Parallelization of Graph Mining using Backtrack Search Algorithm

Title

Author(s)

Okuno, Shingo

Citation

Kyoto University (京都大学)

Issue Date

2017-03-23

URL

https://doi.org/10.14989/doctor.k20518

Type

Thesis or Dissertation

Publisher

出版社: 情報処理学会・IEEE 登録条件：・出典及び利用上の注意事項を明記・掲載論文の一部を学位論文に転載する場合は、出典を明記することで可能

Textversion

ETD

Kyoto University
Parallelization of Graph Mining using Backtrack Search Algorithm

by

Shingo Okuno

Submitted to the Graduate School of Informatics in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Informatics

Department of Systems Science
Graduate School of Informatics, Kyoto University

March 2017
Parallelization of Graph Mining using Backtrack Search Algorithm

by
Shingo Okuno

Abstract

This paper describes parallel implementations of a highly complex graph mining problem to extract Common Itemset connected subGraphs (CIGs) from a graph whose vertices are labeled by their own sets of items, or itemsets in short. The problem is to extract all connected subgraphs in a given graph, each of which satisfies that the cardinality of its common itemset, i.e., the intersection of the itemsets of all of its vertices, is not less than a given threshold. Our implementations are to parallelize this mining problem for both shared and distributed memory environments. This kind of graph mining can be applied to the analysis of social and biological networks.

An efficient sequential backtrack search algorithm named COPINE has already been proposed for this problem. COPINE avoids unnecessary searches using a pruning mechanism that depends on the knowledge acquired during the left- and depth-first traversal of its search tree. In a parallel search where a unique set of subtrees is assigned to each worker as its task, the branches in the subtrees must be pruned as well for efficiently reducing the search space, but not excessively by a blind consultation of the knowledge acquired by another worker. To avoid such excessive pruning, we found a restriction imposed on the workers when referring to the knowledge acquired by other workers. In consideration of this restriction, we designed a parallel extension of COPINE.

Because the search trees in COPINE have an irregular structure, dynamic load balancing should be applied in parallelized implementations. Applications with these properties are often implemented by task-parallel languages, by which we can dynamically spawn tasks to be automatically assigned to workers as parallel threads and/or processes. We implemented the parallel COPINE algorithm using the task-parallel language Tascell, which offers high performance in various backtrack search algorithms.

The parallel COPINE algorithm requires workers to share the acquired knowledge for the pruning mechanism. For shared memory environments, we implemented a sharing method in which a single table controlled by locks is shared among all workers. This method enables workers to refer to the knowledge acquired by another worker immediately. In addition, we proposed a task creation strategy whereby useful pruning knowledge can be acquired as early as possible in a parallel search.

On the implementation of this algorithm in distributed memory environments, we should consider the cost of internode communication. The sharing method shown above is impractical in distributed memory environments because internode communication is required every time a worker acquires new knowledge. Therefore, we also implemented a sharing method in which each computing node manages its own table and sends table updates to other nodes periodically. Furthermore, the conventional work-stealing strategy in Tascell, which aims to minimize the number of internode work-steals, could cause a load imbalance by increasing the number of intranode work-steals for small tasks. We solved this problem by implementing new work-stealing strategies in Tascell to enable workers to obtain larger tasks.
The traversal order in the parallel COPINE algorithm also affects its performance: in the sequential COPINE algorithm a useless subtree is always pruned before it is traversed, while in the parallel one a worker may prune the subtree after another worker has started to traverse it, resulting in a large number of redundant searches. Such redundancy can be significantly reduced by letting a worker know that the subtree it is traversing has been pruned, so that the traversal can be aborted. This abortion can be implemented elegantly and efficiently using task-parallel languages, which have a mechanism for exception handling whereby all parallel tasks running in a try block with an exception are automatically aborted. Since Tascell did not have such a capability, we enhanced it with a parallel try-catch mechanism to abort useless works in our parallel COPINE algorithm.

We evaluated our parallel implementations using a real protein network to confirm that they achieve good speedup performance in both shared and distributed memory environments.
Acknowledgments

This thesis summarizes my research results at Nakashima Laboratory, Department of Systems Science, Graduate School of Informatics, Kyoto University from 2012 to 2017. My research would not have been possible without the support I have received from many people.

I would like to especially express my deeply-felt gratitude to my greatest advisor, Prof. Hiroshi Nakashima of the Academic Center for Computing and Media Studies (ACCMS), Kyoto University, who gave me the environment and the opportunity to do my research. His invaluable advice and encouragement have been important guiding lights towards my personal and professional developments. I am proud to do my research under the direction of him.

Besides my advisor, I am grateful to the other distinguished members of my thesis committee: Prof. Hiroshi Nagamochi and Prof. Toshiyuki Tanaka both of the Graduate School of Informatics, Kyoto University, for their great deal of valuable advice to help me improve the quality of this thesis.

My gratitude also goes to Dr. Tasuku Hiraishi of ACCMS, Kyoto University, who led me to this fascinating research area. He has always been willing to dedicate his time to help me with my research.

I would like to thank Prof. Masahiro Yasugi of the Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology, and Dr. Jun Sese of National Institute of Advanced Industrial Science and Technology. They provided useful feedback on my research. In addition, Dr. Sese provided me with protein network data for performance evaluation.

I am appreciative to Prof. Takeshi Iwashita of the Information Initiative Center, Hokkaido University, Dr. Keichiro Fukazawa of ACCMS, Kyoto University, and Dr. Akihiro Ida and Dr. Masatoshi Kawai both of the Information Technology Center, the University of Tokyo. They gave me helpful suggestions for my research from various perspectives. I am also indebted to our secretaries Ms. Shigemi Mitsuzawa and Ms. Maki Takayama, and my labmates for enriching my life in the laboratory.

