineering, University of Toronto

jcb@mie.utoronto.ca

Abstract

In this paper, we develop distributed and parallel general-purpose solvers for combinatorial optimization through the framework of domain-independent dynamic programming (DIDP), a model-based paradigm based on dynamic programming. In particular, we parallelize heuristic state space search algorithms to develop such solvers. Benefiting from the general-purpose nature of DIDP, we apply our solvers to four problem classes: the traveling salesperson problem with time windows (TSPTW), the type1 simple assembly line balancing problem (SALBP-1), the one-to-one multi-commodity pickup and delivery traveling salesperson problem (m-PDTSP), and the type2 assembly line balancing problem with sequence-dependent setup times (SUALBP-2). We demonstrate the scalability of our solvers using up to 192 TB of RAM and 49,152 CPU cores. Using the developed solvers, we close 14 open instances of TSPTW, 49 of m-PDTSP, and 152 of SUALBP-2.

Keywords: Dynamic Programming, Combinatorial Optimization, Massively Parallel and Distributed Algorithms

Funding: Ryo Kuroiwa was supported by JSPS KAKENHI grant number JP25K24378. Part of the work for this article has been conducted in the Research Campus MODAL funded by the German Federal Ministry of Education and Research (BMBF) (fund numbers 05M14ZAM, 05M20ZBM, 05M2025).

1 Introduction

Dynamic Programming (DP) (Bellman 1957) is a problem-solving methodology used in multiple fields of computer science, including combinatorial optimization. In our recent work, we have developed domain-independent dynamic programming (DIDP), a model-based DP paradigm designed for combinatorial optimization (Kuroiwa and Beck 2026). In DIDP, a problem is formulated as a declarative DP model using a

modeling language and then solved by a general-purpose solver. The existing DIDP solvers are based on heuristic state-space search algorithms (Edelkamp et al. 2010), such as A* (Hart et al. 1968), which solve a problem by finding a path in an implicitly defined graph. Empirically, the DIDP solvers have achieved better performance than existing general-purpose solvers, including mixed-integer programming (MIP), across multiple problem classes in single- and multi-thread environments (Kuroiwa and Beck 2026, 2024).

In this paper, we develop massively parallel DIDP solvers for a distributed environment in which multiple machines are connected via a network. We apply the solvers to four problem classes, for which DIDP achieves state-of-the-art performance in single- and multi-thread environments in previous work (Kuroiwa and Beck 2026, 2024, Zhang and Beck 2025): the traveling salesperson problem with time windows (TSPTW) (Savelsbergh 1985), the type1 simple assembly line balancing problem (SALBP-1) (Salveson 1955), the multi-commodity pickup-and-delivery traveling salesperson problem (m-PDTSP) (Hernández-Pérez and Salazar-González 2009), and the type 2 assembly line balancing problem with sequence-dependent setup times (SUALBP-2) (Andrés et al. 2008). While distributed parallelization of existing DIDP solvers does not scale with thousands of CPU cores, our new solver, hybrid A* cyclic (HAC), achieves speedup using up to 49,152 CPU cores across 512 machines. Since HAC quickly runs out of memory even in a large-scale distributed environment, we also develop more memory-efficient parallel solvers that focus on proving optimality rather than finding better solutions, thereby solving seven instances that parallel HAC cannot solve. In total, we solve 14 open instances of TSPTW, 49 of m-PDTSP, and 152 of SUALBP-2. Taking advantage of the anytime nature of parallel HAC, we also improve the best-known solution costs in 116 and the best-known solution cost bound in 65 open instances of SUALBP-2.

In what follows, we first review related work and introduce DIDP. Then, we present an algorithmic framework for the distributed parallelization of DIDP solvers. We empirically show that distributed parallelization of existing DIDP solvers is not scalable, using TSPTW and SALBP-1. Motivated by this result, we propose HAC and evaluate its parallelization using all of the four problem classes. We also develop the memory-efficient parallel DIDP solvers and solve more problem instances. Finally, we summarize open instances solved in each problem class and conclude the paper.

2 Related Work

One prominent approach to developing general-purpose parallel and distributed optimization solvers is to use parallel branch-and-bound for MIP (Eckstein 1994, Eckstein et al. 2001, Shinano et al. 2016). In particular, Shinano et al. (2016) used up to 80,000 CPU cores, demonstrating scalability and closing multiple open instances. While parallel branch-and-bound achieved such success in distributed parallelization, we focus on a different approach, heuristic state space search, which has a complementary strength: DIDP solvers that use heuristic state space search outperform a commercial MIP solver that uses branch-and-bound in multiple problem classes in single- and multi-thread environments (Kuroiwa and Beck 2026, 2024).

Distributed parallelization of heuristic state space search has been studied, mainly focusing on A*. Early work on distributed parallel A* investigated two approaches: dynamic work distribution (Kumar et al. 1988, Mahapatra and Dutt 1993, Dutt and Mahapatra 1994) and hash-based, static work distribution (Evetts et al.

1995). Romein et al. (2002) combined asynchronous message passing and the hash-based work distribution method to parallelize iterative deepening A* (IDA*), a linear-space variant of A*. Hash-distributed A* (HDA*), a state-of-the-art distributed parallelization of A*, also uses this combination (Kishimoto et al. 2013). In this paper, we generalize HDA* to parallelize anytime variants of A*. In previous work, greedy best-first search, a variant of A* that aims to quickly find any path, not necessarily a shortest path, was parallelized based on HDA* (Kuroiwa and Fukunaga 2019). However, distributed parallelization of anytime variants of A* has not been studied to our knowledge. In addition, we use up to 49,152 CPU cores and 192 TB RAM, while existing work used at most 2,400 cores and 10.5 TB with HDA* (Kishimoto et al. 2013).

Early work on distributed parallel A* with dynamic work distribution (Kumar et al. 1988, Dutt and Mahapatra 1994, Mahapatra and Dutt 1993) used the traveling salesperson problem (Dantzig et al. 1954) and the vertex cover problem Karp (1972) as benchmarks. In contrast, Evett et al. (1995) evaluated their method using the sliding-tile puzzle. Previous work on HDA* and its variants (Kishimoto et al. 2013, Kobayashi et al. 2011, Jinnai and Fukunaga 2017, Kuroiwa and Fukunaga 2019) used automated planning problems studied in the AI community (Ghallab et al. 2004) and multiple sequence alignment (Carrillo and Lipman 1988) in addition to the sliding-tile puzzle. Compared with such existing work, we consider four combinatorial optimization problems that have not been solved by distributed parallel heuristic state space search methods.

3 Background

We introduce the modeling formalism of DIDP and heuristic state space search framework for it.

3.1 Domain-Independent Dynamic Programming

In DIDP, a problem is described as a declarative DP model using Dynamic Programming Description Language (DyPDL) (Kuroiwa and Beck 2026). A model is defined by *state variables*, where a *state* is a complete value assignment to all state variables and *transitions* update the values of the state variables. A solution is a sequence of transitions that transforms a special state called the *target state* S^I into a *base state*, which satisfies particular conditions.

Each state variable v has a domain D_v , depending on its *type*, either *element*, *set*, or *numeric*. The domain of an element variable is the set of nonnegative integers ($D_v = \mathbb{Z}_0^+$), the domain of a set variable is the power set of nonnegative integers ($D_v = 2^{\mathbb{Z}_0^+}$), and the domain of a numeric variable is the set of rational numbers ($D_v = \mathbb{Q}$). We can consider conditions on states based on state variables, e.g., $v \geq 0$ where v is a numeric variable. A *base case* is a set of such conditions, and a state satisfying a base case is called a base state.

A transition $\tau \in \mathcal{T}$ transforms a state S into another state $S[[\tau]]$ by updating the state variables. A transition τ is *applicable* in a state S if its *preconditions*, conditions on state variables, are satisfied by S , and the set of applicable transitions in S is denoted by $\mathcal{T}(S)$. A user may define a set of *state constraints*, conditions that must be satisfied by each state. An S -solution is a sequence of applicable transitions $\sigma = \langle \sigma_1, \dots, \sigma_n \rangle$ that transforms a state S into a base state, i.e., $S^{i+1} = S^i[[\sigma_{i+1}]]$ for $i = 0, \dots, n - 1$ and

$\sigma_{i+1} \in \mathcal{T}(S^i)$ where $S^0 = S$, S^n is a base state, and each S^i satisfies state constraints. When S is the target state S^I , S -solution σ is called a solution for the model.

We focus on a subset of the DyPDL formalism, where the cost of a solution is defined by the weight w_τ of each transition τ , a function returning a rational number given a state, and the weight function w_B returning a rational number given a base state. Given an S -solution $\sigma = \langle \sigma_1, \dots, \sigma_n \rangle$, its cost is defined as $\sum_{i=0}^{n-1} w_{\sigma_{i+1}}(S^i) + w_B(S^n)$. An S -solution is optimal if its cost is better than or equal to the cost of any other S -solution. While we assume minimization in this paper, we may also consider a more general model in which the cost is defined by a different binary operator than $+$, or in which the objective is maximization. We focus on minimization with $+$ models, but our implementation in practice supports such generalized cost structures (Kuroiwa and Beck 2026).

Given a state S , we can represent the optimal S -solution cost by $V(S)$ defined by the following recursive equation, called a Bellman equation (Bellman 1957):

$$V(S) = \begin{cases} \infty & \text{if } S \text{ violates a state constraint} \\ w_B(S) & \text{if } S \text{ is a base state} \\ \min_{\tau \in \mathcal{T}(S)} w_\tau(S) + V(S) & \text{otherwise} \end{cases} \quad (1)$$

where we assume that the third line is ∞ if $\mathcal{T}(S) = \emptyset$.

