Journal of Information Processing Vol.25 256–267 (Feb. 2017)

[DOI: 10.2197/ipsjjip.25.256]

Regular Paper

Parallelization of Extracting Connected Subgraphs with Common Itemsets in Distributed Memory Environments

Shingo Okuno[1,a)] Tasuku Hiraishi[2] Hiroshi Nakashima[2] Masahiro Yasugi[3] Jun Sese[4]

Received: May 10, 2016, Accepted: August 4, 2016

Abstract: This paper proposes a parallel implementation of graph mining that extracts all connected subgraphs with common itemsets, of which the size is not less than a given threshold, from a graph and from itemsets associated with vertices of the graph, in distributed memory environments using the task-parallel language Tascell. With regard to this problem, we have already proposed parallelization of a backtrack search algorithm named COPINE and its implementation in shared memory environments. In this implementation, all workers share a single table, which is controlled by locks, that contains the knowledge acquired during the search to obviate the need for unnecessary searching. This sharing method is not practical in distributed memory environments because it would lead to a drastic increase in the cost of internode communications. Therefore, we implemented a sharing method in which each computing node has a table and sends its updates to the other nodes at regular time intervals. In addition to this, the high task creation cost for COPINE is problematic and thus the conventional work-stealing strategy in Tascell, which aims to minimize the number of internode work-steals, significantly degrades the performance since it increases the number of intranode work-steals for small tasks. We solved this problem by promoting workers to enable them to request tasks from external nodes. We also employed a work-stealing strategy based on estimation of the sizes of tasks created by victim workers. This approach enabled us to achieve good speedup performance with up to 8 nodes × 16 workers.

Keywords: backtrack search, graph mining, task-parallel languages, distributed memory environments, dynamic load balancing

1. Introduction

The design and implementation of parallel algorithms to achieve performance acceleration is indispensable because the number of processor cores in computer systems has been steadily increasing. In addition, distributed memory systems consisting of computing nodes with their own memory space and that are connected to a communication network have become mainstream in recent high-performance computing (HPC) systems. Distributed memory systems can improve the performance by carrying out massively parallel computing with multiple computing nodes, and also have the ability to process large-scale problems. Thus, the parallel implementation of graph mining in a distributed environment is important since the amount of data to be analyzed continues to become larger. This paper proposes the parallel implementation of graph mining that extracts connected subgraphs with common itemsets (Common Itemset connected subGraph, CIG) from a graph and itemsets associated with vertices of the graph, in distributed memory environments. Here, a common itemset refers to the intersection of all itemsets with which vertices of a connected subgraph are associated.

This kind of graph mining is applicable to the acquisition of various useful types of knowledge from a large amount of data in various fields [1], [2]. For example, we could identify an appropriate group of users with common interests to which to direct targeted advertising by applying graph mining to a social network where each vertex represents a user and contains his/her interests. Another example would be to find reactional set pairs of genes and drugs by enumerating CIGs in a biological network where each vertex represents a gene and is associated with a set of drugs that react with the gene. Such knowledge is expected to be helpful for drug discovery.

We have already proposed COmmon Pattern Itemset NEtwork mining (COPINE) [1], [2] as an efficient backtrack search algorithm for extracting CIGs. We have also proposed its parallel implementation using the task-parallel language Tascell [3] in shared memory environments [4]. In this research, we enhanced this implementation for distributed memory environments.

Tascell supports distributed memory environments and dynamic load balancing across computing nodes. We confirmed that Tascell achieves good performance for various backtrack search benchmark programs in distributed memory environments [3], [5]. However, the efficient parallel implementation of the COPINE algorithm in distributed memory environments requires us to solve the following two problems.

The first problem involves determining how computing nodes share table information related to pruning. COPINE avoids unnecessary searches by using a pruning mechanism that depends on the knowledge acquired during searches. In a parallel search

[1] Graduate School of Informatics, Kyoto University, Kyoto 606–8501, Japan
[2] Academic Center for Computing and Media Studies, Kyoto University, Kyoto 606–8501, Japan
[3] Department of Artificial Intelligence, Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
[4] National Institute of Advanced Industrial Science and Technology, Koto, Tokyo 135–0064, Japan
[a)] shingo@sys.i.kyoto-u.ac.jp

where a unique set of subtrees (tasks) is assigned to each worker, workers need to share the acquired knowledge efficiently. In our implementation in shared memory environments [4], a single table controlled by locks is shared among workers. This method is unrealistic in distributed memory environments because it drastically increases the cost of internode communications. Therefore, we implemented a sharing method in which each computing node has a table and sends its updates to the other nodes at regular time intervals.

The second problem entails determining how to reduce the total cost of work-steals among workers. When stealing a part of another worker's task with the conventional work-stealing strategy in Tascell, an idle worker (thief) chooses another worker as a victim inside the same computing node whenever possible. This strategy is expected to minimize the number of internode work-steals. However, a thief often obtains a small task within the same node even if larger tasks are available in external nodes. Although we did not experience a significant performance degradation in evaluations using some backtrack search algorithms in [3], [5], serious performance degradation is observed for our parallel COPINE solver due to the high cost of work-steal. We alleviated this problem by discussing and implementing the following work-stealing strategies: (1) workers are promoted to obtain larger tasks by requesting tasks from workers in external nodes, (2) a thief estimates the sizes of tasks that can be created by other workers to enable it to steal a task from the worker with the largest estimated task size, and (3) a worker, which is expected to steal only a small task, waits for a certain time before attempting to steal.

The contributions of this paper are threefold:

- We implemented the parallel COPINE algorithm in distributed memory environments using the task-parallel language Tascell.
- We discussed and implemented new work-stealing strategies in Tascell to enable workers to obtain larger tasks especially in distributed memory environments.
- We evaluated the above implementations of COPINE and the work-stealing strategies, and succeeded in accelerating the computation with up to 8 nodes × 16 workers when analyzing a real protein network.

The remainder of this paper is organized as follows. We introduce the COPINE algorithm in Section 2. In Section 3, we present the load balancing strategy in Tascell. Then, we provide the implementation of COPINE in distributed memory environments in Section 4, and propose the improved work-stealing strategies in Tascell in Section 5. We show the performance evaluations in Section 6. We summarize related work in Section 7. Finally, we conclude this paper and describe future work in Section 8.

2. COPINE Algorithm

In this section, we introduce graph mining targeted at this research. First, we define the problem in Section 2.1. Then, we explain the sequential and parallel COPINE algorithms in Section 2.2 and 2.3, respectively. Further details and proofs of the correctness of these algorithms can be found in Ref. [4].

2.1 Definition of CCIG Enumeration Problem

In this section, we define the *Closed CIG (CCIG) enumeration problem*. This problem involves a graph of which the vertices are associated with itemsets, and the common itemset of a connected subgraph, i.e., the intersection of all itemsets associated with its vertices. A CCIG with respect to an itemset I_c is a maximal subgraph among CIGs that have I_c as their common itemset; in other words, a CIG having no adjacent vertex whose addition to the CIG preserves I_c as the common itemset of the expanded subgraph. The CCIG enumeration problem is to find all CCIGs whose common itemset size is not less than a given threshold. More formal definitions of the connected subgraph, CIG, CCIG, and the CCIG enumeration problem are as follows.

Definition 1 (Connected Subgraph) For a given graph $G = (V, E)$, we term $G' = (V', E')$ 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) $\forall u, v \in V' : \exists\{(u_1, v_1), \cdots, (u_n, v_n)\}$ being 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 let E' be denoted by $E(V')$.

