IPSJ Transactions on Programming Vol.7 No.3 22–39 (July 2014)

Regular Paper

Parallelization of Extracting Connected Subgraphs with Common Itemsets

Shingo Okuno[1,a) Tasuku Hiraishi[2,b) Hiroshi Nakashima[2,c)
Masahiro Yasugi[3,d) Jun Sese[4,e)

Received: November 15, 2013, Accepted: March 19, 2014

Abstract: This paper proposes a parallel algorithm to extract all connected subgraphs, each of which shares a common itemset whose size is not less than a given threshold, from a given graph in which each vertex is associated to an itemset. We also propose implementations of this algorithm using the task-parallel language Tascell. This kind of graph mining can be applied to analysis of social or biological networks. We have already proposed an efficient sequential search algorithm called COPINE for this problem. COPINE reduces the search space of a dynamically growing tree structure by pruning its branches corresponding to the following subgraphs; already visited, having itemsets smaller than the threshold, and having already-visited supergraphs with identical itemsets. For the third pruning, we use a table associating already-visited subgraphs and their itemsets. To avoid excess pruning in a parallel search where a unique set of subtrees (tasks) is assigned to each worker, we should put a certain restriction on a worker when it is referring to a table entry registered by another worker. We designed a parallel algorithm as an extension of COPINE by introducing this restriction. A problem of the implementation is how workers efficiently share the table entries so that a worker can safely use as many entries registered by other workers as possible. We implemented two sharing methods: (1) a victim worker makes a copy of its own table and passes it to a thief worker when the victim spawns a task by dividing its task and assigns it to the thief, and (2) a single table controlled by locks is shared among workers. We evaluated these implementations using a real protein network. As a result, the single table implementation achieved a speedup of approximately a factor four with 16 workers.

Keywords: graph mining, dynamic load balancing, task-parallel language

1. Introduction

The amount of created and stored information is increasing explosively owing to recent advances in information-communication technology and information systems. The size of data generated in this phenomenon called "infoplosion" is too large to make effective use of as it is. Therefore, data mining, i.e., discovering useful knowledge from such large amount of data, is attracting significant attention. In particular, data mining for graphs, i.e., graph mining, has become an active research area since data analyzed for academic and commercial purposes such as network communications, biology, and marketing are often represented by graphs [1].

In most cases, a graph is not analyzed only with its inherent components, i.e., vertices and edges, but together with external data associated with its vertices and/or edges. For example, in a social networking service (SNS), users and friendships among them can be represented as vertices and edges of a graph. User interests can be represented as an itemset associated with a vertex. We can obtain more useful knowledge by analyzing the graph and the itemsets together than by analyzing each of them separately. This paper deals with graph mining that extracts connected subgraphs with common itemsets (Common Itemset connected subGraph, CIG) from data composed of a graph and itemsets associated with vertices of the graph. Here, a common itemset means the product of all itemsets with which vertices of a connected subgraph are associated.

We can obtain a variety of useful knowledge from CIGs. Apparently, they are useful in SNS. Another example is a biological network where each vertex represents a gene and is associated with a set of drugs that react to that gene. A CIG in such a graph denotes a gene combination that reacts to some combination of drugs. Such a knowledge is expected to be helpful for drug discovery.

As the size of a graph increases, the computation time required for extracting CIGs using a naïve algorithm increases exponentially. Thus, it is clearly unrealistic to analyze Facebook, the world's largest SNS with 500 million users [2], using such an exponential time algorithm. One possible approach to extract CIGs from a large-scale graph within a practical time is to employ an efficient sequential algorithm and parallelize it to reduce the com-

1 Graduate School of Informatics, Kyoto University, Sakyo, Kyoto 606–8501, Japan
2 Academic Center for Computing and Media Studies, Kyoto University, Sakyo, Kyoto 606–8501, Japan
3 Department of Artificial Intelligence, Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
4 Graduate School of Information Science and Engineering, Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
a) shingo@sys.i.kyoto-u.ac.jp
b) tasuku@media.kyoto-u.ac.jp
c) h.nakashima@media.kyoto-u.ac.jp
d) yasugi@ai.kyutech.ac.jp
e) sesejun@cs.titech.ac.jp

putation time.

We have already proposed an efficient sequential backtrack search algorithm called COmmon Pattern Itemset NEtwork mining (COPINE) [3], [4]. However, due to the pruning employed in this algorithm, it cannot be parallelized straightforwardly. In order to reduce the search space of a dynamically growing tree structure, COPINE prunes its branches corresponding to the following subgraphs; (1) already visited, (2) having itemsets smaller than a given threshold, and (3) having already-visited supergraphs with identical itemsets. For the third pruning, a table called itemset table is used to record visited subgraphs and their itemsets. In a parallel search where a unique set of subtrees (tasks) is assigned to each worker, a worker could excessively prune the branches in its subtrees if it blindly consulted table entries registered by another worker. To avoid such excessive pruning, we should put a certain restriction on a worker when it is referring to table entries registered by another worker. We designed a parallel algorithm as an extension of COPINE by introducing this restriction.

Since a search tree in COPINE has an irregular structure, it is effective to apply dynamic load balancing in parallelized implementations. Applications having these properties are often implemented by task-parallel languages, by which we can dynamically spawn tasks to be automatically assigned to workers being parallel threads and/or processes. We implemented the parallel COPINE algorithm using the task-parallel language Tascell [5] appreciating the high performance it has been accomplishing for various backtrack search algorithms.

In addition to the load balancing, there is another important implementation issue regarding sharing information in the itemset table. This sharing requires consideration of a trade-off between the increasing opportunity of sharing useful information among workers and reducing the cost to do that safely. We implemented the following three sharing methods and evaluated them using a real protein network:

non-sharing method where each worker has its own table and never sees the tables of others.

replicating method where each worker still has its own table but its contents are imported from the table of the victim worker when the thief worker steals a task from the victim.

fully-sharing method where a single table is shared among all workers with a lock for each subgraph entry.

The remainder of this paper is organized as follows. We define the problem targeted in this research in Section 2. In Section 3, we describe a sequential COPINE algorithm and propose a parallel algorithm designed to extend it. Then, we provide a parallelized implementation of COPINE using Tascell in Section 4, and show the performance evaluations in Section 5. We introduce related work in Section 6, and describe future work in Section 7. Finally, we conclude this paper in Section 8.

2. Definition of CCIG Enumeration Problem

In this section, we define the *Closed CIG* (*CCIG*) *enumeration problem*, which is targeted for parallelization in this research. This problem is defined for a graph whose vertices are associated with itemsets, and the itemset of a connected subgraph, i.e., the product of all the itemsets associated to its vertices. A CCIG with

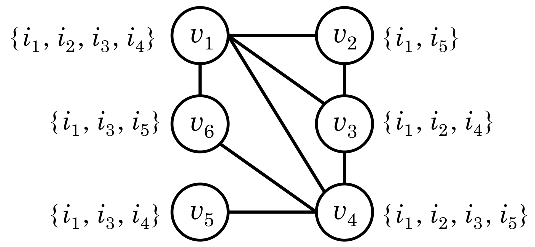


Fig. 1 Example of a graph associated with itemsets.

respect to an itemset I is a maximal subgraph among *CIG* that have I as a common itemset, or a CIG such that, when any adjacent vertex is added to the CIG, the resulting subgraph is not a CIG with respect to I. The CCIG enumeration problem is the problem to enumerate all the CCIGs whose common itemset size is not less than a given threshold. More formal definitions of the connected subgraph, CIG, CCIG, and CCIG enumeration problem are given as follows.

Definition 1 (Connected Subgraph) For a given graph $G = (V, E)$, we call $G' = (V', E')$ a *connected subgraph*[*1] of G iff all of the following criteria hold.

(1) $V' \subseteq V$

(2) $E' = \{(u, v) \mid u, v \in V'\} \cap E$

(3) $\forall u, v \in V' : \exists\{(u_1, v_1), \cdots, (u_n, v_n)\}$ is the path between u and v where $u_1 = u$, $v_n = v$, $(u_i, v_i) \in E'$, $u_i = v_{i-1}$ $(1 < i \leq n)$

Note that E' is uniquely defined by V', and thus, we may denote E' as $E(V')$.

The CCIG enumeration problem is defined as follows.

Definition 2 (CCIG Enumeration Problem) Given a graph $G = (V, E)$, a set of items I, items associated with each vertex v being $I(v) \subseteq I$ $(v \in V)$, and a threshold θ, the problem of extracting all connected subgraphs $G' = (V', E')$ that satisfy the two conditions below is the *CCIG enumeration problem.*

(i) $\left|\bigcap_{v \in V'} I(v)\right| \geq \theta$

(ii) $\left|\bigcap_{v \in V' \cup \{v'\}} I(v)\right| < \left|\bigcap_{v \in V'} I(v)\right|$ for any v' connected to G', i.e., $v' \in V - V'$ such that $\exists v \in V' : (v, v') \in E$.

A connected subgraph G' that satisfies (i) above is called a CIG. A connected subgraph G' that satisfies both (i) and (ii) is called a CCIG. It is closed with respect to the itemset $I(G') = \bigcap_{v \in V'} I(v)$.

An example of an input graph associated with itemsets is shown in **Fig. 1**. **Table 1** contains the outputs when this graph and $\theta = 2$ are given as inputs. Note that $G'' = (V'', E(V''))$ where $V'' = \{v_1, v_4, v_5\}$ is not included in the outputs, because it satisfies (i) since $I(G'') = \{i_1, i_3\}$ but not (ii) since $I(G'_3) = I(G'')$ and $V'_3 \supset V''$.

3. COPINE Algorithm

In this section, we introduce a parallel algorithm that solves the CCIG enumeration problem. First, in Section 3.1, we present the sequential COPINE algorithm proposed in Refs. [3] and [4]. Sec-

[*1] To be exact, this is a connected and induced subgraph due to condition (ii). In this paper, however, we call it a connected subgraph for simplicity.

Table 1 CCIGs obtained from the graph in Fig. 1 with $\theta = 2$.

connected subgraph: G'_i	vertex set : V'_i	common itemset: $I(G'_i)$
G'_1	$\{v_1, v_3, v_4\}$	$\{i_1, i_2\}$
G'_2	$\{v_1, v_3\}$	$\{i_1, i_2, i_4\}$
G'_3	$\{v_1, v_4, v_5, v_6\}$	$\{i_1, i_3\}$
G'_4	$\{v_1, v_4\}$	$\{i_1, i_2, i_3\}$
G'_5	$\{v_1\}$	$\{i_1, i_2, i_3, i_4\}$
G'_6	$\{v_2\}$	$\{i_1, i_5\}$
G'_7	$\{v_4, v_6\}$	$\{i_1, i_3, i_5\}$
G'_8	$\{v_4\}$	$\{i_1, i_2, i_3, i_5\}$
G'_9	$\{v_5\}$	$\{i_1, i_3, i_4\}$

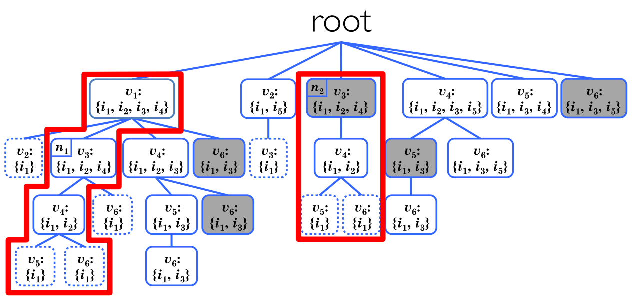


Fig. 2 Search tree for the graph in Fig. 1 ($\theta = 2$).

ond, we prove the correctness of the algorithm and discuss how we can parallelize it in Section 3.2. Based on these discussions, we propose the parallel COPINE algorithm in Section 3.3.

3.1 Sequential COPINE Algorithm

As shown in **Fig. 2**, the COPINE algorithm applies a depth-first search to a search tree consisting of the following components: a pseudo root corresponding to an empty graph, nodes corresponding to graph vertices, and edges corresponding to graph edges. Note that a path from the root to a node represents a connected subgraph, and adding a child node means adding an adjacent vertex to the connected subgraph. The fact that generally there are two or more vertices which can be added to a connected subgraph corresponds to the fact that a node in the search tree can have multiple child nodes. Therefore, after searching all subgraphs derived from a child node, the COPINE algorithm backtracks to the child node, chooses the next child node, and starts searching from the child node. The COPINE algorithm can search all the subgraphs derived from the parent node by repeating this process for all the child nodes.

Since the search tree represents all the subgraphs of an input graph G, we can enumerate all the CCIGs by traversing the tree completely. However, this is in general unrealistic. Therefore in order to reduce the search space, COPINE employs the pruning of the tree edges from which the following three types of subgraphs are derived:

Pruning 1 subgraph we have already visited,

Pruning 2 subgraph whose itemset is smaller than the threshold θ, and

Pruning 3 subgraph not being closed since one of its supergraphs has already been visited and their itemsets are identical.

The first pruning avoids duplicate enumeration by using a technique called gSpan proposed in Ref. [6]. First, we define an arbitrary ordering $v_1 \prec v_2 \prec \cdots \prec v_n$ of all the vertices in the vertex set V. In a straightforward depth-first search to enumerate the connected subgraphs, all the vertices adjacent to a connected subgraph that is represented by a path from the root to a node becomes candidates for the vertex added in the next step. COPINE achieves the first pruning by enumerating all the connected subgraphs according to their total (lexicographical) order derived from the order of vertices. A search tree for the graph in Fig. 1 to which this pruning is applied is shown in Fig. 2. Each label and itemset of a node represents the last added vertex and the common itemset respectively. While $\langle v_1, v_3, v_4 \rangle$ is traversed in Fig. 2, $\langle v_1, v_4, v_3 \rangle = \langle v_1, v_3, v_4 \rangle$ is not traversed because it precedes $\langle v_1, v_4 \rangle$ in the lexicographic order.

The second pruning is achieved by exploiting that the size of the common itemset does not increase when an adjacent vertex is added to a connected subgraph, i.e., the itemset size is monotonically non-increasing from the root to the leaf in a search tree. Therefore, when the size of the common itemset is lower than a threshold θ during a search, no CIG among the descendants has a common itemset size larger than or equal to the threshold. Therefore, the search of the descendants can be skipped. The nodes represented by dashed frames in Fig. 2 are eliminated by this second pruning.