In DyPDL, a user may define additional information, which is implied by the other parts of the model but can be exploited by a solver. For an element or a numeric variable, if its preference, either less or greater, is specified, then the variable is called a *resource variable*. Given two states S and S' , S is preferred to S' , denoted by $S' \preceq_a S$, if $S[v] = S'[v]$ for each non-resource variable v , $S[v] \leq S'[v]$ for each resource variable v where less is preferred, and $S[v] \geq S'[v]$ for each resource variable v where greater is preferred. In such a case, S dominates S' , i.e., for each S' -solution, there exists an S -solution with an equal or better cost. If S is not preferred to S' , we denote it by $S \not\preceq_a S'$. In addition to resource variables, a user may define a *dual bound function* η that returns a dual bound (lower bound for minimization and an upper bound for maximization) on the cost of an S -solution given a state S . The state dominance and dual bound functions can be represented by the following inequalities on V :

$$V(S) \leq V(S') \quad \text{if } S' \preceq_a S \quad (2)$$

$$V(S) \geq \eta(S). \quad (3)$$

3.2 Heuristic State Space Search for DyPDL

A DyPDL model can be solved by finding a path in a state transition graph, an implicit graph where nodes are states, edges are labeled with transitions, and an S -solution corresponds to a path from S to a base state (Kuroiwa and Beck 2026). This problem setting is known as state space search problems in the AI community (Russell and Norvig 2020), and heuristic state space search algorithms, such as A*, can be used (Hart et al. 1968, Edelkamp et al. 2010). Heuristic state space search uses a heuristic function h to estimate the distance from a state S to a base state. Following the previous work in DIDP (Kuroiwa and Beck 2026,

Algorithm 1 Heuristic State Space Search for DyPDL.

```
1:  $G, O \leftarrow \{S^I\}$ ,  $\sigma(S^I) \leftarrow$  an empty sequence,  $g(S^I) \leftarrow 0$ ,  $\bar{\sigma} \leftarrow \text{NULL}$ ,  $\bar{\gamma} \leftarrow \infty$ 
2: while  $O \neq \emptyset$  do
3:    $O \leftarrow O \setminus \{S\}$  where  $S \in O$  is selected depending on a concrete algorithm
4:   if  $S$  is a base state and  $g(S) + w_B(S) < \bar{\gamma}$  then
5:      $\bar{\sigma} \leftarrow \sigma(S)$ ,  $\bar{\gamma} \leftarrow g(S) + w_B(S)$ ,  $O \leftarrow \{S' \in O \mid g(S') + \eta(S') < \bar{\gamma}\}$ 
6:   else
7:     for  $\tau \in \mathcal{T}(S)$  such that  $S[[\tau]]$  satisfies the state constraints do
8:       if no  $S' \in G$  satisfies  $S[[\tau]] \preceq_a S' \wedge g(S') \leq g(S) + w_\tau(S)$  and  $g(S) + w_\tau(S) + \eta(S[[\tau]]) < \bar{\gamma}$  then
9:          $\sigma(S[[\tau]]) \leftarrow \sigma(S)$  extended with  $\tau$ ,  $g(S[[\tau]]) \leftarrow g(S) + w_\tau(S)$ 
10:         $G \leftarrow \{S' \in G \mid S' \not\preceq_a S[[\tau]] \vee g(S') < g(S[[\tau]])\} \cup \{S[[\tau]]\}$ 
11:         $O \leftarrow \{S' \in O \mid S' \not\preceq_a S[[\tau]] \vee g(S') < g(S[[\tau]])\} \cup \{S[[\tau]]\}$ 
12: return  $\bar{\sigma}$ 
```

2024), we use the dual bound function η defined in a model as the heuristic function h .

We show a generic pseudo-code for heuristic state space search in Algorithm 1. In each step, we expand a state, i.e., select a state and generate its successor states by applying transitions. We maintain G , the set of generated states, O , the open list containing states to expand, and $\sigma(S)$, the best path from the target state to a state S found so far, and its cost $g(S)$ called the g -value. We initialize G and O with the target state S^I , $\sigma(S^I)$ is an empty sequence, and $g(S^I) = 0$ (line 1). We also maintain the best known solution $\bar{\sigma}$, initialized with NULL, and its cost $\bar{\gamma}$, initialized with ∞ . Since $\bar{\gamma}$ is an upper bound on the optimal solution cost in minimization, we call it a *primal bound*.

In each step, we select a state S and remove it from O . The selection of the state depends on concrete algorithms. For example, A* selects a state minimizing $f(S) = g(S) + h(S) = g(S) + \eta(S)$. When the transition weight function w_τ satisfies $w_\tau(S) \geq 0$ for all S and τ and $w_B(S) = 0$, then the first found solution by A* is optimal (Hart et al. 1968), and no feasible solution is found before proving the optimality. By using a different criterion from minimizing $f(S)$, we can realize an anytime heuristic state space search algorithm, which tends to quickly find a first solution and continue to improve it until proving the optimality (Kuroiwa and Beck 2026).

If the selected state S is a base state, $\sigma(S)$ is a solution, so we compare its cost $g(S) + w_B(S)$ with the primal bound $\bar{\gamma}$ (line 4). If the new solution is better, we update the current best solution $\bar{\sigma}$ and $\bar{\gamma}$ and remove a state S' with $g(S') + \eta(S') \geq \bar{\gamma}$ from O' ; $g(S') + \eta(S')$ is a dual bound on the cost of a solution extending $\sigma(S')$ (line 5).

If S is not a base state, for each applicable transition τ , we generate a successor state $S[[\tau]]$ and check if it satisfies state constraints (line 7), it is not dominated by another state $S' \in G$ with an equal or better g -value, and that $g(S) + w_\tau(S) + \eta(S[[\tau]])$, the dual bound on the solution cost extending the current path to $S[[\tau]]$, does not exceed the primal bound $\bar{\gamma}$ (line 8). If $S[[\tau]]$ passes these checks, then we initialize or update $\sigma(S)$ and $g(S)$, remove states dominated by $S[[\tau]]$ with an equal or better g -value from G and O , and insert $S[[\tau]]$ into G and O (lines 9–10). When O becomes empty, the optimality of the current best solution is proved, or the problem is infeasible if no solution is found ($\bar{\sigma}$ is NULL), and the algorithm terminates.

Before O becomes empty, at the beginning of the while loop (line 2), $\min_{S \in O} g(S) + \eta(S)$ is the dual bound of the optimal solution cost. Therefore, during search, we can estimate how close the current solution

Algorithm 2 Hash-Distributed State Space Search for DyPDL.

```
1: for  $i = 1, \dots, k$  in parallel do
2:    $G, O \leftarrow \emptyset, \bar{\sigma} \leftarrow \text{NULL}, \bar{\gamma} \leftarrow \infty$ 
3:   if  $(\text{HASH}(S^I) \bmod k) + 1 = i$  then  $G, O \leftarrow \{S^I\}, \sigma(S^I) \leftarrow$  an empty sequence,  $g(S^I) \leftarrow 0$ 
4:   while termination criteria are not met do
5:     while an upper bound  $\bar{\gamma}'$  is received do  $\bar{\gamma} \leftarrow \min\{\bar{\gamma}, \bar{\gamma}'\}, O \leftarrow \{S' \in O \mid g(S') + \eta(S') < \bar{\gamma}\}$ 
6:     while  $S$  is received with  $\sigma^{\text{current}}, g_{\text{current}},$  and  $\eta(S)$  do
7:       if  $g_{\text{current}} + \eta(S) < \bar{\gamma}$  and no  $S' \in G$  satisfies  $S[[\tau]] \preceq_a S' \wedge g(S') \leq g_{\text{current}}$  then
8:          $\sigma(S) \leftarrow \sigma^{\text{current}}, g(S) \leftarrow g_{\text{current}}$ 
9:          $G \leftarrow \{S' \in G \mid S' \not\preceq_a S \vee g(S') < g(S[[\tau]])\} \cup \{S\}$ 
10:         $O \leftarrow \{S' \in O \mid S' \not\preceq_a S \vee g(S') < g(S[[\tau]])\} \cup \{S\}$ 
11:         $O \leftarrow O \setminus \{S\}$  where  $S$  is selected depending on a concrete algorithm
12:        for  $\tau \in \mathcal{T}(S)$  such that  $S[[\tau]]$  satisfies the state constraints do
13:          if  $S[[\tau]]$  is a base state and  $g(S) + w_\tau(S) + w_B(S[[\tau]]) < \bar{\gamma}$  then
14:             $\bar{\sigma} \leftarrow \sigma(S)$  extended with  $\tau$ 
15:            Send the new upper bound  $g(S) + w_\tau(S) + w_B(S[[\tau]])$  to  $j = 1, \dots, k$ 
16:            else if  $g(S) + w_\tau(S) + \eta(S[[\tau]]) < \bar{\gamma}$  then
17:               $\sigma^{\text{current}} \leftarrow \sigma(S)$  extended with  $\tau, g_{\text{current}} \leftarrow g(S) + w(S)$ 
18:              Send  $S[[\tau]], \sigma^{\text{current}}, g_{\text{current}},$  and  $\eta(S[[\tau]])$  to  $j = (\text{HASH}(S[[\tau]]) \bmod k) + 1$ 
19:            Send  $\bar{\sigma}^i = \bar{\sigma}, \bar{\gamma}^i,$  and  $\underline{\gamma}_i = \min_{S \in O} g(S) + \eta(S)$  to the root process
20: Receive  $\bar{\sigma}^i, \bar{\gamma}^i,$  and  $\underline{\gamma}^i$  from  $i = 1, \dots, k$ 
21: return  $\bar{\sigma}^j$  with  $j \in \arg \min_{i=1, \dots, k} \bar{\gamma}^i, \min_{i=1, \dots, k} \underline{\gamma}^i$ 
```

$\bar{\sigma}$ is to optimal by comparing $\bar{\gamma}$ and $\min_{S \in O} g(S) + \eta(S)$.

4 HDS3: A Parallelization Framework for Heuristic State Space Search

We propose *hash-distributed state space search* (HDS3), a distributed parallelization framework for state space search, generalizing an existing method, HDA* (Kishimoto et al. 2013). In HDA*, each process maintains the set of generated states G and the open list O . A generated state is sent to its owner process, determined by the hash value of the state, by message passing. Since the same state is always sent to the same process, if the hash function uniformly distributes states, HDA* simultaneously achieves load balancing and duplicate detection. Message passing is done asynchronously, so each process continues to search without waiting for sent states to be received. While the basic architecture of HDS3 is the same as HDA*, it has three main differences to realize anytime heuristic state space search: the state expanded by HDS3 depends on a concrete realization of an algorithm, unlike HDA*, which always expands a state S minimizing $f(S)$; a newly found solution is reconstructed and saved during search before the termination criteria are met; and the dual bound on the optimal solution cost is returned when optimality is not proved.

We present a pseudo-code for HDS3 in Algorithm 2. While each process i performs search similarly to Algorithm 1, there are several differences. Each generated successor state $S[[\tau]]$ is sent to a process $j = (\text{HASH}(S[[\tau]]) \bmod k) + 1$ by asynchronous message passing, where $\text{HASH}(S[[\tau]])$ is the hash value of the state (line 18). The hash value is computed from the values of non-resource state variables so that dominating and dominated states are assigned to the same process. When a new solution is found, its cost is

sent to other processes, and each process updates the primal bound (lines 15 and 5). These communications are also performed by asynchronous message passing. To reduce the number of communications, when a state $S[[\tau]]$ is generated, we first check if it is a base state (line 13) and then check if it can be pruned using the bounds $\eta(S[[\tau]])$ and $\bar{\gamma}$ (line 16). If $S[[\tau]]$ is not pruned, we send it with $\eta(S[[\tau]])$ to avoid recomputation in the receiver process. While a process sends a message to itself in the pseudo-code for simplicity, it can be immediately handled without message passing. In practice, when a generated successor state $S[[\tau]]$ is assigned to the local process i in line 18, we immediately perform dominance detection, reuse $\eta(S[[\tau]])$ if $S[[\tau]]$ is already included in G or compute it if not, and then check the bound.