Last but not the least, I would like to thank my father, Keiichiro Okuno, and my mother, Nobuko Okuno. I could not accomplish my research work without their love and support.
## Contents

1 Introduction ............................. 1
  1.1 Background .................................. 1
  1.2 Our Proposal ................................ 2
  1.3 Contributions ............................. 3
  1.4 Organization of the Thesis ............... 4

2 COPINE Algorithm ......................... 5
  2.1 Definition of the Closed CIG Enumeration Problem .......... 5
  2.2 Sequential COPINE Algorithm .................... 7
  2.3 Correctness and Application to Parallel Search of Pruning .... 9
    2.3.1 Correctness of Pruning 1 ..................... 9
    2.3.2 Correctness of Pruning 2 ..................... 12
    2.3.3 Correctness and Application to Parallel Search of Pruning 3 .... 13
    2.3.4 Soundness of Search ......................... 14
  2.4 Parallel COPINE Algorithm ................... 15
  2.5 Conclusion ............................... 16

3 Task-Parallel Language Tascell .......... 17
  3.1 Overview .................................... 17
  3.2 Dynamic Load Balancing ..................... 18
  3.3 Work-Stealing Strategy ..................... 20
    3.3.1 Task Request ............................. 20
    3.3.2 Stealing Back .............................. 20
  3.4 Tascell Language ............................ 20
  3.5 Implementation .............................. 23
    3.5.1 Nested Functions ......................... 23
    3.5.2 Translation to C with Nested Functions ......... 24

4 Parallel Implementation in Shared Memory Environments .... 27
  4.1 Introduction ............................... 27
  4.2 Itemset Table Sharing ...................... 27
    4.2.1 Sharing Method 0: Non-Registering Method .......... 28
    4.2.2 Sharing Method 1: Non-Sharing Method ............. 28
    4.2.3 Sharing Method 2: Replicating Method ............. 28
    4.2.4 Sharing Method 3: Fully-Sharing Method ........... 28
  4.3 Advantageous Feature of Tascell for Parallel COPINE ........ 29
  4.4 Task Creation Strategy ..................... 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Performance Evaluation</td>
<td>31</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Evaluation Setup</td>
<td>31</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Performance with Standard Task Creation Strategy</td>
<td>32</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Effect of Dividing Ratio of Task</td>
<td>37</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Effect of Vertex Ordering</td>
<td>41</td>
</tr>
<tr>
<td>4.6</td>
<td>Conclusion</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>Parallel Implementation in Distributed Memory Environments</td>
<td>44</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>Itemset Table Sharing</td>
<td>45</td>
</tr>
<tr>
<td>5.3</td>
<td>Improvement of Work-Stealing Strategy</td>
<td>46</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Strategy for Choosing a Victim</td>
<td>46</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Reducing the Number of Tasks Stolen Back across Computing Nodes</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance Evaluation</td>
<td>48</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Evaluation Setup</td>
<td>48</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Performance in Distributed Memory Environments</td>
<td>48</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Comparison with Performance in Shared Memory Environments</td>
<td>50</td>
</tr>
<tr>
<td>5.5</td>
<td>Conclusion</td>
<td>55</td>
</tr>
<tr>
<td>6</td>
<td>Enhancement of Tascell for Abort Operation</td>
<td>56</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>56</td>
</tr>
<tr>
<td>6.2</td>
<td>Language Extension</td>
<td>59</td>
</tr>
<tr>
<td>6.3</td>
<td>Implementation of Exception Handling</td>
<td>62</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Tascell Compiler</td>
<td>62</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Task Scheduler</td>
<td>62</td>
</tr>
<tr>
<td>6.4</td>
<td>Performance Evaluation</td>
<td>65</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Evaluation Setup</td>
<td>65</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Overheads</td>
<td>66</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Task Abortion Time</td>
<td>66</td>
</tr>
<tr>
<td>6.5</td>
<td>Conclusion</td>
<td>68</td>
</tr>
<tr>
<td>7</td>
<td>Abortion of Redundant Search</td>
<td>71</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>71</td>
</tr>
<tr>
<td>7.2</td>
<td>Implementation of Abort Operations</td>
<td>71</td>
</tr>
<tr>
<td>7.3</td>
<td>Performance Evaluation</td>
<td>75</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Evaluation Setup</td>
<td>75</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Performance in Shared Memory Environments</td>
<td>75</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Performance in Distributed Memory Environments</td>
<td>80</td>
</tr>
<tr>
<td>7.4</td>
<td>Conclusion</td>
<td>82</td>
</tr>
<tr>
<td>8</td>
<td>Related Work</td>
<td>83</td>
</tr>
<tr>
<td>8.1</td>
<td>Parallel Backtrack Search Implementation</td>
<td>83</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Parallel Game Tree Search</td>
<td>83</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Parallel SAT Solver</td>
<td>84</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Task Scheduling in Parallel Tree Search</td>
<td>84</td>
</tr>
<tr>
<td>8.2</td>
<td>Parallel Programming Language</td>
<td>85</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Cilk</td>
<td>85</td>
</tr>
</tbody>
</table>
8.2.2 Intel Cilk Plus .................................................. 85
8.2.3 OpenMP ......................................................... 85
8.2.4 Java ................................................................. 85
8.2.5 X10 ................................................................. 86
8.2.6 Manticore ......................................................... 86
8.2.7 Uni-Address Threads ........................................... 87
8.3 Non-Backtracking Based Data Mining ........................................ 87
  8.3.1 Graph Mining using Mathematical Calculation .................... 87
  8.3.2 Frameworks for Graph Mining .................................. 87
  8.3.3 Frequent Itemset Mining ....................................... 88

9 Conclusion and Future Work ................................................ 89

A Proof of Correctness of COPINE ........................................... 91
  A.1 Proof of Theorem 1 ............................................... 91
  A.2 Proof of Theorem 3 ............................................... 94
  A.3 Proof of Theorem 4 ............................................... 96

Bibliography ........................................................................ 97