Focusing on the two subtrees surrounded by red frames in Fig. 2 we find that the right subtree is identical to the tail of the left one, including the itemset labels on their nodes. In this case, no descendants of n_2 are needed to be visited since the subgraphs represented by them have supergraphs represented by the left subtree whose itemsets are identical to those of the subgraphs. That is, the subgraphs represented by the right subtree are not closed. To avoid such a duplicate search, we introduce the third pruning as follows.

We prepare an itemset table with entries corresponding to the vertices of a graph. When adding a vertex to a current subgraph during a search, the common itemset of the resulting subgraph is added to the entry corresponding to the added vertex. On this occasion, if the table entry contains a super-itemset of the itemset being added, the search of the descendants of the current search tree node can be skipped. For example, the shaded nodes in Fig. 2 are eliminated by this kind of pruning. The itemset $\{i_1, i_2, i_4\}$ is added to the table entry corresponding to v_3 when n_1 is visited. When n_2 is visited, $\{i_1, i_2, i_4\}$ is to be added to the same table entry again. At this time, since this itemset has been already registered; i.e., a super-itemset (in a broad sense) has been already registered, the search in the direction from n_2 to the leaf is skipped.

On the other hand, if a proper subset of the itemset being added has been registered, it is removed from the entry [*2]. An itemset in the entry that has no inclusive relation with the itemset being added remains stored. For example, the itemset $\{i_1, i_2\}$ is added to the table entry corresponding to v_4 when $\langle v_1, v_3, v_4 \rangle$ is visited. Then, $\{i_1, i_2\}$ is removed from this entry when $\langle v_1, v_4 \rangle$ is visited and the itemset $\{i_1, i_2, i_3\}$, which contains $\{i_1, i_2\}$, is added to the entry.

[*2] Though this removal does not affect the correctness of the algorithm, it prevents an itemset table from bloating.

3.2 Correctness and Application to Parallel Search of Pruning

In this section, we describe the COPINE algorithm more formally and prove its correctness. Then we discuss its parallelization. Specifically, we present the correctness of the sequential search using the pruning methods explained in Section 3.1, and show that we cannot blindly apply Pruning 3 in a parallel search. Next, we show that a correct parallel search is obtained by a partial application of Pruning 3, putting a restriction on a worker when referring to a table entry registered by another worker. Note that by correct pruning we mean that the completeness of the search is preserved when the pruning is applied, i.e., all the subgraphs that satisfy the required conditions are enumerated. On the other hand, our parallel algorithm does not preserve soundness; it can enumerate subgraphs that do not satisfy the conditions. We discuss the soundness in more detail in Section 3.2.4.

3.2.1 Correctness of Pruning 1

In this section, we provide a more formal definition of Pruning 1 and prove its correctness. First, we define a sequence called a *Canonical Spanning Tree* (CST).

Definition 3 (Spanning Tree) For a connected subgraph $G' = (U, E(U))$ of $G = (V, E)$, a sequence $\langle u_1, \cdots, u_n\rangle$ such that $\{u_1, \cdots, u_n\} = U$ is called a *spanning tree* of G' iff all of the following conditions hold.

(1) $1 \leq \forall i \neq \forall j \leq n : u_i \neq u_j$

(2) $1 < \forall j \leq n : 1 \leq \exists i < j$ such that $(u_i, u_j) \in E(U)$

Note that there can be two or more spanning trees for a subgraph G'.

Definition 4 (Canonical Spanning Tree) A spanning tree $T(U) = \langle u_1, \cdots, u_n\rangle$ of the connected subgraph $G' = (U, E(U))$ such that $U = \{u_1, \cdots, u_n\}$ is called *canonical* iff $u_1 = \min[\prec](U)$ and $u_i = \min[\overset{U_{i-1}}{\leftharpoondown}]\,(N(U_{i-1}) - U_{i-1})$ for all i such that $1 < i \leq n$, where U_j, $N(U_j)$, and $\overset{U_j}{\leftharpoondown}$ are defined as follows.

(1) $U_j = \{u_1, \cdots, u_j\}$

(2) $N(U_j) = \displaystyle\bigcup_{k=1}^{j}\{u \mid (u_k, u) \in E(U)\}$

(3) (Depth First Ordering)

$prev(u, U_j) = \arg\max_k\{u_k \mid (u_k, u) \in E(U), 1 \leq k \leq j\}$

$u \overset{U_j}{\leftharpoondown} v \Leftrightarrow u, v \in N(U_j) - U_j \wedge (prev(u, U_j) > prev(v, U_j) \vee (prev(u, U_j) = prev(v, U_j) \wedge u \prec v))$

Note that, for $T(U) = \langle u_1, \cdots, u_n\rangle$, its *ancestor* $\langle u_1, \cdots, u_m\rangle$ $(m < n)$ is canonical by definition. On the other hand, it is not always true that there exists a $u \in V$ such that $T(U) \cdot \langle u\rangle$, i.e., a *direct descendant* of $T(U)$, is canonical. We can form the CST of U by the algorithm in **Fig. 3**.

Second, we define the lexicographic order of connected subgraphs as follows.

Definition 5 (Lexicographical Order of Subgraphs) For a pair of two connected subgraphs $G^1 = (U^1, E(U^1))$ and $G^2 = (U^2, E(U^2))$ of $G = (V, E)$ such that $G^1 \neq G^2$, we denote $G^1 \prec G^2$ iff the following holds for $T(U^1) = \langle u^1_1, \cdots, u^1_m\rangle$ and $T(U^2) = \langle u^2_1, \cdots, u^2_n\rangle$;

$$k = m < n \vee (k < \min(m, n) \wedge u^1_{k+1} \overset{U^{1,2}_k *}{\leftharpoondown} u^2_{k+1})$$

where $k = \arg\max_i\{u^1_i \mid \forall j \leq i : u^1_j = u^2_j\} = |prefix(T(U^1), T(U^2))|$, $U^{1,2}_k = \{u^1_1, \cdots, u^1_k\} = \{u^2_1, \cdots, u^2_k\}$, and $u \overset{U_i *}{\leftharpoondown} v$ is defined as follows for a CST $T(U_i) = \langle u_1, \cdots, u_i\rangle$.

```
function $CST(U)$ begin
  $v \leftarrow \min[\prec](U)$; $U \leftarrow U - \{v\}$; $T \leftarrow \langle v\rangle$; $E \leftarrow E(U)$;
  $C \leftarrow sort(neighbors(v, E))$;
  while $C \neq \langle\rangle$ do begin
    $v \leftarrow car(C)$; $C \leftarrow cdr(C)$;
    if $v \notin U$ then continue;
    $T \leftarrow T \cdot \langle v\rangle$; $U \leftarrow U - \{v\}$; $N \leftarrow sort(neighbors(v, E))$;
    $C \leftarrow N \cdot C$;
  end
  return($T$);
end
```

Fig. 3 Algorithm to form the CST of a connected subgraph.

(1) $N^*(U_i) = \begin{cases} V & i = 0 \\ \displaystyle\bigcup_{j=1}^{i}\{u \mid (u_j, u) \in E\} & i > 0 \end{cases}$

(2) (Depth First Ordering)

$prev^*(u, U_i) = \begin{cases} 0 & i = 0 \\ \arg\max_j\{u_j \mid (u_j, u) \in E,\ 1 \leq j \leq i\} & i > 1 \end{cases}$

$u \overset{U_i *}{\leftharpoondown} v \Leftrightarrow u, v \in N^*(U_i) - U_i \wedge (prev^*(u, U_i) > prev^*(v, U_i) \vee (prev^*(u, U_i) = prev^*(v, U_i) \wedge u \prec v))$

Theorem 1 (Direct Successor of a Subgraph) Let $\{G_1, \cdots, G_N\}$ be the set of all connected subgraphs of G such that $G_1 \prec \cdots \prec G_N$. For $G_i = (V_i, E(V_i))$ $(1 \leq i < N)$ where $T(V_i) = \langle u_1, \cdots, u_n\rangle$, $T(V_{i+1})$ is given by:

$$T(V_{i+1}) = \langle u_1, \cdots, u_k\rangle \cdot \left\langle \min\left[\overset{U_k *}{\leftharpoondown}\right](C_k)\right\rangle$$

where k and C_k are defined by:

(1) $P_i = \{v \mid v \overset{U_{i-1} *}{\leftharpoondown} u_i\} \cup \{u_i\}$

(2) $D_i = \displaystyle\bigcup_{j=1}^{i}\{N^*(U_{j-1}) \cap P_j\}$

(3) $C_i = N^*(U_i) \cap \left(V - \begin{cases} D_{i+1} & i < n \\ D_n & i = n \end{cases}\right)$

(4) $k = \arg\max_j\{C_j \mid C_j \neq \emptyset, 0 \leq j \leq n\}$

Proof: See Appendix A.1.

According to Theorem 1, we can enumerate the CSTs of all the connected subgraphs of a graph $G = (V, E)$ (in the lexicographic order of the corresponding connected subgraphs) by the algorithm in **Fig. 4**. Since each connected subgraph has just one CST, we can enumerate all connected subgraphs of G with no duplications using this algorithm. This means that, by traversing a search tree derived from this algorithm, we can correctly achieve the purpose of Pruning 1, i.e., avoid visiting a subgraph that has been visited already.

3.2.2 Correctness of Pruning 2

The correctness of Pruning 2 is proved by Theorem 2 as follows.

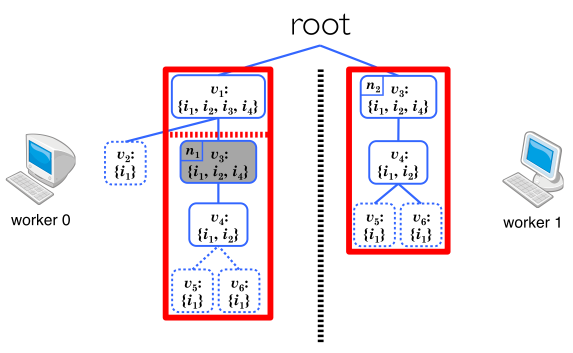
Theorem 2 (Monotonicity of Common Itemset) Let $G^1 = (U^1, E(U^1))$ and $G^2 = (U^2, E(U^2))$ be connected subgraphs of a graph $G = (V, E)$. If $T(U^2) = T(U^1) \cdot T'$, or in other words G^1 and G^2 form an *ancestor-descendant* pair respectively, all of the following conditions hold.

(1) $I(G^1) \supseteq I(G^2)$

```
function $EnumSG(G)$ begin
  $(V, E) \leftarrow G$; $\boldsymbol{L} \leftarrow \emptyset$;
  while $V \neq \emptyset$ do begin
    $v = \min(V)$; $V \leftarrow V - \{v\}$;
    $\boldsymbol{L} \leftarrow ExploreSG(\langle v \rangle, V, E, \langle\rangle, \boldsymbol{L})$;
  end
  return($\boldsymbol{L}$);
end

function $ExploreSG(T, V, E, C, \boldsymbol{L})$ begin
  $\boldsymbol{L} \leftarrow \boldsymbol{L} \cup \{T\}$;
  $N \leftarrow sort(neighbors(v, E))$;
  $C \leftarrow N \cdot C$;
  while $C \neq \langle\rangle$ do begin
    $u \leftarrow car(C)$; $C \leftarrow cdr(C)$;
    if $u \notin V$ then continue;
    $V \leftarrow V - \{u\}$;
    $\boldsymbol{L} \leftarrow ExploreSG(T \cdot \langle u \rangle, V, E, C, \boldsymbol{L})$;
  end
  return($\boldsymbol{L}$);
end
```

Fig. 4 Algorithm to enumerate connected subgraphs.



Itemset table of worker 1

vertex: v	$v.I$
v_1	$\emptyset$
v_2	$\emptyset$
v_3	$\{i_1, i_2, i_4\}$
v_4	$\{i_1, i_2\}$
v_5	$\emptyset$
v_6	$\emptyset$

Fig. 5 Example of excessive pruning ($\theta = 2$).

(2) $\left|I(G^1)\right| \geq \left|I(G^2)\right|$
(3) $\left|I(G^1)\right| < \theta \rightarrow \left|I(G^2)\right| < \theta$

Proof:
(1) It is obvious that $U^1 \subset U^2$. Therefore $I(G^1) = \bigcap_{v \in U^1} I(v) \supseteq \bigcap_{v \in U^2} I(v) = I(G^2)$. □
(2) Trivial by (1). □
(3) Trivial by (2). □

According to Theorem 2, when the size of a current common itemset is less than a threshold during a search, there is no connected subgraph whose common itemset size is not less than the threshold among the descendants. Therefore, Pruning 2 is correct.

3.2.3 Correctness and Application to Parallel Search of Pruning 3

Theorem 3 (Lexicographical Order of CCIG) Let $G^1 = (U^1, E(U^1))$ and $G^2 = (U^2, E(U^2))$ be connected subgraphs of a graph $G = (V, E)$ being $G^1 \prec G^2$. If the tails of $T(U^1) = \langle u^1_1, \cdots, u^1_m \rangle$ and $T(U^2) = \langle u^2_1, \cdots, u^2_n \rangle$ are common, i.e., $tail(U^1) = u^1_m = tail(U^2) = u^2_n$, and $I(G^1) \supseteq I(G^2)$, all of the following criteria hold.
(1) G^2 is not closed with respect to $I(G^2)$.
(2) Any descendant $G' = (U', E(U'))$ of G^2 is not closed with respect to $I(G')$.

Proof: See Appendix A.2.

Theorem 3 proves the correctness of Pruning 3 in a sequential search. Let us look at the search-tree nodes n_1 and n_2 in Fig. 1. We can see that n_2 is searched later than n_2 ($G_1 \prec G_2$), their tails are the same ($tail(U^1) = tail(U^2)$), and the common itemset at n_2 is a subset of the common itemset at n_1 ($I(G^1) \supseteq I(G^2)$). Note that we can recognize that the third condition is satisfied from the table entry corresponding to the vertex $u^1_m = u^2_n$ when we visit n_2. Since the prerequisites of Theorem 3 are satisfied, n_2 and any descendant of n_2 are not CCIGs. Therefore, we can skip searching them.

