DOI https://doi.org/10.24297/ijct.v25i.9709

Parallel Metaheuristic Algorithms for Task Scheduling Problems

Sirui Mao[1], *Masato Edahiro*[1]

[1]Nagoya University

Email: eda@ertl.jp

Abstract

This study addresses the task-scheduling optimization challenges in parallel computing systems using a novel metaheuristic framework. We analyze the differential evolution in task scheduling and propose an advanced shift-chain methodology to improve the cooperation between scheduling components. The proposed framework introduces a waiting time-based neighborhood exploration strategy for handling complex task dependencies, along with two parallel implementation approaches: basic matching vector (MV) parallelization and an event-driven strategy. The experimental results demonstrated superior solution quality and computational efficiency compared with existing methods, particularly in large scale problems. The modular design of this framework enables practical applications in modern computing environments.

Keywords: Task scheduling optimization, Parallel computing, Metaheuristics

1 Introduction

With the rapid advancement of computing systems and increasing complexity of data processing requirements, efficient resource allocation and optimization have become critical challenges in modern computing environments. Many real-world optimization problems are classified as nondeterministic polynomial time (NP)-hard, making it impractical to find optimal solutions for practical instances using exact methods. This complexity is particularly evident in cloud- computing environments, where dynamic workload patterns and heterogeneous resource configurations create multifaceted optimization challenges that require effective cooperation between different solution components.

Metaheuristic algorithms have emerged as effective approaches for solving complex optimization problems. These algorithms provide near-optimal solutions within a reasonable computational time through iterative improvement processes that rely on cooperative interactions between solution components. The growing scale of data processing demands has led to an increased interest in parallel implementations of metaheuristics to achieve faster convergence and better solution quality. However, maintaining effective cooperation among solution components in parallel environments presents additional challenges that must be addressed.

Traditional parallel metaheuristic approaches have limitations in managing cooperative effects, particularly for complex neighborhood structures. These limitations are manifested in several ways, including premature convergence to local optima, inefficient utilization of parallel resources, and difficulties in maintaining solution diversity. This has motivated the development of more sophisticated search mechanisms, such as the ejection chain method, which enables the structured exploration of complex moves through a cooperative series of interlinked modifications.

In this study, we propose a method for developing an efficient parallel metaheuristic framework to solve task scheduling problems in homogeneous multicore systems, with a particular focus on enhancing and maintaining cooperative effects throughout the optimization process. Based on previous research [?], we propose an extended shift-chain methodology to explore the task-scheduling solution space while maintaining cooperative effects between solution components. This method builds upon established ejection chain concepts and adapts them specifically to handle the complex dependency relationships inherent to task-scheduling problems. Additionally, we investigate parallel implementation strategies that preserve cooperative effects while using a multicore architecture to examine both basic parallelization approaches and event-driven execution patterns.

The contributions of this study are outlined as follows:

1. The limitations of existing metaheuristic approaches are analyzed, focusing on the differential evolution method for task scheduling problems, further examining how cooperative effects influence solution quality. Through systematic experimentation and analysis, insights are provided into how the concurrent optimization of multiple solution components affects cooperation between scheduling decisions.

2. An extended ejection chain methodology is developed to explore the task-scheduling solution space while maintaining cooperative effects between solution components. Our approach enables effective exploration through structured cooperation between task ordering and processor assignment decisions while addressing the computational challenges associated with dependency constraint handling.

3. Parallel strategies are implemented to preserve cooperative effects across multiple cores, identifying efficient parallel execution mechanisms that maintain beneficial cooperation between solution components in task-scheduling contexts.

4. Comprehensive experimental evaluations were conducted to validate the proposed method against existing approaches, demonstrating both solution quality and computational efficiency in parallel environments.

2 Previous and Related Work

2.1 Metaheuristics

Metaheuristics represent a family of approximate optimization techniques that combine basic heuristic methods in higher-level frameworks to efficiently explore and exploit solution spaces [4]. These algorithms provide a general problem-solving methodology applicable to a wide range of complex optimization problems. Rather than guaranteeing optimal solutions, metaheuristics aim to obtain high-quality solutions within reasonable computational time. The theoretical foundations of metaheuristics have been drawn from various fields, including evolutionary biology, statistical physics, and artificial intelligence [7].

2.1.1 Parallelization Approaches

Parallel implementations of metaheuristics have garnered significant attention as a means of accelerating optimization processes [6]. Traditional parallel approaches typically focus on population-level parallelization in which multiple solution candidates are evaluated simultaneously [6]. However, some studies have demonstrated [3] that simply distributing the computational load without considering cooperation mechanisms can degrade search effectiveness.

2.1.2 Cooperative Search Strategies

Advanced parallel metaheuristic implementations incorporate cooperative search strategies to maintain search effectiveness by leveraging parallel computing resources [11]. These strategies can be broadly categorized into synchronous and asynchronous approaches. Synchronous methods maintain structured communication between parallel processes, ensuring a coordinated exploration of the solution space. Asynchronous approaches, which are more complex to implement, often demonstrate better performance by allowing individual search processes to progress independently while periodically sharing promising solutions [3] [6].

2.1.3 Classical Metaheuristic Algorithms

The field of metaheuristics encompasses various nature-inspired optimization algorithms. Ant colony optimization (ACO) emulates the collective intelligence of ant colonies, whereby artificial ants construct solutions through pheromone-based communication mechanisms [2]. This approach has been particularly effective in discrete optimization problems and network routing. Particle swarm optimization (PSO) models the social behavior patterns observed in bird flocks and fish schools, where particles navigate the solution space guided by both individual and collective experiences [10]. Differential evolution (DE) operates through an evolutionary process in which new solutions are generated by combining existing population members using vector differences. This has demonstrated robust performance, particularly in continuous optimization problems [1]. Genetic algorithms (GA), one of the earliest and most widely studied metaheuristics, draw from the principles of natural selection and genetic evolution, whereby solution populations evolve through mechanisms of selection, crossover, and mutation [8]. The cuckoo search (CS) algorithm, inspired by the brood parasitism of cuckoo species, incorporates Lévy flights to balance local and global search capabilities [12]. Simulated annealing (SA), analogous to the physical annealing process in metallurgy, employs a temperature-controlled probabilistic acceptance mechanism that allows bypassing local optima by gradually focusing on promising regions [4]. These algorithms have established theoretical foundations and have been successfully applied across diverse optimization domains, contributing significantly to the advancement of metaheuristic optimization techniques.

2.2 Task-Scheduling Problem

Task scheduling represents a fundamental challenge in parallel computing systems, with the primary objective of efficiently allocating computational tasks across multiple processing units, to satisfy various constraints and optimization criteria. Task scheduling has garnered significant attention because of its critical role in high-performance computing, cloud computing, and distributed systems.

In its general form, the task scheduling problem can be formalized as follows: Given a set of tasks $T = \{t_1, t_2, ..., t_n\}$ and a set of processors $P = \{p_1, p_2, ..., p_m\}$, the goal is to find a mapping $\sigma : T \to P$ that minimizes the overall completion time (makespan) while adhering to the precedence constraints between tasks. These precedence constraints can be represented as a directed acyclic graph (DAG) $G = (V, E)$, where vertices V correspond to tasks, and edges E represent the dependencies between tasks.

The completion time of task t under schedule σ must consider both the processing and completion times of its predecessor tasks:

$$C(t,\sigma) = w(t) + \max_{p \in pred(t)} \{C(p,\sigma)\} \tag{1}$$

where $w(t)$ denotes the computational weight of task t, and $pred(t)$ represents the set of immediate predecessor tasks of t in DAG. The overall makespan of the schedule can then be expressed as:

$$C_{max}(\sigma) = \max_{t \in T} \{C(t,\sigma)\} \tag{2}$$

Several variants of the task-scheduling problem have been studied in [7]. The homogeneous variant assumes identical processing capabilities across all processors, whereas the heterogeneous variant considers processors with different computational capabilities. The heterogeneous task scheduling problem and its efficient parallelization are the focus of this study.

2.3 Ejection Chain

The *ejection chain method* represents an advanced neighborhood search technique first introduced by Glover [5] for solving complex combinatorial optimization problems. This sophisticated approach extends beyond simple neighborhood structures by creating a sequence of coordinated moves, where each move initiates a cascade of other moves through solution space. The distinctive feature of this method is its ability to explore complex compound moves while maintaining solution feasibility through carefully structured chains of operations.

The fundamental principle of ejection chains can be formalized through a sequence of moves that transform an initial solution x_0 into a final solution x_k through a series of intermediate states.

$$x_0 \xrightarrow{m_1} x_1 \xrightarrow{m_2} x_2 \xrightarrow{m_3} \cdots \xrightarrow{m_k} x_k \tag{3}$$

where each move m_i represents a compound operation that modifies multiple solution components simultaneously. The transformation at each step can be expressed as:

$$x_{i+1} = T_i(x_i, e_i, r_i) \tag{4}$$

where T_i represents the transformation operator; e_i denotes the ejection operation; and r_i represents the corresponding reception operation that maintains solution feasibility.