List of Publications .............................................................. 102
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example of a graph associated with itemsets.</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Search tree for the graph in Figure 2.1 ($\theta = 2$).</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Algorithm to form the CST of a connected subgraph.</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Algorithm to enumerate connected subgraphs.</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Example of excessive pruning ($\theta = 2$).</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Sequential algorithm to enumerate all CCIGs of a graph.</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Multistage overview of the Tascell framework.</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Spawning a task lazily in Tascell.</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Program (pseudo-code written in C) that performs backtrack search for finding all possible solutions to the Pentomino puzzle.</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>A Tascell program that performs a backtrack search for the Pentomino puzzle.</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Program with nested functions.</td>
<td>24</td>
</tr>
<tr>
<td>3.6</td>
<td>Translation result from the worker function search for Pentomino in Figure 3.4, including translation of a parallel for statement and a dynamic_wnd statement.</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>Speedup and number of visits to vertices with the standard task creation strategy (non-registering method, $\theta = 13$).</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>Speedup and number of visits to vertices with the standard task creation strategy (non-sharing method, $\theta = 5$).</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Speedup and number of visits to vertices with the standard task creation strategy (replicating method, $\theta = 5$).</td>
<td>35</td>
</tr>
<tr>
<td>4.4</td>
<td>Speedup and number of visits to vertices with the standard task creation strategy (fully-sharing method, $\theta = 5$).</td>
<td>35</td>
</tr>
<tr>
<td>4.5</td>
<td>Execution time and number of visits to vertices versus the number/ratio of iterations left for a victim worker (replicating method with 16 workers).</td>
<td>38</td>
</tr>
<tr>
<td>4.6</td>
<td>Execution time and number of visits to vertices versus the number/ratio of iterations left for a victim worker (fully-sharing method with 16 workers).</td>
<td>39</td>
</tr>
<tr>
<td>4.7</td>
<td>Speedup and number of visits to vertices with the setting that leaves one iteration for the victim worker on task creation (replicating method, $\theta = 5$).</td>
<td>40</td>
</tr>
<tr>
<td>4.8</td>
<td>Speedup and number of visits to vertices with the setting that leaves one iteration for the victim worker on task creation (fully-sharing method, $\theta = 5$).</td>
<td>40</td>
</tr>
<tr>
<td>4.9</td>
<td>Execution time and speedup with 16 workers versus vertex ordering (sorted by execution time with 16 workers).</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>Flow of the communication thread for updates to an itemset table.</td>
<td>45</td>
</tr>
<tr>
<td>5.2</td>
<td>Performance when changing the value of $t_{\text{comm}}$ (communication interval for broadcasting table updates) using 8 nodes $\times$ 16 workers.</td>
<td>51</td>
</tr>
</tbody>
</table>
5.3 Performance when changing the value of $\tau$ (threshold determining whether a task request is sent to external node) using 8 nodes $\times$ 16 workers. 
5.4 Performance when changing the value of $\kappa$ (# of workers where # of task divisions is queried) using 8 nodes $\times$ 16 workers.  
5.5 Performance when changing the value of $t_{sb}$ (waiting time before stealing back) using 8 nodes $\times$ 16 workers. 
5.6 Performance comparison between shared and distributed memory environments. 
5.7 Performance comparison between many-core environments and multiple computing nodes. 

6.1 Example of task-parallel program with exceptions. 
6.2 Execution states in an exception of the program in Figure 6.1. 
6.3 Parallel binary tree search for one solution. 
6.4 Parallel binary tree search for all solutions where exceptions are used to reduce redundant search. 
6.5 Tascell program that performs backtrack search for Pentomino and terminates the search as soon as a worker finds that the number of solutions is larger than $\text{THRESHOLD}$.  
6.6 Execution context of the program in Figure 6.5. 
6.7 Translation result from the function `task_exec` for Pentomino in Figure 6.5, including translation of a `try-catch` statement. 
6.8 Translation result from the worker function `search` for Pentomino in Figure 6.5, including translation of a parallel `for` statement and a `throw` statement. 
6.9 Comparison of speedup relative to C between the baseline and the enhanced Tascell. 
6.10 Elapsed time between the time when an exception is thrown and the termination of the computation. 
6.11 Elapsed time between the first throw operation and the termination of the program execution, and the number of aborted tasks (16-worker executions). 

7.1 Aborting a redundant search in the parallel search of COPINE. 
7.2 Pseudo-code for the COPINE algorithm with abort operations. 
7.3 Flow of the communication thread for the detection of workers performing a redundant search. 
7.4 Execution time and number of visits to vertices for `Random`(1500, 12) with and without abort operations. 
7.5 Execution time and number of visits to vertices for `Hypercube`(13) with and without abort operations.  
7.6 Execution time and number of visits to vertices for `2D-torus`(500) with and without abort operations. 
7.7 Execution time and number of visits to vertices for `Protein` with and without abort operations.
List of Tables

2.1 All CCIGs of the graph in Figure 2.1 with respect to $\theta = 2$. 7
4.1 Evaluation environment on Appro GreenBlade 8000. 32
4.2 Characteristics of the real protein network. 32
4.3 Results of performance evaluation with the standard task creation strategy (method 0, $\theta = 13$). 33
4.4 Results of performance evaluation with the standard task creation strategy (methods 1–3, $\theta = 5$). 33
4.5 Results of performance evaluation with the setting that leaves one iteration for a victim worker on task creation ($\theta = 5$). 41
5.1 Evaluation environment on Cray XC40. 48
5.2 Results of performance evaluation with multiple computing nodes. 49
5.3 Cumulative processing time of a communication thread. 53
5.4 Performance when changing the values of $\tau$ and $t_{sb}$ (using 8 nodes $\times$ 16 workers). 53
5.5 Results of performance evaluation on the Xeon Phi processor. 54
6.1 Comparison of execution time and relative time to sequential C programs with one worker between the baseline and the enhanced Tascell. 66
7.1 Evaluation environment on Cray XC30. 75
7.2 Characteristics of the artificial graphs. 76
7.3 Results of performance evaluation for the real protein network with and without abort operations. 79
7.4 Performance for Protein ($\theta = 2$) with abort operations when changing the values of $\tau$ and $t_{sb}$ (using 8 nodes $\times$ 16 workers). 80
7.5 Results of performance evaluation for Protein ($\theta = 2$) with abort operations in distributed memory environments. 81
Chapter 1

Introduction

1.1 Background

Backtrack search algorithms are used in many applications. For example, they can be employed in graph mining to analyze the vertices and/or edges of a graph and discover useful knowledge. In a two-person zero-sum game, the best move is generally found by searching a game tree whose nodes are positions in the game and whose edges are moves. The satisfiability problem (SAT) is solved by finding an assignment of variables that makes a given formula true using a backtrack search. With the increasing size of target problems, it is important to speed up the search in order to solve the problems within a practical time. Therefore, various studies have been conducted on backtrack search algorithms.

From the theoretical side, a number of mechanisms such as pruning have been developed to make the search process more efficient. Such methods are almost always required in practical applications to reduce the search space. As computing systems become more powerful, the implementation of backtrack search algorithms is of increasing importance. In particular, computation power has been enhanced by the rapid increase in the number of processor cores in cloud-type general purpose servers and high-performance computing (HPC) systems. Therefore, we need to exploit this computation power by parallelizing backtrack search algorithms, because this tendency is expected to continue into the future. The goal of the work presented in this thesis is to propose a methodology to implement parallel backtrack search algorithms efficiently using pruning techniques.