Similarly to Algorithm 1, the state that is selected for expansion in line 11 depends on a concrete realization of Algorithm 2. We also do not specify the termination criteria in line 4. Algorithm 1 terminates when the open list becomes empty, and the optimality is proved in such a case. In HDS3, even if the open list in one process i is empty, the open lists in other processes may not be empty, and new states may be sent to i in the future. Moreover, even when the open lists in all processes are empty, states might be being sent. Thus, we need to verify that all sent messages have been received, and no new messages will be sent. For this purpose, following HDA*, we use Mattern’s time algorithm (Mattern 1987). In practice, we also terminate HDS3 when the time limit is reached, and we use Mattern’s time algorithm in that case too to ensure that all sent messages are received.

Once all processes terminate, the best solution is selected and returned (lines 20 and 21). To reconstruct a solution, as in Algorithm 1, we maintain $\sigma(S)$, the best path to each state S . In practice, we maintain a distributed tree data structure, where each node has the ID of the last transition applied, the ID of the parent node, and the ID of the process that owns the parent node. Thus, while we send σ^{current} in line 18, in actual implementation, we send the ID of the last transition τ , the ID of the parent node corresponding to $\sigma(S)$, and the ID of the current process so that the receiver process can create a tree node. When a process i finds a new best solution, i reconstructs it by traversing the tree and sends it to a single root process, e.g., process 1, through asynchronous message passing. The root process then updates the best solution stored if the new solution is better and writes the solution to a file. In this way, the solution is continuously updated during search.

HDS3 also provides the dual bound. In Algorithm 1, when O is not empty, $\min_{S \in O} g(S) + \eta(S)$ is a lower bound on the solution cost. In HDS3, since the open lists are distributed in different processes, we cannot keep track of the dual bound during search. We compute the dual bound at the beginning and end of search: given the target state S^I , $g(S^I) + \eta(S^I)$ is a dual bound; when the termination criteria are met before the open list becomes empty, the algorithm ensures that no state is being sent using Mattern’s time algorithm, and the root process receives $\min_{S \in O} g(S) + \eta(S)$ from all processes and takes the minimum to compute a dual bound (lines 19–21). Unlike Algorithm 1, HDS3 cannot obtain the dual bound during search.

5 Scalability of Existing Anytime Heuristic State Space Search Algorithms

We measure the scalability of existing anytime DIDP solvers in a distributed environment. Our implementation (<https://github.com/Kurorororo/didp-mpi>) is based on didp-rs 0.8.0, a software framework

for DIDP, with Rust 1.70.0. We use the message passing interface (MPI) for parallelization, concretely, OpenMPI 4.1.6 with rsmapi 0.8.0, an MPI binding for Rust. We realize asynchronous message passing by using MPI_Bsend for sending and MPI_Iprobe and MPI_Recv for receiving with a buffer size of 1GB. Other implementation details of each algorithm, e.g., data structures used for the open list and dominance detection, follow didp-rs 0.8.0. For the hash function to distribute states, we use FxHash from rustc-hash 1.1.0 following Kuroiwa and Beck (2026). All experiments are performed on the Lise supercomputer cluster at Zuse Institute Berlin, where each machine has 384GB RAM and 2 Intel Xeon Platinum 9242 processors, resulting in 96 physical CPU cores in total, with Omni-Path OPA100 for communication. We use the same number of processes as the number of physical CPU cores.

In this section, we focus on two problem classes from eleven problem classes evaluated in previous work (Kuroiwa and Beck 2026): the traveling salesperson problem with time windows (TSPTW) (Savelsbergh 1985) and the type1 simple assembly line balancing problem to minimize the number of stations (SALBP-1) (Salveson 1955). From the 340 TSPTW instances (López-Ibáñez and Blum 2023) used by Kuroiwa and Beck (2026), we remove all 135 instances of the Dumas set (Dumas et al. 1995) as all of them are easy, i.e., optimally solved by sequential DIDP solvers in the previous work. Similarly, from the 2,100 SALBP-1 instances (Morrison et al. 2014) used by Kuroiwa and Beck (2026), we remove the smallest 525 instances that are easy. We take the DyPDL models of TSPTW and SALBP-1 from a public repository (<https://github.com/Kurorororo/didp-models>), implemented in YAML-DyPDL (Kuroiwa and Beck 2026).

We evaluate the following metrics: speedup from a baseline, the number of optimally solved instances, and the optimality gap. To compute speedup, we divide the time to optimally solve an instance using a baseline by that of a parallel algorithm. When an instance is not optimally solved by a time limit, the optimality gap is evaluated as $(\bar{\gamma} - \underline{\gamma})/\bar{\gamma}$ where $\bar{\gamma}$ is the primal bound and $\underline{\gamma}$ is the dual bound. When $\bar{\gamma}$ or $\underline{\gamma}$ is not provided, we use 1.0.

5.1 Complete Anytime Beam Search (CABS)

Kuroiwa and Beck (2024) proposed hash distributed beam search (HDBS), a parallelization of beam search for DIDP. Beam search performs search layer by layer. The target state belongs to layer 0. When a state S in layer l is expanded, its successor state $S[[\tau]]$ belongs to layer $l + 1$. Beam search has a parameter b , called beam width, and expands at most b states in the current layer and then proceeds to the next layer. When there are more than b states in the open list, the best b states minimizing the sum of g - and h -values, $f(S) = g(S) + h(S)$, are selected, where the dual bound function η is used as the heuristic function h . When there is no successor state, beam search terminates and returns the best solution found, which is not guaranteed to be optimal. While Algorithm 1 keeps all generated states in G , for memory efficiency, beam search keeps only states in the next layer for dominance detection, and we call this approach *layered dominance detection*.

In HDBS, each process expands the best b states assigned to it according to the hash function. While HDBS is similar to Algorithm 2, it uses a *layer synchronization* mechanism, which ensures that each process proceeds to the next layer after receiving all states in the next layer. Thanks to layer synchronization, HDBS can use layered dominance detection and compute a dual bound on the optimal solution cost after finishing

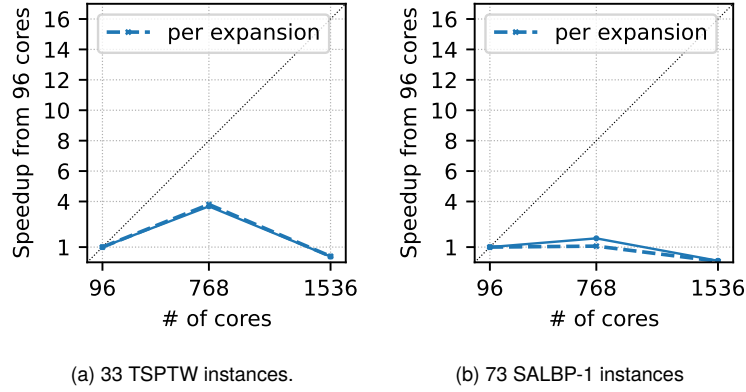


Figure 1: Geometric mean speedup of CAHDBS2 in instances where using 96 cores takes 10 to 300 seconds to solve optimally. The speedup per expansion is shown by a dashed line.

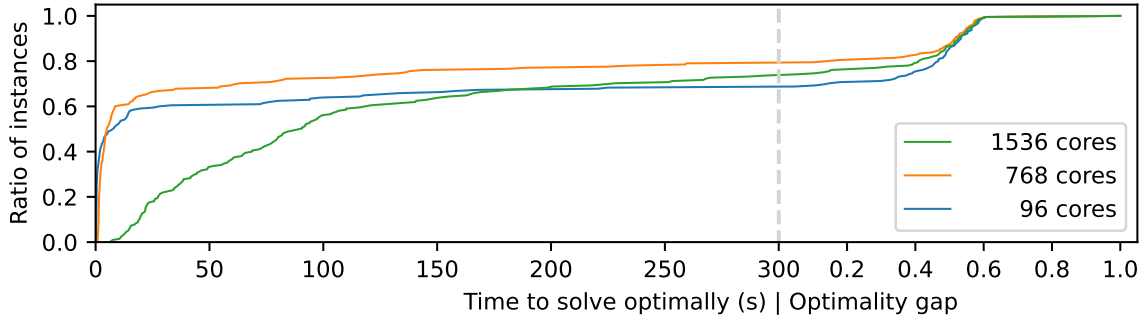


Figure 2: Ratio of instances solved optimally over time and the optimality gap with CAHDBS2 in TSPTW.

each layer. In addition, HDBS can check if search is terminated after synchronizing a layer, so it does not require a dedicated termination detection mechanism such as Mattern’s time algorithm. Kuroiwa and Beck (2024) proposed two variants, HDBS1 and HDBS2. In HDBS1, each process i proceeds to layer $l + 1$ after all processes have finished expanding states in l and receiving states in $l + 1$. In HDBS2, process i proceeds to layer $l + 1$ once it finishes layer l without waiting for others to finish receiving states in $l + 1$. However, process i may not immediately send a generated state S in layer $l + 2$; it keeps S in a buffer until the owner process, $(\text{HASH}(S) \bmod k) + 1$, finishes layer l . Empirically, HDBS2 is slightly better than HDBS1 (Kuroiwa and Beck 2024)

Complete anytime beam search (CABS) (Zhang 1998) repeats beam search multiple times from scratch while doubling b , starting from a constant. It terminates when no states are discarded due to the beam width, and the optimality is proved in such a case. Kuroiwa and Beck (2024) evaluated a multi-thread implementation of CAHDBS2, parallel CABS using HDBS2, using up to 32 threads.