The effectiveness of the method stems from its structured approach to neighborhood exploration. Unlike traditional neighborhood search methods, which consider only single moves or simple combinations, ejection chains create sophisticated reference structures that guide the search through promising regions in solution space. These reference structures can be represented as:

$$R(x) = \{(e_1, r_1), (e_2, r_2), ..., (e_k, r_k)\} \tag{5}$$

where each pair (e_i, r_i) represents a valid ejection-reception combination that preserves solution feasibility.

The method has demonstrated effectiveness in solving complex scheduling and routing problems, where solution feasibility constraints often create challenging optimization landscapes. Yagiura et al. [14] successfully applied ejection chains to the generalized assignment problem (GAP), demonstrating their potential for handling complex resource allocation scenarios, which provides valuable insights into task scheduling problems.

2.4 Differential Evolution Approach for Task Scheduling Problem

Differential Evolution has emerged as an effective method for task scheduling optimization due to its robust exploration capabilities and ability to maintain diverse solution populations [9].

2.4.1 Solution Representation with SV and MV

The dual-vector encoding scheme uses complementary scheduling vector (SV) and matching vector (MV) representations [9]. The SV encodes a topological ordering of tasks that satisfies the precedence constraints defined in the task-dependency graph. For a task set $T = \{t_1, t_2, ..., t_n\}$, SV represents a permutation vector of length n where $SV[i]$ denotes the task scheduled at position i.

The MV component captures the processor assignments with $MV[i]$ representing the target processor for task t_i. In a system with m processors, $MV[i] \in \{0, 1, ..., m-1\}$, this encoding ensures unique processor assignment and enables the exploration of different allocation combinations.

The completion time calculation for schedules under this representation can be formalized as:

$$C_{max}(SV, MV) = \max_{p \in P}\{ \sum_{i:MV[i]=p} w(t_{SV[i]}) + \max_{j \in pred(i)} C(t_{SV[j]})\} \tag{6}$$

where $w(t_i)$ represents the computational weight of task i, and $pred(i)$ denotes the set of immediate predecessors of task i in the dependency graph. This formulation integrates both processor workload distribution and precedence constraints between tasks.

Discrete MV representation can be enhanced by transforming each element into a real number in the range $[0, 1)$. Under this transformation, the actual processor assignment for task i is determined by multiplying $MV[i]$ by the total number of available processors and taking the floor of the result. The processor assignment p_i for task i can be expressed as:

$$p_i = \lfloor MV[i] \cdot m \rfloor \tag{7}$$

where m is the total number of available processors. This representation maintains the benefits of discrete encoding while providing a continuous search space that better aligns with the numerical optimization capabilities of DE. This transformation enables a more effective exploration of processor assignments and facilitates adaptation to systems with varying numbers of processors.

The DE approach employs a population-based evolutionary approach that specializes in task scheduling optimization. In each generation, the algorithm creates candidate solutions through a combination of mutation and crossover operations based on SV and MV.

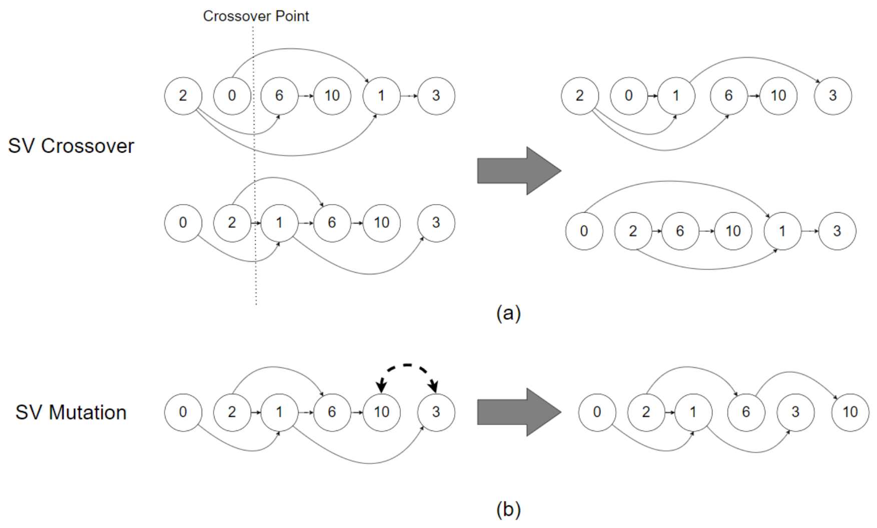


Figure 1: DE operations for task scheduling: (a) Scheduling vector crossover that preserves task dependencies; (b) Mutation operation with valid task shifts.

For each parent solution P_i, the algorithm selects three distinct solutions, P_{r1}, P_{r2}, and P_{r3}, from the current population to generate a mutant vector according to

$$V_i = P_{r1} + F \cdot (P_{r2} - P_{r3}) \tag{8}$$

where F is a scaling factor that controls the amplification of the differential variation. However, direct arithmetic operations cannot be applied to scheduling vectors because of their permutational nature. Instead, the algorithm measures the difference between solutions using the Ulam distance, which counts the minimum number of move operations required to transform one permutation into another.

After mutation, crossover operations are performed independently of the SV and MV components. For the scheduling vector, the crossover must preserve task dependencies while allowing for effective exploration of different task orderings. As shown in Fig. 1(a), the crossover operation maintains its feasibility by ensuring that all precedence relationships are satisfied in the resulting offspring.

The mutation operation illustrated in Fig. 1(b) randomly selects and shifts tasks within the valid ranges determined by the dependency constraints. This operation enables local adjustments to the schedule while ensuring that the solutions are feasible. The matching vector undergoes standard DE operations with modifications to ensure processor assignments remain within valid bounds.

Selection proceeds by comparing each candidate with its parent using a dominance-based criterion. A candidate replaces its parent if its fitness is better within the problem constraints. This process continues until a predefined number of generations is reached or the convergence criteria are met.

3 Proposed Method

This section presents our optimization approach for solving task scheduling problems, with a particular emphasis on decoupling SV and MV optimization and maintaining cooperative effects. However, previous approaches such as differential evolution [9] attempt to simultaneously optimize both components. Our approach draws inspiration from Yagiura et al.'s ejection chain methodology for the generalized assignment problem [14], further introducing several key innovations to address the challenges of dependency-constrained task scheduling.

The overall optimization process, which represents the basic algorithm, can be expressed as follows:

Algorithm 1 Overall optimization process

```
1: procedure OptimizeTaskSchedule($tasks, dependencies$)
2:     Initialize $SV$ by topologically sorting $tasks$ based on $dependencies$
3:     Initialize $MV$ randomly
4:     Initialize solution pool $P$ with size $k$        ▷ Maintain top $k$ solutions
5:     $initial\_solution \leftarrow (SV, MV)$
6:     $best\_makespan \leftarrow$ Evaluate($initial\_solution$)
7:     Add $initial\_solution$ to $P$
8:     while termination criterion not met do
9:         Select newest solution $(SV_{curr}, MV_{curr})$ from $P$
10:        $neighbors \leftarrow$ GenerateShiftNeighbors($SV_{curr}, MV_{curr}$)
11:        for each $(SV_{new}, MV_{new})$ in $neighbors$ do
12:            $MV_{opt} \leftarrow$ OptimizeMV($SV_{new}, MV_{new}$)
13:            $makespan \leftarrow$ Evaluate($(SV_{new}, MV_{opt})$)
14:            if $makespan$ better than the worst in $P$ then
15:                Update $P$ with $(SV_{new}, MV_{opt})$
16:            end if
17:        end for
18:        $(SV_{long}, MV_{long}) \leftarrow$ LongTermShift($SV_{curr}, MV_{curr}$)
19:        $makespan \leftarrow$ Evaluate($(SV_{long}, MV_{long})$)
20:        if $makespan$ better than the worst in $P$ then
21:            Update $P$ with $(SV_{long}, MV_{long})$
22:        end if
23:    end while
24:    ↵ best solution from $P$
25: end procedure
```

The evaluation function computes the overall makespan of a given schedule by considering both the task dependencies and processor assignments. The OptimizeMV function focuses on finding the optimal matching vector for tasks while maintaining their current ordering in the scheduling vector. In the following subsections, the key components of this algorithm are detailed, including the shift move operation, waiting time-based shift neighbor, and long-term shift mechanism for escaping local optima.

3.1 Shift Move in SV

The *shift move* operation serves as the fundamental building block of the proposed task scheduling optimization approach, which is designed to promote cooperative interactions between task ordering and processor assignment decisions. Unlike mutation operators that modify SV and MV independently, *shift move* provides a targeted modification mechanism that considers both task dependencies and the impact of potential processor assignments.

A *shift move* operation takes the source task position and target position in the current SV, relocating the task from its source position to the target position while maintaining the SV sequence. Formally, given a scheduling vector $SV = [t_1, t_2, ..., t_n]$ where each t_i represents a task ID, and n is the total number of tasks, a *shift move* relocates a task from position p_{src} to position p_{target} in the sequence.

The operation must track task dependencies to identify the constraint violations introduced by the move. For each task being moved, its latest predecessor position p_{pred} must be identified based on the task dependency graph. After performing the shift operation, the algorithm calculates which tasks, if any, have their dependency constraints violated, by examining the task positions between the target and source positions in the modified sequence.