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 *CCIG enumeration problem* is to extract all connected subgraphs $G' = (V', E')$ that satisfy the following two conditions.

(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' adjacent 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 named a CIG. A CIG that satisfies (ii) is named a CCIG. CCIGs are 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 output when this graph and $\theta = 2$ are given as input. Note that $G'' = (V'', E(V''))$ where $V'' = \{v_1, v_4, v_5\}$ is not included in the output, because it satisfies (i) since $I(G'') = \{i_1, i_3\}$ but not (ii) since $I(G'_3) = I(G'')$ and $V'_3 \supset V''$.

2.2 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

[*1] To be exact, this is a connected and induced subgraph due to condition (2). In this paper, however, we term it a connected subgraph for simplicity.

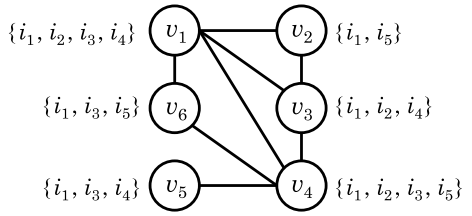


Fig. 1 Example of a graph associated with itemsets.

Table 1 Outputs when the graph in Fig. 1 and $\theta = 2$ are given as inputs.

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	$\{v_5\}$	$\{i_1, i_3, i_4\}$

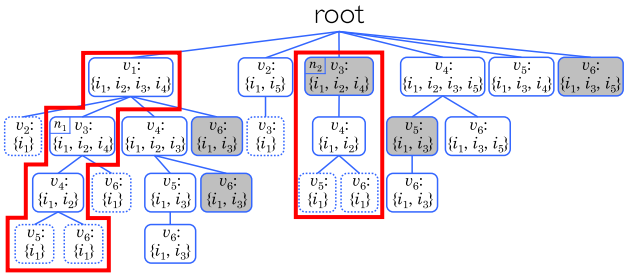


Fig. 2 Search tree for the graph in Fig. 1 ($\theta = 2$).

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 there are generally two or more vertices that can be added to a connected subgraph corresponds to the fact that a node in the search tree can have multiple child nodes. Therefore, to examine all the connected subgraphs G' such that $G' \supset G_n$, where G_n is the subgraph represented by a tree node n, COPINE repeats the search rooted by every child c of n and backtracks to n to choose a sibling of c.

Given that the search tree represents all the subgraphs of an input graph G, we can enumerate all CCIGs by traversing the tree completely. However, this is generally unrealistic. Therefore, to reduce the search space, COPINE prunes those tree edges from which the following three types of subgraphs are derived;

Pruning 1 subgraphs that have already been visited,

Pruning 2 subgraphs of which the itemset is smaller than the threshold θ, and

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

Pruning 1 avoids duplicate enumeration using the gSpan technique [6]. In a straightforward depth-first search to enumerate the connected subgraphs, all vertices adjacent to a connected subgraph represented by a path from the root to a node become candidates for the vertex added in the next step. Pruning 1 limits the addition to ensure that subgraphs continue to be traversed in ascending order in terms of the lexicographical order given to the path from the root to a node and which is derived from an arbitrary ordering $v_1 \prec v_2 \prec \cdots v_{|V|}$ of all the vertices in V. That is, we represent a subgraph $G' = (V', E(V'))$ by the sequence of vertices $\langle v'_1, \ldots, v'_k \rangle$ where $V' = \{v'_1, \ldots, v'_k\}$ and $v'_1 \prec \cdots \prec v'_k$, and traverse subgraphs in the ascending lexicographical order of these sequences. A search tree for the graph in Fig. 1 to which this pruning is applied is shown in Fig. 2. The label and itemset of each tree node represent the last vertex added to the subgraph corresponding to the node and its common itemset, respectively. Although $\langle v_1, v_3, v_4 \rangle$ in Fig. 2 is traversed, $\langle v_1, v_4, v_3 \rangle = \langle v_1, v_3, v_4 \rangle$ is not traversed because its precedence over $\langle v_1, v_4 \rangle$ in the lexicographic order violates the ascending order of traversal.

Pruning 2 exploits the property that the common itemset size 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 the search tree. Therefore, if we encounter a node of which the corresponding subgraph has a common itemset smaller than the threshold θ, it is obvious that further traversal to any descendants of the node would be meaningless because any expansion to the subgraph results in a common itemset smaller than θ. 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 subtree on the right, rooted by n_2, is identical to the tail of the one on the left, rooted by n_1, including the itemset labels of their corresponding nodes. This means that visiting descendants of n_2 is unnecessary because the subgraphs they represent have supergraphs represented by the subtree on the left of which the itemsets are identical to those of the subgraphs. That is, the subgraphs represented by the subtree on the right are not closed. We avoid such a duplicated search by introducing the third type of pruning (Pruning 3) as follows.

We prepare an *itemset table* of which the entries correspond to the vertices of the graph. When a vertex is added to the subgraph we are visiting, the common itemset of the resulting subgraph is added to the entry corresponding to the added vertex unless the table entry contains a super-itemset of the itemset to be added. Otherwise, if a super-itemset exists, the search of the descendants of the current tree node can be skipped. The shaded nodes in Fig. 2 are eliminated by this kind of pruning. For example, 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 super-itemset (in a broad sense) has already been registered, the search from n_2 in the direction of 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 we visit $\langle v_1, v_4 \rangle$ and add the itemset $\{i_1, i_2, i_3\}$, which contains $\{i_1, i_2\}$, to the entry.

The sequential COPINE algorithm shown in **Fig. 3** enumerates

[*2] Even though this removal is not necessary for the correctness of the algorithm, it prevents the itemset table from becoming unnecessarily large.

all CCIGs of a graph $G = (V, E)$ whose common itemset size is not less than θ. In *ExploreCCIG*( ), T is the sequence of vertices corresponding to the connected subgraph that the function is visiting, C is the sequence of vertices adjacent to T, and V is the set of all vertices not in T. In this algorithm, Pruning 1 corresponds to the fact that subgraphs are scanned, 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 21. Pruning 3 is executed by adding an itemset to the itemset table entry (lines 8 and 23) and detecting an inclusive relation between itemsets (lines 7 and 22).

2.3 Parallel COPINE Algorithm

We parallelize the algorithm in Fig. 3 by dividing the search tree and assigning a unique set of subtrees to each worker. This can be implemented by dividing the two `for` loops (lines 4–10 and 18–26) into appropriate units and executing them in parallel.

Each worker traverses the assigned subtrees in almost the same way as in the sequential search. Prunings 1 and 2 can be directly applied to the parallel algorithm. However, we need to impose a certain restriction on Pruning 3. In Fig. 3, a worker refers to $c_i.I$ (a set of itemsets registered when the vertex c_i was added to a subgraph) to check whether Pruning 3 is applicable. In a parallel search, a worker could excessively prune the branches in its subtrees if it blindly consulted table entries registered by another worker. We can avoid such excessive pruning by the restriction that *a worker can refer to itemsets for Pruning 3 only if those itemsets had been registered earlier in a sequential search*. A worker needs to check whether an element, i.e., an itemset, registered to the table had been registered earlier to the table, even if the search was executed sequentially rather than in parallel. This verification by a worker requires each itemset to have some sequential ordering information concerning its registration.

