A Parallel Framework for the Maximum Common Induced Subgraph Problem

Jieyu Wu[1] , **Quan Zhang**[1] , **Yiyuan Wang**[1,2*] , **Shiwei Pan**[1,2*] and **Jian Gao**[1]
[1]School of Information Science and Technology, Northeast Normal University, China
[2]Key Laboratory of Applied Statistics of MOE, Northeast Normal University, Changchun, China
wujy939@nenu.edu.cn, wangyy912@nenu.edu.cn, pansw779@nenu.edu.cn

Abstract

Finding the maximum common induced subgraph (MCIS) between two graphs is a well-known NP-hard problem. While sequential MCIS algorithms have been extensively studied, parallel computing has emerged as an important direction for further performance enhancement with the advancement of computing resources. In this paper, we propose a parallel MCIS framework integrating a dynamic task decomposition method guided by search information and a novel pruning strategy based on shared information. The experimental results demonstrate that the algorithm enhanced by our framework achieves superior performance over both state-of-the-art sequential and existing parallel algorithms. Extensive results further demonstrate the wide scalability and generality of our framework, and the effectiveness of our strategies.

1 Introduction

Graphs provide a powerful abstraction for representing relationships between entities in various domains, such as social networks [Milgram, 1967], web searches [Brin and Page, 1998], and chemistry [Dalke and Hastings, 2013]. Consequently, graph data analysis has attracted significant attention in recent years, and a key challenge is the identification of common structures across graphs. Given a pattern graph G_p and a target graph G_t, the aim of the maximum common induced subgraph (MCIS) problem is to find two induced subgraphs from G_p and G_t that are isomorphic and contain the maximum number of vertices. Although the MCIS problem is NP-hard [Akutsu and Tamura, 2012] and computationally challenging in theory, it plays a crucial role in many practical applications, such as software analysis [Park and Reeves, 2011], computer vision [Solnon *et al.*, 2015], video analysis and image processing [Shearer *et al.*, 2001], as well as graph database systems [Yan *et al.*, 2005].

The efficiency of the MCIS algorithm has been improved significantly in the past decades, and related research mainly focuses on sequential and parallel algorithms. The existing sequential algorithms for the MCIS can be categorized into exact and heuristic algorithms. Although heuristic algorithms [Choi *et al.*, 2012; Rutgers *et al.*, 2010; Fuchs and Riesen, 2025] can quickly obtain a feasible solution, empirical evidence shows that exact algorithms generally achieve superior solving performance in practice. Thus, this paper focuses on exact algorithms for the MCIS. The primary exact algorithm for solving the MCIS is the branch-and-bound (BnB) algorithm [Raymond and Willett, 2002; McCreesh *et al.*, 2016; Hoffmann *et al.*, 2017; Calabrese *et al.*, 2026], which exhaustively explores the search space by matching vertices of G_p to G_t to maximize the isomorphic subgraph size. The most classic BnB algorithm for the MCIS is McSplit [McCreesh *et al.*, 2017]. In recent years, several studies have enhanced McSplit by incorporating reinforcement learning techniques, such as McSplitRL [Liu *et al.*, 2020], McSplitLL [Zhou *et al.*, 2022], GLSearch [Bai *et al.*, 2021], McSplitDAL [Liu *et al.*, 2023], McSplitDSB [Kothalawala *et al.*, 2025] and RRSplit [Yu *et al.*, 2025]. These variants significantly enhance the McSplit algorithm's performance.

For parallel methods, researchers aim to decompose the original MCIS problem into smaller, independent subproblems for parallel solving [Minot *et al.*, 2015; Hoffmann *et al.*, 2018; Quer *et al.*, 2020]. However, in most existing parallel MCIS algorithms, task decomposition and the solving process are tightly coupled. For example, the approaches employed by [Hoffmann *et al.*, 2018; Quer *et al.*, 2020] rely on the search tree to generate subproblems. Consequently, the quality and scale of generated subproblems depend on the current state of the search tree, leading to imbalanced workloads and limiting the algorithm's parallel scalability.

To tackle the above challenges, this paper proposes a parallel framework that aims to decouple task decomposition from the solving process and enhances search efficiency through shared useful information. Specifically, we propose a dynamic task decomposition method that utilizes search information to partition the pattern vertices into prioritized and excluded subsets for matching. Subsequently, by leveraging useful shared information, we design two pruning rules that effectively prune branches and guide the algorithm to focus on promising search subspaces. Finally, we propose a master-worker architectural framework, in which the master uses the dynamic task decomposition method to generate and assign tasks, while workers execute the tasks using the pruning strategy. The results demonstrate that our framework effectively

*Corresponding author.

enhances the performance of sequential MCIS algorithms and exhibits strong scalability and generality.

2 Preliminaries

Given an undirected graph $G = (V, E)$, V is the set of vertices and E is the set of edges. Two vertices v and u are neighbors if $\{v, u\} \in E$. For each $v \in V$, the neighborhood of vertex v is denoted as $N_G(v) = \{u|\{v, u\} \in E\}$ and its degree is denoted as $deg_G(v) = |N_G(v)|$. Given a graph $G = (V, E)$, the induced subgraph $G' = (V', E')$ of G has $V' \subseteq V$ and $E' = \{\{u, v\} \in E|u, v \in V'\}$.