Because the actual search trees in backtrack search algorithms usually grow dynamically, and thus unpredictably, it is necessary to apply dynamic load balancing to parallelized implementations. Applications having these properties are often implemented by task-parallel languages such as Cilk [1] and Tascell [2], by which we can dynamically spawn tasks to be automatically assigned to workers as parallel threads and/or processes. Thus, each worker has an exclusive set of subtrees as its task set. For a simple backtrack search algorithm in which the search tree can be traversed in an arbitrary order, such an implementation is easy and works efficiently.

However, we cannot straightforwardly parallelize backtrack search algorithms with pruning. The first problem is that the pruning mechanisms employed in many practical backtrack search algorithms assume a sequential search. In a straightforward parallel implementation, we could prune subtree excessively to lead us an incorrect solution. To prevent excessive pruning, we need to design a parallel algorithm to retain the correctness as well as the efficient reduction of the search space.
The second problem is how to achieve efficient pruning in a parallel search. Some pruning mechanisms depend on the knowledge acquired during searches. In a parallel search, workers need to share the acquired knowledge efficiently. In addition, while pruning operations aim to eliminate useless subtrees before they are traversed, a worker may prune the subtree after another worker has started to traverse it. If the latter worker ignores the pruning and continues its traversal, a large number of useless searches will be performed, canceling out the performance gain obtained from parallelization.

There is another problem related to a tendency in recent parallel computing environments. Distributed memory systems, consisting of computing nodes with their own memory space that are connected to a communication network, have become mainstream in recent HPC systems. These memory systems can improve performance by carrying out massively parallel computing with multiple computing nodes, and also have the ability to process large-scale problems. Thus, we also need to implement parallel algorithms so that the parallel implementations are compatible with distributed memory environments.

There have been a number of studies on the parallelization of backtrack search algorithms. For example, Danaher et al. developed the task-parallel language JCilk [3], which has a mechanism for exception handling whereby all parallel tasks running in a try block with an exception are automatically aborted. Though they claim that a mechanism to abort redundant search can be implemented elegantly and efficiently using the capability of exception handling in JCilk, they showed only a limited number of simple programs as the application of the mechanism. In addition, JCilk does not support distributed memory environments. There are other implementations that support the abortion of a redundant search in backtrack search applications for board-game playing [4, 5]. The incorporated abort operations are implemented at the program level at the expense of time and resources.

There has been little previous work on implementing real backtrack search applications that require knowledge sharing for pruning using existing parallel frameworks. Therefore, we need to solve problems related to both performance and productivity for parallel implementations of practical backtrack search algorithms.

1.2 Our Proposal

This thesis discusses our proposed methodology to implement practical parallel backtrack search algorithms with pruning using a task-parallel language. We developed parallelization techniques that achieve higher pruning effectiveness using new work-stealing strategies, aborting speculative searches, and so on. We also simplified the programming complications resulting from the implementation by adding new language mechanisms to a task-parallel language.

We evaluated our methodology in terms of performance and productivity using a practical backtrack search application. As a practical example, we applied our methodology to the COPINE graph mining algorithm, which aims to obtain useful knowledge from biological networks [6, 7]. The COPINE algorithm has not been parallelized, because this is essentially a difficult task. We designed a parallel algorithm as an extension of COPINE, and implemented it using Tascell, which offers high performance for various backtrack search algorithms [2, 8].
1.3 Contributions

This thesis describes the following contributions we have made in our research work.

- We designed a parallel COPINE algorithm. COPINE reduces the search space by pruning branches corresponding to the following subgraphs: those already visited, those having itemsets smaller than the threshold, and those included in already-visited supergraphs with identical itemsets. For the third pruning, we use a table that associates already-visited subgraphs and their itemsets. To prevent excess pruning in a parallel search, we found a restriction to be imposed on workers when referring to table entries registered by other workers. We designed a parallel algorithm as an extension of COPINE by introducing this restriction.

- We implemented the parallel COPINE algorithm in shared and distributed memory environments. One problem concerns how workers can efficiently share the table entries so that they can safely use as many entries as possible registered by other workers. We implemented a sharing method in which all workers share a single table with a mutual execution lock for each table entry. This sharing method is practical in shared memory environments, but not in distributed memory environments, where it would lead to a drastic increase in the cost of internode communication. Therefore, we also implemented a sharing method in which each computing node has its own table and sends its updates to the other nodes at regular intervals.

- We investigated task creation and work-stealing strategies to improve the performance of our parallel solver. The effectiveness of the third pruning depends on the task creation strategy, whereby part of a search tree is divided and assigned to another worker. We implemented a task creation strategy optimized for the pruning. In addition, since the task creation cost for COPINE is relatively high, the conventional work-stealing strategy in Tascell, which aims to minimize the number of internode work-steals, significantly degrades the performance by increasing the number of intranode work-steals for too small tasks. We solved this problem by enabling a worker to request a task from an external node even when the worker’s residence node has some tasks to be stolen. We also employed a work-stealing strategy based on an estimation of the size of the task created by a victim worker from which the task is stolen.

- We improved the parallel COPINE algorithm by adding a mechanism to abort redundant searches. The redundancy described in Section 1.1 can be alleviated by informing workers that a subtree has been pruned, allowing the redundant search to be aborted. This abortion should also be applied to other workers, which can be identified by a bottom-up contraction of the search context to be aborted, if they were assigned some portions of the pruned subtree as their tasks. In addition, the bottom-up contraction may result in the complete destruction of a task for part of the pruned subtree. In this case, workers must notify the abortion to the worker that assigned them with that task. These operations can be easily implemented using task-parallel languages that have exception handling capabilities with “collateral” task abortion, whereby all parallel tasks running in a try block whose catch clause is the target of the throw of an abortion are automatically aborted. We incorporated abort operations in the parallel COPINE algorithm, and implemented them using Tascell, to which we added exception handling features with collateral task abortion.
• We evaluated our parallel implementations of COPINE to find that it successfully accelerates the analysis of a real protein network. We also confirmed the superiority of our proposal in terms of productivity and performance.

1.4 Organization of the Thesis

This thesis is organized as follows.