On the other hand, from Theorem 3 we see that we cannot apply Pruning 3 to a parallel search straightforwardly. The prerequisite of Theorem 3 means that *a worker can refer to itemsets for Pruning 3 only if it had been registered earlier in a sequential search with Pruning 1*. Following this restriction, we can correctly apply Pruning 3 in a parallel search.

In **Fig. 5** we show an example of losing completeness in the absence of this restriction. We suppose that worker 0, traversing the left subtree, and worker 1, traversing the right subtree, have visited $\langle v_1, v_2 \rangle$ and $\langle v_3, v_4, v_6 \rangle$ respectively. At this time, the itemset table of worker 1 is shown at the bottom of Fig. 5. After that, worker 0 will visit $\langle v_1, v_3 \rangle$. If worker 0 refers to $\{i_1, i_2, i_4\}$ in the itemset table of worker 1, worker 0 will skip the search from $\langle v_1, v_3 \rangle$ to the leaf nodes because worker 0 recognizes that the conditions for Pruning 3 are satisfied. As a result, $\{v_1, v_3\}$ and subgraphs including $\{v_1, v_3\}$ are excluded from the search result. For example, $\{v_1, v_3, v_4\}$ is not enumerated erroneously when $\theta = 2$.

3.2.4 Soundness of Search

We discuss the soundness of the search in this section.

Theorem 4 (CCIG) A connected subgraph $G' = (U, E(U))$ of a graph $G = (V, E)$ is closed with respect to $I(G')$ iff all of the followings hold.
(1) For any direct descendant $G^d = (U^d, E(U^d))$ of G', it holds that $I(G') \neq I(G^d)$.
(2) For any connected subgraph $G^p = (U^p, E(U^p))$ of G such that $G^p \prec G'$ and $tail(U^p) = tail(U')$, it holds that $I(G^p) \not\supseteq I(G')$.

Proof: See Appendix A.3.

Theorem 4 guarantees that, when Prunings 1–3 are completely applied, we can enumerate all the CCIGs without any redundant outputs, i.e., soundly and completely, by asserting a visiting subgraph G' to be closed if either of the following is satisfied for any direct descendent G'' of G'; G'' is pruned by one of Prunings 1–3 or $|I(G')| > |I(G'')|$. Thus, using the sequential search algorithm in **Fig. 6**, we can completely enumerate all the CCIGs of a graph $G = (V, E)$ whose common itemset size is not less than θ. In this algorithm, Pruning 1 corresponds to the fact that connected

```
 1: function $EnumCCIG(G)$ begin
 2:   $(V, E) \leftarrow G$;  $\boldsymbol{L} \leftarrow \emptyset$;
 3:   for $\forall v \in V$ do $v.I \leftarrow \emptyset$;
 4:   while $V \neq \emptyset$ do begin
 5:     $v = \min(V)$;  $V \leftarrow V - \{v\}$;
 6:     if $|I(v)| < \theta$ then continue;
 7:     if $\exists I$ s.t. $I \in v.I \wedge I \supseteq I(v)$ then continue;
 8:     $\boldsymbol{L} \leftarrow ExploreCCIG(v, \langle v \rangle, I(v), V, E, \langle\rangle, \boldsymbol{L})$;
 9:   end
10:   return($\boldsymbol{L}$);
11: end
12:
13: function $ExploreCCIG(v, T, I, V, E, C, \boldsymbol{L})$ begin
14:   $closed \leftarrow$ true;
15:   $v.I \leftarrow v.I \cup \{I\}$;
16:   $N \leftarrow sort(neighbors(v, E))$;
17:   $C \leftarrow N \cdot C$;
18:   while $C \neq \langle\rangle$ do begin
19:     $u \leftarrow car(C)$;  $C \leftarrow cdr(C)$;
20:     if $u \notin V$ then continue;
21:     $V \leftarrow V - \{u\}$;  $I' \leftarrow I \cap I(u)$;
22:     if $|I'| < \theta$ then continue;
23:     if $\exists I''$ s.t. $I'' \in u.I \wedge I'' \supseteq I'$ then continue;
24:     if $I' = I$ then $closed \leftarrow$ false;
25:     $\boldsymbol{L} \leftarrow ExploreCCIG(u, T \cdot \langle u \rangle, I', V, E, C, \boldsymbol{L})$;
26:   end
27:   if $closed$ then $\boldsymbol{L} \leftarrow \boldsymbol{L} \cup \{T\}$;
28:   return($\boldsymbol{L}$);
29: end
```

Fig. 6 Sequential algorithm to enumerate all CCIGs of graph.

subgraphs are scanned in the same way as in Fig. 4, managing V, N, and C. Pruning 2 is performed by detecting the inferiority of the itemset cardinality to the threshold θ at lines 6 and 22. Pruning 3 is achieved by adding an itemset to the itemset table entry at line 15 and detecting an inclusive relation between itemsets at line 23.

In a parallel search, on the other hand, a search tree node visited by a worker may have some *precedent* nodes left unvisited by other workers though they should have been visited in the sequential search. Therefore, it is virtually impossible to perform Pruning 3 perfectly and thus the parallel version of the algorithm in Fig. 6 will assert a CIG to be closed though in reality it is not. In order to obtain a sound set of CCIGs, we need to eliminate CIGs that do not satisfy condition (ii) of Definition 2 after the search. Note that although the elimination can be done by constructing the complete itemset table from the set of imperfect ones and examining the itemset of the tail vertex of each candidate CIG against the table, the elapsed time for this process is not considered in the performance evaluation in Section 5.

3.3 Parallel Algorithm

We parallelized the algorithm in Fig. 6 by dividing a search tree and assign a unique set of subtrees to each worker. This can be implemented by dividing the two `while` loops in Fig. 6 (lines 4–9 and 18–26) into appropriate units and executing them in parallel.

Each worker traverses assigned subtrees almost in the same way as in the sequential search. Prunings 1 and 2 can directly be applied to the parallel algorithm. On the other hand, we need to apply the restriction described in Section 3.2.3 to Pruning 3. In Fig. 6, a worker refers to $u.I$ (a set of itemsets registered when the vertex u is added to a subgraph) to check whether it can apply Pruning 3. In our parallel algorithm, however, it refers to each element (itemset) in $u.I$ only if the element had been registered earlier in a sequential search.

In order to check whether the element had been registered earlier in a sequential search, we must devise an efficient way to let each itemset have some sequential ordering information of its registration. From a performance perspective, it is also important to consider how to divide a search tree and assign subtrees to workers, and how to share information in the itemset table among workers. We discuss these issues in Section 4.

4. Parallel Implementation

In order to parallelize the COPINE algorithm, we divide a search tree growing in the COPINE execution process and assign a set of subtrees to each worker running in parallel. One of the most important issues in a parallel search is to balance load among workers. However, before the search, we cannot evaluate the size of each subtree whose root is a search tree node. Therefore, it is too difficult to assign subtrees to balance load among workers in advance. For example, it is clear that the static load balancing that divides the node sets consisting of the children of the root into the same number of subsets and assign them to the workers should cause a significant load imbalance due to the divergent sizes of the subtrees.

In this research, we use the dynamic load balancing strategy called "work-stealing," where an idle worker steals a part of another worker's task. While it is difficult to implement this strategy using a programming framework that is oriented to static load balancing such as OpenMP, it is known that a task-parallel language, such as Cilk [7] or Tascell [5], drastically eases the implementation difficulty. In most of these languages, programmers need to describe: (1) a series of operations, part of which can be assigned to another worker as a task, e.g., a loop to be parallelized, and (2) information passing along a task assignment to a worker, e.g., data referred to or updated by the worker. During the execution of the program, a runtime creates tasks and assigns them to workers automatically. Thus, we can implement a parallel COPINE that treats each subtree as a parallelization unit and uses dynamic load balancing, by parallelizing the two `while` loops in Fig. 6.

Among the several task-parallel languages, we chose Tascell, which can achieve a higher performance, especially in parallel backtrack search. In the remainder of Section 4, we present the mechanism of our parallel version of COPINE implemented in Tascell.

4.1 Dynamic Load Balancing in Tascell

A Tascell worker executes its own task sequentially and does not spawn a task until it receives a work-stealing request (task request) from another worker. That is, when the worker reaches a statement where a task can be spawned (e.g., a parallel loop), it just remembers the possibility at this point, and then executes the statement *as if it chose a completely sequential execution*. For example, in a parallelized implementation of the CCIG enumer-

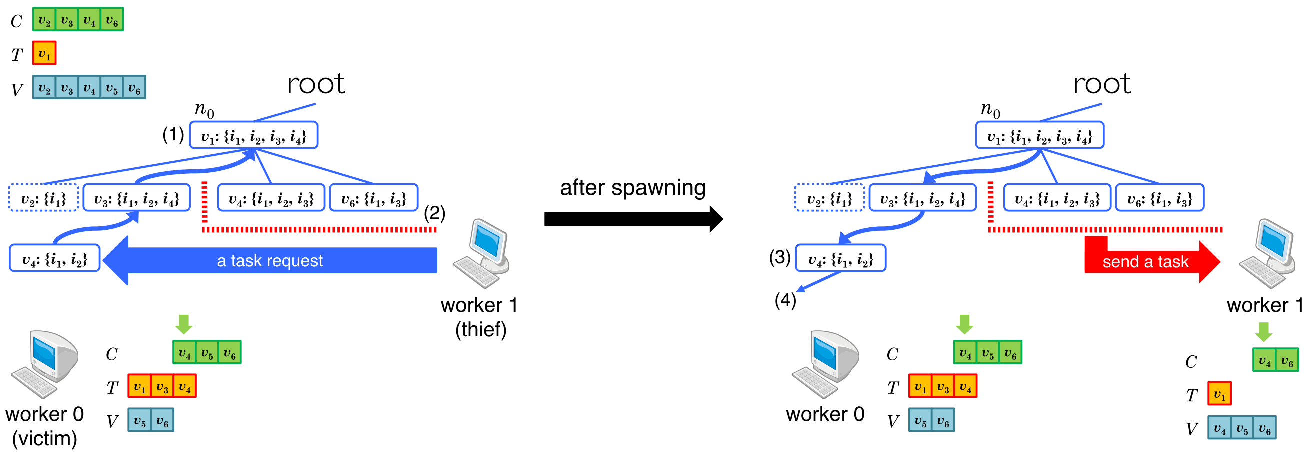


Fig. 7 Spawning a task lazily in the backtrack search of COPINE.

Table 2 Workspaces managed by each worker in CCIG enumeration.

Variable Name in *ExploreCCIG*( )	Content
T	CST of the visiting subgraph
C	set of vertices adjacent to *T*
V	set of candidates for the vertex added in the next step

ation based on the algorithm in Fig. 6, each worker has its own workspace that holds data as shown in **Table 2**, and it performs the search, updating the data at each step. In addition, in order to achieve Pruning 3, workers maintain their own itemset tables or a single shared one apart from the workspace in Table 2. Here, whether making tables private to workers or sharing a single table among workers depends on the sharing methods shown in Section 4.3.

When a worker (victim) receives a task request from another worker (thief), it backtracks to the oldest point among the parallelizable (task-spawnable) points, that is, the point where the largest task can be spawned, and then spawns a task *as if it changed the choice of execution to parallel from sequential made past*. Then it allocates and initializes a new workspace for the task by making a copy of the victim's workspace in the state after the backtracking (the connected subgraph, the adjacent vertex, etc.).

Figure 7 illustrates how a task is spawned lazily, that is, a task is spawned only after a worker receives a task request. Suppose that n_0 is the oldest task-spawnable point of worker 0. When worker 0 receives a task request from worker 1,

(1) it backtracks to n_0 performing *undo* operations to restore the state of its workspace at n_0, and then

(2) spawns a task to traverse the right subtree.

Note that worker 0 creates a part of unexecuted iterations of a parallel `for` loop at n_0 as a task and the number of iterations left to worker 0 considerably affects the performance of our implementation. We discuss this in Section 4.4.

After sending the task to the thief worker (worker 1),

(3) worker 0 returns from the backtracking while performing *redo* operations to restore the state of its workspace before backtracking, and then

(4) it resumes its own task.

In contrast to Tascell, there is another task creation technique called Lazy Task Creation (LTC) [8], which is employed in "multithreaded languages" such as Cilk. In LTC, parallel execution units called "logical threads" [*3] are created and tasks are assigned to them at the beginning of a series of operations, part of which can be spawned as a parallel task, such as a parallelizable loop. When a thief worker requires a task, the oldest logical thread of a victim worker is selected and assigned to the thief. Since a worker in LTC can steal a task only by selecting and taking out one of logical threads that have been already created, the cost of a stealing operation in LTC is smaller than that of Tascell, which involves backtracking. On the other hand, the overhead in a sequential execution in LTC in the absence of task requests is larger than that in Tascell, since, in LTC, a logical thread that may not be assigned to another worker can be created as a task at each beginning of any task-spawnable operation. Although there is a trade-off between the cost of a steal operation and the serial overhead, Tascell is better in many search problems since the number of created tasks is much smaller than the number of potential tasks [5]. As shown later in Section 5, while the number of potential tasks (the number of visits to vertices during the search) ranges from millions to tens of millions, the number of created tasks ranges from thousands to tens of thousands at the most. Therefore, it is clear that Tascell has an advantage over LTC.

In Section 6.1, we perform a detailed comparison of Tascell with multithreaded languages from other viewpoints.

4.2 Challenges in Parallelized Implementation

As stated in Section 3.3, Prunings 1 and 2 can directly be applied to the parallel search. On the other hand, in order to achieve Pruning 3 efficiently, a worker needs to refer to itemset table entries registered by another worker under the restriction described in Section 3.2.3. Therefore, besides the implementation issue about efficient sharing of the information in the table among workers, we need to devise a method to let each worker know whether a table entry registered by another worker is safely referred to according to the location of the registration in the search

[*3] It is called this since it is executed under management of a language system and differs from a "physical thread," which is executed under OS management and corresponds to a worker to which actual execution of tasks is assigned.

tree.