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are said to be isomorphic if there exists a bijection $\phi : V_1 \to V_2$ such that for any $u, v \in V_1$, $\{u, v\} \in E_1$ if and only if $\{\phi(u), \phi(v)\} \in E_2$. Given a pattern graph $G_p = (V_p, E_p)$ and a target graph $G_t = (V_t, E_t)$, let G'_p and G'_t denote induced subgraphs of G_p and G_t, respectively. If G'_p and G'_t are isomorphic, they are called a common induced subgraph of G_p and G_t. Such an isomorphism naturally induces a matching between vertices of G_p and G_t, which can be represented as $M = \{(v_1, \phi(v_1)), (v_2, \phi(v_2)), \cdots, (v_l, \phi(v_\ell))\}$. Each pair $(v, \phi(v))$ is a matched vertex pair, and v and $\phi(v)$ are said to be matched. The maximum common induced subgraph (MCIS) problem aims to find a matching between G_p and G_t that maximizes the number of matched vertex pair.

3 Dynamic Task Decomposition

In this section, we propose a dynamic task decomposition method based on task tree. Each node of the task tree represents one task, where each task is associated with two vertex subsets: one prioritized for exploration and one excluded from exploration. The method partitions pattern vertices into these subsets by expanding nodes in the task tree.

3.1 Task Tree Structure

We maintain a binary tree as the task tree to record the decomposition history. Each node in the task tree represents an independent task and contains the following four attributes:

- Set A: The set of pattern vertices that are prioritized for exploration within the search subspace corresponding to the task, where $A \subseteq V_p$.
- Set B: The set of pattern vertices that are excluded from exploration within the search subspace corresponding to the task, where $B \subseteq V_p$.
- $state$: The current status of the task, $state \in \{running, closed, waiting\}$. A $running$ task is currently being processed by a worker. A task becomes $closed$ if any of the following conditions is met: it is completed by a worker, its parent task is completed, or both of its child tasks are completed. A $waiting$ task has not yet been assigned to any worker.
- Node ID: Each node in the task tree is assigned a unique ID. If a node has ID x, its left child node will have the ID $2x$, while its right child node will have the ID $2x + 1$.

The attributes of a node in the task tree can be represented as $\langle A, B, state, ID\rangle$. Initially, the task tree contains only one root node. For the root node, sets A and B are initialized as empty sets, its state is set to $waiting$, and the ID is 1, i.e., $\langle\emptyset, \emptyset, waiting, 1\rangle$.

3.2 Decomposition Method

We first introduce a partition vertex selection strategy based on search information, followed by the presentation of our dynamic task decomposition method.

Partition Vertex Selection Strategy

In our method, a crucial step in splitting a task is to choose a partition vertex from the pattern vertices that have not yet been selected within the given task. Although the search space of the MCIS is usually large, the high-quality matchings often share common structural characteristics. If a vertex frequently appears in multiple independent matchings, it has a high probability of belonging to the optimal matching. Thus, prioritizing such high-potential vertices can lead to better matchings and accelerate the entire search process.

To exploit this observation, we design a scoring function that dynamically guides vertex selection. During the solving process, the master collects matchings after the workers complete their tasks. For each pattern vertex, it maintains a $score$, initialized to 0. Whenever a worker returns a matching that improves the global best matching, the $score$ of all pattern vertices in that matching is incremented by 1. The partition vertex selection rule is defined as follows.

Vertex Selection Rule: Select a pattern vertex $v \in V_p \backslash (A \cup B)$ with the highest $score$ value, breaking ties by choosing the vertex with the largest deg_{G_p} value.

Details of the Decomposition Method

During the decomposition phase, the master follows these steps to select nodes for expansion and assign their corresponding tasks to the idle workers:

Step 1. Select a leaf node with the smallest ID x from all leaf nodes in the $waiting$ state.

Step 2. Update the task state of the selected node to $running$, and assign the task to the idle worker to execute.

Step 3. Select a partition vertex v from the task of the selected node, following the vertex selection rule.

Step 4. Based on v, generate the left child node $\langle A \cup \{v\}, B, waiting, 2x\rangle$ and the right child node $\langle A, B \cup \{v\}, waiting, 2x + 1\rangle$ for the selected node.

The key operations in the above steps are further explained as follows. Initially, all workers are idle. The master dispatches tasks in the $waiting$ state to idle workers for execution. Once a task is assigned, its state is updated to $running$, and the corresponding worker becomes busy. To maintain an adequate task supply, the master immediately decomposes a task after assigning it to a worker. During the decomposition phase, the master selects a partition vertex v based on the vertex selection rule and then partitions the task based on whether the search space contains the vertex v. The subtasks inherit the sets A and B from their parent task, and the partition vertex is added to the sets A and B of the left and right subtasks, respectively. This results in two subtasks where one

prioritizes searching the space containing v, and the other explores the space excluding v. The two newly generated subtasks are added as new leaf nodes under the corresponding parent node in the task tree.

When a worker completes the task of a node and returns its result to the master, the worker becomes idle again. Concurrently, the task state of that node is updated to $closed$, and the task states of all its child nodes are also updated to $closed$ accordingly. Similarly, a parent node is marked $closed$ once both of its child nodes are closed. Nodes whose task state is $closed$ are not expanded further, as their corresponding subspaces are fully explored and further expansion would cause redundant computation.

Compared with existing MCIS task decomposition methods [Hoffmann *et al.*, 2018; Quer *et al.*, 2020], our method offers two main advantages. First, our method decouples task decomposition from the problem solving process, overcoming the limitation of tight coupling in the previous works. Second, whereas previous works rely on branch selection strategies for task partitioning, our method uses search information to guide task partitioning, prioritizing exploration of promising spaces.

4 A Novel Pruning Strategy

In this section, we propose a novel pruning strategy based on two types of shared information, which enables each worker to focus on exploring promising search spaces.

4.1 Shared Information

In our framework, workers regularly share useful information with each other through the master. Two types of useful information are shared: the size of the current global best matching and the reuse information. For the first type, the master periodically collects each worker's local best matching size and broadcasts the maximum value to all workers. This provides a tighter lower bound for pruning, thereby focusing computational resources on promising spaces. For the second type, reuse information consists of key information extracted from a completed search branch that can be reused by other workers, and is formally defined as follows.