In a parallel search, a search tree node visited by a worker may have some *precedent* nodes left unvisited by other workers even 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. 3 will assert a CIG to be closed even though this is not true in reality. 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 [*3].

```
1:  function EnumCCIG(G) begin
2:    (V, E) ← G;  L ← ∅;  V' ← V;
3:    for ∀v ∈ V do v.I ← ∅;
4:    for i = 1 to |V| do begin  // V = {v_1, v_2, ···}
5:      V' ← V' − {v_i};
6:      if |I(v_i)| < θ then continue;
7:      if ∃I s.t. I ∈ v_i.I ∧ I ⊇ I(v_i) then continue;
8:      v_i.I ← v_i.I ∪ {I(v_i)};
9:      L ← ExploreCCIG(v_i, ⟨v_i⟩, I(v_i), V', E, ⟨⟩, L);
10:   end
11:   return(L);
12: end
13:
14: function ExploreCCIG(v, T, I, V, E, C, L) begin
15:   closed ← true;
16:   N ← sort(neighbors(v, E));
17:   C ← N · C;
18:   for i = 1 to |C| do begin  // C = ⟨c_1, c_2, ···⟩
19:     if c_i ∉ V then continue;
20:     V ← V − {c_i};  I_c ← I ∩ I(c_i);
21:     if |I_c| < θ then continue;
22:     if ∃I' s.t. I' ∈ c_i.I ∧ I' ⊇ I_c then continue;
23:     c_i.I ← c_i.I ∪ {I_c};
24:     if I_c = I then closed ← false;
25:     L ← ExploreCCIG(c_i, T · ⟨c_i⟩, I_c, V, E, cdr(C), L);
26:   end
27:   if closed then L ← L ∪ {T};
28:   return(L);
29: end
```

Fig. 3 Sequential algorithm to enumerate all CCIGs of graph.

3. Task-Parallel Language Tascell

Before describing the implementation of COPINE, we explain the mechanism of dynamic load balancing in Tascell.

3.1 Overview

Figure 4 shows a multistage overview of Tascell. Compiled Tascell programs are executed on one or more computing nodes. Each computing node has one or more workers in the shared memory environment, and is TCP/IP connected to a relay server named Tascell server. Thus, Tascell realizes parallel computation in distributed memory environments by connecting computing nodes via Tascell servers.

A Tascell server relays messages among computing nodes, processes input and output to the user, and manages workloads of computing nodes. As shown in Fig. 4, a Tascell server can be connected to another Tascell server. Thus, computing nodes and Tascell servers generally form a tree, although we do not include performance evaluations with more than one Tascell server in this paper.

3.2 Dynamic Load Balancing

Tascell balances workloads among workers using the dynamic load balancing mechanism of "work-stealing," whereby an idle worker steals part of another worker's task. In Tascell, 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) the information passed to a worker during task assignment, e.g., data referred to or updated by the worker. During execution, the runtime program creates tasks and assigns

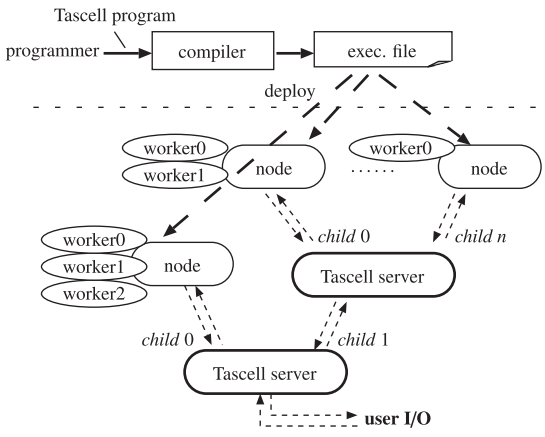


Fig. 4 Multistage overview of the Tascell framework.

[*3] The time required for this elimination is not considered in the performance evaluation in Section 6 because we can obtain a sound set of CCIGs in short time independently of the parallel search method that is used. This can be achieved by traversing the search tree again by referring to the "perfect" itemset table, which is obtained by the first traversal.

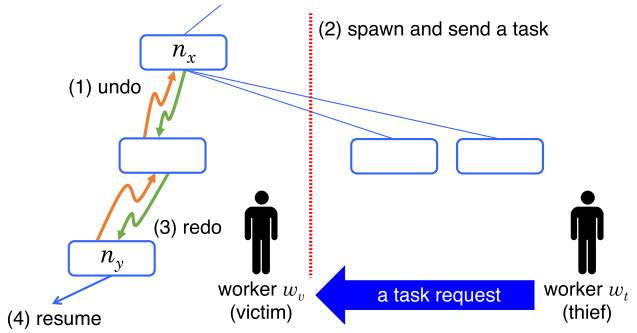


Fig. 5 Spawning a task lazily in Tascell.

them to workers automatically.

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 for which a task can be spawned (e.g., a parallel loop), it simply remembers the possibility at this point, and then executes the statement *as if choosing a completely sequential execution*. Each worker has its own workspace containing the data required for the search, and the search data are updated at each step.

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 at which the largest task can be spawned, and then spawns a task *as if changing the choice of execution to parallel from sequential*. The victim then allocates and initializes a new workspace for the task by making a copy of its workspace after backtracking.

Figure 5 illustrates how a task is spawned lazily, that is, only after a worker receives a task request. Suppose that n_x is the oldest task-spawnable point passed by the worker w_v. When w_v receives a task request from the worker w_t,

(1) it backtracks to n_x performing *undo* operations to restore the state of its workspace at n_x, and then
(2) spawns a task to traverse the right subtree.

w_v creates part of the unexecuted iterations of a parallel `for` loop at n_x as a task. After sending the task to w_t,

(3) w_v returns from the backtracking with *redo* operations to restore the state of its workspace before backtracking, and then
(4) resumes its own task.

Each task and its result are transmitted as a *task object* among workers. It can be transferred by passing the pointer in shared memory environments, or by serializing it via Tascell servers in distributed memory environments.

3.3 Work-Stealing Strategy

3.3.1 Task Request

In conventional Tascell, a thief worker uses the following strategy to choose a victim worker to which to send a task request.

(1) A thief randomly chooses a victim among other workers in the same computing node and sends a task request to it. If the victim can spawn a task, it spawns a task and sends it to the thief, as described in Section 3.2. Otherwise, the victim sends a refusal message to the thief.
(2) If the thief received a task as a response to the request in (1), it executes the task. If it received a refusal message, it chooses another worker in the same node as a victim and sends a task request to it.
(3) If the thief received refusal messages from all workers in the same computing node after repeating (1) and (2), that is, there are no task-spawnable workers in the node, the representative worker in the node sends a task request to the Tascell server directly connected to the node.
(4) The Tascell server that received a task request randomly chooses a computing node among nodes connected to the server excluding the request sender, and forwards the request to it.
(5) The computing node that received a task request forwarded by a Tascell server checks workers in the node in a random order. If the node contains a worker with a task that can be spawned, the worker spawns the task and sends the task to the thief via the Tascell server. Otherwise, the node sends a refusal message to the thief, again via the Tascell server.
(6) If the thief cannot acquire a task from external computing nodes, it returns to (1) and retries stealing a task after halting for a while.

According to this strategy, a thief always steals a task from a worker inside the same computing node if there is a worker with a task that can be spawned. We discuss the problems associated with this strategy and propose solutions to them in Section 5.1.