To formalize this, we use the following procedure to execute a *shift move*:

Algorithm 2 Shift Move Operation

```
1:  procedure ShiftMove($scheduler, SV, p_{src}, p_{target}$)
2:      Remove task from position $p_{src}$ in $SV$
3:      Insert task at position $p_{target}$ in $SV$
4:      $invalid \leftarrow \emptyset$                 ▷ Track tasks with violated dependencies
5:      for each position $i$ in $SV$ do
6:          if Dependencies violated for task at $SV[i]$ then
7:              $invalid \leftarrow invalid \cup \{i\}$
8:          end if
9:      end for
10:     return $invalid$                 ▷ Return positions of tasks with violated dependencies
11: end procedure
```

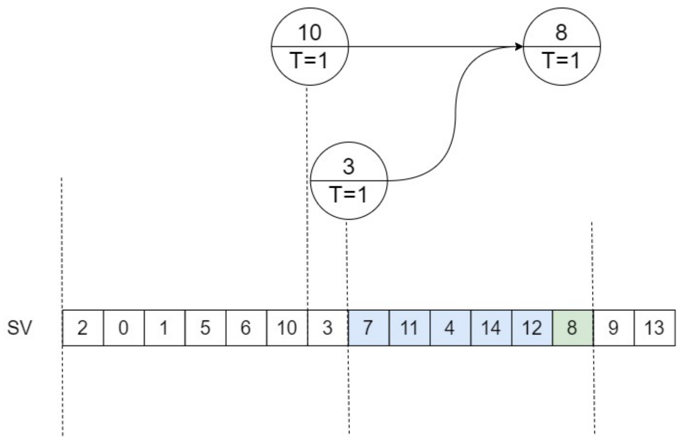


Figure 2: Shift move operation

For example, in Fig. 2, consider a task graph in which Task 8 has Tasks 3 and 10 as predecessors, with the current SV = [2,0,1,5,6,10,3,7,11,4,14,12,8,9,13]. A valid *shift move* can relocate task 8 to any position after tasks 3 and 10, such as immediately after task 3, producing SV = [2,0,1,5,6,10,3,8,7,11,4,14,12,9,13].

The *shift move* operation provides several advantages over traditional genetic operators by enabling fine-grained control over task reordering while explicitly tracking dependency violations. This operation allows detailed local searches while enabling significant schedule restructuring through a series of moves. The *shift move* operation forms the foundation of more advanced search strategies such as shift neighbor and long-term shifts, as discussed in the following sections.

3.2 Shift Neighbor

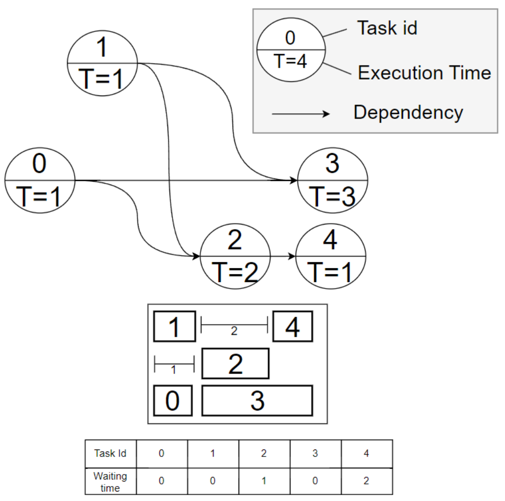
The *shift neighbor* operation extends the basic *shift move* mechanism to explore a comprehensive neighborhood of potential task schedule modifications. This operation systematically generates and evaluates multiple candidate solutions by applying valid *shift moves* to the current SV.

3.2.1 Waiting Time

To effectively identify the tasks that would benefit the most from rescheduling, we introduce a waiting time metric that is used to select target tasks for the *shift neighbor* operation as shown in Fig. 3. Given the current solution represented by SV and MV, the *shift neighbor* operation first identifies a target task t_{target} for potential relocation. The selection of t_{target} is based on the waiting time metric, which measures delays in task execution arising from dependency constraints and resource availability. Tasks with longer waiting times represent scheduling inefficiencies that can potentially be improved through relocation. For each task t_i, the waiting time w_i is calculated as follows:

$$w_i = \text{start}_i - \max(\text{finish}_{\text{pred}(i)}, \text{ready}_{\text{core}(i)}) \tag{9}$$

where $start_i$ is the start time of task i; $finish_{pred(i)}$ is the maximum completion time among predecessor tasks; and $ready_{core(i)}$ indicates when the assigned processor core becomes available.



Task Id	0	1	2	3	4
Waiting time	0	0	1	0	2

Figure 3: Example of waiting time calculation in task scheduling

3.2.2 Shift Neighbour Generation

The algorithm for generating and evaluating the shift neighborhood can be expressed as follows:

Algorithm 3 Shift Neighbour Generation

```
 1: procedure GenerateShiftNeighbours($SV, MV$)
 2:     $t_{target} \leftarrow$ SelectTargetTask($SV, MV$)
 3:     $p_{curr} \leftarrow$ current position of $t_{target}$ in $SV$
 4:     $p_{pred} \leftarrow$ position of latest predecessor of $t_{target}$
 5:     $neighbours \leftarrow \emptyset$
 6:     for $p_{new}$ from $(p_{pred} + 1)$ to $(p_{curr} - 1)$ do
 7:         $SV_{new} \leftarrow$ Copy($SV$)
 8:         ShiftMove($SV_{new}, t_{target}, p_{new}$)
 9:         $MV_{new} \leftarrow$ OptimizeMV($SV_{new}, MV$)
10:         $neighbours \leftarrow neighbours \cup \{(SV_{new}, MV_{new})\}$
11:     end for ↵ neighbours
12: end procedure
13: procedure SelectTargetTask($SV, MV$)
14:     Calculate waiting times $w$ for all tasks
15:     for each task $t_i$ in $SV$ do
16:         if $w_i > threshold$ then
17:             candidates $\leftarrow$ candidates $\cup \{t_i\}$
18:         end if
19:     end for ↵ task with maximum waiting time from candidates
20: end procedure
```

Each generated neighbor solution consists of a modified SV and an optimized MV. The optimization of MV for each neighbor is performed using a metaheuristic solver, such as simulated annealing, which efficiently explores the processor assignment space while keeping the task ordering fixed.

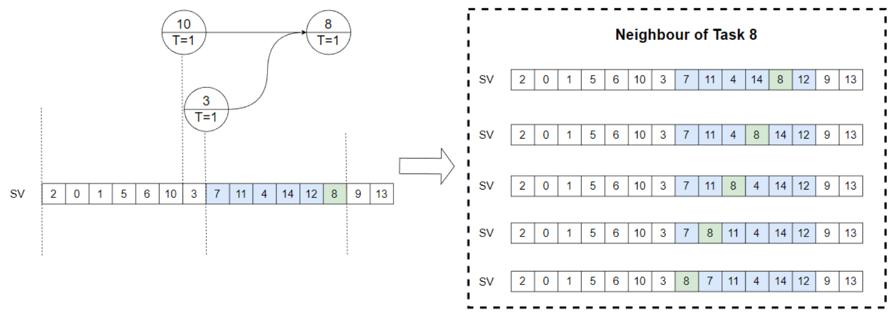

10 T=1
8 T=1
3 T=1
SV 2 0 1 5 6 10 3 7 11 4 14 12 8 9 13
Neighbour of Task 8
SV 2 0 1 5 6 10 3 7 11 4 14 8 12 9 13
SV 2 0 1 5 6 10 3 7 11 4 8 14 12 9 13
SV 2 0 1 5 6 10 3 7 11 8 4 14 12 9 13
SV 2 0 1 5 6 10 3 7 8 11 4 14 12 9 13
SV 2 0 1 5 6 10 3 8 7 11 4 14 12 9 13

Figure 4: Generation of valid neighbours for Task 8

Figure 4 illustrates the neighbor generation process for Task 8, which has two predecessor tasks (Tasks 3 and 10) with an execution time T=1. Given the initial scheduling vector, the algorithm identifies valid positions for Task 8 that adhere to the precedence constraints. The right panel shows five possible neighbor solutions created by shifting Task 8 to different positions, demonstrating how the *shift neighbor* operation systematically explores the solution space while preserving the dependency relationships.

A key feature of the *shift neighbor* operation is its ability to maintain solution feasibility throughout the neighborhood exploration. This is achieved via two mechanisms:

$$\begin{aligned} p_{new} &> p_{pred} & \forall\, p_{new} \text{ considered for } t_{target} \\ deps(t_i) \cap succ(t_{target}) &= \emptyset & \forall\, t_i \text{ between } p_{new} \text{ and } p_{curr} \end{aligned} \tag{10}$$

where $deps(t_i)$ represents the set of dependencies for task i, and $succ(t_{target})$ represents the set of successor tasks for the target task.

The size of the generated neighborhood is bounded by $O(n)$ where n is the number of tasks, as each target task can be shifted to at most $n-1$ new positions. However, the actual neighborhood size is typically much smaller due to dependency constraints and tabu list restrictions, making the operation computationally efficient even for large problem instances.