Definition 1 (Reuse Information)**.** *The reuse information is defined as a triple $\langle (v_i, u_j), ms, S\rangle$, where (v_i, u_j) denotes the initial matched vertex pair of the current search branch, ms represents the maximum matching size that can be found under the search branch of (v_i, u_j), and S is the set of excluded pattern vertices that need not be explored when the search branch starts with (v_i, u_j).*

As shown in Figure 1, consider a pattern graph $G_p = \{0, 1, 2\}$ and a target graph $G_t = \{a, b, c, d\}$. Suppose a worker exhaustively explores the branch starting with $(0, a)$ and finds $ms = 2$. Then, the worker generates reuse information $\langle (0, a), 2, \emptyset\rangle$. Subsequently, the worker matches pattern vertex 0 with other target vertices, generating reuse information for each. Once all possibilities for vertex 0 are exhausted, vertex 0 is excluded. Later, when the branch starting with $(1, a)$ is exhaustively searched, the worker generates reuse information $\langle (1, a), 2, \{0\}\rangle$. The set $\{0\}$ indicates that there is no need to explore vertex 0 within the current branch.

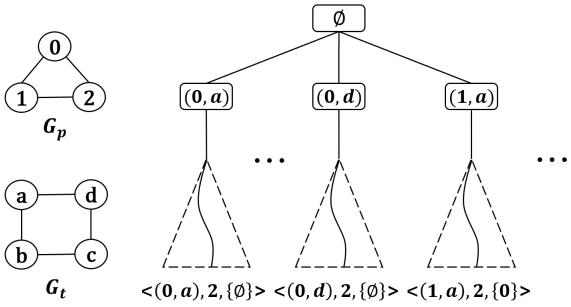

Figure 1: Example of reuse information.

4.2 Pruning Strategy

Our framework leverages shared information to determine whether the current search branch is worth exploring. We design pruning rules based on two types of shared information: the size of the current global best matching bs_{go} and the reuse information set RI. To implement this pruning strategy, each worker w maintains its own private variables, including the local best matching found so far M^w_{lo}, the set of excluded pattern vertices B^w in the current task, and the upper bound UB^w of the current branch (computed using the adopted MCIS algorithm). The following introduces two pruning rules using the shared information.

Pruning Rule 1. If the upper bound of the current branch does not exceed the currently known best matching size (i.e., $UB^w \leq max(|M^w_{lo}|, bs_{go})$), the search backtracks to the deepest unexplored node.

By definition of UB^w, no matching obtained from the current branch can have size greater than UB^w. Now consider the case where the pruning condition $UB^w \leq max(|M^w_{lo}|, bs_{go})$ holds. In this case, even in the most optimistic scenario (i.e., achieving the upper bound UB^w), the best possible matching obtainable from this branch is of size at most $max(|M^w_{lo}|, bs_{go})$, which is no larger than the size of the best matching already known. Hence, there is no benefit in further exploring this branch, and it can be safely pruned. The search then backtracks to the deepest unexplored node.

Pruning Rule 2. If the initial vertex pair of the current branch matches that of a reuse information $\langle (v_i, u_j), ms, S\rangle \in RI$ and satisfies either:

- $S \subset B^w$ and $ms \leq max(|M^w_{lo}|, bs_{go})$
- $B^w \subseteq S$ and $ms + |S| - |B^w| \leq max(|M^w_{lo}|, bs_{go})$

, then the search backtracks to the vertex v_i.

Suppose the initial vertex pair of the current branch matches a reuse entry $\langle (v_i, u_j), ms, S\rangle \in RI$. The pruning condition depends on the relation between B^w and S. In the first case, if $S \subset B^w$, then the current branch excludes at least all vertices that were excluded in the reuse information. Hence, its achievable matching size cannot exceed the recorded value ms. In the second case, if $B^w \subseteq S$, then the reuse information has a search space no larger than that associated with the current branch. Thus, in the most optimistic expansion, the best matching size attainable in this branch is $ms + |S| - |B^w|$. If the corresponding upper bound in either case does not exceed the currently known best matching size,

Algorithm 1: Prune

Input: The shared information bs_{go} and RI
Output: The backtracking vertex v

1 $v \leftarrow -1$;
2 **if** *satisfies* ***Pruning Rule 1*** **then**
3 $\quad v \leftarrow$ the deepest unexplored pattern vertex;
4 **else if** $RI \neq \emptyset$ **then**
5 $\quad (v_i, u_i) \leftarrow$ initial vertex pair in the current branch;
6 $\quad$ **if** (v_i, u_i) *satisfies* ***Pruning Rule 2*** **then** $v \leftarrow v_i$;
7 **return** v;

then the branch cannot yield a better matching. Therefore, it is safely pruned, and the search backtracks to vertex v_i.

4.3 Details of Pruning Procedure

The pruning strategy is implemented in Algorithm 1. The backtracking vertex v is initialized to -1 (Line 1). The algorithm first computes an upper bound following the adopted MCIS algorithm and checks whether pruning rule 1 is satisfied (Line 2). If so, the search backtracks to the deepest unexplored pattern vertex (Line 3). When the reuse information set RI is not empty, the algorithm obtains the initial vertex pair (v_i, u_i) of the current branch (Line 5). Then, it checks if this pair satisfies pruning rule 2. If satisfied, the algorithm should backtrack to the vertex v_i (Line 6).

5 Our Parallel Framework

Our framework adopts a master–worker architecture , as illustrated in Figure 2.

Master: The master dynamically decomposes and assigns tasks to idle workers, while periodically collecting the size of the local best matchings and reuse information. It also broadcasts valuable shared information to all workers for effective pruning. Upon task completion or reaching the cutoff time, the master terminates all workers.