3.3.2 Stealing Back

When a Tascell worker cannot process its running task without receiving the result of a task that has been sent to another worker as part of the running task, the worker tries to steal another task as a thief rather than becoming idle waiting for the result. On this occasion, the victim of the request to steal is not chosen using the strategy explained in Section 3.3.1 but the thief *steals back* a task from the worker to which the task causing the synchronization is assigned. Using this technique, we can guarantee that the maximum size of execution stacks of workers does not exceed a constant times the maximum stack size encountered during a sequential execution [7]. On the other hand, this approach causes performance problems. We discuss these problems and propose solutions to them in Section 5.2.

4. Itemset Table Sharing

The efficient implementation of Pruning 3 in a parallel search requires workers to share table information under the restriction described in Section 2.3. This necessitates the design of an efficient sharing method by considering a trade-off between increasing the opportunity of using itemsets registered by other workers and reducing the cost of sharing information. In Section 4.1, we explain the sharing method in shared memory environments presented in [4]. Then, we introduce a sharing method in distributed memory environments in Section 4.2.

4.1 Implementation in Shared Memory Environments

In our COPINE implementation in shared memory environments, all workers share a single itemset table with a mutual exclusion lock for each table entry. We satisfy the restriction de-

scribed in Section 2.3 by associating 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 value of 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.

Since there are too many search tree nodes to associate a unique ID with each node, we associate a range [minID, maxID] represented by the 128-bit integers minID ≤ maxID with each task as its task ID and add the ID to each itemset registered in an itemset table entry to denote where in a search tree the itemset is registered. The full range of unsigned 128-bit integers $[0, 2^{128} - 1]$ is associated with the root task, which is first assigned to a certain worker at the beginning of a search. When a task is divided, we divide the range in half, allocating the lower and higher subranges to the tasks in the left and right portions of the subtree to be split. When a worker registers an itemset I to the itemset table, the value of the minID of the running task 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 that of the running task, I' is removed from the entry. The other itemsets remain registered. A worker can only use an itemset in the itemset table for Pruning 3 if the task ID of the itemset is not greater than that of the task being executed.

This management technique enables the order of tasks to be defined by the order of their minIDs. As this order of tasks is equivalent to the order in which corresponding subtrees are visited in the sequential search, we can apply Pruning 3 without the loss of completeness.

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 further, a worker executes a task with such a range sequentially. Although such a sequential task might cause a load imbalance if its size were large, our implementation with 128-bit unsigned integers divides the root task up to 128 times recursively such that the resulting sequential tasks are sufficiently small.

4.2 Implementation in Distributed Memory Environments

A worker can immediately use itemsets registered by other workers by using the sharing method described in Section 4.1. However, it is unrealistic to use this method in distributed memory environments because the communication cost of acquiring mutual exclusion locks is excessively high. Therefore, we implemented the following method, which allows table information to be shared at a realistic cost while allowing some incompleteness in sharing table information across computing nodes.

We prepare an itemset table in each computing node. All workers in each node share the single itemset table controlled by locks as described in Section 4.1. In addition, each computing node sends updates of its own table to other nodes at regular time intervals of t_{comm}. In our implementation, in addition to worker threads, a communication thread is created in each computing node for sending and receiving such updates. The communication thread repeats the following operations at intervals of t_{comm}:

```
1  while(1) {
2    uint64_t send_buf[], recv_buf[];
3    sleep(t_comm);
4    /* extract table updates and pack them into send_buf */
5    for(i=0; i<|V|; i++) {
6      pthread_mutex_lock(&table_entry_lock[i]);
7      extract itemsets registered to the table entry v_i.I after the previous
       communication and append them to send_buf.
8      pthread_mutex_unlock(&table_entry_lock[i]);
9    }
10   /* exchange the sizes of send_buf in all nodes */
11   MPI_Allgather(the size of send_buf, ...);
12   /* exchange the table updates in all nodes */
13   MPI_Allgatherv(send_buf, ..., recv_buf, ...);
14   /* unpack the received table updates from recv_buf */
15   for(k=0; k<# of itemsets in recv_buf; k++) {
16     e = the k-th element in recv_buf;
17     i = the vertex number in e;
18     pthread_mutex_lock(&table_entry_lock[i]);
19     register the itemset in e to the table entry v_i.I.
20     pthread_mutex_unlock(&table_entry_lock[i]);
21   }
22 }
```

Fig. 6 Flow of the communication thread for updates to an itemset table.

(1) extracting table updates of its computing node, (2) sending the updates to all other nodes and receiving updates from them, and (3) registering the received updates to the table of its node.

Figure 6 shows the pseudo-code of the operations of a communication thread. The communication thread of each computing node extracts all itemsets (including their minIDs and maxIDs described in Section 4.1) registered to the table after the previous communication, and stores them in the send buffer. Then, the communication threads in all the computing nodes synchronously exchange the contents in their send buffers using MPI collective communication functions. After receiving updates from external nodes, the communication thread registers the updates to the table of its node. At this time, the communication thread refers to minID and maxID associated with each received itemset and performs the operations described in Section 4.1 to satisfy the restriction described in Section 2.3 considering the order of itemsets based on minIDs.

We represent a set of itemsets stored in each table entry as a linked list in which a new itemset is added to the head. Therefore, the time required to extract table updates is proportional to the number of new itemsets.

5. Improvement of Work-Stealing Strategy

As explained in Section 3.2, a victim worker that received a task request backtracks to the oldest task-spawnable point to spawn a larger task. On the other hand, the strategy of a thief to choose a victim, explained in Section 3.3.1, does not aim to seek a victim that can create a larger task from candidates. In addition, due to the stealing back mechanism mentioned in Section 3.3.2, a thief is often restricted to sending a task request only to a specific worker. This approach did not cause considerable performance problems in the evaluations conducted in Refs. [3], [5]. However, the cost of a work-steal in our parallel COPINE implementation is high because the size of a task object in COPINE, which contains a current subgraph from which a thief starts its search, a set of vertices adjacent to the subgraph, etc., is large. Since such a task object is created and transferred among workers at every work steal, an increase in the number of small tasks that are stolen

leads to serious performance degradation.

As described in the remainder of this section, we improved the strategy for choosing a victim and tried to reduce the number of tasks that are stolen back to allow workers to obtain larger tasks.

5.1 Strategy for Choosing a Victim

5.1.1 Task Request to External Computing Node

In the conventional strategy explained in Section 3.3.1, a thief does not send a task request to a worker in an external computing node as long as a worker with a spawnable task exists in the same node. This means that, once the representative worker in a node acquired a task, none of the workers in the node send task requests to external nodes until the first task is completed (except for stealing back). This strategy is effective from the viewpoint of reducing the number of internode tasks that are stolen. On the other hand, it causes the problem that a thief always sends a task request to a worker in the same node even if it can only acquire a small task in the same node but can obtain a larger task from an external node. The thief worker completes such a small task in a short time and obtains another small task. Such repetition continues until all tasks assigned to the node are completed; a number of small tasks are transferred among workers inside the node during the repetition.

We solved this problem by implementing a strategy that promotes workers to request tasks from external nodes. When sending a task request, a thief omits step (1) in Section 3.3.1 and chooses a worker in an external node as a victim if the number of uncompleted tasks taken from external nodes (except tasks taken by stealing back) is less than a threshold τ. The larger the value of τ, the more frequently workers request tasks from external nodes. Note that this strategy is equivalent to the conventional strategy when $\tau = 1$.