Chapter 2 introduces the COPINE algorithm. We first define the problem targeted in this thesis, and then describe the sequential COPINE algorithm and our proposal of parallel extension.

Chapter 3 presents the load balancing strategy in Tascell.

Chapter 4 discusses the parallel implementation of COPINE in shared memory environments. We implemented a sharing method in which a single table controlled by locks is shared among workers, and developed a task creation strategy that improves the effectiveness of the pruning.

Chapter 5 is for the parallel implementation of COPINE in distributed memory environments. We implemented a sharing method in which each computing node has its own table and the updates made on the table are sent to the other nodes at regular intervals. We also made an improvement on work-stealing strategies in Tascell.

Chapter 6 describes exception handling features with collateral task abortion in Tascell. We made a language extension to Tascell for the exception handling features, whose implementation is also given in the chapter.

Chapter 7 is devoted to the mechanism to abort redundant searches in COPINE by exploiting the exception handling. We implemented this mechanism in shared and distributed memory environments.

Chapter 8 summarizes related work of the parallelization of backtrack search applications, task-parallel languages, and data mining not using backtrack search.

Finally, Chapter 9 concludes this thesis and describes some ideas for future work.
Chapter 2

COPINE Algorithm

We consider graph mining in which connected subgraphs with common itemsets (Common Itemset connected subGraph, CIG) are extracted from a graph whose vertex is labeled by itemset being a set of items. Here, a common itemset means the intersection of all itemsets associated with the vertices of a connected subgraph. This kind of graph mining is applicable to the acquisition of useful knowledge from a large amount of data in various fields [6, 7], such as identifying groups with common interests for targeted advertising from a graph of a social network in which each vertex represents a user and contains his/her interests. As another example, reactional set pairs of genes and drugs can be identified by enumerating the 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.

As the size of a graph increases, the computation time required for extracting CIGs using a naïve algorithm increases exponentially. The most promising way to solve a large-scale CIG extraction problem in a realistic time is to choose an efficient sequential algorithm and parallelize it for further efficiency. An efficient sequential backtrack search algorithm named COPINE (COmmon Pattern Itemset NEtwork mining) [6, 7] has already been proposed for this problem; however, there is no parallel implementation of COPINE, because it is only intended for sequential searches. This motivated us to design a parallel algorithm as an extension of COPINE, as described in Section 2.4, following the formal definition of the problem (Section 2.1). An outline of the sequential algorithm of COPINE is given in Section 2.2, and proofs of the correctness of the algorithm are presented in Section 2.3.

2.1 Definition of the Closed CIG Enumeration Problem

In this section, we define the Closed CIG (CCIG) enumeration problem. This problem involves a graph in 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 vertices 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 undirected 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. \( E' = \{ (u, v) | u, v \in V' \} \cap E \)
3. \( \forall u, v \in V' : \exists (u_1, v_1), \ldots, (u_n, v_n) \) 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 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 \( \mathcal{I}(v) \subseteq I \) (\( v \in V \)), and a threshold \( \theta \), the CCIG enumeration problem is to extract all connected subgraphs \( G' = (V', E') \) that satisfy the following two conditions.

\[
\begin{align*}
\text{(i)} \quad & \left| \bigcap_{v \in V'} \mathcal{I}(v) \right| \geq \theta \\
\text{(ii)} \quad & \left| \bigcap_{v \in V' \cup \{v'\}} \mathcal{I}(v) \right| < \left| \bigcap_{v \in V'} \mathcal{I}(v) \right| \text{ for any } v' \text{ adjacent to } G', \text{i.e., } v' \in V - V' \text{ such that } \exists v \in V' : (v, v') \in E.
\end{align*}
\]

A connected subgraph \( G' \) that satisfies (i) is named a CIG. A CIG that satisfies (ii) is named a CCIG. A CCIG \( G' \) is said closed with respect to the itemset \( \mathcal{I}(G') = \bigcap_{v \in V'} \mathcal{I}(v) \).

An example input graph associated with itemsets is shown in Figure 2.1. Table 2.1 lists all CCIGs of this graph with respect to \( \theta = 2 \). 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 \( \mathcal{I}(G'') = \{i_1, i_3\} \)) but not (ii) (since \( \mathcal{I}(G_3') = \mathcal{I}(G'') \) and \( V_3' \supseteq V'' \)).

\(^1\)To be precise, this is a connected and induced subgraph because of condition (2). In this thesis, however, we term it a connected subgraph for simplicity.
2.2 Sequential COPINE Algorithm

As shown in Figure 2.2, 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 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 because the number of subgraphs is usually exponential to the number of vertices. Therefore, to reduce the search space, COPINE prunes 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 has a smaller cardinality than the threshold \( \theta \); and

Pruning 3 Subgraphs that are not closed because one of their supergraphs has already been visited and their itemsets are identical.

Pruning 1 is to avoid duplicate enumerations using the gSpan technique [9]. 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 to be 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 total ordering of paths from the root to nodes, which is derived from an arbitrary ordering \( v_1 < v_2 < \cdots < v_{|V|} \) of all the vertices in \( V \). That is, we represent a subgraph \( G' = (V', E(V')) \), where \( V' = \{v'_1, \ldots, v'_k\} \), by the canonical sequence of vertices \( \langle v'_1, \ldots, v'_k \rangle \) which we will define in Section 2.3.1, and traverse subgraphs in the ascending order of these sequences also defined in Section 2.3.1. A search tree for the graph in Figure 2.1 to which this pruning is applied is shown in Figure 2.2. 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 \( V' = \{v_1, v_3, v_4\} \), whose canonical sequence is \( \langle v_1, v_3, v_4 \rangle \), in Figure 2.2 is traversed as the successor of \( \langle v_1, v_3 \rangle \), it is not revisited as the successor of \( \langle v_1, v_4 \rangle \) because its precedence over \( \langle v_1, v_4 \rangle \) in the total order defined in Section 2.3.1 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 for which the corresponding subgraph has a common itemset smaller than the threshold \( \theta \), 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 \( \theta \). The nodes represented by dashed frames in Figure 2.2 are eliminated by this second pruning.

Focusing on the two subtrees surrounded by red frames in Figure 2.2, 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 it is not necessary to visit descendants of \( n_2 \), because the subgraphs they represent have supergraphs represented by the subtree on the left, in 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 duplicate search by introducing the third type of pruning (Pruning 3) as follows.