The effectiveness of Pruning 3 is also deeply affected by the task creation strategy of deciding which portion of subtrees, or in other words unscanned iterations of a parallel loop, is chosen for a task to be created. According to the idea of the oldest-first task creation strategy by which a worker intends to create as large a task as possible, it is reasonable to make victim's and thief's unscanned node sets equal-size, i.e., divide victim's unscanned node set in half. However, from the viewpoint of the effect of Pruning 3, i.e., for greater use of itemsets registered by other workers, each search tree node should be visited after as many nodes preceding it in the sequential search have been visited as possible. The half-and-half strategy deprives the chance of a thief worker to use information in table entries registered by a victim worker since many unscanned nodes are left for the victim worker, making the pruning less effective. Thus, the following strategies are expected to improve the effectiveness of the pruning; (1) assigning all the unscanned nodes in a parallel loop to the thief worker to give it all the pruning information except that corresponding to the subgraph being traversed by the victim worker, or (2) introducing a certain sequentiality into the higher layer of the search tree to ensure that the itemset table entries registered by a set of parallel tasks are passed to the serially succeeding set without loss of information. However, both of them could lead to an increase in the number of work-steals and reduce the traversing speed since the former causes the size of victim's task to be reduced while the latter causes the entire size of each sequential task-set to be reduced. We need to develop an efficient task creation strategy in consideration of the trade-off between the effectiveness of the pruning and the traversing speed.

4.3 Itemset Table Sharing

Considering the restriction described in Section 3.2.3 and the implementation issues discussed in Section 4.2, we developed the following sharing methods of the itemset table to examine the performance effect of the sharing.

4.3.1 Sharing Method 0: Non-registering Method

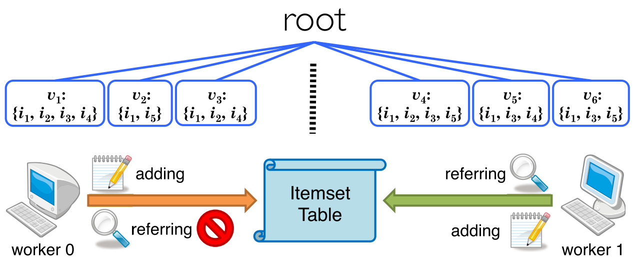
No worker registers any itemsets during a search, that is, Pruning 3 is not applied. With this method, the size of search space remains huge. We implemented this method just to evaluate the effect of Pruning 3 and the performance of Tascell for a search algorithm that does not have any sequential dependencies. Since the management cost of itemset tables is completely eliminated, this is a good method for a graph for which Pruning 3 does not work.

4.3.2 Sharing Method 1: Non-sharing Method

Each task manages its own itemset table, and no table information is shared among tasks. After a thief worker steals a task, it starts the sub-search corresponding to the task with an empty itemset table. Since each worker refers to and updates only its own table at every search step, there is no cost for sharing table information. However, the effect of Pruning 3 is limited since a worker cannot use any itemsets registered in other tasks (including another task that was executed by the worker itself).

4.3.3 Sharing Method 2: Replicating Method

Just as in the non-sharing method, each task manages its own itemset table. The difference from the non-sharing method is that, when a thief worker steals a task from a victim worker, the victim makes a copy of its own table and passes it to the thief. In the work-stealing mechanism of Tascell, the search range of a task spawned by a victim always corresponds to the subsequent search from a certain future point of the victim's search. Therefore, the thief worker executing the spawned task safely refers to itemsets in the table preserving the completeness of the search.



root
v_1: $\{i_1, i_2, i_3, i_4\}$
v_2: $\{i_1, i_5\}$
v_3: $\{i_1, i_2, i_4\}$
v_4: $\{i_1, i_2, i_3, i_5\}$
v_5: $\{i_1, i_3, i_4\}$
v_6: $\{i_1, i_3, i_5\}$
adding
referring
Itemset Table
referring
adding
worker 0
worker 1

Fig. 8 Limitation of references to the itemset table in the fully-sharing method.

In addition to the costs of the non-sharing method, this method adds the cost of making a copy of the itemset table on each steal. A worker can use parts of table information registered by other workers.

4.3.4 Sharing Method 3: Fully-sharing Method

All the workers share a single itemset table with a lock for each table entry for mutual exclusion. In order to satisfy the restriction described in Section 3.2.3, we associate a task ID to each task and add the ID to each itemset registered in an itemset table entry to denote where in a search tree the itemset is registered. That is, the later the itemset is registered in a sequential search, the greater the ID is associated with the itemset. A worker can use an itemset in the itemset table only if the task ID of the itemset is not greater than that of the task being executed. For example, in **Fig. 8**, worker 0 and worker 1 are traversing the left and the right sub-search tree respectively in parallel. Here, worker 1 can use itemsets registered by worker 0, but, worker 0 cannot use itemsets registered by worker 1.

Although this method adds cost for locks and is difficult to implement in distributed memory environments, a worker can use itemsets registered by other workers immediately.

Since there are too many search tree nodes to associate a unique ID with each node, we manage task IDs as follows.

- We associate a pair of two 64-bit unsigned integers, minID and maxID (minID $\leq$ maxID), with each task as its task ID.
  - We associate minID $=0$ and maxID $=2^{64}-1$ (the maximum number that can be represented by a 64-bit unsigned integer) with the root task, which is at first assigned to a certain worker at the beginning of a search.
  - When a task division occurs by a task request, we divide the range [minID, maxID) by an appropriate integer i within the range [minID, maxID) (in our current implementation, $i = \lfloor(\text{minID} + \text{maxID})/2\rfloor$), updating maxID of the task of the victim worker to i, and setting i and the value of maxID of the victim's task before the update to minID and maxID of the spawned task, respectively. An example of such a division is shown in **Fig. 9**.
- When a worker registers an itemset I to the itemset table, the

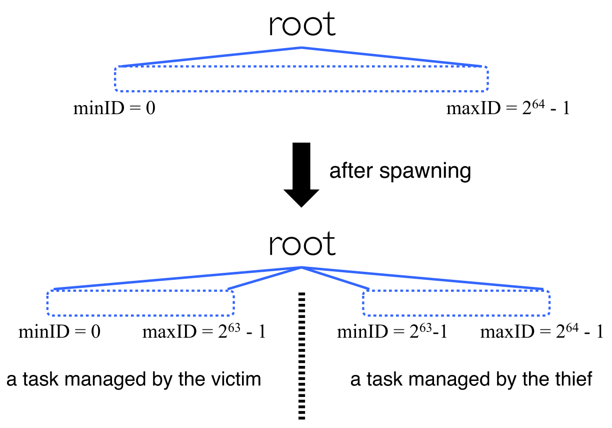


Fig. 9 Division of a task ID range on spawning a task.

value of the running task's minID is added to I. At this time, if a proper subset I' of I has been registered and the minID of I' is greater than the running task's minID, I' is removed from the entry. The others are kept registered.

- When a worker refers to an itemset I in the itemset table for Pruning 3, it compares the minID of I with the minID of the running task. The worker use I for Pruning 3 iff the minID of I is not greater than that of the task.

In this management technique, the order of tasks can be defined by the order of their minIDs. Since this order of tasks is equivalent to the visiting order of corresponding subtrees in the sequential search, we can apply Pruning 3 without loss of the completeness using this technique.

The range [minID, maxID) is divided recursively and minID may become equal to maxID after a certain number of divisions. Since we cannot divide such a range any more, a worker executes a task with such a range sequentially. Though such a sequential task might cause a load imbalance if its size were large, our implementation with 64-bit unsigned integers makes the root task divided up to 64 times recursively so that the resulting sequential tasks are sufficiently small. In fact, we confirmed that the execution time of a sequential task is about 23 ms at maximum in Section 5. Thus this management technique does not cause a considerable load imbalance.

4.4 Task Creation Strategy

As explained in Section 4.1, a Tascell worker backtracks to the oldest task-spawnable point and spawn a task when it receives a task request from another worker. In our Tascell program, there are parallel `for` loops corresponding to the two `while` loops [*4] in Fig. 6. The search function corresponding to *ExploreCCIG*( ) is called recursively in each loop. Therefore, the victim worker chooses the oldest loop [*5] from parallel `for` loops that have unexecuted iterations, and then spawns a part of the iterations (unscanned nodes set) as a new task. The ratio of iterations left for the victim and given to the thief was fixed to half-and-half by Tascell's implementation, disabling us to change it in our program.

In many parallel backtrack search algorithms, this task creation strategy works reasonably well to increase the size of each task and thus to reduce the number of steals. As discussed in Section 4.2, however, this may not be the best strategy for our parallel COPINE algorithm since, in order to make Pruning 3 effective, vertices that are visited early in the sequential search should be visited as early as possible also in the parallel search.

Therefore, we developed a new task creation strategy in which the number of iterations left for a victim worker is less than half of the unexecuted iterations. The more iterations are stolen, the earlier spawned task can traverse a subtree that is traversed earlier in the sequential search. However, when the division ratio is set to an extremely small value, the tasks of the victim and the thief are imbalanced, and the number of steals will increase. In order to enable a programmer to change the division ratio at the program level, we enhanced the parallel-`for` construct of Tascell.

[*4] Each of the `while` loops can be implemented as a `for` loop whose number of iterations is fixed at the beginning of the loop.

[*5] This means the outermost loop in nested loops when we consider a recursive call as a dynamic creation of a loop in the nest.

Table 3 Evaluation environment.

	Appro GreenBlade 8000 (1 node)
CPU	Intel Xeon E5 2.6 GHz 8-core × 2 (1 thread per core)
Memory	DDR3-1600 64 GB
OS	Red Hat Enterprise Linux Server release 6.2 (Santiago)
Compiler	GCC 4.4.6 with `-O3` optimizer
Worker	Created by `pthread_create` with `PTHREAD_SCOPE_SYSTEM`
Lock	A `pthread_mutex_t` lock is attached to each entry (for Sharing Method 3: Fully-Sharing Method)

Table 4 Characteristics of the graph used in the evaluation.

Parameter	Value
$\|V\|$	15,425
$\|E\|$	239,063
$\|I\|$	158
Average degree	29.2
Diameter of G	12
# of vertices in the largest weak connected component	15,061
# of vertices in the smallest weak connected component	1
Average number of items in each vertex	9.42

5. Performance Evaluation

5.1 Evaluation Method

To evaluate the implementations described in Sections 4.3 and 4.4, we measured their performance on a single node of Appro GreenBlade 8000, the supercomputer of ACCMS, Kyoto University. The evaluation environment is summarized in **Table 3**. We used a real protein network as the input. **Table 4** shows the characteristics of this graph. The threshold of the common itemset θ was set to $\theta = 5$ except in the non-registering method (sharing method 0). With the non-registering method, since the search did not finish within a realistic time when $\theta = 5$, we set θ to 13. We also compared the performance of each implementation with that of the sequential COPINE implementation written in C.

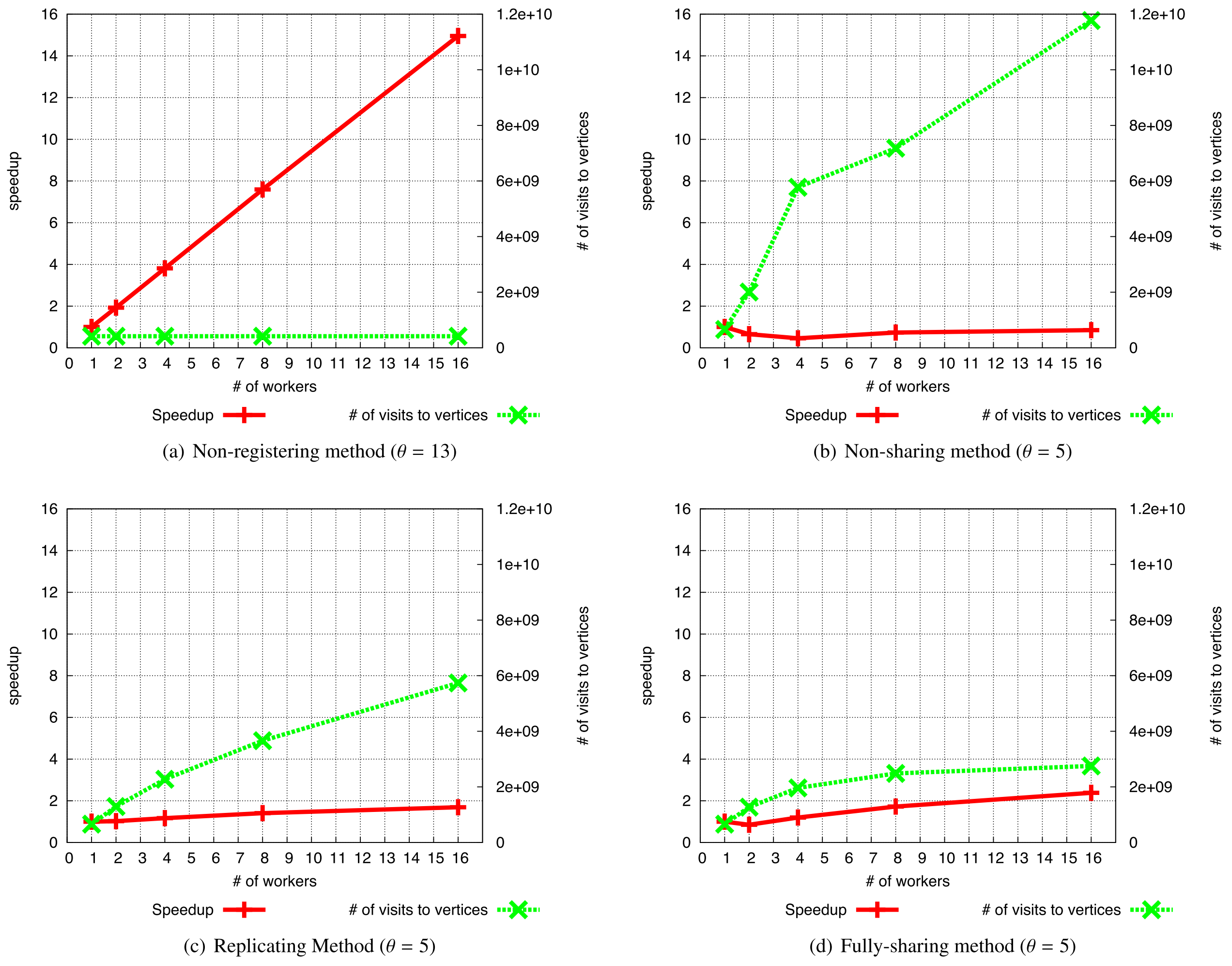
Note that we executed a program three times for each measurement of elapsed time and employed the arithmetic mean of the measured results to be shown in the tables and the charts in this section. There was no great variability among the three samples in each evaluation.