5.1.2 Choosing a Victim inside a Computing Node

In addition to the strategy described in Section 5.1.1, we implemented a strategy where, at steps (1) and (5) in Section 3.3.1, a thief estimates the sizes of tasks being executed by other workers in the computing node and chooses a worker with the maximum estimated size as a victim from which to steal a larger task. More concretely, the following operations are performed.

- Each worker always remembers the number of divisions of its running task counting from the root task. In the COPINE case, the root task corresponds to a task to traverse the whole search tree.
- Steps (1) and (5) in Section 3.3.1 are modified as follows:
  - The thief randomly chooses κ workers ($1 \leq \kappa \leq$ the number of workers in the node) from the workers in the node, and checks the numbers of task divisions of their running tasks.
  - It chooses a worker with the smallest number of task divisions among the κ workers as a victim.

This strategy is based on the hypothesis that the larger the number of task divisions, the smaller the resulting task. The optimal value of κ should be found considering a trade-off between the expected sizes of stolen tasks and the cost of checking other workers. Note that this strategy is equivalent to the conventional strategy when $\kappa = 1$.

5.2 Reducing the number of Stealing Backs across Computing Nodes

As discussed in Section 3.3.2, a worker waiting for the result of another task is restricted to stealing back a task from the worker to which the task causing the synchronization is assigned, even if another worker can spawn a larger task. As stated in Section 5.1.1, when a worker completes a small task stolen back in a short time, it tries to steal back another task. This repetition produces a number of work-steals for small tasks between the two workers. Note that such a situation can occur between workers in different computing nodes. Furthermore, a victim of stealing back can also steal back another worker's small task at the same time. Such a chain of stealing back tasks involving a number of workers across computing nodes significantly degrades the performance.

Unlike the problem discussed in Section 5.1.1, we cannot solve this problem by changing the destination of a request for a task that has been stolen back. Therefore, we tried to alleviate the performance degradation caused by the stealing-back chain by reducing the number of tasks stolen back. More concretely, a worker that waits for the result of another task that is assigned to a worker in an external node waits for a certain time t_{sb} before sending a stealing-back request. It is expected that we can prevent the thief from stealing a small task whose creation and transfer cost exceeds the performance gain obtained by parallelization, since the thief would receive the result of the spawned task within the waiting time if the task is too small to be worth being stolen.

6. Performance Evaluation

6.1 Evaluation Setup

We measured the performance of our implementations in distributed memory environments using an Appro GreenBlade 8000 supercomputer. For comparison, we also used a Cray XC30 supercomputer with Xeon Phi coprocessors to measure the performance in shared memory environments with many workers. Both of these systems are supercomputers of ACCMS, Kyoto University. The evaluation environments are summarized in **Table 2**.

Table 2 Evaluation environment.

	Appro GreenBlade 8000	Cray XC30 with Xeon Phi
CPU	Intel Xeon E5 2.6 GHz 8-core × 2 (16 cores in total per node)	Xeon Phi 5120D 1.053 GHz 60-core
Memory	DDR3-1600 64 GB	GDDR5 8 GB
Compier	GCC 4.4.7 with `-O3`	Intel Compiler 15.0.6 with `-O3`
Worker	Created by `pthread_create` with `PTHREAD_SCOPE_SYSTEM`	
Lock	A `pthread_mutex_t` lock is attached to each table entry	
Tascell Server	Implemented in Allegro Common Lisp 8.1 with `(speed 3) (safety 1) (space 1)` Executed on one of the same computing nodes	—
Network	InfiniBand FDR × 2	—
MPI	Intel MPI Library 5.1.3	—

Table 4 Results of performance evaluation with multiple computing nodes.

Implementation	Parameter	$(n \times w)$	Exec. time [s]	Speedup (vs. C)	Speedup (vs. one worker)	# of visits to vertices (total amt. of all workers)	# of visits to vertices / s (avg. among workers)	# of task creations	Task exec. rate [%]
C	—	(1×1)	30.3	1	—	614,153,293	20,269,085	—	100 ± 0
One worker	—	(1×1)	47.0	0.644	1	614,153,293	13,072,213	—	100 ± 0
Conventional	$t_{\mathrm{comm}} = 500\,\mathrm{ms}$, $\tau = 1$, $\kappa = 1$, $t_{\mathrm{sb}} = 0\,\mathrm{s}$	(1×16)	11.4	2.66	4.13	1,343,142,530	7,383,659	8,658	81.3 ± 4.20
		(2×16)	10.6	2.85	4.43	1,727,404,195	5,087,872	8,180	54.0 ± 21.1
		(4×16)	10.3	2.95	4.58	2,259,465,895	3,440,079	16,014	38.3 ± 12.6
		(8×16)	10.5	2.88	4.48	3,142,857,961	2,340,209	31,310	27.7 ± 15.3
		(16×16)	10.8	2.81	4.36	3,280,427,117	1,189,989	27,798	13.9 ± 12.8
Proposed	$t_{\mathrm{comm}} = 100\,\mathrm{ms}$, $\tau = 2$, $\kappa = 16$, $t_{\mathrm{sb}} = 100\,\mathrm{ms}$	(1×16)	10.5	2.87	4.45	1,281,925,347	7,596,562	10,321	83.4 ± 5.69
		(2×16)	8.74	3.46	5.38	1,581,756,502	5,658,729	8,430	62.4 ± 10.2
		(4×16)	7.72	3.92	6.09	2,293,151,188	4,640,698	12,526	60.0 ± 11.4
		(8×16)	6.51	4.65	7.22	2,839,150,383	3,555,790	21,938	44.3 ± 11.8
		(16×16)	7.37	4.11	6.38	3,285,588,321	1,742,400	25,760	22.6 ± 11.3

Table 3 Characteristics of the graph used in the evaluation.

Parameter	Value
$\lvert V \rvert$	15,227
$\lvert E \rvert$	225,458
$\lvert I \rvert$	158
Average degree	29.6
Diameter of G	12
# of vertices in the largest connected component	15,061
# of vertices in the smallest connected component	1
Average # of items in each vertex	9.42

We used a real protein network and an itemset created by referring to the database [8] and the results shown in [9], respectively, as the input. **Table 3** shows the characteristics of this graph. We set the threshold θ to 5.

Note that we executed the program three times for each measurement setting to select the median of the execution times to be shown in the tables and the charts in this section. The relative errors of the median of the three samples were less than ±10% for every setting.

6.2 Performance in Distributed Memory Environments

We evaluated the performance improvement attained by the work-stealing strategies proposed in Section 5 by measuring the performance with the Appro GreenBlade 8000 on multiple computing nodes with the following two parameter settings:

(1) $t_{\mathrm{comm}} = 500\,\mathrm{ms}$, $\tau = 1$, $\kappa = 1$, and $t_{\mathrm{sb}} = 0\,\mathrm{s}$, that is, the setting to simulate the conventional work-stealing strategy, and

(2) $t_{\mathrm{comm}} = 100\,\mathrm{ms}$, $\tau = 2$, $\kappa = 16$, and $t_{\mathrm{sb}} = 100\,\mathrm{ms}$, which we found by the following optimal parameter search using 8 nodes × 16 workers.

   (a) We found the optimal values of t_{comm} and κ independently by varying t_{comm} or κ within the range of $t_{\mathrm{comm}} \in \{100\,\mathrm{ms}, 250\,\mathrm{ms}, 500\,\mathrm{ms}, 750\,\mathrm{ms}, 1\,\mathrm{s}\}$ or $\kappa \in \{1, 2, 4, 8, 16\}$, respectively, while fixing the other parameters to the same values as (1).