We prepare an itemset table in 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 Figure 2.2 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, as 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.

### 2.3 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 demonstrate the correctness of the sequential search using the pruning methods explained in Section 2.2, 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, imposing a restriction on workers 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 CIGs not closed and thus not CCIGs. We discuss the soundness in more detail in Section 2.3.4.

#### 2.3.1 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 named 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 \(U = \{u_1, \cdots, u_n\}\) 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. \(1 < \forall j \leq n : 1 \leq \exists i < j \text{ such that } (u_i, u_j) \in E(U)\)

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 \(u_i = \min[U^\downarrow(N(U_{i-1}) - U_{i-1})]\) for all \(i\) such that \(1 < i \leq n\), where \(U_j\), \(N(U_j)\), and \(U^\downarrow\) are defined as follows.

1. \(U_j = \{u_1, \cdots, u_j\}\)
2. \(N(U_j) = \bigcup_{k=1}^{j} \{u \mid (u_k, u) \in E(U)\}\)

\(^2\)Even though this removal is not necessary for the correctness of the algorithm, it prevents the itemset table from becoming unnecessarily large.
1: **function** $CST(U)$ **begin**
2: $v \leftarrow \text{min}[\prec](U)$; $U \leftarrow U - \{v\}$; $T \leftarrow \langle v \rangle$; $E \leftarrow E(U)$;
3: $C \leftarrow \text{sort}(\text{neighbors}(v, E))$;
4: **while** $C \neq \langle \rangle$ **do** begin
5: $v \leftarrow \text{car}(C)$; $C \leftarrow \text{cdr}(C)$;
6: **if** $v \notin U$ **then continue**;
7: $T \leftarrow T \cdot \langle v \rangle$; $U \leftarrow U - \{v\}$; $N \leftarrow \text{sort}(\text{neighbors}(v, E))$;
8: $C \leftarrow N \cdot C$;
9: **end**
10: **return**($T$);
11: **end**

Figure 2.3: Algorithm to form the CST of a connected subgraph.

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

$u \leftarrow v \iff u, v \in N(U_j) - U_j \land$

$(\text{prev}(u, U_j) > \text{prev}(v, U_j)) \lor (\text{prev}(u, U_j) = \text{prev}(v, U_j) \land u < v))$

More intuitively, the CST of $U$ is obtained by a depth-first traversal of $U$ starting from the smallest vertex of $U$. In each descending of the traversal, we choose the most recently visited vertex having non-visited neighbors and then add its smallest non-visited neighbor to the tree, as shown in Figure 2.3 in which we use the operator ‘·’ to concatenate two sequences, the notation ‘⟨⟩’ for an empty sequence, and the following functions.

- **sort($s$)** to sort a sequence $s$ of vertices according to the ascending order defined by ‘⟨⟩’.
- **neighbors($v, E$)** to have all vertices of a vertex $v$ with respect to $E$.
- **car($s$)** to have the first element of a sequence $s$.
- **cdr($s$)** to have the sequence of the second and succeeding elements of $s$.

Note that, for $T(U) = \langle u_1, \ldots, u_n \rangle$, its ancestor $\langle u_1, \ldots, u_m \rangle$ ($m < n$) is canonical by definition. However, it is not always true that there exists some $u \in V$ such that $T(U) \cdot \langle u \rangle$, i.e., a direct descendant of $T(U)$, is canonical. That is, it may be the case that a CST $T(U)$ cannot have any direct descendants when any neighboring vertices of $U$ cannot be added to the tail of $T(U)$ conforming the canonicity.

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

**Definition 5 (Total Order of Subgraphs)** For a pair of 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 $T(U^1) = \langle u^1_1, \ldots, u^1_m \rangle$ and $T(U^2) = \langle u^2_1, \ldots, u^2_n \rangle$:

$$k = m < n \lor (k < \min(m, n) \land u^1_k \leftarrow u^2_k$$

where $k = \arg \max \{u^1_i | \forall j \leq i : u^1_j = u^2_j\} = \lceil \text{prefix}(T(U^1), T(U^2)) \rceil$, $U^1_k = \{u^1_1, \ldots, u^1_k\} = \{u^2_1, \ldots, u^2_k\}$, and $u \leftarrow v$ is defined as follows for a CST $T(U) = \langle u_1, \ldots, u_i \rangle$.
(1) \( N^*(U_i) = \begin{cases} V & i = 0 \\ \bigcup_{j=1}^{i} \{u \mid (u_j, u) \in E\} & i > 0 \end{cases} \)

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

\[ u \xleftarrow{\text{prev}^*} v \iff \exists u, v \in N^*(U_i) - U_i \land \ (\text{prev}^*(u, U_i) > \text{prev}^*(v, U_i) \lor (\text{prev}^*(u, U_i) = \text{prev}^*(v, U_i) \land u < v)) \]

More intuitively, \( G^1 < G^2 \) if \( T(U^1) \) is a prefix of \( T(U^2) \) \((k = m < n)\), or the vertex \( u_{k+1}^1 \) has the precedence over \( u_{k+1}^2 \) in the CST of \( U_k^{1,2} = U_k^1 \cup \{u_{k+1}^1, u_{k+1}^2\} \), i.e., \( T(U_k^{1,2}) = T(U_k^1) \cdot \langle u_{k+1}^1, u_{k+1}^2 \rangle \) \((k < m, n)\). The latter condition means that the largest index of \( u_{k+1}^1 \)'s neighbors in \( T(U_k^1) \) \((\text{i.e., } \text{prev}^*(u_{k+1}^1, U_k^1))\) is larger than that of \( u_{k+1}^2 \)'s \((\text{i.e., } \text{prev}^*(u_{k+1}^2, U_k^2))\), or they are equal and \( u_{k+1}^1 < u_{k+1}^2 \). In other words, \( u_{k+1}^1 \) has the depth-first \((\text{prev}^*(u_{k+1}^1, U_k^{1,2}) > \text{prev}^*(u_{k+1}^2, U_k^{1,2}))\) or the smallest-first \((\text{prev}^*(u_{k+1}^1, U_k^{1,2}) = \text{prev}^*(u_{k+1}^2, U_k^{1,2}) \land u_{k+1}^1 < u_{k+1}^2)\) precedence over \( u_{k+1}^2 \) in the traversal of the CST.