In Algorithm 2, the master initializes the global best matching size bs_{go}, the reuse information set RI, and the global best matching M_{go} accordingly (Line 1). For each worker

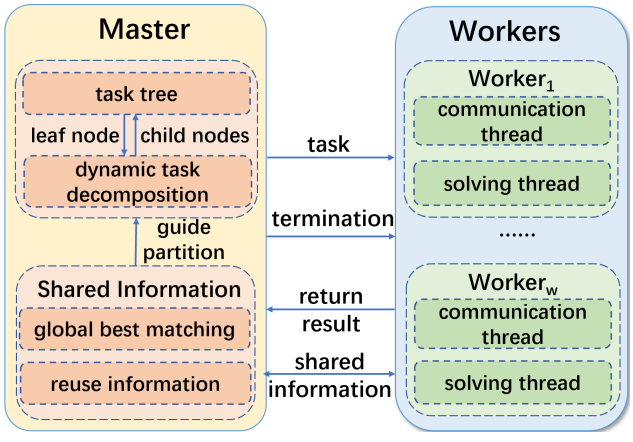

Figure 2: Our parallel framework.

Algorithm 2: Master

Input: $G_p = (V_p, E_p)$, *cutoff*
Output: the global best matching M_{go}

1 $bs_{go} \leftarrow 0$, $RI \leftarrow M_{go} \leftarrow \emptyset$;
2 $idle[w] = 1$ for each worker w;
3 **while** $runtime < cutoff$ **do**
4 $\quad$ **if** $\exists$ *worker* w *with* $idle[w] = 1$ **then**
5 $\quad\quad$ send a task with sets A and B to the worker w based on dynamic task decomposition;
6 $\quad\quad$ $idle[w] = 0$;
7 $\quad$ **if** *meet the requesting condition* **then**
8 $\quad\quad$ **if** $\exists$ *worker* w *with* $idle[w] = 0$ **then**
9 $\quad\quad\quad$ require $|M_{lo}^w|$ and RI_{new}^w from worker w;
10 $\quad\quad\quad$ **if** $|M_{lo}^w| > bs_{go}$ **then** $bs_{go} \leftarrow |M_{lo}^w|$;
11 $\quad\quad\quad$ $RI \leftarrow RI \cup RI_{new}^w$;
12 $\quad\quad$ broadcast bs_{go} and RI to workers;
13 $\quad$ **if** $\exists$ *worker* w *with* $idle[w] = 0$ *and task is finished* **then**
14 $\quad\quad$ receive M_{lo}^w from worker w and $idle[w] = 1$;
15 $\quad\quad$ **if** $|M_{lo}^w| > |M_{go}|$ **then**
16 $\quad\quad\quad$ $M_{go} \leftarrow M_{lo}^w$ and $bs_{go} \leftarrow |M_{lo}^w|$;
17 $\quad$ **if** *finished all tasks* **then break** ;
18 send termination signal to the workers;
19 receive M_{lo}^w from workers and update M_{go};
20 **return** M_{go};

w, we use $idle[w]$ to indicate its status, where $idle[w] = 1$ means the worker is idle, and $idle[w] = 0$ means it is busy. Initially, all workers are set to idle, i.e., $idle[w] = 1$ for each worker (Line 2). Within the time limit, the master dynamically decomposes tasks and assigns them to idle workers, setting $idle[w] = 0$ for each worker (Lines 4–6). In Lines 7–9, the master periodically requests the local best matching size $|M_{lo}^w|$ and newly generated reuse information RI_{new}^w from all workers (set to every 10 seconds). It then updates bs_{go} and RI (Lines 10–11), and broadcasts the latest bs_{go} and RI to all workers to achieve effective pruning (Line 12). When a worker completes its task, the master receives its local best matching M_{lo}^w and marks the worker as idle again (Line 14). If this M_{lo}^w outperforms the current global best matching, the master updates M_{go} and bs_{go} accordingly (Lines 15–16). Once all tasks are completed or the cutoff time is reached, the master sends a termination signal to all workers (Line 18). Finally, the master receives all results and updates the global best matching M_{go} based on these results (Line 19).

Worker: To improve computational and communication efficiency, each worker uses two threads: a communication thread for task reception, result reporting, and shared information exchange with the master, and a solving thread that runs the MCIS algorithm with the proposed pruning strategy.

Algorithm 3 describes the cooperation between the communication and the solving threads within a worker. The global variables in a worker include the local best matching M_{lo}, the newly generated reuse information set RI_{new}, the reuse information set RI, the global best matching size

Algorithm 3: Worker

Input: $G_p = (V_p, E_p), G_t = (V_t, E_t)$
Output: the local best matching M_{lo}

1 **Global** $M_{lo} \leftarrow RI_{new} \leftarrow RI \leftarrow \emptyset, bs_{go} \leftarrow ms \leftarrow 0$;
2 **return** $Communication()$;

/* communication thread */
3 **Thread** Communication();
4 **while do**
5 **if** *solving thread is idle* **then**
6 receive a task with sets A and B from master;
7 $Solving(A, B)$;
8 **else if** *task is finished* **then** send M_{lo} to master ;
9 **else if** *meet the requesting condition* **then**
10 send $|M_{lo}|$ and RI_{new} to master and $RI_{new} \leftarrow \emptyset$;
11 receive bs_{go} and RI from master;
12 **else if** *receive a termination signal* **then break** ;
13 send M_{lo} to master;

/* solving thread */
14 **Thread** Solving(A, B);
15 $Search(\emptyset, A, V_p \setminus (A \cup B), B, bs_{go}, RI)$;

Algorithm 4: Search

Input: $M_c, A, V_r, S, bs_{go}, RI$
Output: The backtracking vertex v'