3.3 Long-Term Shift

Although the *shift neighbor* operation maintains strict feasibility during exploration, this constraint can limit the algorithm's ability to discover beneficial cooperative interactions between solution components. The long-term shift operation addresses this limitation by temporarily allowing infeasible solutions through a controlled violation of dependency constraints, followed by a repair mechanism that restores feasibility while preserving the beneficial cooperative effects.

Algorithm 4 Long Term Shift Operation

```
1: procedure LongTermShift($SV, MV$)
2:     $t_{target} \leftarrow$ SelectTargetTask($SV, MV$)
3:     $p_{curr} \leftarrow$ current position of $t_{target}$
4:     $K \leftarrow$ violation step
5:     $p_{pred} \leftarrow$ position of latest predecessor of $t_{target}$
6:     $SV_{temp} \leftarrow$ Copy($SV$)
7:     $invalid\_tasks \leftarrow$ ShiftMove($SV_{temp}, t_{target}, p_{pred} - K$)    ▷ Intentionally violate constraints
8:     $SV_{new} \leftarrow$ RepairDependencies($SV_{temp}, invalid\_tasks$)
9:     $MV_{new} \leftarrow$ OptimizeMV($SV_{new}$) ↩($SV_{new}, MV_{new}$)
10: end procedure
```

The long-term shift operation represents an advancement over basic shift moves by enabling the exploration of otherwise inaccessible regions in solution space. The algorithm accomplishes this via a two-phase process: temporary constraint violation and dependency repair.

During the *temporary constraint violation* phase, the violation step K determines how far backward the target task can be moved beyond its dependency constraints. This parameter controls the trade-off between exploration potential and repair complexity. The operation creates a temporary schedule SV_{temp} by moving the target task to position $p_{pred} - K$, thereby deliberately violating the dependency constraints to explore new schedule configurations.

The algorithm then enters the *dependency repair* phase. These invalid tasks are collected and processed through a repair mechanism that restores feasibility while attempting to preserve as many of the intended schedule modifications as possible. Finally, the matching vector MV is re-optimized to account for the new task ordering, ensuring efficient processor utilization under the modified schedule.

This approach differs from shift neighbors by allowing temporary constraint violations to discover high-quality solutions that would be unreachable through feasibility-preserving moves alone. The repair process ensures that the operation remains manageable while expanding the searchable solution space.

3.3.1 Temporary Constraint Violation

The long-term shift operation begins by identifying the target task t_{target} with the highest waiting time, as in the *shift neighbor* operation. However, instead of limiting the move to positions that maintain feasibility, the operation allows the task to be shifted to positions that temporarily violate the dependency constraints.

The temporary violation of constraints creates a state in which dependency relationships are broken.

$$\exists\, t_i, t_j \in SV_{temp} : (t_i, t_j) \in E \wedge pos(t_j) < pos(t_i) \tag{11}$$

where E represents the set of edges in the task dependency graph, and $pos(t_i)$ denotes the position of task i in the SV.

3.3.2 Dependency Repair Process

The repair process involves a series of corrective shifts that restore feasibility while maintaining as many of the potential benefits as possible from the original move. This is achieved through an iterative repair mechanism:

Algorithm 5 Dependency repair process

```
1: procedure RepairDependencies($SV, invalid\_tasks$)
2:     while $invalid\_tasks$ not empty do
3:         $t_i \leftarrow$ SelectNextInvalidTask($invalid\_tasks$)
4:         $p_{valid} \leftarrow$ position of latest predecessor of $t_i$
5:         while $p_{valid} \in invalid\_tasks$ do
6:             $t_i \leftarrow p_{valid}$
7:             $p_{valid} \leftarrow$ position of latest predecessor of $p_{valid}$
8:         end while
9:         $affected\_tasks \leftarrow$ ShiftMove($SV, t_i, p_{valid}$)
10:        UpdateInvalidTasks($invalid\_tasks, affected\_tasks$)
11:    end while ↵ $SV$
12: end procedure
```

The while loop in the repair procedure plays a crucial role in handling cascading dependency violations that may occur during the repair process. This mechanism is essential because when attempting to repair an invalid task by moving it to the earliest valid predecessor position, this position itself might correspond to another invalid task.

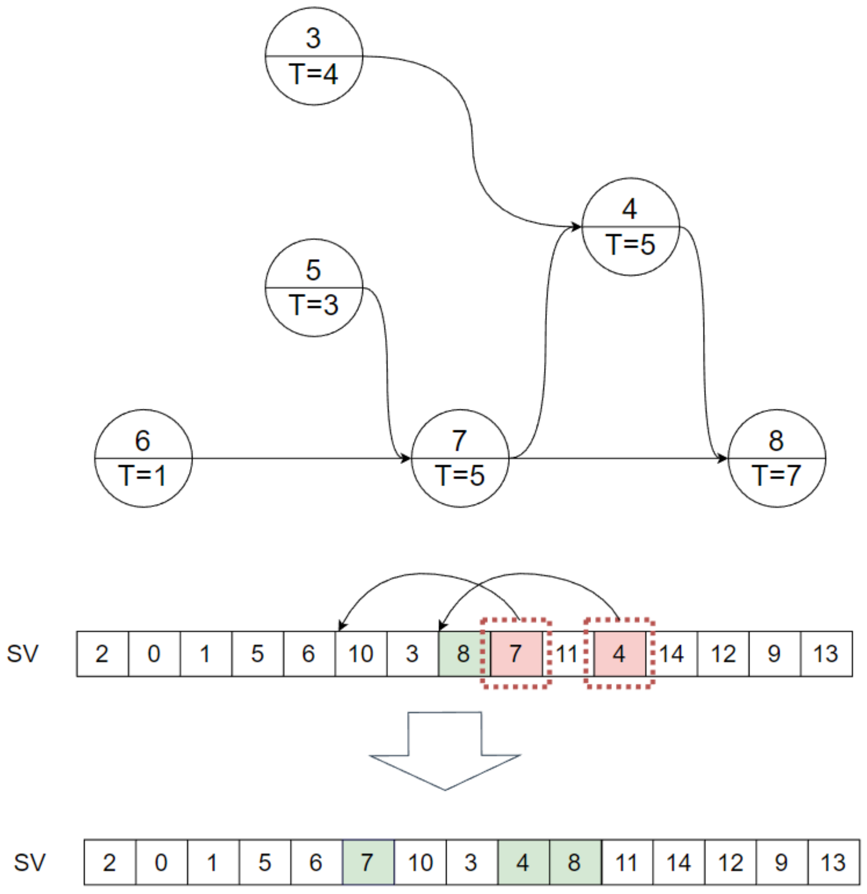


Figure 5: Example of long-term shift operation with repair process

Figure 5 illustrates the long-term shift operation. In a task graph, the vertices represent tasks, whereas the edges indicate dependencies. When Tasks 7 and 11 violate the dependency constraints in the initial scheduling vector, the repair process relocates these tasks to maintain their feasibility. The final scheduling vector shows the results after the repair process, with tasks reordered to satisfy all dependency relationships while preserving the beneficial aspects of the initial shift operation.

This mechanism is important for maintaining solution feasibility when dealing with complex dependency structures, with multiple tasks possibly requiring repositioning to accommodate a single shift operation.

3.4 Search Strategy

The search strategy integrates the previously described components into a cohesive strategy that effectively explores both the task ordering and processor assignment spaces. Fig. 6 illustrates the complete search mechanism of the proposed method.

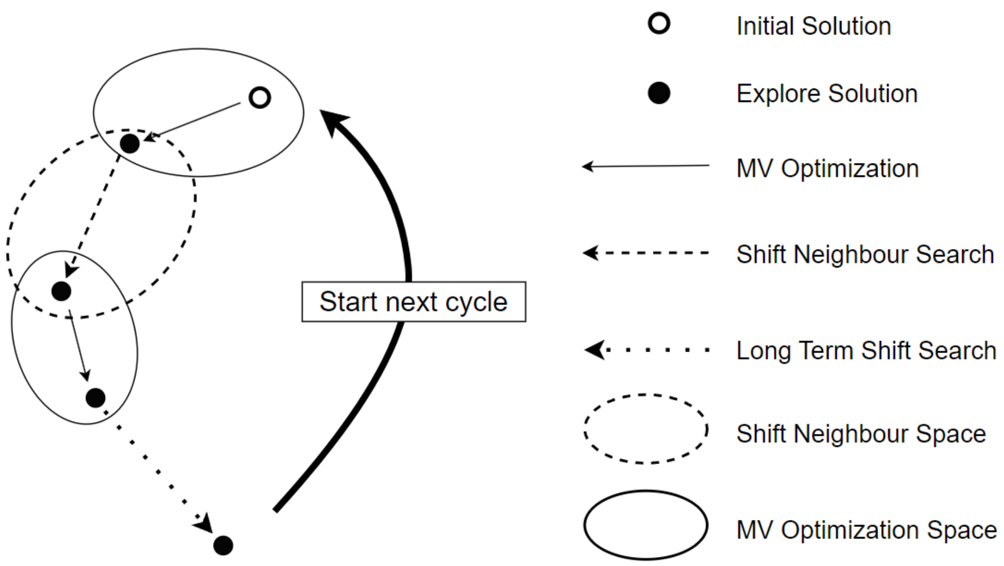


Figure 6: Overall optimization process of the proposed method

The process begins with an initial solution. For each explored task ordering, the algorithm performs MV optimization to determine the optimal processor assignments within the MV optimization space.