5.1.1 Results

We implement CAHDBS2 with MPI and evaluate it in TSPTW and SALBP-1. Each process starts with $b = 1$ and doubles it after each iteration. We compare CAHDBS2 with 96, 768, and 1,536 cores using

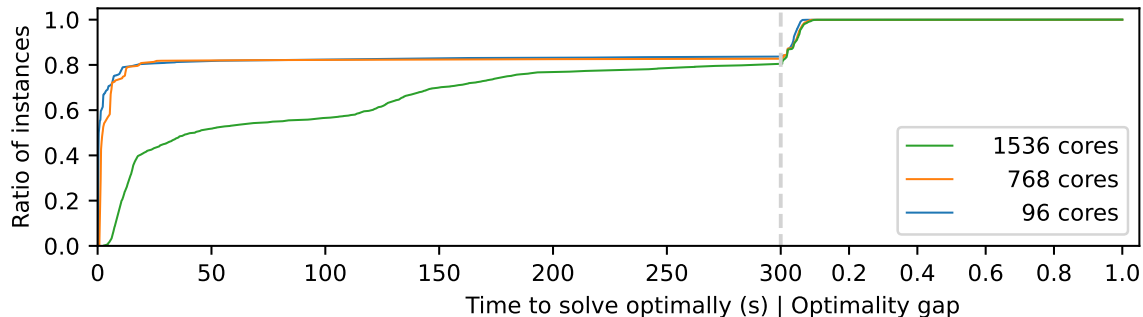


Figure 3: Ratio of instances solved optimally over time and the optimality gap with CAHDBS2 in SALBP-1.

300 second time limit for search. The overall time limit is 400 seconds, including time for initialization and finalization of MPI. Figure 1 presents the geometric mean speedup of CAHDBS2 with 768 and 1,536 cores from 96 cores using instances where CAHDBS2 with 96 cores takes at least 10 seconds. Figures 2 and 3 show the ratio of optimally solved instances over time and the cumulative ratio of instances against the optimality gap. In the right-hand side of these plots, a point (x, y) represents that an optimality gap of at most x is achieved in $100y\%$ of the instances. While using 768 cores is better than using 96 cores, using 1,536 cores results in performance degradation in both TSPTW and SALBP-1.

5.1.2 Parallel Overheads

We observe that using more processes results in a slowdown of CAHDBS2. Previous work (Kishimoto et al. 2013, Jinnai and Fukunaga 2017) classified overheads of parallel A* as follows:

- Search overhead: increase in the number of expansions caused by parallelization.
- Synchronization overhead: idle time waiting to synchronize with other processes.
- Communication overhead: cost of sending and receiving information such as generated states.

To evaluate search overhead, we show speedup per expansion in Figure 1, computed from the time to solve an instance divided by the number of expansions. In TSPTW, the speedup per expansion is almost the same as the actual speedup. In SALBP-1, the actual speedup is larger than the speedup per expansion, indicating that the number of expansions decreases as we use more cores, and this phenomenon was also observed by previous work in multi-threading (Kuroiwa and Beck 2024). Overall, the slowdown is not due to search overhead.

CAHDBS2 possibly suffers from the other two overheads. The layer synchronization mechanism introduces synchronization overhead. Moreover, for layer synchronization, each process sends control messages to other processes in addition to states, e.g., a process notifies all other processes that it has finished expanding all states in the current layer. The number of such messages does not depend on the number of states in a layer and is proportional to the number of processes. Due to the restarting mechanism, CAHDBS2 performs beam search multiple times starting from $b = 1$. When b is small, the communication cost for control messages can be dominating, and performing beam search with a larger number of processes can be slower.

5.2 Anytime Column Progressive Search and Anytime Pack Progressive Search

With HDS3, we parallelize two anytime heuristic search algorithms that do not use layer synchronization and restarting: anytime column progressive search (ACPS) (Vadlamudi et al. 2012) and anytime pack progressive search (APPS) (Vadlamudi et al. 2016). These algorithms are next to CABS in terms of performance in previous work (Kuroiwa and Beck 2026), and they have not been parallelized in a distributed environment.

ACPS partitions the open list into layers, maintaining O_l for each layer l . Starting from $l = 0$, $O_0 = \{S^I\}$, and $b = 1$, ACPS expands at most b states from O_l , inserts the successor states into O_{l+1} , and proceeds to the next layer by incrementing l by 1. Our implementation expands the b states minimizing $f(S) = g(S) + h(S) = g(S) + \eta(S)$ when $|O_l| > b$ and all states otherwise. Unlike in beam search, states that are not expanded in O_l are kept and may be expanded in future iterations. When l reaches the maximum depth, i.e., $O_i = \emptyset$ for all $i \geq l$, ACPS increments b by 1 and repeats the procedure from $l = 0$. In our parallelization with HDS3, each process independently maintains l and b without synchronization. A state S is sent with the index l of the layer to which it belongs, and the receiver process inserts S into the open list O_l .

APPS partitions the open list into three sets, O_b , O_c , and O_s , initialized with $O_b = \{S^I\}$ and $O_c = O_s = \emptyset$. APPS expands each state in O_b and inserts its successor states into O_c . After all states in O_b are expanded, the best b states in O_c are moved to O_b , and the remaining states are moved to O_s . When O_b remains empty after this step, b is increased by 1, and the best b states in O_s are moved to O_b . Again, the best b states minimizing $f(S)$ are selected in the above procedures. In our parallelization, each process separately maintains b and repeats iterations without synchronization. In addition to the successor states assigned to the local process, states received from other processes are also put in O_c .

5.2.1 Results

We observe that parallel ACPS in TSPTW and parallel APPS in SALBP-1 outperform CAHDBS2, not showing a slowdown with 1,536 or more cores, as shown in the appendix. However, ACPS in SALBP-1 and APPS in TSPTW show problematic behavior. Figures 4 and 5 compare CAHDBS2, ACPS, and APPS using 768 cores.

In some TSPTW instances, parallel APPS does not find solutions within a time limit, so the optimality gap is 1.0. We also observe that APPS fails in some SALBP-1 instances when we use 1,536 cores or more. We hypothesize that this behavior is because, when parallelized, APPS becomes closer to A*, which does not find a solution until proving the optimality. In each iteration, APPS expands b states in O_b and inserts generated successor states into O_c , from which the best b states are moved to O_b . In this way, APPS can expand a state S in a deeper layer even if $f(S)$ is large, and it possibly finds a feasible solution before proving optimality, particularly when solutions have a fixed length as in the DyPDL model of TSPTW. In contrast, in our parallel implementation, states received from other processes are also inserted into the list O_c . If a process receives many states with $f(S)$ in shallow layers from other processes, it may not keep deeper states in O_b and may fail to find feasible solutions in the early stages. Indeed, it is possible that states in deeper layers have larger $f(S) = g(S) + \eta(S)$ since the g -value is monotonically increasing in the DyPDL model for

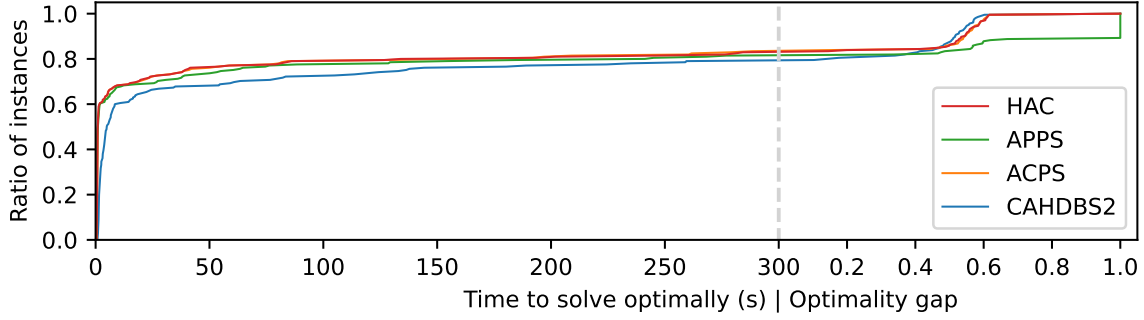


Figure 4: Ratio of instances solved optimally over time and the optimality gap with 768 cores in TSPTW.

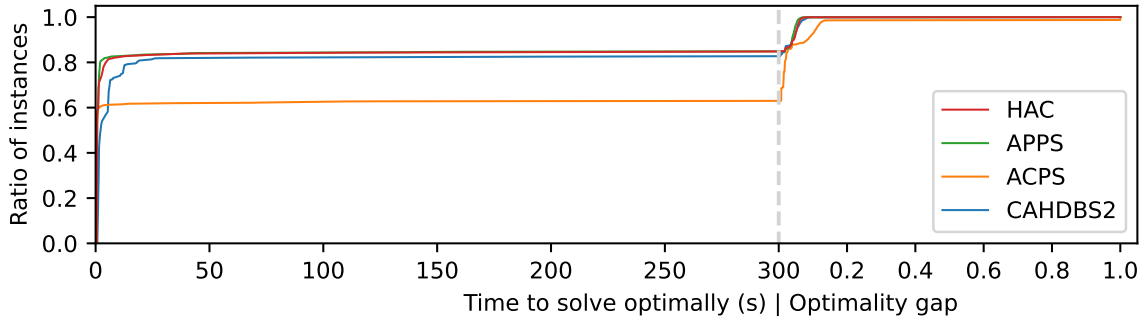


Figure 5: Ratio of instances solved optimally over time and the optimality gap with 768 cores in SALBP-1.

TSPTW and nondecreasing in SALBP-1. Furthermore, in TSPTW, the dual bound function η is not strong; previous work reported that replacing the dual bound function of the DyPDL model for TSPTW with a zero dual bound function, which always returns 0, does not change the number of optimally solved instances by sequential CABS (Kuroiwa and Beck 2026). Overall, communications with other processes may prevent each process from searching deeper layers to find feasible solutions.

In SALBP-1, parallel ACPS performs poorly and is significantly outperformed by CAHDBS2 and parallel APPS. We hypothesize that the poor performance of ACPS is due to ignorance of guidance by $f(S)$. Previous work reported that replacing the dual bound function of the DyPDL model for SALBP-1 with the zero dual bound function significantly degrades the performance of CABS (Kuroiwa and Beck 2026). This result suggests that the dual bound function for SALBP-1 is strong, and following its guidance leads to better performance. Indeed, the previous work also reported that sequential A^* , which always follows the guidance by $f(S)$, solves more instances optimally than sequential APPS and ACPS (Kuroiwa and Beck 2026). However, parallel ACPS spends almost the same search effort in all layers.