1 **if** $|M_c| > |M_{lo}|$ **then** $M_{lo} \leftarrow M_c$;
2 **if** $|M_c| > ms$ **then** $ms \leftarrow |M_c|$;
3 $v' \leftarrow Prune(bs_{go}, RI)$;
4 **if** $v' \neq -1$ **then return** v' ;
5 **if** $A \neq \emptyset$ **then** $v_i \leftarrow pop(A)$;
6 **else** $v_i \leftarrow$ select a pattern vertex from V_r ;
7 **for** *each* $u_j \in V_t$ *can be matched to* v_i **do**
8 $M_c \leftarrow M_c \cup \{(v_i, u_j)\}$;
9 $v' \leftarrow Search(M_c, A, V_r, S, bs_{go}, RI)$;
10 $M_c \leftarrow M_c \setminus \{(v_i, u_j)\}$;
11 **if** $v' \neq -1$ *and* $v' \neq v_i$ **then return** v' ;
12 **if** $|M_c| == 0$ *and* $|S| < thrd$ **then**
13 $RI_{new} \leftarrow RI_{new} \cup \{\langle (v_i, u_j), ms, S \rangle\}$ and $ms \leftarrow 0$;
14 $V_r \leftarrow V_r \setminus \{v_i\}$ and $S \leftarrow S \cup \{v_i\}$;
15 $v' \leftarrow Search(M_c, A, V_r, S, bs_{go}, RI)$;
16 **return** v';

bs_{go}, and the maximum matching size ms under the current branch. First, these variables are initialized accordingly (Line 1). Then, the worker launches the communication thread (Line 2). When a task is received, the communication thread activates the solving thread to perform the computation.

As for the communication thread (Lines 3–13), it repeatedly performs the following operations: (1) When the solving thread is idle, it receives a new task from the master and wakes up the solving thread to start computation (Lines 5–7). (2) When the solving thread completes the task, it sends the local best matching M_{lo} to the master (Line 8). (3) When the master requests information (every 10 seconds), it sends the current best matching size $|M_{lo}|$ and the newly reuse information RI_{new} to the master, and receives the global best matching size bs_{go} and reuse information set RI from the master (Lines 9–11). (4) Upon receiving a termination signal, it exits the loop (Line 12). Within the loop, if no communication is required, the communication thread sleeps for 1 second to avoid invalid queries. Finally, it sends the local best matching M_{lo} to the master (Line 13).

As for the solving thread, it invokes the $Search$ function to perform the solving process (Line 15). The $Search$ function is described in Algorithm 4. The six input parameters include: the current matching M_c (initialized as empty), the priority matching pattern vertex set A, the remaining pattern vertex set V_r (initialized as $V_p \setminus (A \cup B)$), the excluded pattern vertex sets S (initialized as the current task's set B), as well as the globally shared information bs_{go} and RI.

If the current matching M_c has a larger size than M_{lo} and exceeds the value of ms, then M_{lo} and ms are updated (Lines 1–2). The algorithm then determines whether to continue the search based on the shared information bs_{go} and RI from the master (Line 3). If the returned vertex is not -1, the search backtracks to vertex v' (Line 4). Next, the algorithm prioritizes obtaining a vertex from the set A (Line 5). The algorithm selects the last vertex from the set A each time. If A is empty, it selects a vertex v_i from V_r based on the adopted MCIS algorithm (Line 6). For the vertex v_i, the algorithm traverses all vertices u_j in the target graph that can be matched with v_i according to the adopted MCIS algorithm, and recursively invokes itself to extend the matching (Line 9). If pruning occurs during the search process, the search backtracks to vertex v' (Line 11). When the search of a branch is completed, a piece of reuse information is generated and stored in RI_{new} (Line 13). As reuse information with a large set S has low utility and can only prune a relatively limited search space, we only collect reuse information where $|S| < thrd$. In our experiments, $thrd$ is set to 10. Subsequently, the algorithm explores the search space that does not include vertex v_i, moves v_i from the remaining vertex set V_r to the set S, and continues the recursive search (Lines 14–15).

6 Experimental Evaluation

In this section, we apply our framework to enhance the state-of-the-art MCIS algorithm and evaluate the performance of the proposed algorithm across a broad range of benchmarks.

6.1 Experiment Preliminaries

We adopt 8 benchmarks[1] with 15,396 instances in total, following previous works [Liu *et al.*, 2020; Liu *et al.*, 2023; Wang *et al.*, 2024; Kothalawala *et al.*, 2025; Yu *et al.*, 2025]. (1) images-CVIU11 [Damiand *et al.*, 2011] has 6,278 segmented image instances, with 43 pattern graphs (22–151 vertices) and 146 target graphs (1,072–5,972 vertices). (2) meshes-CVIU11 [Damiand *et al.*, 2011] has 3,018 instances

[1] http://liris.cnrs.fr/csolnon/SIP.html

from meshes modeling 3D objects, with 6 pattern graphs (40–199 vertices) and 503 target graphs (208–5,873 vertices). (3) images-PR15 [Solnon *et al.*, 2015] has 24 segmented image instances, with 24 pattern graphs (4–170 vertices) and 1 target graph (4,838 vertices). (4) scalefree [Solnon, 2010] includes 100 random instances, where target graphs have 200–1,000 vertices and pattern graphs comprise 90% of the target vertices. (5) si [Solnon, 2010] has 1,170 instances, with target graphs (200–1,296 vertices) and pattern graphs (20%–60% of target vertices). (6) phase [McCreesh *et al.*, 2017] comprises 200 instances selected close to the satisfiable–unsatisfiable phase transition, with a pattern graph with 30 vertices and a target graph with 150 vertices. (7) LV [Solnon, 2010] has 1,176 instances, using 49 graphs (10–128 vertices) as both pattern and target graphs. (8) LargerLV extends LV with 3,430 instances, using the same 49 pattern graphs and 70 new target graphs (138–6,671 vertices).