5.2 Performance with Standard Task Creation Strategy

We measured the performance of each sharing method with the standard task creation strategy of Tascell, that is, the num-

Table 5 Results of performance evaluation with the standard task creation strategy.

impl.	θ	n	exec. time [s]	speedup (vs. C)	speedup (vs. 1 worker)	# of visits to vertices (total amt. of all workers)	# of visits to vertices/s (avg. among workers)	# of task creations	task creation time [s]	lock rate [%]
C	5	1	41.3	1	—	659,689,891	15,973,121	—	—	—
method 1	5	1	48.4	0.854	1	659,689,891	13,636,611	0	0	—
method 1	5	2	74.0	0.558	0.653	2,007,586,879	13,557,887	31	0.0616	—
method 1	5	4	105	0.392	0.459	5,772,863,716	13,683,207	294	0.355	—
method 1	5	8	66.0	0.625	0.733	7,179,640,913	13,589,994	1,213	2.00	—
method 1	5	16	70.0	0.725	0.849	11,770,801,051	12,905,555	3,962	5.81	—
method 2	5	1	48.4	0.854	1	659,689,891	13,636,611	0	0	—
method 2	5	2	47.2	0.875	1.03	1,293,331,815	13,705,380	27	0.103	—
method 2	5	4	41.4	0.997	1.17	2,275,527,823	13,733,201	109	0.414	—
method 2	5	8	34.3	1.20	1.41	3,660,133,631	13,327,540	433	2.04	—
method 2	5	16	28.6	1.44	1.69	5,739,857,890	12,532,215	1,177	5.71	—
method 3	5	1	50.3	0.821	1	659,689,891	13,118,018	0	0	0
method 3	5	2	59.0	0.700	0.853	1,273,282,195	10,797,569	33	5.44×10^{-4}	0.0172
method 3	5	4	42.0	0.984	1.12	1,962,681,152	11,688,358	435	0.00431	0.0908
method 3	5	8	29.2	1.42	1.72	2,483,872,565	10,644,751	675	0.00960	0.281
method 3	5	16	21.1	1.96	2.38	2,753,644,092	8,150,603	2,113	0.00440	0.811
C	13	1	0.00593	1	—	85,165	14,361,720	—	—	—
method 0	13	1	139	4.27×10^{-5}	1	417,311,281	3,006,393	0	0	—
method 0	13	2	72.0	8.24×10^{-5}	1.93	417,311,281	2,898,815	47	4.35×10^{-4}	—
method 0	13	4	36.4	1.63×10^{-4}	3.81	417,311,281	2,863,362	589	0.00609	—
method 0	13	8	18.3	3.24×10^{-4}	7.59	417,311,281	2,779,165	3,937	0.0364	—
method 0	13	16	9.28	6.39×10^{-4}	15.0	417,311,281	2,809,745	8,346	0.113	—



(a) Non-registering method ($\theta = 13$)
(b) Non-sharing method ($\theta = 5$)
(c) Replicating Method ($\theta = 5$)
(d) Fully-sharing method ($\theta = 5$)

Fig. 10 Speedup and the number of node visits with the standard task creation strategy.

ber of iterations left for a victim when dividing a parallel loop is half of the unexecuted iterations. We evaluated each of the four sharing methods; the non-registering, non-sharing, replicating, and fully-sharing methods with 1, 2, 4, 8, and 16 workers.

The evaluation results are shown in **Table 5**. **Figure 10** shows the speedup relative to one-worker execution and the number of visits to vertices for each execution with a worker population. Note that, in Table 5, methods 0–3 denote the non-registering, non-sharing, replicating, and fully-sharing methods respectively. θ denotes the threshold of the common itemset, and n denotes the number of workers. "Task creation time" means the accumulated total time required for initializing task objects and copying workspaces. "Lock rate" is the percentage of lock contentions in all acquisitions.

5.2.1 Result of Non-registering Method

Since Pruning 3 is not applied, the search space of the parallel search with multiple workers is identical to that of the sequential search. Therefore, the speedups relative to one worker are almost ideal and we confirm that the overhead caused by the Tascell mechanism is sufficiently small. However, the size of the search space (the number of visits to vertices) is approximately 5,000 times as large as that of the sequential COPINE implementation in C which uses Pruning 3. As a result, the execution time with one worker is more than 20,000 times as long as that of the sequential implementation, and it is clear that this method is definitely inapplicable to the graph used in this evaluation.

5.2.2 Result of Non-sharing Method

The execution time with one worker is approximately 17.2% worse (higher by 7.1 s) than that of the sequential implementation in C. This is mainly due to the cost of the Tascell mechanisms such as polling. Since itemset tables are not shared among workers, the effect of Pruning 3 is limited and the total number of visits to vertices in multiple worker executions is still much larger than that of one worker. Therefore, we could not obtain the performance improvement.

In Table 5, we can see that the per-task creation time in the non-sharing method (method 1) is longer than that in the non-registering method (method 0). This is due to the cost of initializing a new itemset table at each task creation in the non-sharing method. Since the number of task creations in the non-sharing method is large, the total time required for task creation in this method is close to that of the replicating method, which requires making a copy of an itemset table at each task creation.

5.2.3 Result of Replicating Method

In the replicating method, a victim worker makes a copy of its own table at each task creation so that a thief worker can use a part of table information registered by other workers. Therefore, the number of visits to vertices in multiple worker executions is smaller than that in the non-sharing method, and we achieved a 1.69 times speedup with 16 workers (1.44 times speedup relative to C).

The accumulated total time required for task creation with 16 workers is 5.71 s, that is, the average time per worker is 0.357 s, being insignificant in the execution time of 21.1 s.

5.2.4 Result of Fully-sharing Method

The performance of this method is better than for the replicating method (method 2). However, we could not obtain a sufficient performance improvement in parallel executions. Although the accumulated total time required for task creation is quite short since it is not necessary to make a copy of an itemset table at each task creation, the traversing speed (the number of visits to vertices per second) in multiple worker executions is lower than that in the non-sharing method (method 1), the replicating method (method 2), and this method with one worker. This is due to the cost of acquiring a lock at each search step.

The total number of visits to vertices with 16 workers is 4.17 times as large as that with one worker. To reduce this, we need to improve the task creation strategy in consideration of the effect of Pruning 3.

5.3 Effect of Dividing Ratio of Task

We evaluated the effect of the number of iterations left for a victim worker to the effectiveness of Pruning 3. We measured the performance of the replicating method (method 2) and the fully-sharing method (method 3) since a worker can use table information registered by other workers in these methods. We set the number of iterations left for a victim worker from unexecuted iterations as follows:

setting by the number of iterations 1 [*6], and 10–200 in units of 10, and

setting by the ratio $k/2$, $k/4$, $k/8$, $k/16$, $k/32$, $k/64$, and $k/128$ for the number of unexecuted iterations k.

When the number of unexecuted iterations is less than the specified number of iterations, a parallel loop is divided using the standard creation strategy. We measured the performance of both sharing methods only with 16 workers. In addition, with the setting with only one iteration left for the victim worker (the setting expected to bring the highest effectiveness of Pruning 3), our evaluation was carried out with 1, 2, 4, and 8 workers.

Figure 11 (replicating method) and **Fig. 12** (fully-sharing method) show the execution time and number of visits to vertices for each setting of the number of iterations (a) and the ratio left for a victim worker (b). The results with the setting in which one iteration is left for a victim are shown in **Table 6** and **Fig. 13**.

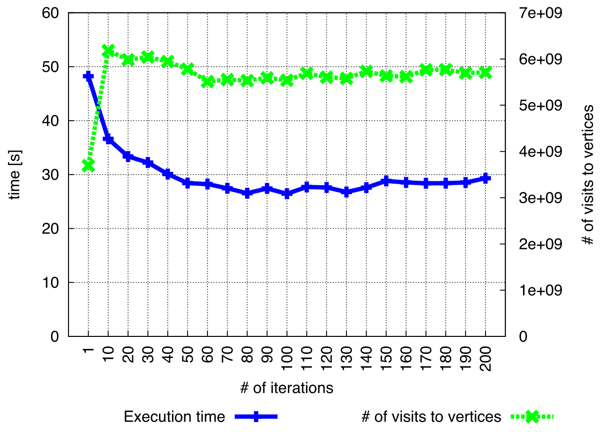
5.3.1 Result of Replicating Method

Comparing the results of the replicating method in Tables 5 and 6, it is found that the total number of visits to vertices is reduced by leaving only one iteration for a victim worker. In contrast, the traversing speed considerably decreases due to increase in the number of task creations. Since a victim worker makes a copy of its own table at each task creation in the replicating method, the cost of task creation is higher than for the other sharing methods. It thus has a negative and serious impact on the overall performance. For example, the total task creation time in method 2 with 16 workers in Table 6 (54.7 s) is 9.58 times as long as that in Table 5 (5.81 s), resulting in a protraction of the overall execution time by 20 second or about 70%. This tendency, that the reduction of the iterations worsens the overall performance while it shrinks the search space, is also visible in Fig. 11.

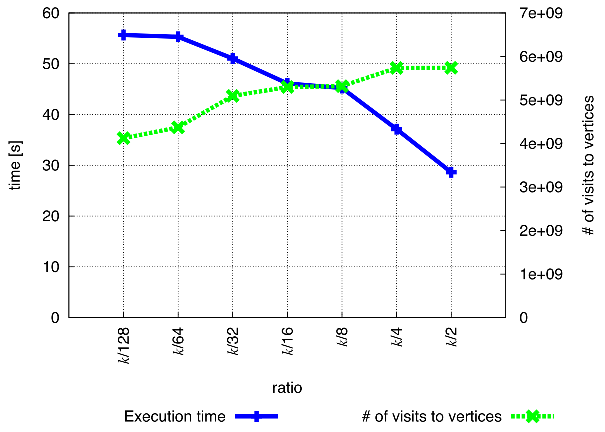
5.3.2 Result of Fully-sharing Method

The comparison of the results of the fully-sharing method in Tables 5 and 6 gives us an observation different from Section 5.3.1. That is, by leaving only one iteration for a victim worker, we successfully reduce *both* the search space size and the overall execution time to have the highest 4.07-fold 16-worker speedup among our experiments. This is partly because the traversing speed degradation compared to the half-and-half is

[*6] This means that only the iteration being executed by a victim worker is left. That is, "unexecuted iterations" here include an iteration being executed.

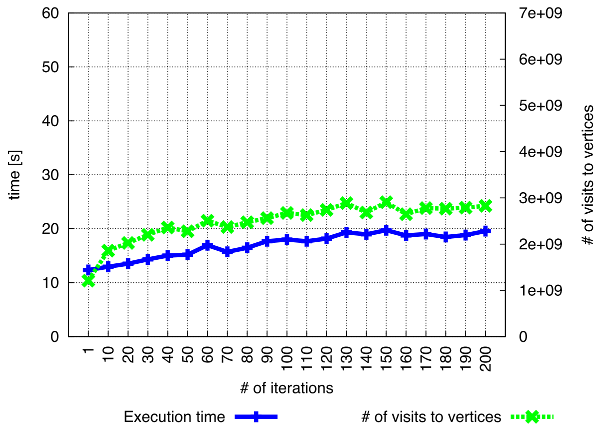


(a) Setting the number of iterations left for a victim directly

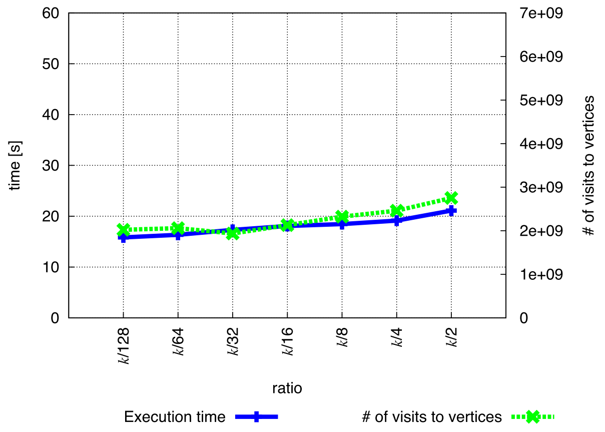


(b) Setting the number of iterations left for a victim by the ratio to the number of unexecuted iterations

Fig. 11 Execution time and the number of visits to vertices versus the number/ratio of iterations left for a victim worker (replicating method with 16 workers).



(a) Setting the number of iterations left for a victim directly



(b) Setting the number of iterations left for a victim by the ratio to the number of unexecuted iterations

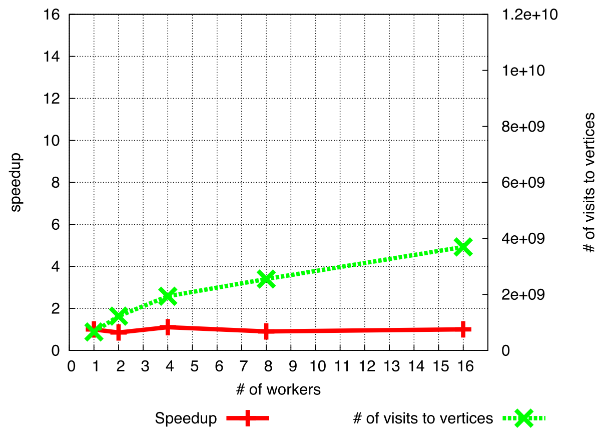
Fig. 12 Execution time and the number of visits to vertices versus the number/ratio of iterations left for a victim worker (fully-sharing method with 16 workers).