When shift neighborhood search explores all the solutions in the neighborhood, a long-term shift search is activated. This mechanism enables exploration to break free from local optima by allowing temporary constraint violations. The long-term shift operation identifies distant but promising regions within the solution space that are inaccessible through conventional neighborhood moves.

The search process continues cyclically, with each cycle beginning with the best solution obtained in the previous iteration. This cyclic nature allows the algorithm to progressively explore different regions of the solution space while maintaining the best solutions. The combination of local exploration through *shift neighbor* search and broader exploration via long-term shift search creates a balanced approach that can handle both simple and complex task-scheduling scenarios.

Through this integrated approach, the algorithm maintains a dynamic balance between intensification, which is achieved through MV optimization, *shift neighbor* search, and diversification, which is facilitated by long-term shift operations. This balance enables the method to navigate the complex solution space of task-scheduling problems while avoiding premature convergence to suboptimal solutions.

3.5 Implementation of Parallel Optimization

The overall optimization process described at the beginning of this section reveals several opportunities for parallelization in the iterative evaluation. In each iteration, the process performs multiple MV optimizations for different candidate solutions generated through a *shift neighbor* operation. As these optimizations are independent of each other, they create a natural opportunity for parallel execution. The long-term shift operation can also potentially run concurrently with ongoing neighborhood explorations.

Given these characteristics, we propose two parallel implementation strategies: an MV parallel approach that focuses on concurrent optimization of matching vectors and an event-driven approach that extends this parallelism through asynchronous execution patterns. Although the MV parallel approach provides a straightforward method for distributing computational load across multiple threads, the event-driven implementation introduces sophisticated mechanisms to manage thread allocation and facilitate cooperative solution space exploration.

3.5.1 MV Parallel Approach

The MV parallel approach exploits the independence of processor core assignment optimization for different scheduling vectors. Given a set of candidate SVs generated through shift neighbors or long-term shift operations, the MV optimization for each candidate can be performed concurrently.

For each scheduling vector SV_i, the corresponding matching vector MV_i must be optimized to minimize the makespan while maintaining the task order specified by SV_i. This optimization can be formalized as

$$\begin{aligned} MV_i^* &= \arg\min_{MV_i} \text{makespan}(SV_i, MV_i) \\ &\text{subject to:} \quad 0 \leq MV_i[j] < 1 \quad \forall j \in \{1, \ldots, |T|\} \end{aligned} \tag{12}$$

where T represents the set of tasks, and $makespan(SV_i, MV_i)$ computes the schedule length for the given combination.

The parallel optimization process initially follows a fork-join pattern. For a set of k candidate solutions, the process can be described by the following transformation:

$$\{SV_1, \ldots, SV_k\} \xrightarrow{\text{parallel optimization}} \{(SV_1, MV_1^*), \ldots, (SV_k, MV_k^*)\} \tag{13}$$

The implementation utilizes a thread-pool architecture with a configurable number of worker threads, $n_{threads}$. Each worker thread independently executes a metaheuristic solver such as simulated annealing to optimize the MV for its assigned SV. However, this fork-join pattern has several limitations:

1. Processing time variations across threads lead to uneven load distribution, resulting in suboptimal resource utilization.
2. In *shift chain* operations, the requirement to wait for all neighborhood solutions before proceeding to the next iteration creates idle periods.
3. During *shift chain* computations, other threads remain inactive while waiting for the chain operation to complete.

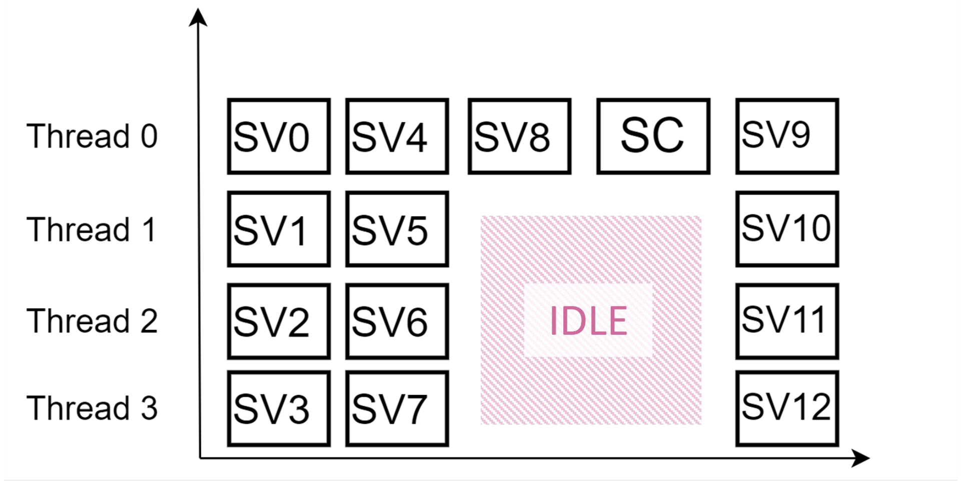


Figure 7: MV Parallel Optimization

To illustrate the resource utilization issues, consider a scenario with a 4-thread solver processing an initial set of nine SVs generated from the shift neighborhood computation, as shown in Fig. 7. The execution timeline reveals two critical inefficiencies:

1. During the final phase of SV processing, three out of four threads enter an IDLE state, as the remaining workload (single SV) cannot utilize the available parallel resources.
2. Following the completion of all SV computations, a long-term shift chain (SC) operation must be executed sequentially on a single thread, forcing all other threads into an IDLE state.

This pattern of forced idling leads to underutilization of computational resources. The problem is acute during the *shift chain* phase, where despite having multiple capable processing threads available, the sequential nature of the operation prevents parallel execution.

3.5.2 Event-Driven Approaches

The event-driven approach leverages the inherent independence between *shift chain* operations and MV optimization processes.

The fundamental interaction pattern consists of four primary message flows through a central publish-subscribe server:

1. SV publication from shift chain.
2. SV reception and MV optimization in solver.
3. optimized result publication from solver.
4. result collection in shift chain.

To address the resource utilization issues present in fork-join patterns, the architecture is enhanced using an observation system, as illustrated in Fig. 8. The observation system introduces two distinct feedback loops: a solver loop for direct computational flow and an observation loop for resource monitoring.

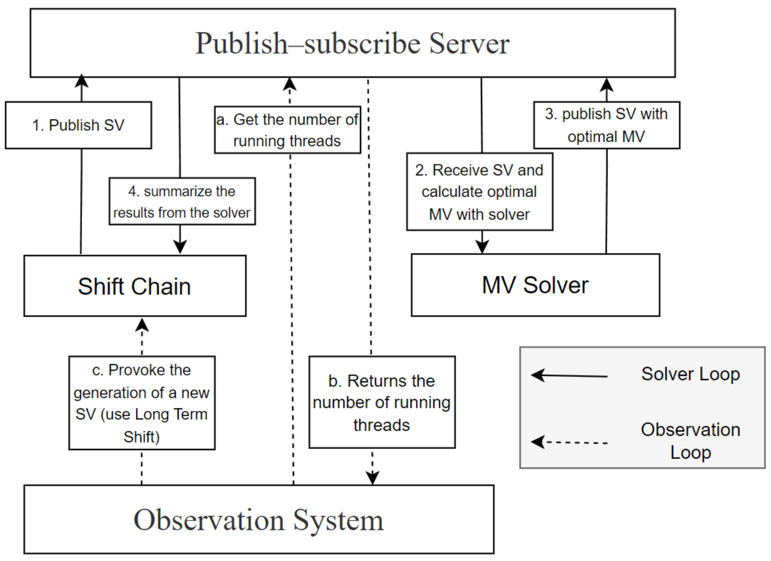

Publish–subscribe Server
1. Publish SV
a. Get the number of running threads
3. publish SV with optimal MV
4. summarize the results from the solver
2. Receive SV and calculate optimal MV with solver
Shift Chain
MV Solver
c. Provoke the generation of a new SV (use Long Term Shift)
b. Returns the number of running threads
Solver Loop
Observation Loop
Observation System

Figure 8: Event-driven architecture with observation system

The observation system monitors both the active thread count and pending SV quantities:

$$O(t) = \{n_{threads}(t), n_{pending}(t)\} \tag{14}$$

where $O(t)$ represents the set of observations at time t; $n_{threads}(t)$ denotes the number of active threads at time t; and $n_{pending}(t)$ denotes the number of pending SV at time t.

Resource availability is determined based on the following conditions:

$$n_{threads}(t) < n_{max} \vee n_{pending}(t) < \text{Threshold} \tag{15}$$

where n_{max} represents the maximum allowable thread count, and Threshold defines the critical level for pending tasks. This equation establishes two criteria for initiating new computations: either the system must have available threads, or the number of pending tasks must be below the specified threshold.