6 HAC: A New Anytime Heuristic State Space Search Algorithm

While a deeper investigation of why CAHDBS2, ACPS, and APPS fail in a distributed environment is an interesting topic, in this paper, we focus on developing a working distributed and parallel DIDP solver

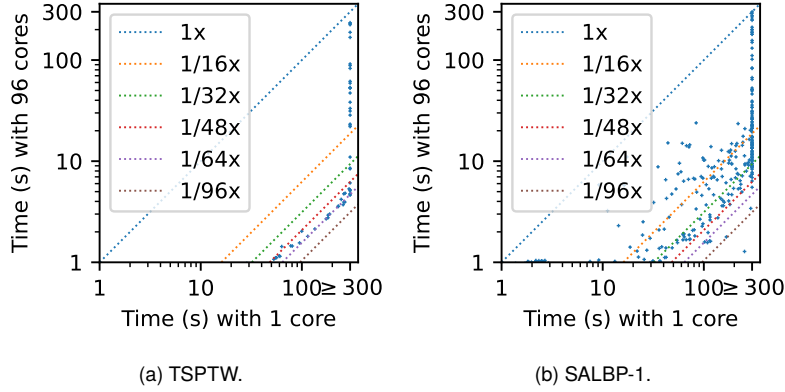


Figure 6: Comparison of HAC with 1 and 96 cores in time to solve each instance of TSPTW and SALBP-1.

to leverage the power of distributed computing. We propose a new anytime heuristic search algorithm, hybrid A* cyclic (HAC). Unlike CAHDBS2 and similarly to ACPS and APPS, HAC does not use layer synchronization and restarting. While ACPS sometimes ignores a state S with the best $f(S)$, APPS focuses too much on such states and does not search deeper layers, particularly in TSPTW. HAC addresses these issues by alternating two expansion turns. Similarly to ACPS, HAC partitions O into layers and inserts a state in layer l into O_l . First, HAC expands a state $S \in \arg \min_{S' \in O} f(S')$ among all layers, similarly to A*. In the next turn, it expands a state $S \in \arg \min_{S' \in O_l} f(S')$ where l is the index of the current layer, initialized with $l = 0$. When $O_l = \emptyset$, HAC updates l to the minimum $l' \geq l$ such that $O_{l'} \neq \emptyset$ and then expands a state from $O_{l'}$. If no such l' exists, HAC uses the minimum $l' < l$ such that $O_{l'} \neq \emptyset$. After expanding a state from O_l , HAC increases l by 1. In our parallelization using HDS3, each process independently maintains l , and each state is sent with the depth of the layer.

As shown in Figures 4 and 5, with 768 cores, HAC performs similarly to ACPS in TSPTW and to APPS in SALBP-1, outperforming CAHDBS2 in the number of optimally solved instances, and always succeeds in finding a feasible solution.

We evaluate parallel HAC with different numbers of cores using two problem classes in addition to TSPTW and SALBP-1: one-to-one multi-commodity pickup and delivery traveling salesperson problem (m-PDTSP) (Hernández-Pérez and Salazar-González 2009) and the type2 assembly line balancing problem with sequence-dependent setup times (SUALBP-2) (Andrés et al. 2008). Previous work demonstrated state-of-the-art performance of sequential DIDP solvers in these problem classes (Kuroiwa and Beck 2026, Zhang and Beck 2025). For m-PDTSP, we use 248 class1 instances (Hernández-Pérez and Salazar-González 2009) and 240 larger class1 instances generated by Kuroiwa and Beck (2023). We take the YAML-DyPDL model of m-PDTSP from the same repository as TSPTW and SALBP-1. For SUALBP-2, from the 788 instances by (Scholl et al. 2013), we remove all 272 instances of class A and B that are optimally solved by a sequential DIDP solver (Zhang and Beck 2025). We take the DyPDL model of SUALBP-2 from the supplementary material of Zhang and Beck (2025) (<https://github.com/INFORMSJoC/2024.0603>), converting the model implemented in the Python library, DIDPPy, into YAML-DyPDL.

First, we compare sequential HAC and parallel HAC with 96 cores in terms of the time to solve each

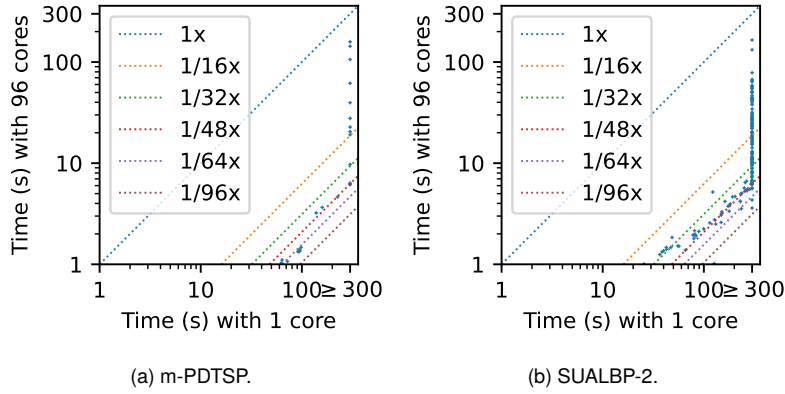


Figure 7: Comparison of HAC with 1 and 96 cores in time to solve each instance of m-PDTSP and SUALBP-2.

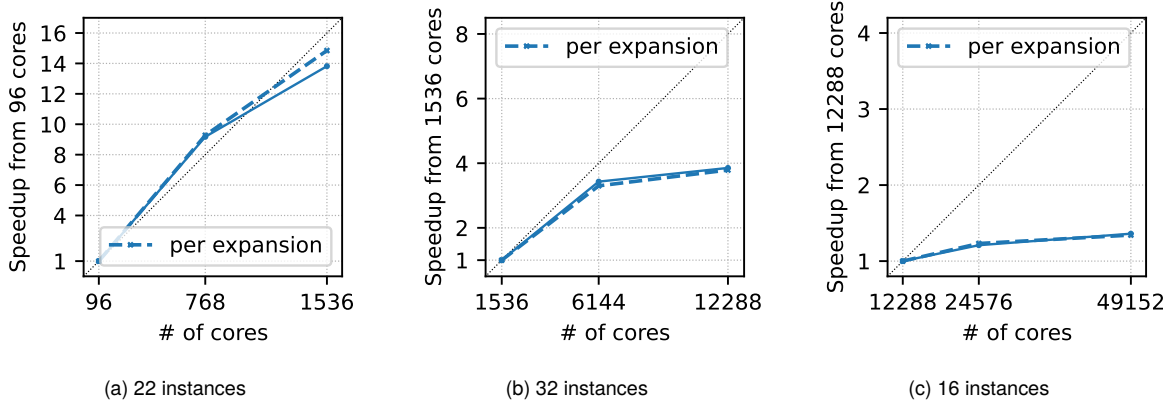


Figure 8: Geometric mean speedup of parallel HAC in TSPTW instances, where a baseline takes 10 to 300 seconds to solve. The speedup per expansion is shown by a dashed line.

instance optimally, shown in Figures 6 and 7. Parallel HAC typically achieves a speedup of 48 to 64 times on TSPTW, m-PDTSP, and SUALPB-2. In SALPB-1, speedup varies widely across instances, consistent with the previously observed result in multithread CAHDBS2, possibly due to the tight dual bound function (Kuroiwa and Beck 2024), but is typically from 16 to 64 times.

For TSPTW, m-PDTSP, and SUALBP-2, we evaluate HAC using 96, 768, 1,536, 6,144, 12,288, 24,576, and 49,152 cores. For SALPB-1, we are not able to use 49,152 cores due to resource limitations of our cluster. We sometimes encounter an error due to parameters of OpenMPI, PSM2_MQ_RECVREQS_MAX or PSM2_MQ_SENDREQS_MAX. In such cases, we use 2,097,152 (twice the default value) or double it if it is still insufficient. As shown in Figures 8–11, while the relative speedup becomes smaller as we increase the number of cores, HAC continues to speed up with up to 49,152 cores in TSPTW and m-PDTSP and up to 24,576 cores in SALPB-1 and SUALBP-2. Figures 12–15 show that using more cores helps solve more instances optimally and helps achieve a smaller optimality gap, even in SUALBP-2, where we observe a slowdown with 49,152 cores.

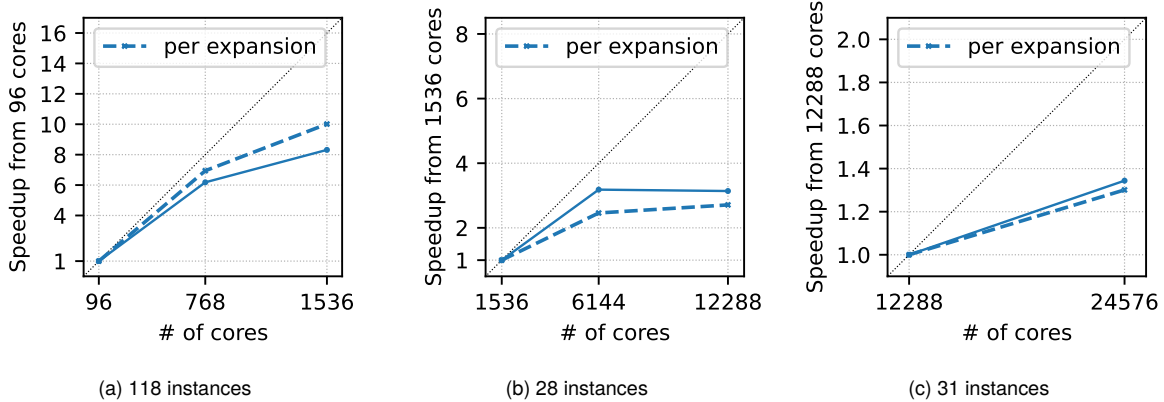


Figure 9: Geometric mean speedup of parallel HAC in SALBP-1 instances, where a baseline takes 10 to 300 seconds to solve. The speedup per expansion is shown by a dashed line.

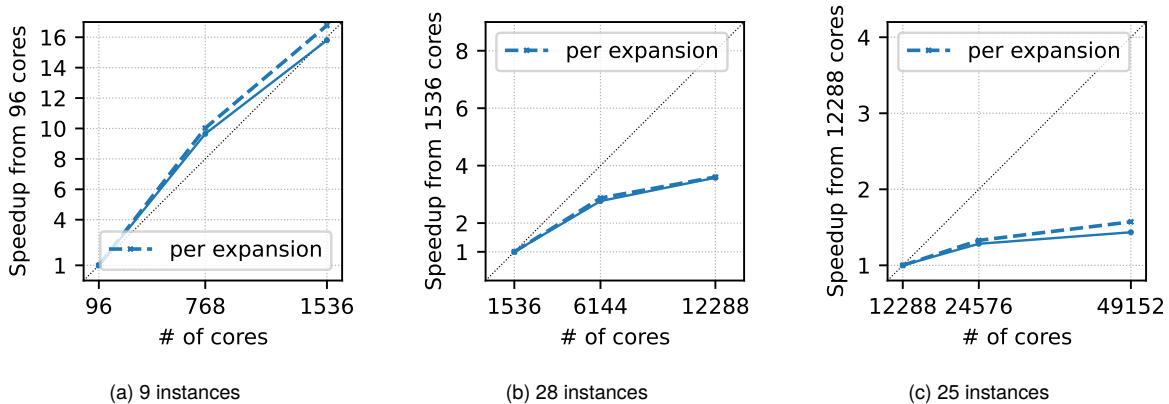


Figure 10: Geometric mean speedup of parallel HAC in m-PDTSP instances, where a baseline takes 10 to 300 seconds to solve. The speedup per expansion is shown by a dashed line.