Table 6 Results of performance evaluation with the setting that leaves one iteration for a victim worker on task creation.

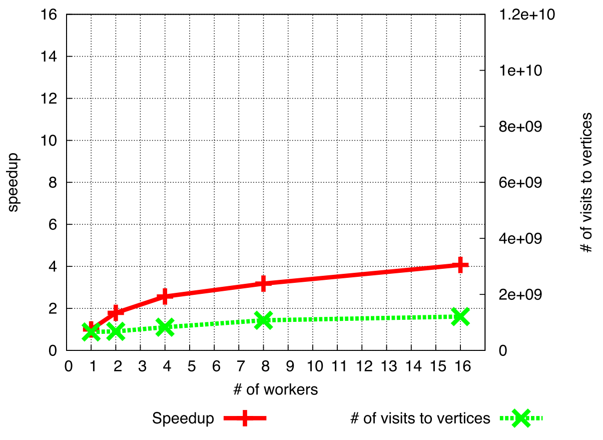
impl.	θ	n	exec. time [s]	speedup (vs. C)	speedup (vs. 1 worker)	# of visits to vertices (total amt. of all workers)	# of visits to vertices/s (avg. among workers)	# of task creations	task creation time [s]	lock rate [%]
C	5	1	41.3	1	—	659,689,891	15,973,121	—	—	—
method 2	5	1	48.4	0.854	1	659,689,891	13,636,611	0	0	—
method 2	5	2	56.4	0.732	0.858	1,226,497,028	10,871,687	4,140	12.8	—
method 2	5	4	43.4	0.946	1.11	1,936,382,708	11,087,120	2,506	8.04	—
method 2	5	8	53.4	0.773	0.906	2,549,093,537	5,966,202	10,085	32.0	—
method 2	5	16	48.2	0.856	1.00	3,693,889,961	4,787,149	14,575	54.7	—
method 3	5	1	50.3	0.821	1	659,689,891	13,118,018	0	0	0
method 3	5	2	28.1	1.47	1.79	683,238,709	12,177,129	52	0.00353	0.0155
method 3	5	4	19.6	2.11	2.57	830,420,056	10,611,666	185	0.0144	0.0568
method 3	5	8	15.8	2.61	3.18	1,076,032,858	8,511,302	1,137	0.0973	0.216
method 3	5	16	12.4	3.34	4.07	1,209,697,021	6,121,471	7,201	0.198	0.528

less significant in this method than in the replication method, owing to the fact that a victim worker is free from copying its itemset table, which the replication method requires for every task creation.

The good speedup is also confirmed by the comparison of the search space size and traversing speed of 1-worker and 16-worker executions with the half-and-half and this only-one strategies. That is, the rates of search space expansion and of traversing speed degradation from 1-worker to 16-worker are 1.83 and 2.14 with the only-one strategy, making the product of them smaller (better) than that of the half-and-half to improve the 16-worker speedup greatly. In addition, Fig. 12 confirms that the only-one strategy is most appropriate for the fully-sharing method. Both of two graphs in the figure clearly show that the smaller the number of iterations are left for a victim, the more the execution time is shortened.



(a) Replicating method ($\theta = 5$)



(b) Fully-sharing method ($\theta = 5$)

Fig. 13 Speedup and the number of visits to vertices with the setting that leaves one iteration for the victim worker on task creation.

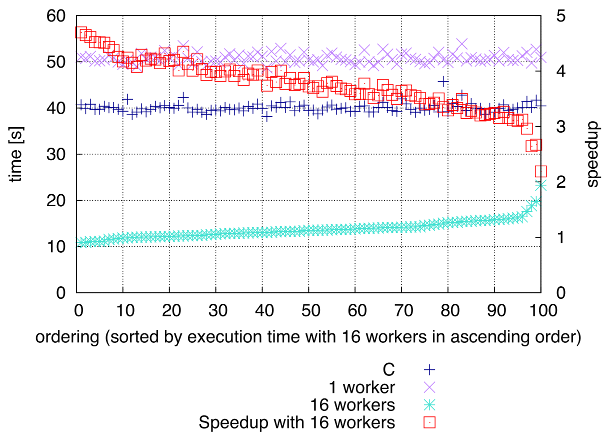


Fig. 14 Execution time and speedup with 16 workers versus vertex ordering (sorted by execution time with 16 workers).

5.4 Effect of Vertex Ordering

The shape of a search tree in the COPINE algorithm depends on the order of vertices which we introduced in Section 3.1 for Pruning 1. Since the elimination of a branch by Pruning 3 in our parallel search depends on the shape of the tree to be traversed, the parallel performance may also depend on the shape and thus on the vertex ordering. To evaluate this effect, we generated 100 random orders of the vertices of the graph used in this evaluation, and measured the search time for each ordering.

The evaluations are done under the condition for which we obtained the best performance among the previous evaluations, that is, the fully-sharing method with the setting that leaves one iteration for a victim when dividing a parallel loop. We executed the Tascell program with 1 and 16 workers and the sequential C program for each ordering. We determined each order by generating uniform random numbers in the range $[1, |V|]$ by the Mersenne Twister [9] assigning a unique number to each vertex.

The execution times are shown in **Fig. 14**. As shown in the graph, the times of the 1-worker execution and sequential one are almost stable because the search space size is virtually independent of the tree shape. On the other hand, 16-worker execution times show a slightly larger deviation but, in 95 cases out of 100, still in the range from 11.1 s to 16.4 s resulting in a 3.10- to 4.52-fold speedup, sufficiently close to the results shown in Table 6. Therefore, we may conclude that our parallel implementation stably exerts a good parallel performance regardless of the vertex ordering for the graph used in our evaluation.

6. Related Work

6.1 Lazy Task Creation

For implementing our parallel search, Tascell is not the sole language but there are other LTC-based multithreaded candidates such as Clik [7] and Intel Cilk Plus [10]. The advantages of Tascell over these candidates, however, should justify our choice as follows.

1) As mentioned in Section 4.1, since Tascell does not create any logical threads at an execution point where a parallel task can be spawned (e.g., at a parallel loop statement), the cost of managing them is eliminated.
2) In a multithreaded language, each (logical) thread requires its own workspace; in our COPINE implementations, a workspace for a current connected subgraph, a set of adjacent vertices, a set of candidate vertices to be visited in the next step, and an itemset table (in the non-sharing and replicating methods) are required. In contrast, a Tascell worker can reuse a single workspace while it performs a sequential computation to improve the locality of reference.
3) When we implement a backtrack search algorithm in a multithreaded language, each thread often needs its own copy of its parent thread's workspace. In contrast, Tascell's temporary backtracking mechanism allows a worker to delay the copy operation until it becomes really necessary.
4) Tascell enables us to realize dynamic load balancing among computing nodes in distributed memory environments more easily. Note that, as for this research, future work is needed to implement a parallel solver supporting distributed memory environments.
5) The replicating method explained in Section 4.3.3 is easily and efficiently implemented by letting a victim worker suspend its task execution and make a copy of its itemset table up-to-date at the time it receives a task request from a thief worker. On the other hand, it is difficult to implement the same mechanism in an LTC language, because a worker

steals a task from the task queue of a victim worker, while the victim that created the task does not have any concern with the steal operation. In a straightforward implementation of the replicating method in an LTC language, a worker would make a copy of its itemset table when creating a logical thread rather than when the task is stolen. This implementation reduces the effect of pruning since information of the itemset table registered between the thread creation and the task stealing is not passed to the thief.

6) In the fully-sharing method explained in Section 4.3.3, a victim worker divides the ID range of a task being executed when it receives a task request from another worker. Since this division is done only when part of the task is actually stolen by another worker, we can keep the number of divisions much smaller than the number of operations that potentially create tasks (e.g., parallel loops). In an LTC language, it is difficult to make a victim worker divide an ID range when a task stealing occurs, for the same reason as in 5). In a straightforward implementation of the fully-sharing method in an LTC language, a worker would divide an ID range when creating a logical thread rather than when having a task stolen, thus exhausting the task IDs rapidly, facing the consequence that minID becomes equal to maxID.

Note that Tascell's advantages 1)–4) were listed in Ref. [5], while 5) and 6) were discovered through our research.

As shown in Section 5, the fully-sharing method is superior to the replicating method in terms of performance. However, implementing a parallel COPINE in distributed memory environments, using advantage 4), it is unrealistic for computing nodes that do not share memory to share a single itemset table controlled by locks. Therefore, we need a mechanism that is based on the replicating method and updates table information in each node at appropriate intervals.

6.2 Scheduling and Task Creation

Tascell and Cilk share the oldest-first task scheduling policy to let a worker steal a task as close to the search tree root as possible, thus reducing the task stealing frequency.

In contrast, the task given to a thief can be found at the node closest to the leaf of the subtree that a victim is traversing, as done in the Parallel Depth First (PDF) scheduling [11] for left- and depth-first search. In PDF scheduling, the leftmost unassigned branch of such a node is the task for a thief, achieving good utilization of on-chip cache of chip multiprocessors and better performance than the oldest-first for some applications [12], [13].

This paper presented a variation of the oldest-first strategy in which a stolen task consists of the leftmost and subsequent unassigned branches, and demonstrated the advantage of this variation over the conventional half-and-half splitting in terms of the effectiveness of sequentially dependent pruning.

6.3 Sharing of Pruning Information among Workers

There has been a number of studies regarding the parallelization of Satisfiability Problem (SAT) as a major application of the backtrack search with pruning [14], [15].

In these works, sharing the information about pruning is regarded as an important issue [16], but, unlike our problem, the pruning information can be freely referred to by any workers regardless of the locations of the registration and reference in the search space. MiraXT [17], PaSAT [18], and ySAT [19] are proposed as implementations of SAT in shared memory environments. For sharing table information, MiraXT employs a strategy similar to our fully-sharing method. In PaSAT and ySAT, each worker has its own table and exchanges its contents with other workers periodically to make the table up-to-date.

In SAT there is another implementation issue to find an appropriate shared portion of an enormous number of conflict clauses, especially for distributed memory implementations. This issue is less significant in COPINE, since the size of a table is $O(|V| \exp(\max_{v \in V} |I(v)|))$ at the worst, since the number of itemsets registered in an entry corresponding to a vertex v is at most $2^{|I(v)|}$, or more precisely ${}_{|I(v)|}\mathrm{C}_{|I(v)|/2}$.

6.4 Frequent Itemset Mining

With a given itemset $I = \{i_1, i_2, \ldots, i_n\}$, a set of transactions $T = \{t_1, t_2, \ldots, t_m\}$, itemsets associated with each transaction $\mathcal{I}(t) \subseteq I$ $(t \in T)$, and a threshold θ, the problem extracting all itemsets whose members are commonly contained by θ or more transactions is called Frequent Itemset Mining (FIM) [20], [21]. There are many implementations of FIM solvers proposed [22]. Among them, Linear time Closed itemset Miner (LCM) [23], [24], [25] realizes efficient enumeration of closed itemsets [*7] by defining and utilizing parental relation among itemsets.

The CCIG enumeration can be considered a tougher variation of FIM since the connections (edges) among transactions (vertices) are imposed as additional requirements.

7. Future Work

In order to improve the performance of our parallel solver, it is necessary to reduce redundant visits to vertices more aggressively. One possible approach to achieve this is to abort the execution of a worker traversing a subtree pruned by another worker.

For example, in a parallel search using the fully-sharing method explained in Section 4.3.4, a worker w executing a task t can let another worker w' executing a task t' abort its search by sending a signal when w registers an itemset S to the entry corresponding to a vertex v in the itemset table, if the following conditions are satisfied: (1) a subset S' of S is already registered in the entry, (2) minID of the task t' attached to S' is greater than minID of t, and (3) the worker w' executing t' is traversing the subtree whose root is v.

The condition (3) can be recognized by setting a flag indicating "searching" to each entry of the itemset table. For example, suppose worker 1 is traversing the subtree whose root is v_5 for which the common itemset $\{i_1, i_3, i_4\}$ was registered, as illustrated in **Fig. 15**. Then, worker 0 can abort worker 1's traversal of the subtree by sending a signal when worker 0 finds the common itemset $\{i_1, i_3, i_4\}$ associated with v_5 when it visits the vertex

[*7] Let $\mathcal{I}(S) = \bigcap_{t \in S} \mathcal{I}(t)$ for a set of transactions $S \subseteq T$, and $\mathcal{T}(H) = \{t \in T | H \subseteq \mathcal{I}(t)\}$ for an itemset $H \subseteq I$. H is called closed iff $\mathcal{I}(\mathcal{T}(H)) = H$ [23].

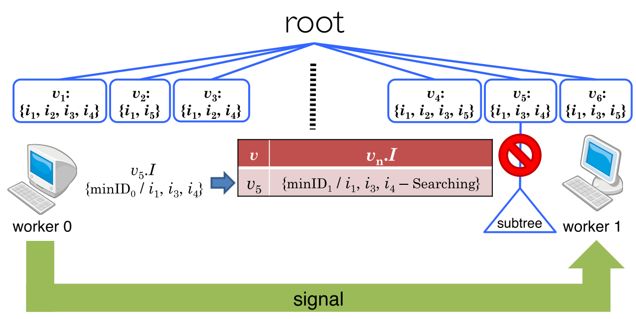

root
v_1: $\{i_1, i_2, i_3, i_4\}$
v_2: $\{i_1, i_5\}$
v_3: $\{i_1, i_2, i_4\}$
v_4: $\{i_1, i_2, i_3, i_5\}$
v_5: $\{i_1, i_3, i_4\}$
v_6: $\{i_1, i_3, i_5\}$
$v_5.I$ $\{\mathrm{minID}_0 / i_1, i_3, i_4\}$
v | $v_n.I$
v_5 | $\{\mathrm{minID}_1 / i_1, i_3, i_4$ – Searching$\}$
subtree
worker 0
worker 1
signal

Fig. 15 Example of aborting a redundant search. MinID_0 and minID_1 are the task IDs being executed by worker 0 and 1, respectively ($\mathrm{minID}_0 < \mathrm{minID}_1$).

with the itemset. We can describe such an aborting operation in a language that supports exceptions, though Tascell does not. Therefore, we will add an exception support to Tascell, and then implement and evaluate this new sharing method.