The system retrieves the current thread count using an observation loop. Subsequently, it assesses the resource availability. If resources are deemed sufficient, the long-term shift process is triggered. This sequence ensures that new shift-chain computations are efficiently initiated based on the current computational load and available resources.

Essentially, the system monitors its ongoing processes, evaluates whether there is sufficient capacity to handle more computations, and initiates additional tasks when appropriate.

This event-driven architecture addresses resource utilization issues through asynchronous event processing, eliminating forced synchronization points and allowing the concurrent execution of shift chain and MV optimization operations. The monitoring system ensures dynamic workload distribution, preventing extended idle periods by initiating new computations when resources become available.

Figure 9 illustrates the improved resource utilization achieved by the event-driven approach. In contrast to the previous fork-join pattern, in which threads remain idle during shift chain operations, the event-driven system maintains continuous thread activity. The SC is seamlessly integrated with other MV optimization tasks, eliminating the previously observed extended idle periods. This dynamic scheduling allows the processing of additional SVs (SV7-SV13)

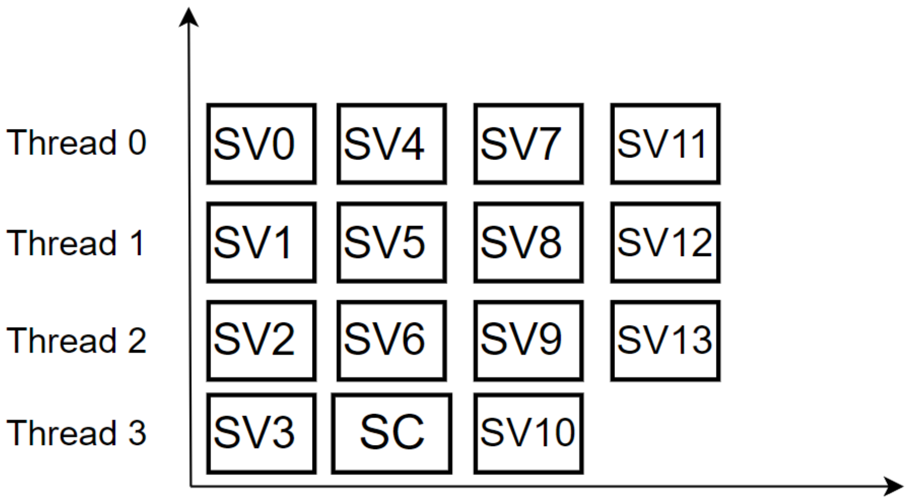


Figure 9: Ideal thread utilization achieved through event-driven approach

concurrently with shift chain operations, demonstrating enhanced resource utilization compared with the fork-join implementation shown in Fig. 7.

4 Evaluation

4.1 MV Optimization

Experiments were conducted to evaluate how different metaheuristic algorithms leverage cooperative search patterns when optimizing the matching vector while keeping the scheduling vector fixed in Fig. 10. Six algorithms were compared: SA , DE (population size 50), GA (population size 50), CS (population size 25), PSO (particle size 80), and ACO (population size 50). The evaluations used a problem instance from the benchmark dataset [13] with 300 tasks scheduled across four CPU cores and a theoretical optimal makespan of 448.

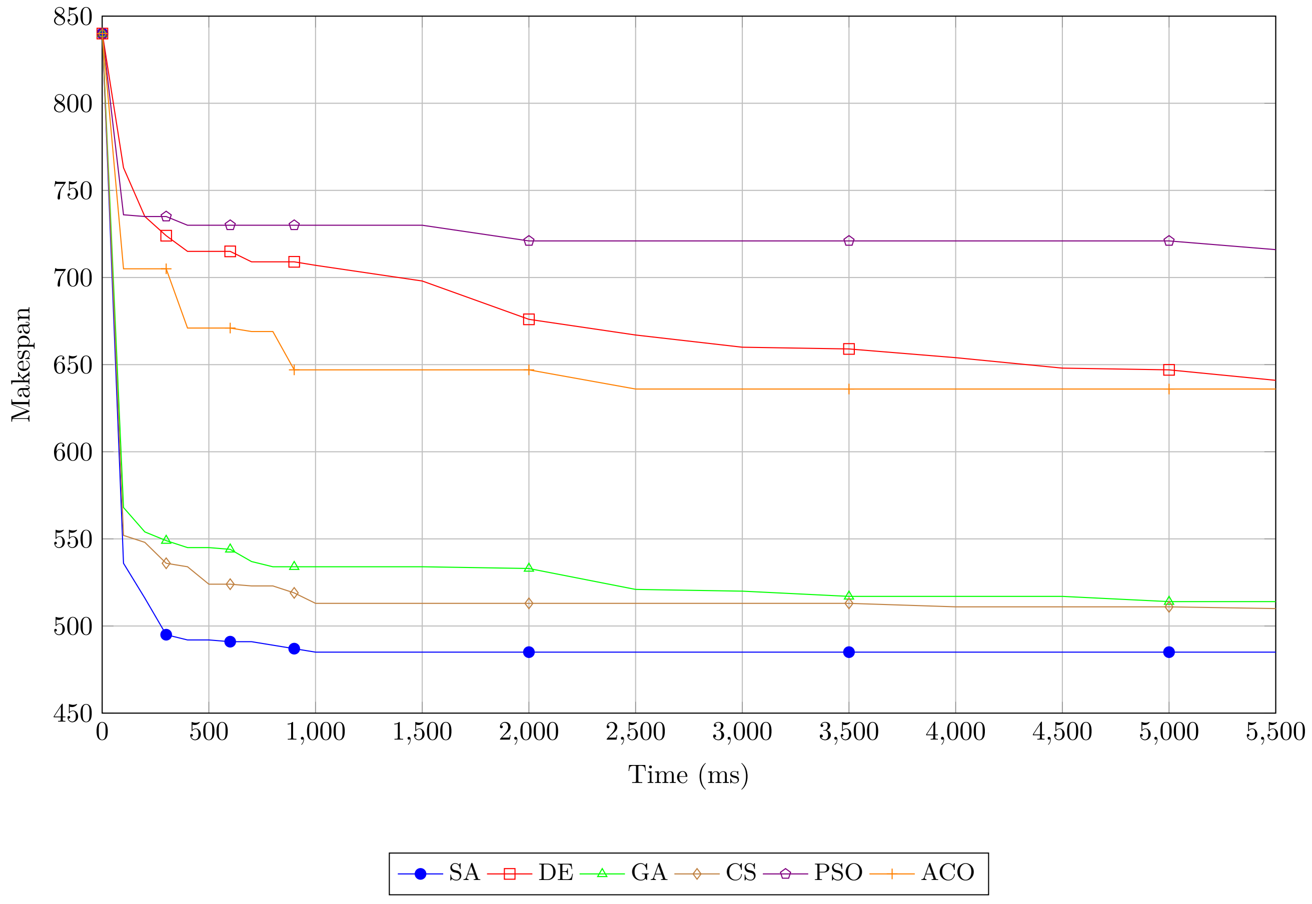


Figure 10: Convergence comparison of metaheuristic algorithms for MV optimization

The SA demonstrated the best performance, achieving the lowest makespan value of 485 within 1000 ms. This represents a significant improvement over other methods. The GA and CS showed similar performance patterns, converging to makespan values of 514 and 510, respectively. PSO demonstrated the slowest convergence, stabilizing at a relatively high makespan of 716, whereas ACO plateaued at 636.