**Theorem 1 (Direct Successor of a Subgraph)** Let \( \{G_1, \cdots, G_N\} \) be the set of all connected subgraphs of \( G \) such that \( G_1 < \cdots < G_N \). 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:

\[ T(V_{i+1}) = \langle u_1, \cdots, u_k \rangle \cdot \left\lfloor \min \left[ \frac{U_i}{k} \right] \right\rfloor (C_k) \]

where \( k \) and \( C_k \) are defined as follows with \( U_i = \{u_1, \cdots, u_i\} \) \((1 \leq i \leq n)\).

1. \( P_i = \{v \mid u \xleftarrow{\text{prev}^*} u_i\} \cup \{u_i\} \)
2. \( D_i = \bigcup_{j=1}^{i} P_j \)
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.

Intuitively, \( G_{i+1} = (V_{i+1}, E(V_{i+1})) \) is obtained by a depth-first traversal step from \( G_i = (V_i, E(V_i)) \) by at first choosing the most recently visited vertex \( u_k \) in \( V_i \), where \( U_k = \{u_1, \cdots, u_k\} \) has non-visited neighbors which can be added to the tail of \( T(U_k) \) preserving the canonicity. Such \( k \) is found as the largest index such that \( C_k \neq \emptyset \) where \( C_i \) has the set of vertices able to be added to \( U_i \). That is, \( C_i \) is the set of neighbors of \( U_i \) excluding those in \( D_i \) which we have already visited at or before we had \( T(U_j) \) such that \( 1 \leq j \leq i \), i.e., the union of \( P_j \) for all \( j \). Then, we choose \( u = \min[C_k] \) so that \( T(U_k) \cdot u \) is the minimum among all candidates \( T(U_k) \cdot u' \)
1: function EnumSG(G) begin
2:   (V, E) ← G;  L ← ∅;
3:   while V ≠ ∅ do begin
4:     v = min(V);  V ← V − {v};
5:     L ← ExploreSG(v, ⟨v⟩, V, E, ⟨⟩, L);
6:   end
7:   return(L);
8: end

9: function ExploreSG(v, T, V, E, C, L) begin
10:   L ← L ∪ {T};
11:   N ← sort(neighbors(v, E));
12:   C ← N · C;
13:   while C ≠ ⟨⟩ do begin
14:     c ← car(C);  C ← cdr(C);
15:     if c ∉ V then continue;
16:     V ← V − {c};
17:     L ← ExploreSG(c, T · ⟨c⟩, V, E, C, L);
18:   end
19:   return(L);
20: end

Figure 2.4: Algorithm to enumerate connected subgraphs.

where \( u' \in C_k \). The proof shown in Appendix A.1 is to prove \( T(V_{l+1}) = T(U_k) \cdot u \) is canonical, \( T(V_l) < T(V_{l+1}) \), and any sequence \( t \) such that \( T(V_l) < t < T(V_{l+1}) \) is not canonical.

The algorithm shown in Figure 2.4 implements the traversal step shown above in the recursive function \( ExploreSG() \). That is, \( C_k \) is found as the leading portion of the intersection of the depth-first neighbor list \( C \) and the set \( V \), the latter of which has all vertices able to be added to \( T(U_k) \) if they are the neighbor of \( U_k \). Therefore, the function \( EnumSG() \) enumerates all connected subgraphs, in the ascending order of their CSTs and thus without any duplications, letting \( L \) have them in the form of CST. 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 already been visited.

2.3.2 Correctness of Pruning 2

The correctness of Pruning 2 is proved by Theorem 2.

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, all of the following conditions hold.

1. \( \mathcal{S}(G^1) \supseteq \mathcal{S}(G^2) \)
2. \( |\mathcal{S}(G^1)| \geq |\mathcal{S}(G^2)| \)
3. \( |\mathcal{S}(G^1)| < \theta \rightarrow |\mathcal{S}(G^2)| < \theta \)
Proof:

(1) It is obvious that $U^1 \subset U^2$. Therefore, $\mathcal{I}(G^1) = \bigcap_{v \in U^1} \mathcal{I}(v) \supseteq \bigcap_{v \in U^2} \mathcal{I}(v) = \mathcal{I}(G^2)$. \hfill □

(2) Trivial by (1). \hfill □

(3) Trivial by (2). \hfill □

According to Theorem 2, when we encounter a search tree node for a subgraph whose common itemset size is less than the threshold $\theta$, we can be sure that there is no subgraph whose common itemset size is not less than $\theta$ among the descendants of the node. Therefore, Pruning 2 is correct.

2.3.3 Correctness and Application to Parallel Search of Pruning 3

Theorem 3 (Order of CIGs and their Closeness) Let $G^1 = (U^1, E(U^1))$ and $G^2 = (U^2, E(U^2))$ be connected subgraphs of a graph $G = (V, E)$ such that $G^1 \prec G^2$. If the tails of $T(U^1) = \langle u_1, \ldots, u_n \rangle$ and $T(U^2) = \langle u_1', \ldots, u_m \rangle$ are common, i.e., $tail(U^1) = u_1^1 = tail(U^2) = u_1^2$, and $\mathcal{I}(G^1) \supseteq \mathcal{I}(G^2)$, all of the following criteria hold.

(1) $G^2$ is not closed with respect to $\mathcal{I}(G^2)$.

(2) Any descendant $G' = (U', E(U'))$ of $G^2$ such that $T(U') = T(U^2) \cdot T'$ is not closed with respect to $\mathcal{I}(G')$.

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 Figure 2.2 and let $G^1$ and $G^2$ be the subgraphs corresponding to $n_1$ and $n_2$, respectively. We can see that $n_2$ is visited later than $n_1$ ($G^1 \prec G^2$), their tails are the same ($tail(U^1) = tail(U^2)$), and the common itemset of $G^2$ is a subset of the common itemset of $G^1$ ($\mathcal{I}(G^1) \supseteq \mathcal{I}(G^2)$). Note that the third condition is satisfied because the table entry corresponding to the vertex $u_1^1 = u_1^2$ has $\mathcal{I}(G^1)$ 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 directly apply Pruning 3 to a parallel search. The prerequisite of Theorem 3 means that a worker can only refer to itemsets for Pruning 3 that 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 Figure 2.5 we show an example in which the soundness is lost in the absence of this restriction. We assume that worker $w_0$, traversing the left subtree, and worker $w_1$, traversing the right subtree, have visited $\langle v_1, v_2 \rangle$ and $\langle v_3, v_4, v_5 \rangle$, respectively. At this time, the itemset table of $w_1$ is shown on the right of Figure 2.5. After that, $w_0$ will visit $\langle v_1, v_3 \rangle$. If $w_0$ refers to $\{i_1, i_2, i_4\}$ in the itemset table of $w_1$, $w_0$ will skip the search from $\langle v_1, v_3 \rangle$ to the leaf nodes, because $w_0$ 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 $\theta = 2$. 