7 Memory-Efficient Parallel State Space Search

While we have used the 300-second time limit so far, once we run parallel HAC without a time limit, it runs out of memory within a few thousand seconds when an instance is not solved. To solve more problem instances, we propose a more memory-efficient approach.

7.1 Layer-Synchronized Breadth-First Search

Compared to HAC, CAHDBS2 is memory-efficient, thanks to layered dominance detection, which discards states in previous layers. However, CAHDBS2 is not scalable, as shown in our evaluation, possibly due to layer synchronization and restarting. A straightforward approach is to avoid restarting. Instead of doubling the beam width b after each iteration starting from small b , we use $b = \infty$, i.e., do not discard any states based on the priority. This algorithm is equivalent to breadth-first search (BrFS), and its layer synchronization mechanism is the same as HDBS2, so we name it BrFS2. The drawback of eliminating restarting is that BrFS2 cannot find solutions quickly using small beam widths. Therefore, we run parallel HAC until it

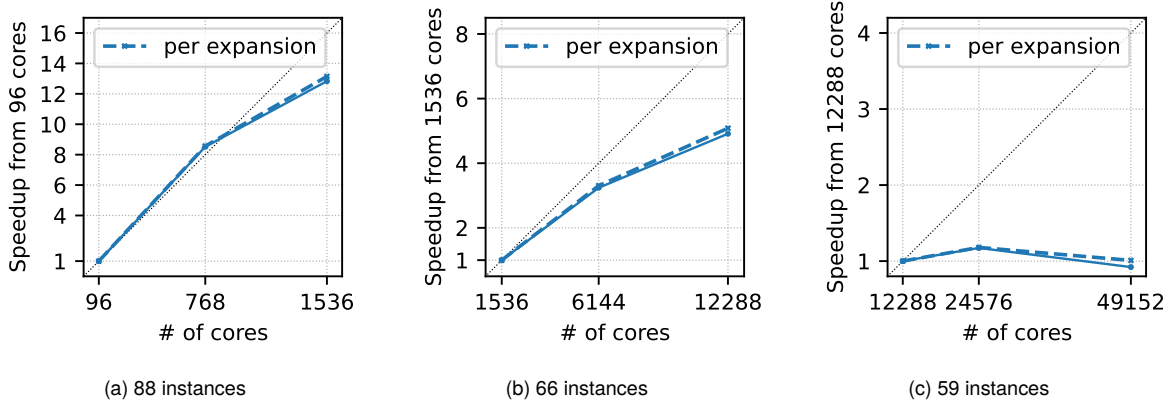


Figure 11: Geometric mean speedup of parallel HAC in SUALBP-2 instances, where a baseline takes 10 to 300 seconds to solve. The speedup per expansion is shown by a dashed line.

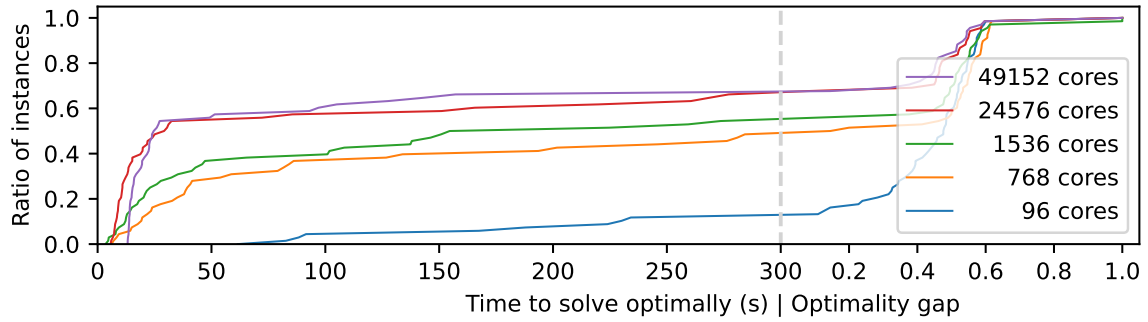


Figure 12: Ratio of instances solved optimally over time and the optimality gap with HAC in 68 TSPTW instances, where HAC with 96 cores takes at least 60 seconds.

runs out of memory, and then run BrFS2 using the best solution cost found by HAC as a primal bound. Since BrFS2 may prune states using the primal bound and the dual bound function, it can be considered a distributed parallelization of breadth-first heuristic search (Zhou and Hansen 2006).

7.2 New Parallel Breadth-First Search Method

We also propose BrFS3, a parallel BrFS method that uses layered dominance detection without layer synchronization, based on HDS3. In BrFS3, the open list O is partitioned into layers, O_l for layer l . In each step, each process expands a state from O_l where l is the minimum value such that $O_l \neq \emptyset$ and is possibly different for different processes. Similar to BrFS2, BrFS3 uses a primal bound given as input and may prune states with the dual bound function.

For layered dominance detection, each process partitions the set of generated states G into layers, G_l for layer l . In the first layer, only one process has the target state in its open list O_0 , and the open lists in other processes are empty. After process i expands all states assigned to it in layer l (we will explain how to confirm this situation below), i notifies each process j of the number of successor states in layer $l + 1$ sent to j , c_{ij}^{l+1} , using asynchronous message passing. Each process j keeps track of r_{ij}^l , the number of states in

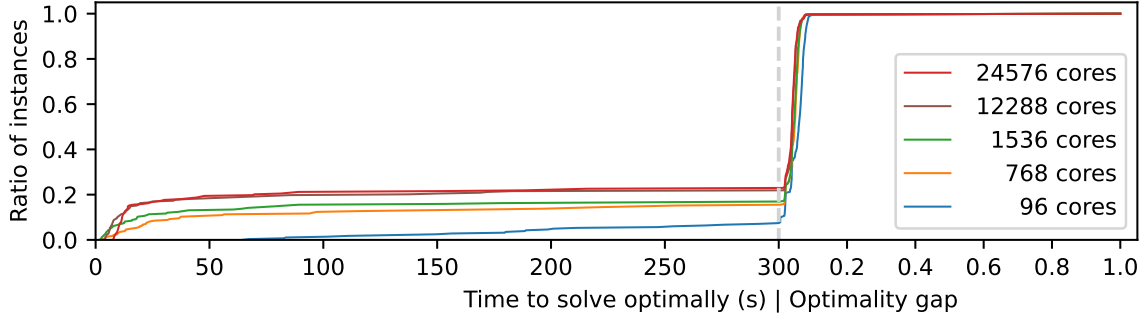


Figure 13: Ratio of instances solved optimally over time and the optimality gap with HAC in 283 SALBP-1 instances, where HAC with 96 cores takes at least 60 seconds.

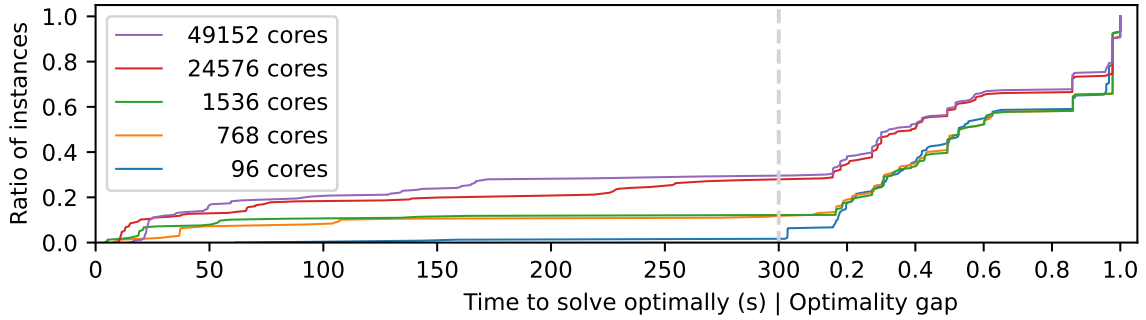


Figure 14: Ratio of instances solved optimally over time and the optimality gap with HAC in 237 m-PDTSP instances, where HAC with 96 cores takes at least 60 seconds.

layer l received from process i , and increments it when a state in layer l is received from i . After receiving c_{ij}^{l+1} , at the point when $c_{ij}^{l+1} = r_{ij}^{l+1}$ holds, process j recognizes that no more states in layer $l + 1$ will be sent from process i . At the point when $c_{ij}^{l+1} = r_{ij}^{l+1}$ for all i and $O_{l+1} = \emptyset$, process j has finished expanding all states in layer $l + 1$, so it sends each process i the number of successor states in layer $l + 2$, c_{ji}^{l+2} , and discards all states in G_{l+1} from memory.

In the above procedure, process i sends c_{ij}^{l+1} if and only if i expands at least one state in layer l , receives at least one state in layer $l + 1$, or receives c_{ji}^{l+1} from another process j , i.e., at least one state in layer l must be expanded in some process. Otherwise, no process i sends c_{ij}^{l+1} , and termination is detected by Mattern's time algorithm. Without this condition, processes would repeat sending $c_{ij}^l = 0$ to each other infinitely many times while incrementing l and be unable to terminate.

7.3 Results

To obtain primal bounds for BrFS2 and BrFS3, in addition to the result from the previous experiments, we run parallel HAC using 24,576 cores until it runs out of memory. Due to resource limitations on our cluster, we are unable to use 49,152 cores in this experiment. We are also unable to use 24,576 cores for SALBP-1 and SUALBP-2. As a result, we run BrFS2 and BrFS3 with 12,288 and 24,576 cores for TSPTW

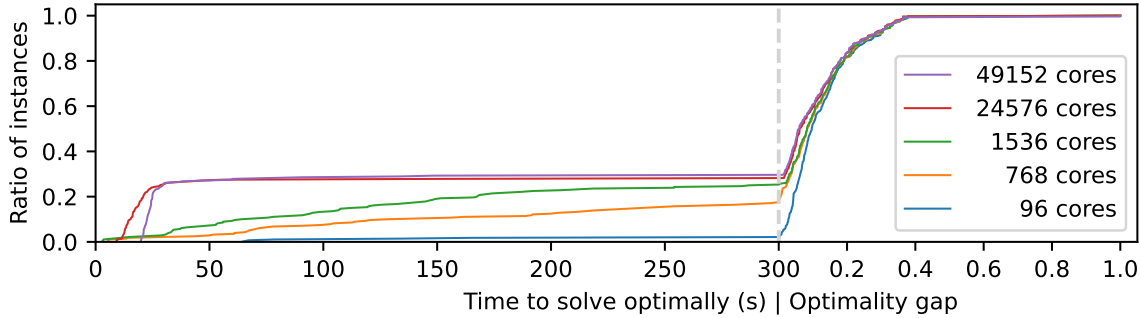


Figure 15: Ratio of instances solved optimally over time and the optimality gap with HAC in 280 SUALBP-2 instances, where HAC with 96 cores takes at least 60 seconds.