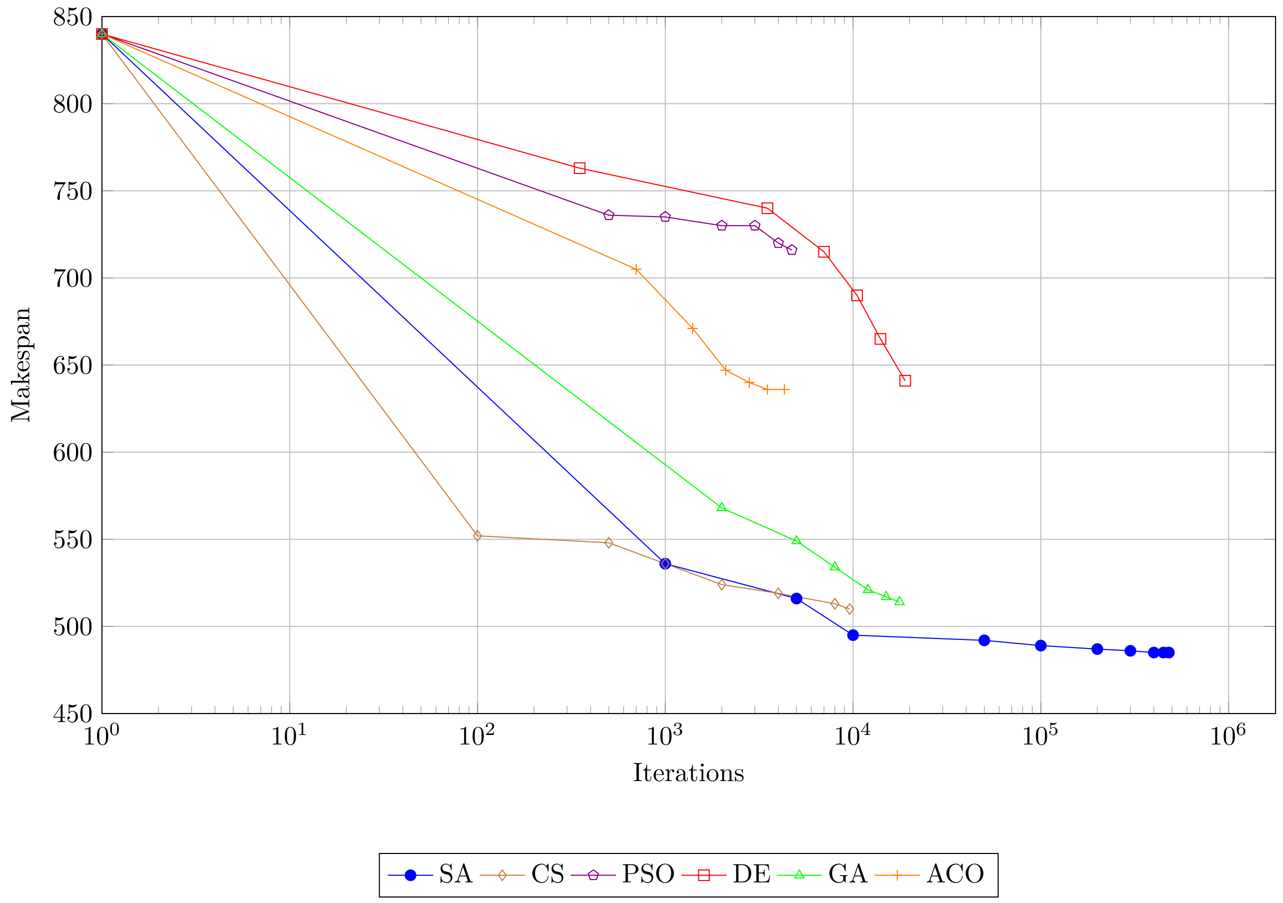


Figure 11: Relationship between makespan and iteration count for different metaheuristic algorithms

Figure 11 shows the convergence behavior across iteration counts for the different metaheuristic approaches. The SA required the highest number of iterations (480,598). Notably, CS demonstrated efficient early stage convergence, reaching competitive solutions in 9601 iterations with a final makespan of 510. The population-based approaches, DE and GA, showed similar iteration counts (18,957 and 17,693, respectively). PSO and ACO completed fewer iterations (4,721 and 4,309) owing to their complex update mechanisms, with both methods showing early plateaus in their convergence behavior.

These performance differences can be attributed to computational complexity and population management strategies. Population-based algorithms show slower convergence despite their explicit cooperative mechanisms through population diversity. This suggests that maintaining population diversity may hinder effective cooperation in the MV optimization context, with rapid local adjustments based on immediate feedback proving beneficial.

Based on these results, we selected simulated annealing as the primary MV optimization algorithm for our proposed method, given its solution quality and convergence speed.

4.2 Comparison with DE Approach

To evaluate the proposed *shift chain* method using eight threads on an AMD 7840HS processor against the baseline DE approach, we conducted experiments using publicly available benchmark datasets with 50, 300, and 1000 tasks, each containing 100 problem instances [13]. The optimality gap was calculated as:

$$\text{Gap from OPT} = \frac{\text{Algorithm Solution} - \text{Optimal Solution}}{\text{Optimal Solution}} \times 100\% \tag{16}$$

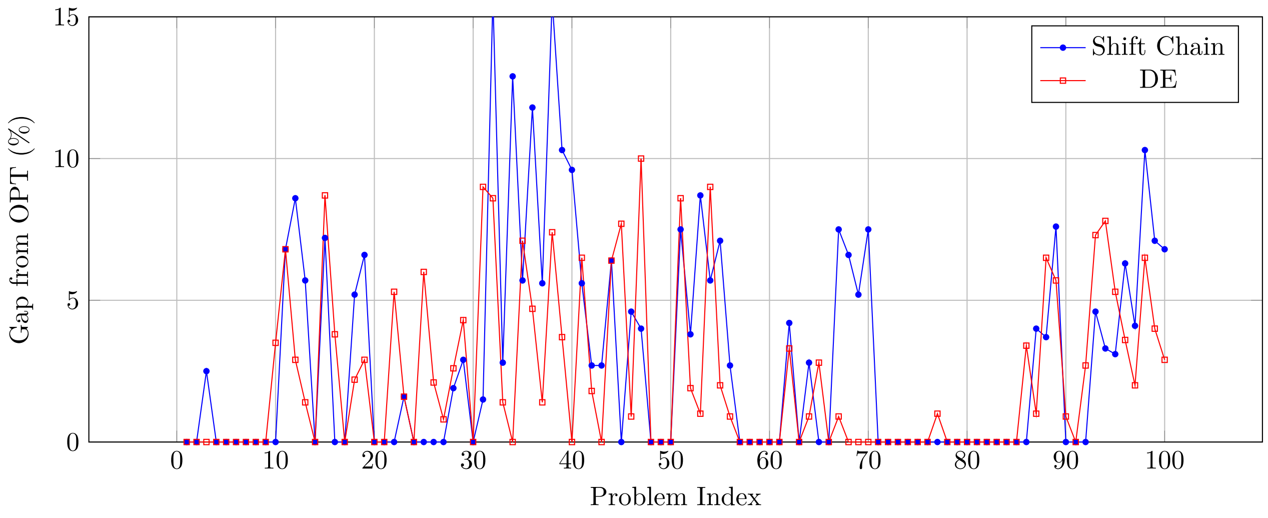


Figure 12: Optimality gap comparison for 50-task problems

For the 50-task dataset in Fig. 12, both algorithms demonstrated comparable performance levels, which were attributed to the smaller solution space and fewer opportunities for optimization. However, the advantages of the proposed method are evident for the 300-task and 1000-task datasets in Fig. 13 and Fig. 14, respectively. The *shift chain* method consistently maintained low optimality gaps, whereas the DE approach showed a significant degradation in solution quality, particularly in problems with complex dependency structures.

Analysis of the results reveals that the ability of the *shift chain* method to handle task dependencies directly through its specialized mechanisms allows a more effective exploration of the solution space while maintaining feasibility. The event-driven parallel implementation enables the efficient utilization of computational resources, which is particularly beneficial for larger problem instances.