Our future work also includes implementing our parallel search for distributed memory environments. For Pruning 3, we have to devise a new method of itemset table sharing among workers on computing nodes without shared memory, substituting the fully-sharing method which is feasible only inside a shared-memory computing node. It is expected that we can obtain minimal pruning effect by using the replicating method to share information among workers in different nodes. Furthermore, we should improve the pruning effect by exchanging table information among nodes at intervals. It requires finding optimal intervals and timings to have good scalability in an environment with certain communication costs.

In addition, we will evaluate our implementations using other various graphs including artificial ones. This further evaluation is necessary to prove the efficiency of our implementation since its performance may depend strongly on the characteristics of input graphs, especially in terms of the effectiveness of Pruning 3.

8. Conclusion

In this paper, we proposed a parallel algorithm and implementations for graph mining that extracts all connected subgraphs, each of which shares a common itemset whose size is not less than a given threshold.

We had already proposed an efficient sequential algorithm called COPINE, but its straightforward parallelization results in excessive pruning. We proved that we can avoid such excessive pruning by the restriction that a worker can refer to an itemset registered by another worker only if the registration-reference flow conforms to the sequential search order, and we designed a parallel extension of COPINE introducing this restriction.

We implemented the parallel COPINE algorithm using the task-parallel language Tascell. In order to prune a branch corresponding to a subgraph having an already-visited supergraph with identical itemset, workers need to share table information efficiently in such a manner that a worker can use as many itemsets registered by other workers as possible under the restriction. We implemented two sharing methods: (1) the replicating method, in which a victim worker makes a copy of its own table and passes it to a thief worker when a steal occurs, and (2) the fully-sharing method, in which a single table controlled by locks is shared among workers. In addition, we implemented a task creation strategy optimized for table sharing, as a substitution of the standard strategy of Tascell. As a result, by using the implementation with the fully-sharing method and the task creation strategy where the number of iterations left for a victim worker is minimized when dividing a parallel loop, we achieved an approximately four-fold speedup with 16 workers when analyzing a real protein network.

As stated in Section 7, we will improve our implementation by introducing the abortion of redundant search and for distributed memory environments, and evaluate the current and new implementations with various graphs. In addition, we will implement our parallel algorithm also in Cilk and compare it to the Tascell version in terms of performance and productivity.

Acknowledgments This work was supported in part by JSPS KAKENHI Grant Numbers 21300008 and 25730041.

References

[1] Newman, M.E.J.: The Structure and Function of Complex Networks, *SIAM REVIEW*, Vol.45, No.2, pp.167–256 (2003).

[2] Russell, M.: *Mining the Social Web: Analyzing Data from Facebook, Twitter, LinkedIn, and Other Social Media Sites*, O'Reilly Media (2011).

[3] Seki, M. and Sese, J.: Identification of Active Biological Networks and Common Expression Conditions, *Proc. 8th IEEE International Conference on BioInformatics and BioEngineering, BIBE '08*, pp.1–6 (2008).

[4] Sese, J., Seki, M. and Fukuzaki, M.: Mining Networks with Shared Items, *Proc. 19th ACM International Conference on Information and Knowledge Management, CIKM '10*, pp.1681–1684 (2010).

[5] Hiraishi, T., Yasugi, M., Umatani, S. and Yuasa, T.: Backtracking-based Load Balancing, *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '09*, pp.55–64 (2009).

[6] Yan, X. and Han, J.: gSpan: Graph-Based Substructure Pattern Mining, *Proc. 2002 IEEE International Conference on Data Mining, ICDM '02*, pp.721–724 (2002).

[7] Frigo, M., Leiserson, C.E. and Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language, *Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI '98*, pp.212–223 (1998).

[8] Mohr, E., Kranz, D.A. and Halstead, R.H., Jr.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *IEEE Trans. Parallel and Distributed Systems*, Vol.2, No.3, pp.264–280 (1991).

[9] Matsumoto, M. and Nishimura, T.: Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator, *ACM Trans. Modeling and Computer Simulation*, Vol.8, No.1, pp.3–30 (1998).

[10] Intel Corporation: A Quick, Easy and Reliable way to Improve Performance — Intel Cilk Plus, available from ⟨http://software.intel.com/en-us/intel-cilk-plus/⟩.

[11] Blelloch, G., Gibbons, P. and Matias, Y.: Provably Efficient Scheduling for Languages with Fine-Grained Parallelism, *Proc. Symposium on Parallel Algorithms and Architectures*, pp.1–12 (1995).

[12] Liaskovitis, V., Chen, S., Gibbons, P.B., Ailamaki, A., Blelloch, G.E., Falsafi, B., Fix, L., Hardavellas, N., Kozuch, M., Mowry, T.C. and Wilkerson, C.: Parallel Depth First vs. Work Stealing Schedulers on CMP Architectures, *Proc. 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '06*, pp.330–330 (2006).

[13] Chen, S., Gibbons, P.B., Kozuch, M., Liaskovitis, V., Ailamaki, A., Blelloch, G.E., Falsafi, B., Fix, L., Hardavellas, N., Mowry, T.C. and Wilkerson, C.: Scheduling Threads for Constructive Cache Sharing on CMPs, *Proc. 19th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '07*, pp.105–115 (2007).

[14] Singer, D.: *Parallel Resolution of the Satisfiability Problem: A Survey*, pp.123–147, John Wiley & Sons, Inc. (2006).

[15] Martins, R., Manquinho, V. and Lynce, I.: An Overview of Parallel SAT Solving, *Constraints*, Vol.17, No.3, pp.304–347 (2012).

[16] Hamadi, Y. and Wintersteiger, C.M.: Seven Challenges in Parallel SAT Solving, *AI Magazine*, Vol.34, No.2, pp.99–106 (2013).

[17] Lewis, M., Schubert, T. and Becker, B.: Multithreaded SAT Solving, *Proc. 2007 Asia and South Pacific Design Automation Conference, ASP-DAC '07*, pp.926–931 (2007).

[18] Sinz, C., Blochinger, W. and Küchlin, W.: PaSAT - Parallel SAT-Checking with Lemma Exchange: Implementation and Applications, *Proc. LICS 2001 Workshop on Theory and Applications of Satisfiability Testing, SAT '01*, Vol.9, Elsevier Science Publishers (2001).

[19] Feldman, Y., Dershowitz, N. and Hanna, Z.: Parallel Multithreaded Satisfiability Solver: Design and Implementation, *Electronic Notes in Theoretical Computer Science*, Vol.128, No.3, pp.75–90 (2005).

[20] Agrawal, R., Imieliński, T. and Swami, A.: Mining Association Rules Between Sets of Items in Large Databases, *SIGMOD Record*, Vol.22, No.2, pp.207–216 (1993).

[21] Agrawal, R., Mannila, H., Srikant, R., Toivonen, H. and Verkamo, A.I.: Fast Discovery of Association Rules, *Advances in Knowledge Discovery and Data Mining*, Fayyad, U.M., Piatetsky-Shapiro, G., Smyth, P. and Uthurusamy, R. (Eds.), American Association for Artificial Intelligence, pp.307–328 (1996).

[22] Goethals, B.: FIMI Repository, available from ⟨http://fimi.ua.ac.be/⟩.

[23] Uno, T., Asai, T., Uchida, Y. and Arimura, H.: LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets, *Proc. IEEE ICDM Workshop on Frequent Itemset Mining Implementations, FIMI '03* (2003).

[24] Uno, T., Kiyomi, M. and Arimura, H.: LCM ver.2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets, *Proc. IEEE ICDM Workshop on Frequent Itemset Mining Implementations, FIMI '04* (2004).

[25] Uno, T., Kiyomi, M. and Arimura, H.: LCM ver.3: Collaboration of Array, Bitmap and Prefix Tree for Frequent Itemset Mining, *Proc. 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations, OSDM '05*, pp.77–86 (2005).

Appendix

A.1 Proof of Theorem 1

First, we state the following Lemma 1 for the proof.

Lemma 1 For a connected subgraph $G' = (U, E(U))$ and its CST $T(U) = \langle u_1, \cdots, u_n\rangle$, all of the following hold for all i, j, and k such that $1 \le i, j, k, \le n$.

(1) $u_i \in N^*(U_j) \to u_i \in N(U_j)$

(2) $u_i \in N^*(U_j) - U_j \to u_i \in N(U_j) - U_j$

(3) $prev^*(u_i, U_j) = prev(u_i, U_j)$

(4) $u_i \overset{U_k*}{\leftharpoondown} u_j \to u_i \overset{U_k}{\leftharpoondown} u_j$

Proof:

(1) Since $u_i \in N^*(U_j)$, there must exist u_k such that $(u_k, u_i) \in E$ and $k \le j$ by Definition 5 (1). Since $u_k, u_i \in U$, $(u_k, u_i) \in E(U)$ and thus $u_i \in E(U_j)$ by Definition 4 (2). □

(2) Since $u_i \in N^*(U_j) - U_j$, $u_i \in N^*(U_j)$ and $u_i \notin U_j$. Therefore, $u_i \in N(U_j)$ by (1) and thus $u_i \in N(U_j) - U_j$. □

(3) Since $u_i \in U$, $\{u_k \mid (u_k, u_i) \in E, 1 \le k \le j\} = \{u_k \mid (u_k, u_i) \in E(U), 1 \le k \le j\}$. Therefore;

$$\begin{aligned} prev^*(u_i, U_j) &= \arg\max_k\{u_k \mid (u_k, u) \in E, 1 \le k \le j\} \\ &\qquad\qquad \text{(Definition 5 (2))} \\ &= \arg\max_k\{u_k \mid (u_k, u) \in E(U), 1 \le k \le j\} \\ &= prev(u_i, U_j) \qquad \text{(Definition 4 (3))} \end{aligned}$$

□

(4)
$$\begin{aligned} u_i \overset{U_k*}{\leftharpoondown} u_j &\to u_i, u_j \in N^*(U_k) - U_k \wedge \\ &\Big(prev^*(u_i, U_k) > prev^*(u_j, U_k) \vee \\ &\Big(prev^*(u_i, U_k) = prev^*(u_j, U_k) \wedge u_i \prec u_j\Big)\Big) \\ &\qquad\qquad \text{(Definition 5 (2))} \\ &\to u_i, u_j \in N(U_k) - U_k \wedge \qquad \text{((2))} \\ &\Big(prev(u_i, U_k) > prev(u_j, U_k) \vee \\ &\Big(prev(u_i, U_k) = prev(u_j, U_k) \wedge u_i \prec u_j\Big)\Big) \\ &\qquad\qquad \text{((3))} \\ &\to u_i \overset{U_k}{\leftharpoondown} u_j \qquad \text{(Definition 4 (3))} \end{aligned}$$

□

With the help of Lemma 1, we can prove Theorem 1 as follows.

Proof of Theorem 1: Let $u = \min[\overset{U_k*}{\leftharpoondown}](C_k)$ and $V_{i+1} = U' = \{u'_1, \cdots, u'_{k+1}\}$ where $u'_j = u_j$ for all j such that $1 \le j \le k$ and $u'_{k+1} = u$.

(1) $T(V_{i+1})$ is canonical because of the following.

(a) If $k = 0$, $T(V_{i+1}) = \langle u\rangle$ is canonical by definition. For the case of $k > 0$ we have:

(b) Since $T(V_i)$ is canonical, so is $T(U_k)$, $u'_1 = u_1 = \min[\prec](U_k) = \min[\prec](U'_k)$ holds. Since $P_1 = \{v \mid v \overset{U_0*}{\leftharpoondown} u_1\} \cup \{u_1\} = \{v \mid v \prec u_1\} \cup \{u_1\} = \{v \mid v \preceq u_1\}$, it holds that $D_1 = N^*(U_0) \cap P_1 = V \cap P_1 = \{v \mid v \preceq u_1\}$. Therefore, $D_k \supseteq \{v \mid v \preceq u_1\}$ and thus $C_k \cap \{v \mid v \preceq u_1\} = \emptyset$, to mean $u'_{k+1} \notin \{v \mid v \preceq u_1\}$ since $u'_{k+1} = u \in C_k$ and thus $u'_1 = u_1 \prec u'_{k+1}$ to hold $u'_1 = u_1 = \min[\prec](U')$ satisfying the canonicity requirement for u'_1.

(c) Since $T(V_i)$ is canonical, it holds that for all j such that $1 \le j \le k$ that $u_j \in N(U_{j-1}) - U_{j-1}$ by Definition 4 (3) to mean $u_j \in N^*(U_{j-1})$ since $N(U_{j-1}) - U_{j-1} \subseteq N(U_{j-1}) \subseteq N^*(U_{j-1})$, and $u_j \in P_j$ by definition. Therefore, $u_j \in N^*(U_{j-1}) \cap P_j \subseteq D_j$. Since $u \in C_k$ to mean $u \notin D_k$, it holds for all j such that $1 \le j \le k$ that $u \ne u_j$ and thus $u \notin U_j = U'_j$.

(d) Since $u \in C_k \subseteq N^*(U_k)$, there must be l such that $1 \le l \le k$, $u \in N^*(U_j) = N^*(U'_j)$ for all j such that $l \le j \le k$, and $u \notin N^*(U_j) = N^*(U'_j)$ for all j such that $1 \le j < l$.

(e) For $j < l$, $u \notin N^*(U'_j)$ means $u \notin N(U'_j)$ and thus $N(U'_j) = N(U_j)$, and $u \notin N(U'_j) - U'_j$, because $N(U'_j) - U'_j \subseteq N(U'_j) \subseteq N^*(U'_j)$. Since $T(V_i)$ is canonical, $u'_{j+1} = u_{j+1} = \min[\overset{U_j}{\leftharpoondown}](N(U_j) - U_j) = \min[\overset{U'_j}{\leftharpoondown}](N(U'_j) - U'_j)$ to satisfy the canonicity requirement for u'_{j+1}.