We evaluated the proposed algorithm against sequential and parallel algorithms. For sequential algorithms, we select four state-of-the-art McSplit variants: McSplitDAL [Liu *et al.*, 2023], McSplitDSB [Kothalawala *et al.*, 2025], RRSplit [Yu *et al.*, 2025], and McSplitDALPR [Calabrese *et al.*, 2026]. For parallel algorithms, we adopted a CPU-based multi-threaded parallel algorithm v2 [Quer *et al.*, 2020] and a parallel version of the McSplit algorithm [Hoffmann *et al.*, 2018], which we denote as ParaMcSplit in this paper. To enhance the comparison, we integrate the four sequential algorithms into a portfolio named PortfolioMCIS. Specifically, for each instance, PortfolioMCIS runs the four algorithms in parallel, and its performance is the optimal matching obtained by the first algorithm that completes within the time limit.

Since RRSplit performs best among the sequential algorithms, we extended it into a parallel version using our framework, denoted as ParaRRS [2]. Our algorithm is parallelized with OpenMPI. ParaRRS and all compared algorithms are implemented in C or C++ and compiled with g++ using the -O3 optimization flag. All experiments are conducted on the servers equipped with an Intel Xeon Gold 6238 CPU (2.10 GHz) and 512 GB of RAM, running Ubuntu 22.04. For each instance, the time limit of all algorithms is 1800 seconds. In the comparative experiments, PortfolioMCIS requires at least 4 threads to run its four constituent algorithms in parallel. Therefore, we configured PortfolioMCIS, ParaMcSplit and v2 to use 4 threads. Since v2 runs with 8 threads in their work, we also include this version in our experimental evaluation. Accordingly, our ParaRRS algorithm was set to use 4 processes (one master and three workers).

6.2 Comparative Experiment

To clearly show the overall performance of ParaRRS and its competitors, we followed previous studies [Liu *et al.*, 2020; Liu *et al.*, 2023] by excluding two types of instances: (1) 976 easy instances solved by all algorithms within 10 seconds; (2) 12,222 extremely hard instances unsolvable by any algorithm within the time limit. There are 2,198 remaining instances.

According to the time taken by each algorithm to find the optimal matching, we illustrate the number of solved instances for all algorithms in Figure 3. ParaRRS solves 2,052 instances, outperforming all competitors. Notably, ParaRRS can solve 52 instances unsolved by all competitors. When compared with the parallel algorithms PortfolioMCIS, v2(4 threads), v2(8 threads), and ParaMcSplit, ParaRRS solved 59, 105, 25, and 124 more instances, corresponding to performance improvements of 3.0%, 5.4%, 1.2%, and 6.4%, respectively. When compared with the sequential algorithms, ParaRRS achieves superior performance, solving 89, 362, 354, and 1,651 more instances than RRSplit, McSplitDSB, McSplitDAL, and McSplitDALPR, respectively. The results show that ParaRRS exhibits superior solving ability compared to both parallel and sequential algorithms. In particular, the solving performance of ParaRRS improves upon RRSplit by 4.5%, demonstrating that the parallelization strategies of our framework effectively speed up the search process.

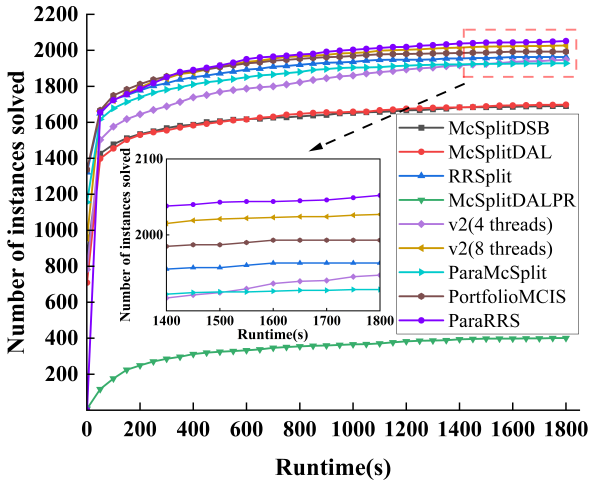

Figure 3: Cumulative numbers of instances solved by ParaRRS and all comparison algorithms within the time limit.

6.3 Scalability Analysis

To evaluate the scalability and generality of our framework, we parallelize three state-of-the-art sequential MCIS algorithms, RRSplit, McSplitDAL and McSplitDSB, which are denoted as ParaRRS, ParaDAL and ParaDSB, respectively. To further assess their scalability, we randomly select one extremely hard unsolved instance from each benchmark in the comparative experiment. The images-PR15 benchmark is excluded, as all its instances can be solved within the time limit by the three sequential algorithms. We then tested these parallel algorithms on the selected instances using 4 to 64 processes. The experimental results are presented in Table 1. In the table, $result$ denotes the size of the found best matching, and $time$ denotes the time to find it. Results for instances solved to optimality are marked in bold.