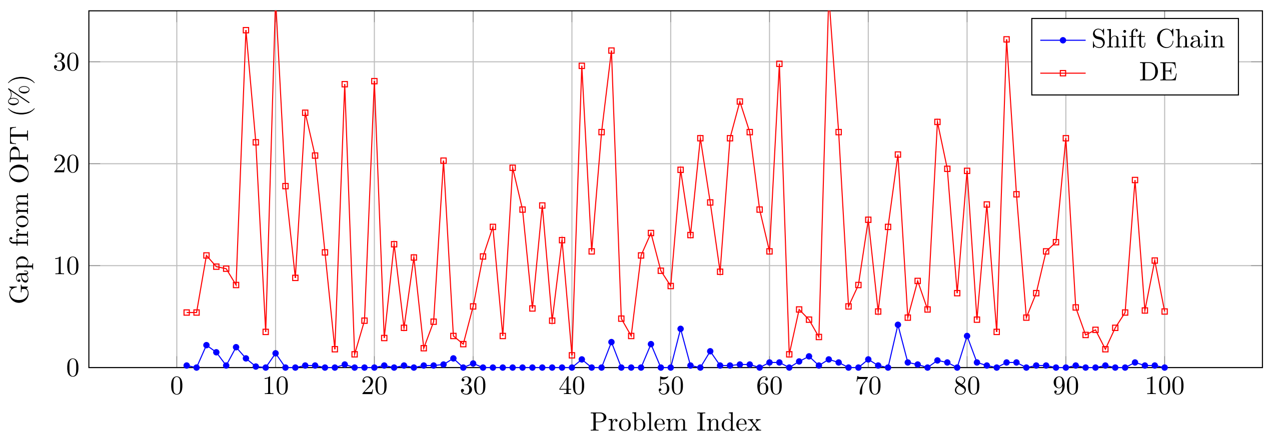


Figure 13: Optimality gap comparison for 300-task problems

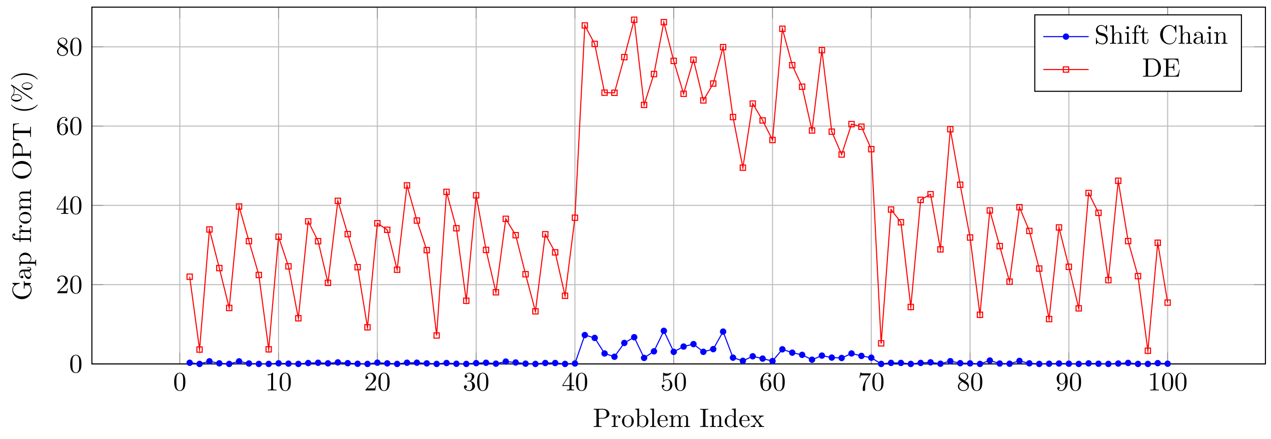


Figure 14: Optimality gap comparison for 1000-task problems

4.3 Parallel Shift Chain Performance

We evaluated four distinct approaches, random SV parallel execution and three shift chain methods: event-driven, fork-join, and series-execution. The test case involved scheduling 300 tasks across four CPU cores, with the experiments running for 230 s on an AMD 7840HS processor using eight threads.

The event-driven approach achieved a theoretical optimal makespan of 448 within 165 s, significantly outperforming the other implementations. The Fork-Join implementation exhibited a slower improvement after the first 50 s, achieving a final makespan of 454. The series-execution approach demonstrated the slowest convergence rate among the structured approaches, with a makespan of 471.

The random SV parallel execution, despite utilizing all eight threads, quickly stagnated at a makespan of 476, indicating that merely parallelizing the search process without structured exploration strategies is insufficient for effective optimization.

These results validate the effectiveness of the event-driven parallel implementation strategy. The success of this approach stems from its ability to maintain cooperative effects through complementary mechanisms working in concert, whereas the event-driven implementation facilitates effective cooperation between different search threads through timely information sharing.

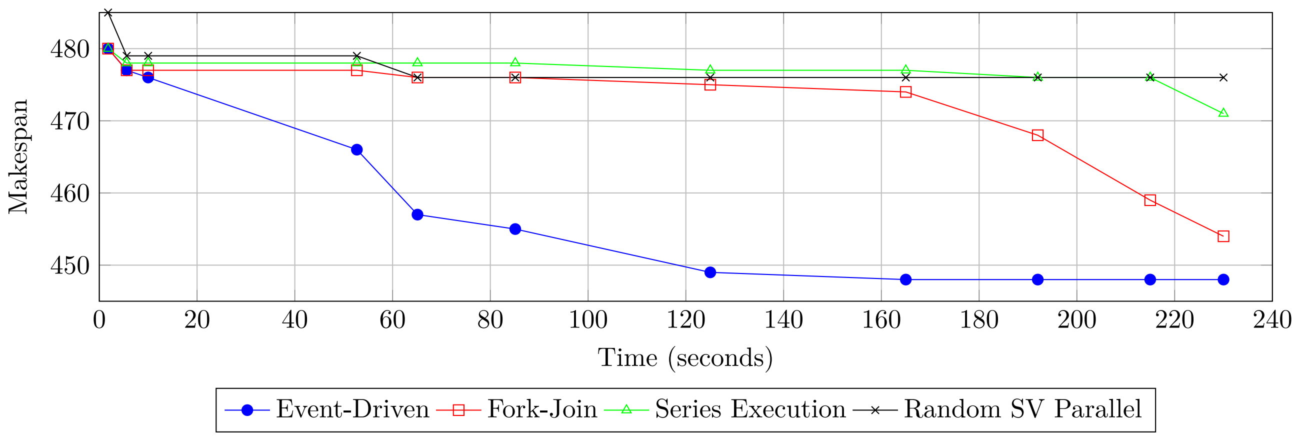


Figure 15: Convergence comparison of different parallel implementation strategies

5 Conclusion

This study introduced a parallel metaheuristic for task scheduling optimization, with a focus on maintaining cooperative effects while utilizing parallel computing resources. Our experimental evaluation yielded several significant findings.

Through experiments with different problem scales (50, 300, and 1000 tasks), we demonstrated that the effectiveness of our method increases with problem complexity.

The MV optimization experiments revealed that simpler algorithms such as simulated annealing can outperform complex population-based methods when cooperative effects are properly managed, suggesting that effective cooperation benefits more from rapid local adjustments than from elaborate population dynamics.

The parallel implementation comparison showed that the event-driven approach achieves superior performance by effectively balancing cooperative mechanisms with computational resource utilization, consistently reaching optimal or near-optimal solutions for large-scale problems.

These results demonstrate that successful task scheduling optimization relies not only on sophisticated algorithms but also on carefully designed mechanisms that preserve cooperative effects throughout the optimization process.

5.1 Limitations

Although the proposed method demonstrated promising results, limitations were encountered when handling task sets with minimal waiting times. In such cases, the effectiveness of the shift-chain mechanism may be reduced as its core operation relies on identifying and optimizing tasks with substantial waiting times. When most tasks have minimal or negligible waiting periods, the algorithms ability to discover meaningful improvements through task reordering is limited. This suggests that alternative optimization strategies may be more suitable for tightly coupled task sets with minimal scheduling flexibility.

Conflicts of Interest

The authors declare no conflicts of interest.

References

[1] Ahmad, M. F., Isa, N. A. M., Lim, W. H., & Ang, K. M. (2022). Differential evolution: A recent review based on state-of-the-art works. *Alexandria Engineering Journal*, 61(5), 3831-3872. https://doi.org/10.1016/j.aej.2021.09.013

[2] Chen, X., Yu, L., Wang, T., Liu, A., Wu, X., Zhang, B., Lv, Z., & Sun, Z. (2020). Artificial intelligence-empowered path selection: A survey of ant colony optimization for static and mobile sensor networks. *IEEE Access*, 8, 71497-71510. https://doi.org/10.1109/ACCESS.2020.2984329

[3] Crainic, T. G. (2019). Parallel metaheuristics and cooperative search. In *International Series in Operations Research and Management Science* (Vol. 272, pp. 419-451). Springer. https://doi.org/10.1007/978-3-319-91086-4_13

[4] Du, K. L., & Swamy, M. N. S. (2016). *Search and optimization by metaheuristics: Techniques and algorithms inspired by nature.* Springer. https://doi.org/10.1007/978-3-319-41192-7

[5] Glover, F. (1996). Ejection chains, reference structures, and alternating path methods for traveling salesman problems. *Discrete Applied Mathematics*, 65(1–3), 223-253. https://doi.org/10.1016/0166-218X(94)00037-E

[6] Hijazi, N. M., Faris, H., & Aljarah, I. (2021). A parallel metaheuristic approach for ensemble feature selection based on multi-core architectures. *Expert Systems with Applications*, 182, 115290. https://doi.org/10.1016/j.eswa.2021.115290

[7] Houssein, E. H., Gad, A. G., Wazery, Y. M., & Suganthan, P. N. (2021). Task scheduling in cloud computing based on meta-heuristics: Review, taxonomy, open challenges, and future trends. *Swarm and Evolutionary Computation*, 62, 100841. https://doi.org/10.1016/j.swevo.2021.100841

[8] Katoch, S., Chauhan, S. S., & Kumar, V. (2021). A review on genetic algorithm: Past, present, and future. *Multimedia Tools and Applications*, 80(5), 8091-8126. https://doi.org/10.1007/s11042-020-10139-6

[9] Khaled Ahsan Talukder, A. K. M., Kirley, M., & Buyya, R. (2009). Multiobjective differential evolution for scheduling workflow applications on global Grids. *Concurrency and Computation: Practice and Experience*, 21(13), 1683-1706. https://doi.org/10.1002/cpe.1417

[10] Marini, F., & Walczak, B. (2015). Particle swarm optimization (PSO). A tutorial. *Chemometrics and Intelligent Laboratory Systems*, 149, 153-165. https://doi.org/10.1016/j.chemolab.2015.08.020

[11] Santander-Jiménez, S., & Vega-Rodríguez, M. A. (2017). Asynchronous non-generational model to parallelize metaheuristics: A bioinformatics case study. *IEEE Transactions on Parallel and Distributed Systems*, 28(7), 1825-1838. https://doi.org/10.1109/TPDS.2016.2645764

[12] Shehab, M., Khader, A. T., & Al-Betar, M. A. (2017). A survey on applications and variants of the cuckoo search algorithm. *Applied Soft Computing Journal*, 61, 498-516. https://doi.org/10.1016/j.asoc.2017.02.034

[13] Standard Task Graph Set. (2025, January 30). Waseda University. https://www.kasahara.cs.waseda.ac.jp/schedule/

[14] Yagiura, M., Ibaraki, T., & Glover, F. (2004). An ejection chain approach for the generalized assignment problem. *INFORMS Journal on Computing*, 16(2), 133-151. https://doi.org/10.1287/ijoc.1030.0036