(f) For j such that $l \le j < k$, $u \in N^*(U'_j)$ means $u \in N(U'_j)$ since $u \in U'$ by Lemma 1 (1). Since $u \in C_k$ and thus $u \notin D_{j+1}$ but $u \in N^*(U'_j) = N^*(U_j)$, then $u \notin P_{j+1}$, $u_{j+1} \overset{(U_j=U'_j)*}{\leftharpoondown} u$ must be true and thus $u'_{j+1} \overset{U'_j}{\leftharpoondown} u$ by Lemma 1 (4). Therefore, $u'_{j+1} = u_{j+1} = \min[\overset{U_j}{\leftharpoondown}](N(U_j) \cup \{u\} - U_j) = \min[\overset{U'_j}{\leftharpoondown}](N(U'_j) - U'_j)$ to satisfy the canonicity requirement for u'_{j+1}.

(g) Since $u \in N^*(U'_k)$ and $u \in U'$, then $u \in N(U'_k)$ must be true by Lemma 1 (1), and thus $N(U'_k) = U'_k \cup \{u\}$. Therefore, $\min[\overset{U'_k}{\leftharpoondown}](N(U'_k) - U'_k) = \min[\overset{U'_k}{\leftharpoondown}](\{u\}) = u = u'_{k+1}$ to satisfy the canonicity requirement for u'_{k+1}.

(2) $T(V_i) \prec T(V_{i+1})$. It is obvious that $k = |\mathit{prefix}(T(V_i), T(V_{i+1}))|$.

(a) If $k = n = |T(V_i)|$, it holds that $|T(V_{i+1})| = k + 1 = n + 1 > n$ and thus $T(V_i) \prec T(V_{i+1})$.

(b) If $k < n$, $u \in C_k$ and thus $u \notin D_{k+1}$ to mean that $u \notin P_{k+1}$

since $u \in N^*(U_k)$. Therefore, $u_{k+1} \xleftarrow{U_k*} u = u'_{k+1}$ proving the proposition $T(V_i) \prec T(V_{i+1})$.

(3) Any spanning tree t such that $T(V_i) \prec t \prec T(V_{i+1})$ is not canonical. Such t should satisfy $|t| > k$ and $|\mathit{prefix}(T(V_i), t)| = k' \geq k$ since $|\mathit{prefix}(T(V_i), T(V_{i+1}))| = k$. Let $t = \langle u''_1, \cdots, u''_{|t|}\rangle$ and $U'' = \{u''_1, \cdots, u''_{|t|}\}$ where $u''_j = u_j$ for all j such that $1 \leq j \leq k'$, and let $w = u''_{k'+1}$.
   (a) If $k' = k$, it must be $w \xleftarrow{U_k*} u$ because $t \prec T(V_{i+1})$. Since $u = \min[\xleftarrow{U_k*}](C_k)$, however, $w \notin C_k$ but definitely $w \in N^*(U_k)$ to mean that there exists l such that $w \in D_l$ and $l \leq k+1$ if $n > k$ or $l \leq k$ if $n = k$.
   (b) If $k' > k$, $C_{k'} = \emptyset$ because of the maximality of k for $C_k \neq \emptyset$, but definitely $w \in N^*(U_{k'})$ to mean that there exists l such that $w \in D_l$ and $l \leq k'+1$ if $n > k'$ or $l \leq k'$ if $n = k'$.
   (c) Existence of l above means the existence of m such that $w \in N^*(U_{m-1}) \cap P_m$, to mean $w \xleftarrow{U_{m-1}*} u_m$, and $m \leq l$. If $m = l = k'+1$, we have $w \xleftarrow{U_{k'}*} u_{k'+1}$ but it contradicts $u_{k'+1} \xleftarrow{U_{k'}*} w$ which should be derived from $T(V_i) \prec t$. Therefore, $m \leq k'$ must be true.
   (d) Since $m \leq k'$, $w \xleftarrow{U_{m-1}*} u_m$ means that $w \xleftarrow{U''_{m-1}*} u''_m$. If $m = 1$, this means that $w \xleftarrow{U''_0*} u''_1$ and thus $w \prec u''_1$ violating $u''_1 = \min[\prec](U'')$ to be satisfied if t is canonical. If $m > 1$, $w \xleftarrow{U''_{m-1}*} u''_m$ means that $w \xleftarrow{U''_{m-1}} u''_m$ since $w \in U''$ by Lemma 1 (4), violating $u''_m = \min[\xleftarrow{U''_{m-1}}](N(U_{m-1}) - U_{m-1})$ to be satisfied if t is canonical. Therefore, t is not canonical.

□

A.2 Proof of Theorem 3

First, we state Lemmas 2 and 3 for the proof.

Lemma 2 (Lexicographical Order of Super-Subgraph) Let $G^1 = (U^1, E(U^1))$ and $G^2 = (U^2, E(U^2))$ be connected subgraphs of $G = (V, E)$. If $G^1 \subset G^2$, either of the followings holds.

(1) $\exists T' : T(U^2) = T(U^1) \cdot T'$

(2) $G^2 \prec G^1$

Proof: It is obvious that $G^1 \subset G^2$ if (1) holds, and if so $G^1 \prec G^2$. Therefore, we prove if G^2 is not a descendant of G^1, (2) must hold, i.e., $G^2 \prec G^1$. Let $m = |T(U^1)|$, $n = |T(U^2)|$, $k = |\mathit{prefix}(T(U^1), T(U^2))|$. Since G^2 is not a descendant of G^1, it must be $k < n$. Since $G^1 \subset G^2$ and thus G^1 cannot be a descendant of G^2, it must be $k < m$ as well. Let us suppose $G^1 \prec G^2$ and let $u = u^1_{k+1}$ and $w = u^2_{k+1}$. Since $G^1 \prec G^2$, $u \prec w$ must be true. However, since $G^1 \subset G^2$, $u \in N(U^2_k) - U^2_k$ must be true and thus $w \neq \min[\xleftarrow{U^2_k}](N(U^2_k) - U^2_k)$ contradicting the premise that $T(U^2)$ is canonical. Therefore, our supposition $G^1 \prec G^2$ leads us to a contradiction and thus $G^2 \prec G^1$. □

Lemma 3 (Lexicographical Order of Union of Subgraphs) Let $G^1 = (U^1, E(U^1))$ and $G^2 = (U^2, E(U^2))$ be connected subgraphs of $G = (V, E)$ with $G_1 \prec G_2$, and let $T(U^1) = \langle u^1_1, \cdots, u^1_m\rangle$ and $T(U^2) = \langle u^2_1, \cdots, u^2_n\rangle$. If their *tails* are common, i.e., $\mathit{tail}(U^1) = u^1_m = \mathit{tail}(U^2) = u^2_n$, all of the following hold.

(1) $G^1 \cup G^2 \prec G^2$

(2) For any connected subgraph G' of G such that $G' \supset G^1 \cup G^2$, it holds that $G' \prec G^2$.

Proof: It is obvious that $G^1 \cup G^2 = G^{\cup} = (U^{\cup}, E(U^{\cup}))$ is a connected subgraph of G since G^1 and G^2 has a common vertex $\mathit{tail}(U^1) = \mathit{tail}(U^2)$. Since $G^1 \prec G^2$ means that $G^1 \neq G^2$ and their tails are common, they cannot be descendants of each other, or have two or more occurrences of a vertex in $T(U^1)$ or $T(U^2)$ contradicting their canonicity. Therefore, $k = |\mathit{prefix}(T(U^1), T(U^2))| < |T(U^1)|, |T(U^2)|$. Since $G^1 \prec G^2$ and G^2 is not a descendant of G^1, it holds that $G^1 \not\subset G^2$ by Lemma 2 and thus $G^2 \subset G^{\cup} \subset G'$.

(1) $G^2 \subset G^{\cup}$ means $G^{\cup} \prec G^2$ unless $G^{\cup}$ is a descendant of G^2 by Lemma 2. Let us suppose that $G^{\cup}$ is a descendant of G^2 and thus $|\mathit{prefix}(T(U^1), T(U^{\cup}))| = |\mathit{prefix}(T(U^1), T(U^2))| = k$. Let $T(U^1) = \langle u^1_1, \cdots\rangle$, $T(U^2) = \langle u^2_1, \cdots\rangle$, and $T(U^{\cup}) = \langle u^{\cup}_1, \cdots\rangle$. Since $G^1 \prec G^2$ and $G^{\cup}$ is a descendant of G^2, it holds that $u = u^1_{k+1} \prec u^2_{k+1} = u^{\cup}_{k+1} = w$. However, since $G^1 \subseteq G^{\cup}$, it must be $u \in N(U^{\cup}_k) - U^{\cup}_k$. Therefore, $u \prec w$ violates $w = \min[\xleftarrow{U^{\cup}_k}](N(U^{\cup}_k) - U^{\cup}_k)$ for the canonicity of $T(U^{\cup})$ leading us to a contradiction. Therefore, $G^{\cup}$ cannot be a descendant of G^2 and thus $G^{\cup} \prec G^2$.

(2) Replacing $G^{\cup}$ with G' in the proof (1) gives us a valid proof for $G' \prec G^2$.

□

With Lemmas 2 and 3, we can now prove Theorem 3.

Proof of Theorem 3:

(1) Let $G^{\cup} = G^1 \cup G^2$ being a connected subgraph of G. Since $I(G^1) \supseteq I(G^2)$, it holds that $I(G^{\cup}) = I(G^1) \cap I(G^2) = I(G^2)$. Since $G^{\cup} \supset G^2$, the subgraph G^2 is not closed with respect to $I(G^2)$.

(2) Let $T(U') = T(U^2) \cdot \langle w_1, \cdots, w_n\rangle$, $I = \bigcap_{i=1}^{n} I(w_i)$, and $G^+ = G^1 \cup G'$. Since $I(G^+) = I(G^1) \cap I(G^2) \cap I \supseteq I(G^2) \cap I = I(G')$, the subgraph G' is not closed with respect to $I(G')$ unless $G' = G^+$. However, since $G^+ \prec G^2$ by Lemma 3 and definitely $G^2 \prec G'$ since G' is a descendant of G^2, we have $G^+ \prec G'$ and thus $G' \neq G^+$. Therefore, G' is not closed with respect to $I(G')$.

□

A.3 Proof of Theorem 4

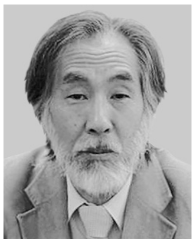
Necessity: Obviously (1) is a necessary condition of the closeness of G' with respect to $I(G')$, and (2) is as well by Theorem 3.

Sufficiency: By Lemma 2, any connected subgraph $G^s = (U^s, E(U^s))$ of G such that $G^s \supset G'$ must be either a descendant of G' or a predecessor $G^s \prec G'$. If G^s is a descendant of G', there must be a direct descendant G^d of G' such that $G^d = G^s$ or G^s is a descendent of G^d. Since $I(G') \neq I(G^d)$ but clearly $I(G') \supseteq I(G^d) \supseteq I(G^s)$, it must hold that $I(G') \supset I(G^d)$ and thus $I(G') \supset I(G^s)$. If $G^s \prec G'$ on the other hand, $T(U^s) = T(U^p) \cdot T'$ with some G^p (possibly G^s itself) such that $G^p \prec G'$ and $\mathit{tail}(U^p) = \mathit{tail}(U')$ because $G^s \supset G'$ and thus $\mathit{tail}(U') \in U^s$. Since $I(G^p) \not\supseteq I(G')$, we have $I(G^s) \subset I(G')$ because $I(G^s) \subseteq I(G^p)$ and $I(G^s) \subseteq I(G')$. Therefore, $I(G^s) \supset I(G')$ for any G^s such that $G^s \supset G'$, and thus the subgraph G' is closed with respect to $I(G')$ by definition, proving that (1) and (2) are sufficient conditions.

□

Shingo Okuno was born in 1989. He received his B.E. in Information Systems from Osaka University in 2012 and his M.E. in Informatics from Kyoto University in 2014. He has been a Ph.D. candidate at Department of Systems Science, Graduate School of Informatics, Kyoto University since 2014. His research interests include parallelization of graph mining.

Tasuku Hiraishi was born in 1981. He received his B.E. in Engineering in 2003, his M.E. in Informatics in 2005, and his Ph.D. in Informatics in 2008, all from Kyoto University. In 2007–2008, he was a fellow of JSPS at Kyoto University. Since 2008, he has been working as an assistant professor at the Academic Center for Computing and Media Studies, Kyoto University. His research interests include parallel programming languages and high performance computing. He won the IPSJ Best Paper Award in 2010. He is a member of IPSJ and the Japan Society for Software Science and Technology (JSSST).



Hiroshi Nakashima received his M.E. and Ph.D. degrees from Kyoto University in 1981 and 1991, respectively, and was engaged in research on inference systems with Mitsubishi Electric Corporation from 1981. He became an associate professor at Kyoto University in 1992, a professor at Toyohashi University of Technology in 1997, and a professor at Kyoto University in 2006. His current research interests are the architecture of and programming in parallel systems. He is a fellow of IPSJ, and a member of IEEE/CS, ACM, ALP, and TUG.

Masahiro Yasugi was born in 1967. He received his B.E. in Electronic Engineering, his M.E. in Electrical Engineering, and his Ph.D. in Information Science from the University of Tokyo in 1989, 1991, and 1994, respectively. In 1993–1995, he was a fellow of the JSPS (at the University of Tokyo and the University of Manchester). In 1995–1998, he was a research associate at Kobe University. In 1998–2012, he was an associate professor at Kyoto University. Since 2012, he is a professor at Kyushu Institute of Technology. In 1998–2001, he was a researcher at PRESTO, JST. His research interests include programming languages and parallel processing. He is a member of IPSJ, ACM, and the Japan Society for Software Science and Technology. He was awarded the 2009 IPSJ Transactions on Programming Outstanding Paper Award.

Jun Sese is an associate professor of Department of Computer Science at Ochanomizu University. He received his Ph.D. in 2005 from Department of Computational Biology, Graduate School of Frontier Science, the University of Tokyo. He joined Undergraduate Program of Bioinformatics and Systems Biology, the University of Tokyo as an associate researcher. In 2006, he moved to Ochanomizu University as an associate professor. Among 2011–2014, he also joined Tokyo Institute of Technology as an associate professor. He developed computational techniques to analyze large biological data, such as DNA sequences and gene expression profiles.