   (b) We found the optimal values of τ and t_{sb} by trying all the combinations of (τ, t_{sb}) that satisfy $\tau \in \{1, 2, 4, 8, 16\}$ and $t_{\mathrm{sb}} \in \{0\,\mathrm{s}, 1\,\mathrm{ms}, 10\,\mathrm{ms}, 100\,\mathrm{ms}, 1\,\mathrm{s}\}$, while fixing t_{comm} and κ to the values found in (a).

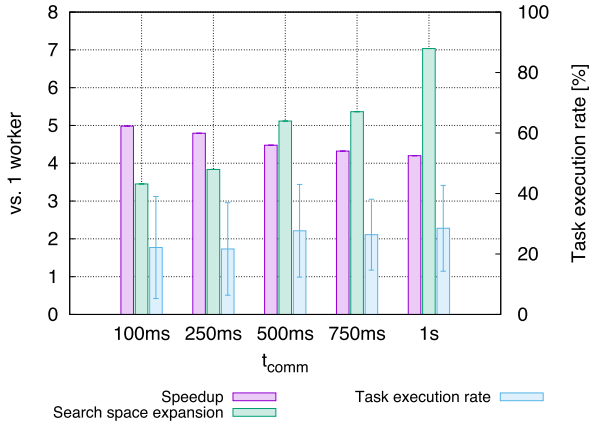
We also compared the performance of each implementation with that of the sequential COPINE implementation written in C. **Table 4** contains the results of the evaluation. Note that, in Table 4, n denotes the number of computing nodes, and w denotes the number of workers per node. "Task execution rate" means the average and the standard deviation of the percentage of time spent on task execution to the total execution time among all workers. "Conventional" and "Proposed" are executions with the performance parameter settings (1) and (2) above, respectively.

The results of "Conventional" in Table 4 indicate that the task execution rates significantly decrease as the number of nodes increases, and we cannot obtain speedups with multiple computing nodes. On the other hand, the "Proposed" strategy enables us to alleviate the degradation of the task execution rates. The improvement in the execution rates means an improvement in the degrees of parallelism. This usually leads to degradation of the completeness of Pruning 3, or an increase in the number of visits to vertices [*4]. However, the number of visits to vertices of "Proposed" are almost the same as those of "Conventional." This is because the communication interval to exchange table updates (t_{comm}) of "Proposed" is shorter than that of "Conventional." As a result, we can obtain speedups with multiple computing nodes using the proposed strategy. For instance, with 8 nodes × 16 workers, we achieved a 7.22-fold speedup compared to the one-worker execution, which is 4.65-fold to C and 1.75-fold (= 7.22/4.13) to the execution by 1 node × 16 workers. However, we were unable to achieve good performance in the 16-node execution of "Proposed" because the task execution rate significantly degrades. Future work is needed to improve the performance with additional computing nodes.

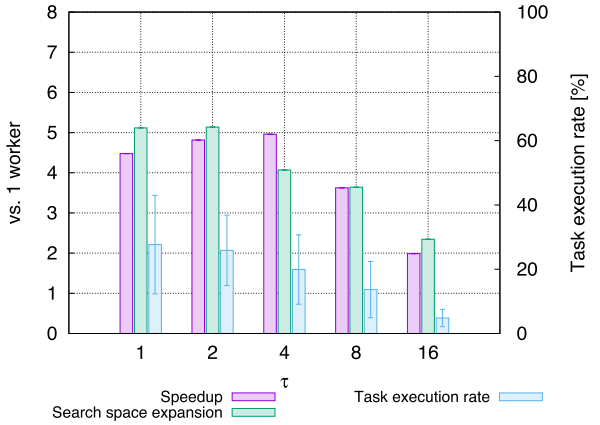
We evaluated the effect of changing the values of $t_{\mathrm{comm}}, \tau, \kappa$, and t_{sb} by measuring the performance with 8 nodes × 16 workers when varying the values of these parameters one by one while fixing the other parameters to the settings of "Conventional." **Figure 7** shows the measurement results. The error bars of the task execution rates show the standard deviation among workers. These results show the following.

- Reduction of t_{comm} enables us to reduce the total number of visits to vertices and improve the overall performance. In **Table 5**, we show the cumulative processing time of the communication thread in the process with MPI rank 0 when t_{comm} is set to 100 ms, 500 ms, and 1 s. We can see that the increase in the cost of communication threads when reducing t_{comm} to 100 ms is sufficiently small relative to the effect of reducing the number of visits to vertices.
- When we promote workers to request tasks from external

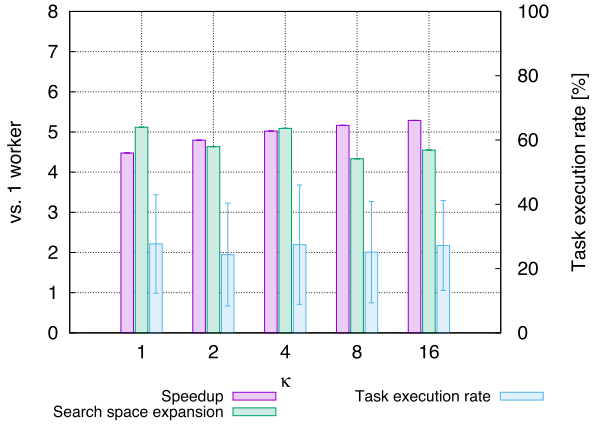
[*4] This problem also occurs in shared memory environments, and is difficult to solve as discussed in [4].



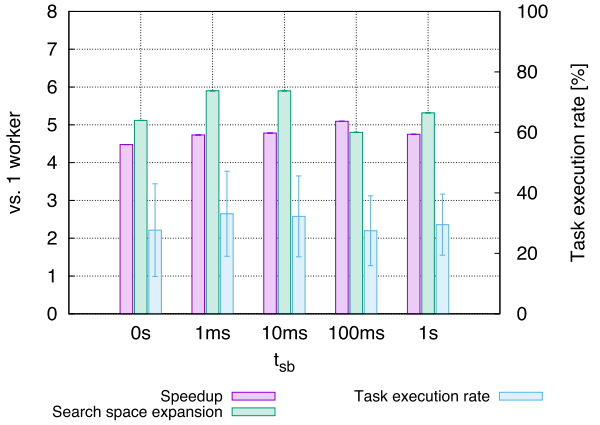
(a) t_{comm}: communication interval to broadcast table updates



(b) τ: threshold determining whether a task request to external node is sent



(c) κ: # of workers where # of task divisions is inquired



(d) t_{sb}: waiting time before stealing back

Fig. 7 Performance when changing the values of performance parameters (using 8 nodes × 16 workers).

Table 5 Cumulative processing time of a communication thread.

t_{comm}	Pack [s]	MPI comm. [s]	Unpack [s]	Total [s]
100 ms	0.0362	1.46	1.30	2.80
500 ms	0.0147	0.426	0.798	1.324
1 s	0.0162	0.203	0.816	1.04

nodes by increasing τ, the task execution rate degrades. This is because the increase in the amount of internode work stolen also results in an increase in the number of internode tasks stolen back. Note that we obtained a slight performance improvement in spite of the degradation of the task execution rates when τ is 2 and 4. This is because the decrease in the degrees of parallelism resulted in a decrease in the number of visits to vertices.