Table 1: Time (s) to solve instances using BrFS with 12,288 and 24,576 cores. ‘OOM’ indicates memory out, and ‘-’ indicates that the configuration is not run. The shortest time is in bold.

Class	Set	Instance	BrFS2		BrFS3	
			12288	24576	12288	24576
TSPTW	Ohlmann and Thomas (2007)	n150w140.004	OOM	OOM	OOM	2070
		n150w160.001	2805	OOM	OOM	1664
SALBP-1	Large (Scholl et al. 2013)	n=100_60	719	-	495	-
		n=100_205	519	-	322	-
		n=100_209	601	-	430	-
		n=100_221	890	-	674	-
		n=100_354	619	-	431	-

and m-PDTSP, and only 12,288 cores for SALBP-1 and SUALBP-2.

BrFS2 and BrFS3 solve multiple additional instances in TSPTW and SALBP-1, while none in m-PDTSP and SUALBP-2. Table 1 shows the time in seconds to solve each instance. With 24,576 cores, BrFS3 succeeds in solving one TSPTW instance not solved by BrFS2. In contrast, while BrFS2 solves another TSPTW instance with 12,288 cores, BrFS3 requires 24,576 cores to solve it. Interestingly, BrFS2 fails to solve the same instance with 24,576 cores, despite the fact that twice as much memory is available in total. In BrFS2, each process may store states assigned to other processes in its local buffer before sending for the layer synchronization mechanism, following CAHDBS2 (Kuroiwa and Beck 2024). When using more processes, due to a higher expansion rate, more states can be stored in the buffer at a time, which may temporarily lead to higher memory consumption per machine. In SALBP-1, BrFS2 and BrFS3 solve the same five instances, and BrFS3 is faster than BrFS2, demonstrating that removing layer synchronization improves running time.

8 Closed Instances

In our evaluation, some of the open problem instances of TSPTW, m-PDTSP, and SUALBP-2 are solved by the proposed algorithms. We report such instances in our repository (<https://github.com/Kurorororo/didp-best-known-solutions>). In Table 2, we summarized the number of instances solved by HAC with the minimum required number of cores. For TSPTW, from the open instances reported by López-Ibáñez and Blum (2023), 13 are closed by HAC, and one (n150w160.001) is solved by BrFS2 with 12,288 cores and

Table 2: Number of closed instances by HAC with the minimum number of cores.

Problem	Instances from	1	96	768	1536	6144	12288	24576	49152	Total
TSPTW	López-Ibáñez and Blum (2023)	0	4	5	0	1	2	1	0	14
m-PDTSP	Hernández-Pérez and Salazar-González (2009)	0	0	2	0	3	2	0	0	7
	Kuroiwa and Beck (2023)	0	6	14	2	8	10	0	2	42
SUALBP-2	Scholl et al. (2013)	1	74	46	19	6	1	1	4	152

Table 3: Number of SUALBP-2 instances whose best known solution cost and bound are improved by HAC with the minimum number of cores.

	1	96	768	1536	6144	12288	24576	49152	Total
Solution cost	1	3	9	4	21	20	26	32	116
Lower bound of the optimal solution cost	44	7	0	0	0	0	0	14	65

BrFS3 with 24,576 cores. In m-PDTSP, from the instances unsolved in previous work (Hernández-Pérez and Salazar-González 2009, Gouveia and Ruthmair 2015, Castro et al. 2020, Kuroiwa and Beck 2026), 7 are closed in total. From the larger m-PDTSP instances proposed by Kuroiwa and Beck (2023), 42 are closed. In SUALBP-2, from the instances unsolved in previous work (Zohali et al. 2022, Zhang and Beck 2025), 152 are closed in total. Moreover, in the remaining open instances, the best-known solution cost is improved in 116, and the dual bound of the optimal solution cost (the lower bound as SUALBP-2 is a minimization problem) is improved in 65, as reported in Table 3.

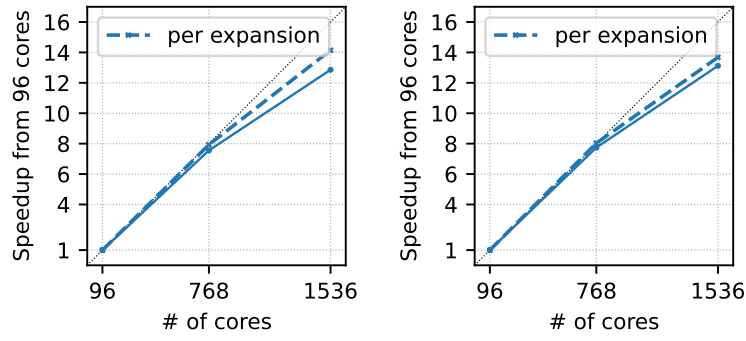
9 Conclusion

We developed massively parallel, distributed, and anytime domain-independent dynamic programming (DIDP) solvers. We generalized an existing method to an algorithmic framework, hash-distributed state space search (HDS3), to parallelize anytime heuristic search algorithms. Since the parallelization of existing algorithms is not scalable, we proposed a new anytime algorithm, hybrid A* cyclic (HAC), and parallelized it using HDS3. We also develop BrFS2 and BrFS3, memory-efficient distributed parallelization methods of breadth-first heuristic search. Using up to 192 TB of RAM and 49,152 processes, we demonstrate the scalability of the developed solvers and solved multiple open problem instances in the traveling salesperson problem with time windows (TSPTW), the one-to-one multi-commodity pickup and delivery traveling salesperson problem (m-PDTSP), and the type2 assembly line balancing problem with sequence-dependent setup times (SUALBP-2).

A limitation of our current approaches is the high consumption of memory. While parallel breadth-first search methods can solve some instances where parallel HAC runs out of memory, they still fail due to memory out in other instances. One approach is to reduce memory usage by discarding some states with the expense of expanding the same state multiple times, e.g., iterative deepening A* with a transposition table (Korf 1985, Reinefeld and Marsland 1994, Romein et al. 2002). Another approach is to use external memory search, which saves states in an external memory such as a solid-state drive, as studied in sequential and multi-thread heuristic state space search (Zhou and Hansen 2004, Korf 2008, Lin and Fukunaga 2018, Hatem et al. 2018).

Appendix

Figure 16 presents the speedup of parallel ACPS using 768 and 1,536 cores compared with their 96 cores in TSPTW and SALPB-1. Figures 17 and 18 compare the ratio of instances over time to optimally solve and the optimality gap. Figures 19–21 evaluate APPS in the same way.



(a) 26 TSPTW instances.

(b) 24 SALBP-1 instances

Figure 16: Geometric mean speedup of ACPS in instances where using 96 cores takes 10 to 300 seconds to solve optimally. The speedup per expansion is shown by a dashed line.

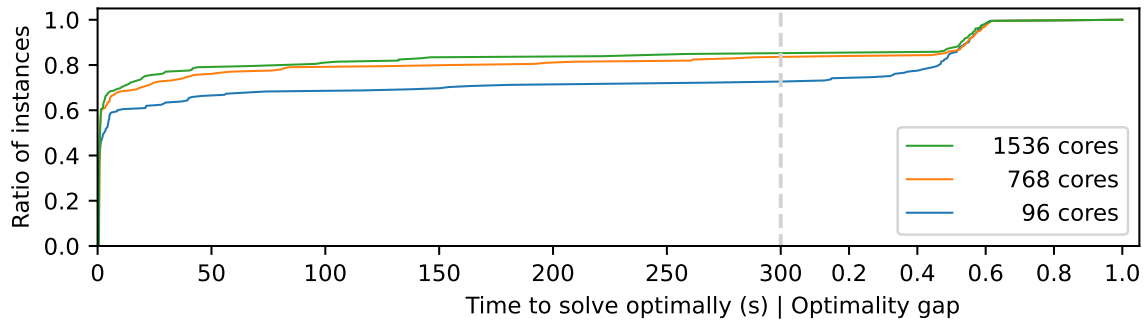


Figure 17: Ratio of instances solved optimally over time and the optimality gap with ACPS in TSPTW.

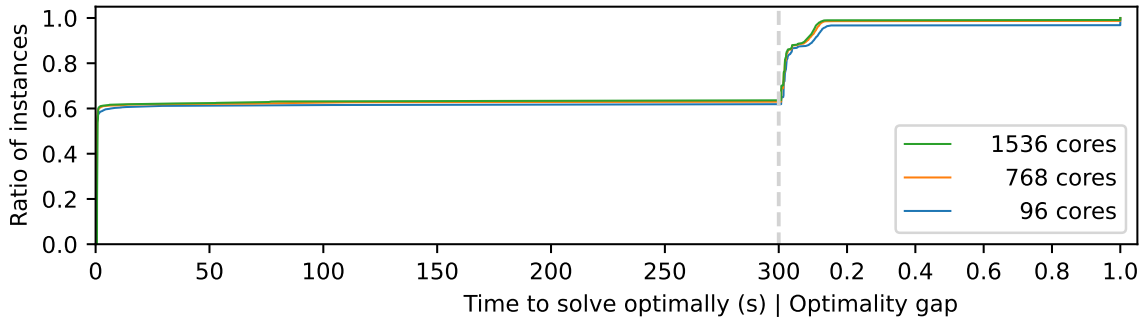


Figure 18: Ratio of instances solved optimally over time and the optimality gap with ACPS in SALBP-1.

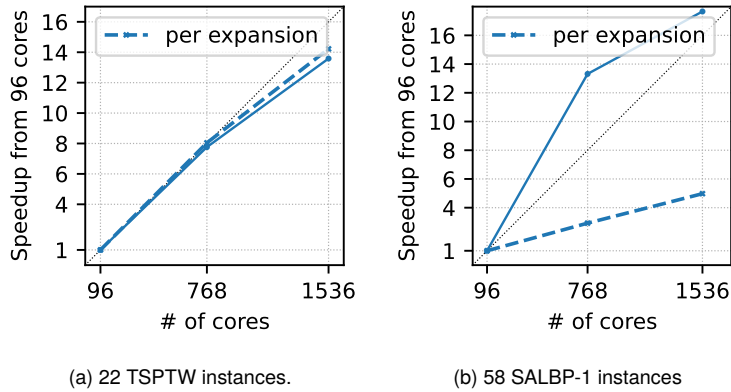


Figure 19: Geometric mean speedup of APPS in instances where using 96 cores takes 10 to 300 seconds to solve optimally. The speedup per expansion is shown by a dashed line.