The results clearly show that all algorithms exhibit a significant performance improvement as the number of processes increases. Specifically, with more processes, these three parallel algorithms find larger matchings or achieve the same matching size in less time. Notably, these three parallel algorithms solved the hard instances pattern10_target27

[2]https://github.com/yiyuanwang1988/Parallel-MCSP.git

Instance	Algorithms	np=4		np=8		np=16		np=32		np=64	
		result	time	result	time	result	time	result	time	result	time
pattern10_target27	ParaDAL	**123**	201.4	**123**	198.3	**123**	69.6	**123**	58.0	**123**	48.4
	ParaDSB	100	39.1	107	1691.2	**123**	1049.6	**123**	266.8	**123**	145.4
	ParaRRS	106	1282.7	**123**	313.6	**123**	218.7	**123**	78.7	**123**	57.7
g38_g76	ParaDAL	101	1643.6	101	94.0	101	75.0	120	34.2	120	1.9
	ParaDSB	99	443.0	99	11.0	99	11.0	120	33.3	120	23.2
	ParaRRS	107	1086.0	**121**	665.6	**121**	299.8	**121**	0.0	**121**	0.0
g42_g43	ParaDAL	64	124.2	64	33.5	64	21.6	64	15.2	66	1687.7
	ParaDSB	64	45.9	64	16.0	64	16.0	64	16.1	64	15.1
	ParaRRS	64	5.5	66	1458.0	66	1444.0	67	251.5	67	74.1
pattern3_target56	ParaDAL	159	355.1	159	3.8	161	65.7	161	0.0	161	0.0
	ParaDSB	159	1224.3	159	0.5	161	174.1	161	0.0	161	0.0
	ParaRRS	159	577.3	159	11.3	161	49.4	161	6.0	161	4.9
phase-30-150-0.84-0.76-0.76-7	ParaDAL	23	84.0	23	65.3	23	50.9	23	50.7	23	40.8
	ParaDSB	23	110.0	23	87.0	23	51.6	23	52.4	23	43.4
	ParaRRS	23	110.3	23	86.6	23	51.4	23	51.7	23	42.0
A.15	ParaDAL	94	1415.8	95	788.2	132	0.0	132	0.0	139	1335.0
	ParaDSB	94	858.4	96	1422.4	132	1.0	132	0.0	139	785.6
	ParaRRS	94	785.5	96	1268.3	132	0.0	132	0.0	139	705.1
si4_m4D_m1296.00	ParaDAL	429	1660.5	429	1655.9	434	505.8	**518**	1174.9	**518**	184.1
	ParaDSB	429	1189.3	429	1183.0	434	1005.4	**518**	678.3	**518**	87.1
	ParaRRS	434	1287.8	434	1235.6	434	569.5	**518**	414.0	**518**	53.4

Table 1: Experimental results on scalability.

and si4_m4D_m1296.00 with 16 and 32 processes, respectively. Furthermore, ParaRRS solved g38_g76 using just 8 processes. For hard phase instances, although all algorithms achieve the same matching size under different process counts, their runtime with 64 processes is only half that with 4 processes. This demonstrates that our framework is not limited to improving a single algorithm, it enhances the number of solvable instances and the overall solving efficiency across all three algorithms. These findings strongly validate the outstanding scalability and generality of our framework.

6.4 Ablation Study

To verify the effectiveness of the proposed strategies, we compare ParaRRS with two variants:(1) ParaRRS1 does not employ dynamic task decomposition method. Instead, the master adopts the same task decomposition approach as in [Quer *et al.*, 2020], where each node in the task tree corresponds to a partial matching and the remaining unmatched vertices. (2) ParaRRS2 removes our pruning strategy, and each worker only prunes using its local best matching.

We sampled 10 instances from each benchmark, resulting in a total of 80 instances. Figure 4 shows the number of instances successfully solved by ParaRRS and its two variants. As the number of processes increases, ParaRRS solves more instances than both variants. Notably, with 32 processes, it solves 26 instances, achieving a performance gain of 23.8% over ParaRRS1 and 36.8% over ParaRRS2. The dynamic task decomposition strategy adopted by ParaRRS relies on vertex scoring information collected during the search process to guide partitioning. This mechanism struggles to fully leverage its advantages when the number of processes is small, and thus it finds one fewer optimal matching than ParaRRS1 when running with 4 processes. Although ParaRRS exhibits comparable performance to both variants when using 4 processes, its dynamic task decomposition method enables more efficient utilization of parallel computational resources as the number of processes increases. Simultaneously, the proposed pruning strategy can generate more useful shared information, thereby significantly improving the search efficiency.

7 Conclusion

This paper proposes a parallel framework for solving the MCIS. The framework adopts a master-worker architecture that integrates a dynamic task decomposition method and a novel pruning strategy. Experimental results show that the algorithm enhanced by our framework outperforms both state-of-the-art sequential and existing parallel algorithms.

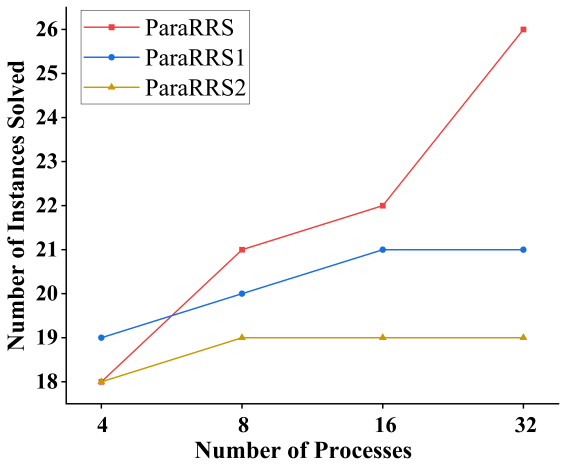

Figure 4: Comparison of ParaRRS with its two variants with different processes.

Acknowledgments

This work is supported by National Cryptologic Science Fund of China 2025NCSF02046, NSFC under Grant No. 61806050.

References

[Akutsu and Tamura, 2012] Tatsuya Akutsu and Takeyuki Tamura. On the complexity of the maximum common subgraph problem for partial k-trees of bounded degree. In *ISAAC*, pages 146–155, 2012.

[Bai *et al.*, 2021] Yunsheng Bai, Derek Xu, Yizhou Sun, and Wei Wang. Glsearch: Maximum common subgraph detection via learning to search. In *ICML*, pages 588–598, 2021.