- When κ increases, there is no great change in the task execution rate, but the overall performance is slightly improved.
- There is no correlation between t_{sb} and the performance or the task execution rate.

We can conclude from these results that we cannot improve task execution rates and overall performance when varying only one of τ and t_{sb}. We evaluated the effect of varying both τ and t_{sb} in detail by rechecking the results of the performance measurements that were conducted to find the settings of "Proposed," that is, the measurements when varying τ and t_{sb} at the same time while fixing t_{comm} and κ to 100 ms and 16, respectively. **Table 6** shows the execution times and the task execution rates with these

Table 6 Performance when changing the values of τ and t_{sb} (using 8 nodes × 16 workers).

Execution time [s]
(Average task execution rate among workers ± S.D. [%])

		t_{sb}				
		0 s	1 ms	10 ms	100 ms	1 s
τ	1	8.62 (32.9 ± 12.3)	8.24 (24.6 ± 19.4)	8.23 (30.3 ± 14.0)	7.99 (37.9 ± 9.27)	7.23 (37.1 ± 8.84)
	2	7.97 (29.2 ± 16.5)	7.89 (33.0 ± 15.6)	7.51 (36.4 ± 12.9)	6.51 (44.3 ± 11.8)	6.68 (38.4 ± 9.46)
	4	7.76 (24.8 ± 13.7)	7.66 (32.5 ± 13.6)	7.50 (33.1 ± 10.4)	7.11 (41.2 ± 13.2)	7.75 (34.3 ± 13.1)
	8	12.9 (14.3 ± 9.16)	12.7 (13.1 ± 8.22)	12.7 (17.8 ± 11.9)	11.4 (17.0 ± 9.11)	10.8 (20.3 ± 9.91)
	16	22.3 (4.83 ± 3.28)	21.1 (4.91 ± 3.30)	23.2 (5.90 ± 3.86)	20.8 (6.86 ± 4.04)	19.5 (6.45 ± 4.07)

settings. It is clear that the task execution rate has considerably improved as t_{sb} increases when τ is set to 2 or 4. One possible reason for these performance improvements is that, not only does the number of stolen internode tasks increase when τ increases, the increase in the number of internode tasks stolen back is suppressed by increasing t_{sb}.

6.3 Comparison with Performance in Shared Memory Environments

In order to compare the performance between shared and distributed memory environments, we measured the performance of each environment with 1 node × 1–16 workers and 1–16 nodes ×

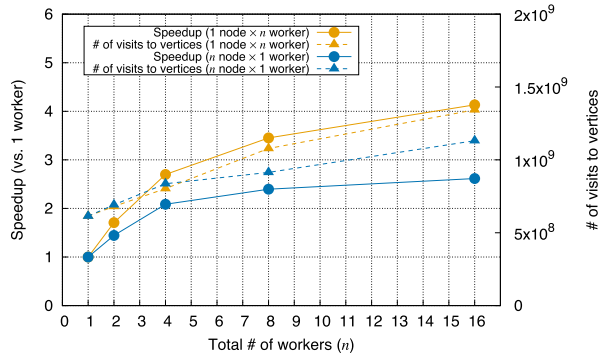


Fig. 8 Performance comparison between shared and distributed memory environments.

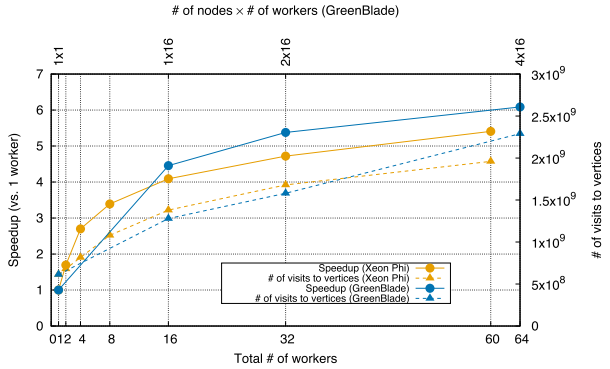


Fig. 9 Results of performance evaluation on the Xeon Phi coprocessor.

1 worker on the Appro GreenBlade 8000. We set the parameters to the values of "Conventional" in Table 4. **Figure 8** shows the measurement results.

A comparison of the results in Fig. 8 indicates that the difference between the performance in shared and distributed memory environments becomes large as the number of workers increases. For instance, the performance in the distributed memory environment is 36.6% worse than that in the shared memory environment when $n = 16$. This is mainly due to the degradation of the task execution rate in distributed memory environments. In fact, the task execution rates in shared and distributed memory environments are $81.3 \pm 4.20\%$ and $32.5 \pm 13.1\%$, respectively, when $n = 16$. The task execution rate degrades in distributed memory environments because the cost of stealing internode work is large. Note that the number of visits to vertices in distributed memory environments is smaller than that in shared memory environments due to the decrease in the degrees of parallelism caused by the degradation of the task execution rates. Thus, only from these results, it is difficult to evaluate the extent to which the incompleteness of table information shared across computing nodes affected the performance.

We also measured the performance in shared memory environments with up to 60 workers on the Xeon Phi coprocessor and compared the results with those on which the "Proposed" strategy was implemented in Table 4 using 1–4 nodes × 16 workers on the Appro GreenBlade 8000. The results are shown in **Fig. 9**. In the executions on the Xeon Phi, a worker can immediately refer to itemsets registered by other workers, and the cost of stealing work is smaller than that in distributed memory environments. Nevertheless, we achieved only a 5.41-fold speedup with 60 workers, because the size of the search space enlarges as the number of workers increases. The speedup with 4 nodes × 16 workers on the Appro GreenBlade 8000, which is 6.09-fold relative to the one-worker execution, is comparable to that in shared memory environments with approximately the same number of workers.

7. Related Work

7.1 Distributed Memory Environment Support of Task-Parallel Languages

Apart from Tascell, there are other task-parallel languages and libraries that support distributed memory environments. For example, Distributed Cilk [10], SilkRoad [11], X10/GLB [12], and Uni-Address Threads [13] are task-parallel execution frameworks that support dynamic load balancing across computing nodes.

However, there has been little previous effort to implement real backtrack search applications that require knowledge sharing for pruning in parallel using such frameworks. In addition, as discussed in [4], it is difficult to efficiently implement the parallel COPINE algorithm targeted in this research without using the temporary backtracking mechanism featured by Tascell.

7.2 Parallel Backtrack Search Implementations in Distributed Memory Environments

Game tree searches are one of the important applications of backtrack search algorithms with pruning. Board games such as chess and shogi (Japanese chess) allow the player to check all the possible moves by traversing a search tree named game tree. Massive parallelization of game tree searches using multiple computing nodes is becoming a popular way to improve the performance. However, it is not easy to obtain effective parallel performance in such environments. One of the challenges is sharing information in a transposition table, which is a hash table with the positions as the keys and is used to avoid duplicate searches of identical positions. Some mechanisms have been proposed to manage such a table in distributed memory environments to alleviate the increase in the search space caused by the incompleteness of information sharing. For example, Transposition table Driven work Scheduling (TDS) [14] uses a distributed hash table. When a task is created, a hash value for the task is calculated, and the task is sent to a computing node to which the hash value is assigned. In TDS, internode communications are required for sending tasks to external nodes but all table accesses are local. However, its performance depends on the quality of the hash function; redundant search is performed when a task is sent to an inappropriate computing node.