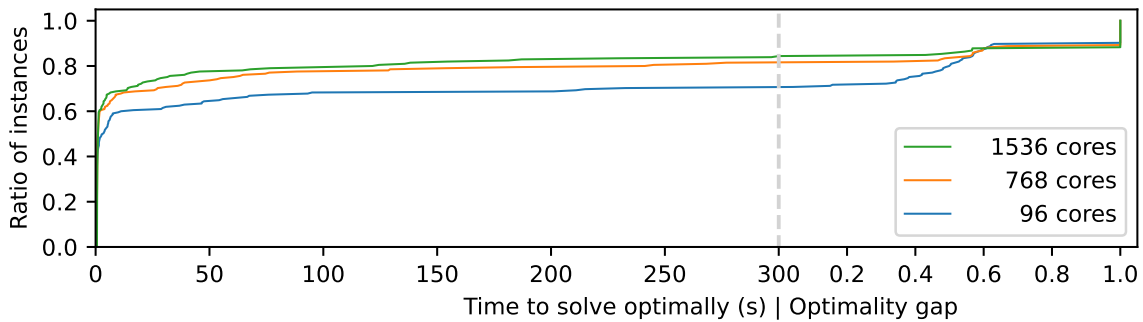


Figure 20: Ratio of instances solved optimally over time and the optimality gap with APPS in TSPTW.

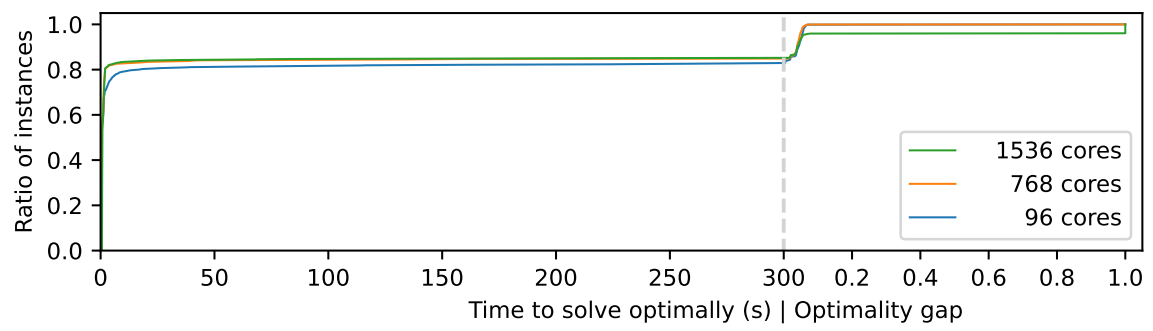


Figure 21: Ratio of instances solved optimally over time and the optimality gap with APPS in SALBP-1.

Acknowledgements The authors gratefully acknowledge the computing time made available to them on the high-performance computer "Lise" at the NHR center NHR@ZIB. This center is jointly supported by the Federal Ministry of Education and Research and the state governments participating in the NHR (www.nhr-verein.de).

References

- Carlos Andrés, Cristóbal Miralles, and Rafael Pastor. Balancing and scheduling tasks in assembly lines with sequence-dependent setup times. *Eur. J. Oper. Res.*, 187(3):1212–1223, 2008.
- Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.
- Humberto Carrillo and David Lipman. The multiple sequence alignment problem in biology. *SIAM J. Appl. Math.*, 48(5):1073–1082, 1988.
- Margarita P. Castro, Andre A. Cire, and J. Christopher Beck. An MDD-based Lagrangian approach to the multicommodity pickup-and-delivery TSP. *INFORMS J. Comput.*, 32(2):263–278, 2020.
- G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *J. Oper. Res. Soc. Am.*, 2(4):393–410, 1954.
- Yvan Dumas, Jacques Desrosiers, Eric Gelinás, and Marius M Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Oper. Res.*, 43(2):367–371, 1995.
- S. Dutt and N.R. Mahapatra. Scalable load balancing strategies for parallel A* algorithms. *J. Parallel Distrib. Comput.*, 22(3):488–505, 1994.
- Jonathan Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5. *SIAM J. Optim.*, 4(4):794–814, 1994.
- Jonathan Eckstein, Cynthia A. Phillips, and William E. Hart. Pico: An object-oriented framework for parallel branch and bound. In Dan Butnariu, Yair Censor, and Simeon Reich, editors, *Inherently Parallel Algorithms in Feasibility and Optimization and their Applications*, volume 8 of *Studies in Computational Mathematics*, pages 219–265. Elsevier, 2001.
- Stefan Edelkamp, Stefan Schroedl, and Sven Koenig. *Heuristic Search: Theory and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.
- M. Evett, J. Hendler, A. Mahanti, and D. Nau. PRA*: Massively parallel heuristic search. *J. Parallel Distrib. Comput.*, 25(2):133–143, 1995.
- Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning*. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, Burlington, 2004.
- Luis Gouveia and Mario Ruthmair. Load-dependent and precedence-based models for pickup and delivery problems. *Comput. Oper. Res.*, 63:56–71, 2015.
- Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.*, 4(2):100–107, 1968.
- Matthew Hatem, Ethan Burns, and Wheeler Ruml. Solving large problems with heuristic search: General-purpose parallel external-memory search. *J. Artif. Intell. Res.*, 62:233–268, 2018.
- Hipólito Hernández-Pérez and Juan José Salazar-González. The multi-commodity one-to-one pickup-and-delivery traveling salesman problem. *Eur. J. Oper. Res.*, 196(3):987–995, 2009.

- Yuu Jinnai and Alex Fukunaga. On hash-based work distribution methods for parallel best-first search. *J. Artif. Intell. Res.*, 60:491–548, 2017.
- Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations: Proceedings of a Symposium on the Complexity of Computer Computations*, pages 85–103. Springer US, Boston, MA, 1972.
- Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Evaluation of a simple, scalable, parallel best-first search strategy. *Artif. Intell.*, 195:222–248, 2013.
- Yoshikazu Kobayashi, Akihiro Kishimoto, and Osamu Watanabe. Evaluations of hash distributed A* in optimal sequence alignment. In *IJCAI*, pages 584–590, 2011.
- Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.
- Richard E. Korf. Linear-time disk-based implicit graph search. *J. ACM*, 55(6), 2008.
- Vipin Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel best-first search of state-space graphs: A summary of results. In *AAAI*, pages 122–127, 1988.
- Ryo Kuroiwa and J. Christopher Beck. Large neighborhood beam search for domain-independent dynamic programming. In *CP*, volume 280, pages 23:1–23:22, 2023.
- Ryo Kuroiwa and J. Christopher Beck. Parallel beam search algorithms for domain-independent dynamic programming. In *AAAI*, pages 20743–20750, 2024.
- Ryo Kuroiwa and J. Christopher Beck. Domain-independent dynamic programming. *Artif. Intell.*, 354:104506, 2026.
- Ryo Kuroiwa and Alex Fukunaga. On the pathological search behavior of distributed greedy best-first search. In *ICAPS*, pages 255–263, 2019.
- Shunji Lin and Alex Fukunaga. Revisiting immediate duplicate detection in external memory search. In *AAAI*, pages 1347–1354, 2018.
- Manuel López-Ibáñez and Christian Blum. Benchmark instances for the travelling salesman problem with time windows (TSPTW), 2023. URL <https://lopez-ibanez.eu/tsptw-instances>.
- N.R. Mahapatra and S. Dutt. Scalable duplicate pruning strategies for parallel A* graph search. In *SPDP*, pages 290–297, 1993.
- Friedemann Mattern. Algorithms for distributed termination detection. *Distrib. Comput.*, 2(3):161–175, 1987.
- David R. Morrison, Edward C. Sewell, and Sheldon H. Jacobson. An application of the branch, bound, and remember algorithm to a new simple assembly line balancing dataset. *Eur. J. Oper. Res.*, 236(2):403–409, 2014.
- Jeffrey W. Ohlmann and Barrett W. Thomas. A compressed-annealing heuristic for the traveling salesman problem with time windows. *INFORMS J. Comput.*, 19(1):80–90, 2007.
- A. Reinefeld and T.A. Marsland. Enhanced iterative-deepening search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(7):701–710, 1994.
- J.W. Romein, H.E. Bal, J. Schaeffer, and A. Plaat. A performance analysis of transposition-table-driven work scheduling in distributed search. *IEEE Trans. Parallel Distrib. Syst.*, 13(5):447–459, 2002.
- Stuart Russell and Peter Norvig. Solving problems by searching. In *Artificial Intelligence: A Modern Approach*, chapter 3, pages 63–109. Pearson, fourth edition, 2020.
- M. E. Salvesson. The assembly-line balancing problem. *J. Ind. Eng.*, 6(3):18–25, 1955.
- M. W. P. Savelsbergh. Local search in routing problems with time windows. *Ann. Oper. Res.*, 4(1):285–305, 1985.
- Armin Scholl, Nils Boysen, and Malte Fließner. The assembly line balancing and scheduling problem with sequence-dependent setup times: problem extension, model formulation and efficient heuristics. *OR Spectr.*, 35(1):291–320, 2013.

- Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, Thorsten Koch, and Michael Winkler. Solving open MIP instances with ParaSCIP on supercomputers using up to 80,000 cores. In *IPDPS*, pages 770–779, 2016.
- Satya Gautam Vadlamudi, Piyush Gaurav, Sandip Aine, and Partha Pratim Chakrabarti. Anytime column search. In *AI 2012*, pages 254–265, 2012.
- Satya Gautam Vadlamudi, Sandip Aine, and Partha Pratim Chakrabarti. Anytime pack search. *Nat. Comput.*, 15(3): 395–414, 2016.
- Jiachen Zhang and J. Christopher Beck. Domain-independent dynamic programming and constraint programming approaches for assembly line balancing problems with setups. *INFORMS J. Comput.*, 37(4):977–997, 2025.
- Weixiong Zhang. Complete anytime beam search. In *AAAI*, pages 425–430, 1998.
- Rong Zhou and Eric A. Hansen. Structured duplicate detection in external-memory graph search. In *AAAI*, pages 683–688, 2004.
- Rong Zhou and Eric A. Hansen. Breadth-first heuristic search. *Artif. Intell.*, 170(4):385–408, 2006.
- Hassan Zohali, Bahman Naderi, and Vahid Roshanaei. Solving the type-2 assembly line balancing with setups using logic-based benders decomposition. *INFORMS J. Comput.*, 34(1):315–332, 2022.