[Brin and Page, 1998] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[Calabrese *et al.*, 2026] Andrea Calabrese, Lorenzo Cardone, Salvatore Licata, Marco Porro, and Stefano Quer. Improving state-of-the-art vertex sorting algorithms to compute the maximum common induced subgraph. *Journal of Heuristics*, 32(1):1, 2026.

[Choi *et al.*, 2012] Jaeun Choi, Yourim Yoon, and Byung-Ro Moon. An efficient genetic algorithm for subgraph isomorphism. In *GECCO*, pages 361–368, 2012.

[Dalke and Hastings, 2013] Andrew Dalke and Janna Hastings. Fmcs: a novel algorithm for the multiple mcs problem. *Journal of Cheminformatics*, 5(Suppl 1):O6, 2013.

[Damiand *et al.*, 2011] Guillaume Damiand, Christine Solnon, Colin De la Higuera, Jean-Christophe Janodet, and Émilie Samuel. Polynomial algorithms for subisomorphism of nd open combinatorial maps. *Computer Vision and Image Understanding*, 115(7):996–1010, 2011.

[Fuchs and Riesen, 2025] Mathias Fuchs and Kaspar Riesen. Fast approximate maximum common subgraph computation. *Pattern Recognition Letters*, 190:66–72, 2025.

[Hoffmann *et al.*, 2017] Ruth Hoffmann, Ciaran McCreesh, and Craig Reilly. Between subgraph isomorphism and maximum common subgraph. In *AAAI*, pages 3907–3914, 2017.

[Hoffmann *et al.*, 2018] Ruth Hoffmann, Ciaran McCreesh, Samba Ndojh Ndiaye, Patrick Prosser, Craig Reilly, Christine Solnon, and James Trimble. Observations from parallelising three maximum common (connected) subgraph algorithms. In *CPAIOR*, pages 298–315, 2018.

[Kothalawala *et al.*, 2025] Buddhi W Kothalawala, Henning Koehler, and Qing Wang. Learning to bound for maximum common subgraph algorithms. In *CP*, pages 22–1, 2025.

[Liu *et al.*, 2020] Yanli Liu, Chu-Min Li, Hua Jiang, and Kun He. A learning based branch and bound for maximum common subgraph related problems. In *AAAI*, pages 2392–2399, 2020.

[Liu *et al.*, 2023] Yanli Liu, Jiming Zhao, Chu-Min Li, Hua Jiang, and Kun He. Hybrid learning with new value function for the maximum common induced subgraph problem. In *AAAI*, pages 4044–4051, 2023.

[McCreesh *et al.*, 2016] Ciaran McCreesh, Samba Ndojh Ndiaye, Patrick Prosser, and Christine Solnon. Clique and constraint models for maximum common (connected) subgraph problems. In *CP*, pages 350–368, 2016.

[McCreesh *et al.*, 2017] Ciaran McCreesh, Patrick Prosser, and James Trimble. A partitioning algorithm for maximum common subgraph problems. In *IJCAI*, pages 712–719, 2017.

[Milgram, 1967] Stanley Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[Minot *et al.*, 2015] Maël Minot, Samba Ndojh Ndiaye, and Christine Solnon. A comparison of decomposition methods for the maximum common subgraph problem. In *ICTAI*, pages 461–468, 2015.

[Park and Reeves, 2011] Younghee Park and Douglas Reeves. Deriving common malware behavior through graph clustering. In *ASIACCS*, pages 497–502, 2011.

[Quer *et al.*, 2020] Stefano Quer, Andrea Marcelli, and Giovanni Squillero. The maximum common subgraph problem: A parallel and multi-engine approach. *Computation*, 8(2):48, 2020.

[Raymond and Willett, 2002] John W Raymond and Peter Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16(7):521–533, 2002.

[Rutgers *et al.*, 2010] Jochem H Rutgers, Pascal T Wolkotte, Philip KF Hölzenspies, Jan Kuper, and Gerard JM Smit. An approximate maximum common subgraph algorithm for large digital circuits. In *DSD*, pages 699–705, 2010.

[Shearer *et al.*, 2001] Kim Shearer, Horst Bunke, and Svetha Venkatesh. Video indexing and similarity retrieval by largest common subgraph detection using decision trees. *Pattern Recognition*, 34(5):1075–1091, 2001.

[Solnon *et al.*, 2015] Christine Solnon, Guillaume Damiand, Colin De La Higuera, and Jean-Christophe Janodet. On the complexity of submap isomorphism and maximum common submap problems. *Pattern Recognition*, 48(2):302–316, 2015.

[Solnon, 2010] Christine Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artificial Intelligence*, 174(12-13):850–864, 2010.

[Wang *et al.*, 2024] Yiyuan Wang, Chenghou Jin, and Shaowei Cai. Pathlad+: Towards effective exact methods for subgraph isomorphism problem. *Artificial Intelligence*, 337:104219, 2024.

[Yan *et al.*, 2005] Xifeng Yan, Philip S Yu, and Jiawei Han. Substructure similarity search in graph databases. In *SIGMOD*, pages 766–777, 2005.

[Yu *et al.*, 2025] Kaiqiang Yu, Kaixin Wang, Cheng Long, Laks Lakshmanan, and Reynold Cheng. Fast maximum

common subgraph search: A redundancy-reduced backtracking approach. *Proceedings of the ACM on Management of Data*, 3(3):1–27, 2025.

[Zhou *et al.*, 2022] Jianrong Zhou, Kun He, Jiongzhi Zheng, Chu-Min Li, and Yanli Liu. A strengthened branch and bound algorithm for the maximum common (connected) subgraph problem. In *IJCAI*, pages 1908–1914, 2022.