In PaSAT [15] and ySAT [16], which are parallel implementations of the Satisfiability Problem (SAT), each worker has its own table and exchanges its contents with other workers periodically to update the table. This sharing method is similar to our implementation. SAT presents another implementation problem in that it is difficult 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, because the number of itemsets registered in an entry corresponding to a vertex v is at most $2^{|I(v)|}$, or more precisely ${}_{|I(v)|}C_{|I(v)|/2}$.

8. Conclusion and Future Work

In this paper, we proposed a parallel implementation of the COPINE algorithm for graph mining that extracts all connected subgraphs with common itemsets of which the size is not less than a given threshold, in distributed memory environments.

We ensured that table information related to pruning is shared across computing nodes by implementing a sharing method whereby computing nodes exchange their table updates periodically by using the collective communication functions of MPI.

Although the task-parallel language Tascell used in the implementation supports distributed memory environments, the conventional work-stealing strategy in Tascell, which aims to minimize the number of internode work-steals and tends to increase the number of work-steals for small tasks, is not efficient from the viewpoint of effective load balancing. Therefore, we employed new work-stealing strategies in which workers request tasks from external nodes more frequently and estimate the sizes of tasks created by victim candidates, to obtain larger tasks.

As a result of these improvements, we achieved a 7.22-fold speedup with 8 nodes × 16 workers compared to the one-worker execution in the performance evaluation using a real protein network.

Our future work includes improving the performance in more highly parallel computing environments. In executions with 16 nodes, we were unable to obtain accelerated performance due to the significant degradation of the task execution rate. We aim to solve this problem by further improving the strategy for choosing a victim. We also need to improve the implementation of Tascell itself, including the stealing back mechanism, which is a major factor that restricts a worker in choosing a victim. Despite our attempt to use many workers to obtain sufficient speedups this was neither successful in shared nor in distributed memory environments. This is because the search space enlarges as the number of workers increases. One possible approach to this problem is to abort the execution of a worker traversing a subtree pruned by another worker. We have already implemented this abort mechanism using exception handling features in a task-parallel language and confirmed the performance improvement in shared memory environments [17], [18]. We plan to enhance this implementation for distributed memory environments.

Acknowledgments This work was supported in part by JSPS KAKENHI Grant Numbers 25730041 and 26280023.

References

[1] 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).

[2] 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).

[3] 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).

[4] Okuno, S., Hiraishi, T., Nakashima, H., Yasugi, M. and Sese, J.: Parallelization of Extracting Connected Subgraphs with Common Itemsets, *IPSJ Trans. Programming*, Vol.7, No.3, pp.22–39 (2014).

[5] Hiraishi, T., Yasugi, M. and Umatani, S.: Evaluation of the Tascell Dynamic Load Balancing Framework in Widely Distributed and Many-Core Environments, *Proc. Symposium on Advanced Computing Systems and Infrastructures 2011*, *SACSIS 2011*, pp.55–63 (2011). (in Japanese).

[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] Wagner, D.B. and Calder, B.G.: Leapfrogging: A Portable Technique for Implementing Efficient Futures, *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, *PPoPP '93*, pp.208–217 (1993).

[8] Razick, S., Magklaras, G. and Donaldson, I.: iRefIndex: A consolidated protein interaction database with provenance, *BMC Bioinformatics*, Vol.9, p.405 (2008).

[9] Su, A.I., Wiltshire, T., Batalov, S., Lapp, H., Ching, K.A., Block, D., Zhang, J., Soden, R., Hayakawa, M., Kreiman, G., Cooke, M.P., Walker, J.R. and Hogenesch, J.B.: A gene atlas of the mouse and human protein-encoding transcriptomes, *Proc. National Academy of Sciences of the United States of America*, Vol.101, No.16, pp.6062–6067 (2004).

[10] Randall, K.H.: Cilk: Efficient Multithreaded Computing, PhD Thesis, Massachusetts Institute of Technology (1998).

[11] Peng, L., Wong, W.F., Feng, M.D. and Yuen, C.K.: SilkRoad: A Multithreaded Runtime System with Software Distributed Shared Memory for SMP Clusters, *Proc. IEEE International Conference on Cluster Computing*, *Cluster 2000*, pp.243–249 (2000).

[12] Zhang, W., Tardieu, O., Grove, D., Herta, B., Kamada, T., Saraswat, V.A. and Takeuchi, M.: GLB: Lifeline-based Global Load Balancing library in X10, *Proc. 1st Workshop on Parallel Programming for Analytics Applications*, *PPAA '14*, pp.31–40 (2014).

[13] Akiyama, S. and Taura, K.: Uni-Address Threads: Scalable Thread Management for RDMA-Based Work Stealing, *Proc. 24th International Symposium on High-Performance Parallel and Distributed Computing*, *HPDC '15*, pp.15–26 (2015).

[14] Romein, J.W., Plaat, A., Bal, H.E. and Schaeffer, J.: Transposition Table Driven Work Scheduling in Distributed Search, *Proc. 16th National Conference on Artificial Intelligence*, *AAAI '99*, pp.725–731 (1999).

[15] 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 (2001).

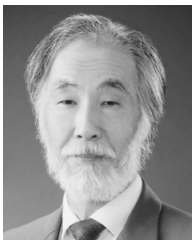
[16] Feldman, Y., Dershowitz, N. and Hanna, Z.: Parallel Multithreaded Satisfiability Solver: Design and Implementation, *Electronic Notes in Theoretical Computer Science*, Vol.128, No.3, pp.75–90 (2005).

[17] Okuno, S., Hiraishi, T., Nakashima, H., Yasugi, M. and Sese, J.: Reducing Redundant Search using Exception Handling in a Task-Parallel Language, *Annual Meeting on Advanced Computing System and Infrastructure 2015*, *ACSI 2015* (2015).

[18] Okuno, S., Hiraishi, T., Nakashima, H., Yasugi, M. and Sese, J.: Reducing Redundant Search in Parallel Graph Mining using Exceptions, *Proc. 2016 IEEE International Parallel and Distributed Processing Symposium Workshops*, *IPDPSW 2016*, pp.328–337 (2016).

Shingo Okuno was born in 1989. He received his B.E. in Information Systems Engineering from Osaka University in 2012, and his M.E. in Informatics from Kyoto University in 2014. He has been a Ph.D. student at Department of Systems Science, Graduate School of Informatics, Kyoto University since 2014. His research interests include parallelization of backtrack search algorithms. He is a student member of IPSJ and ACM. He won the Outstanding Research Award and the Outstanding Student Research Award of ACSI 2015.

Tasuku Hiraishi was born in 1981. He received his B.E., M.E., and Ph.D. degrees in Informatics all from Kyoto University in 2003, 2005, and 2008, respectively. In 2007–2008, he was a fellow of JSPS at Kyoto University. Since 2008, he has been working as an assistant professor at 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, the Japan Society for Software Science and Technology (JSSST), and ACM.



Hiroshi Nakashima received his M.E. and Ph.D. 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 in high-performance computing systems and programming on them. He received the Motooka award in 1988 and the Sakai award in 1993. 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 received the B.S., M.S. and Ph.D. from the University of Tokyo at 1999, 2001 and 2005, respectively. He worked at the University of Tokyo as a research associate from 2003 to 2006, at Ochanomizu University as an associate professor from 2006 to 2011, and at Tokyo Institute technology as an associate professor from 2011 to 2014. He is working at AIST from 2014. His interest is to develop new machine learning methods and their application to life science.