13
implementations with itemset table sharing. The first traversal. The time is very short for graphs for which Pruning 3 works reasonably well as in our parallel set of CCIGs by traversing the search tree again by referring to the “perfect” itemset table, which is obtained by satisfying condition (ii) of Definition 2 after the search is not actually true. To obtain a sound set of CCIGs, we need to eliminate those CIGs that do not the parallel version of the algorithm in Figure 2.6 will assert a CIG to be closed even though this sequential search. Therefore, it is virtually impossible to perform Pruning 3 perfectly, and thus that have been left unvisited by other workers, even though they should have been visited in the parallel search algorithm in Figure 2.6, we can completely enumerate all CCIGs of a graph whose common itemset size is not less than \( \theta \). In \( \text{ExploreCCIG}() \), \( T = T(U) \) is the sequence of vertices corresponding to the connected subgraph that the function is visiting, \( C \) is the sequence of vertices neighboring to \( U \), and \( V \) is the set of all vertices which can be added to \( T \) if they are the neighbor of \( U \). In this algorithm, Pruning 1 corresponds to the fact that subgraphs are scanned maintaining \( C \) and \( V \) as \( \text{EnumSG}() \) and \( \text{ExploreSG}() \) in Figure 2.4 do. Pruning 2 is performed by detecting the inferiority of the itemset cardinality with respect to the threshold \( \theta \) at lines 6 and 22. Pruning 3 is executed by adding an itemset to the itemset table entry (lines 8 and 24) and detecting an inclusive relation between itemsets (lines 7 and 23).

In a parallel search, a search tree node visited by a worker may have some precedent nodes that have been 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 Figure 2.6 will assert a CIG to be closed even though this is not actually true. To obtain a sound set of CCIGs, we need to eliminate those CIGs that do not satisfy condition (ii) of Definition 2 after the search.\(^1\)

\(^1\)The time required for this elimination is not considered in the performance evaluations. We can obtain a sound set of CCIGs by traversing the search tree again by referring to the “perfect” itemset table, which is obtained by the first traversal. The time is very short for graphs for which Pruning 3 works reasonably well as in our parallel implementations with itemset table sharing.

---

**Figure 2.5: Example of excessive pruning (\( \theta = 2 \)).**

### 2.3.4 Soundness of Search

**Theorem 4 (CCIG)** A connected subgraph \( G' = (U, E(U)) \) of a graph \( G = (V, E) \) is closed with respect to \( \mathcal{I}(G') \) iff all of the following conditions hold.

1. For any direct descendant \( G'^d = (U'^d, E(U'^d)) \) of \( G' \), \( \mathcal{I}(G') \neq \mathcal{I}(G'^d) \).

2. For any connected subgraph \( G'' = (U'', E(U'')) \) of \( G \) such that \( G'' \prec G' \) and \( \text{tail}(U'') = \text{tail}(U) \), \( \mathcal{I}(G'') \not\supseteq \mathcal{I}(G') \).

**Proof:** See Appendix A.3.
function EnumCCIG(G) begin
(V, E) ← G; L ← ∅;
for ∀v ∈ V do v.I ← ∅;
while V ≠ ∅ do begin
v = min(V); V ← V − {v};
if |∂(v)| < θ then continue;
if ∃I′ s.t. I′ ∈ v.I ∧ I′ ⊇ ∂(v) then continue;
v.I ← v.I ∪ ∂(v);
L ← ExploreCCIG(v, ⟨v⟩, ∂(v), V, E, ⟨⟩, L);
end
return(L);
end

function ExploreCCIG(v, T, I0, V, E, C, L) begin
closed ← true;
N ← sort(neighbors(v, E));
C ← N · C;
while C ≠ ⟨⟩ do begin
c ← car(C); C ← cdr(C);
if c ∉ V then continue;
V ← V − {c}; Ic ← I0 ∩ ∂(c);
if |Ic| < θ then continue;
if ∃I′ s.t. I′ ∈ c.I ∧ I′ ⊇ Ic then continue;
c.I ← c.I ∪ {Ic};
if Ic = I0 then closed ← false;
L ← ExploreCCIG(c, T · ⟨c⟩, Ic, V, E, C, L);
end
if closed then L ← L ∪ {T};
return(L);
end

Figure 2.6: Sequential algorithm to enumerate all CCIGs of a graph.

2.4 Parallel COPINE Algorithm

We parallelize the algorithm in Figure 2.6 by dividing the search tree and assigning a unique set of subtrees to each worker. This can be implemented by dividing the two while loops (lines 4–10 and 18–27) into some appropriate portions 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 the restriction described in Section 2.3.3 on Pruning 3. In Figure 2.6, a worker refers to c.I (a set of itemsets registered when the vertex c 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. A worker needs to check whether an element, i.e., an itemset, in the table had been registered to the table earlier if the search were conducted sequentially rather than parallel. This verification by a worker
requires each itemset to have some sequential ordering information concerning 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. These issues, together with the verification of the safeness of itemset table references, are discussed in Chapters 4 and 5 for the implementations in shared and distributed memory environments, respectively.

2.5 Conclusion

In this chapter, we extended the sequential COPINE algorithm to the parallel version. COPINE employs the pruning mechanism, for which the itemset table is used to record visited subgraphs and their itemsets. However, its straightforward parallelization causes excessive pruning, resulting in insufficient subgraph extraction. 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 order. We designed the parallel COPINE algorithm by introducing this restriction.
Chapter 3

Task-Parallel Language Tascell

We parallelize the COPINE algorithm by dividing a search tree growing in the COPINE execution process and assigning a set of subtrees to each worker running in parallel. One of the most important issues in a parallel search is to balance the load among workers. However, before the search, we cannot evaluate the size of any subtree whose root is a search tree node. Therefore, it is too difficult to assign subtrees to workers in advance so as to balance the load. 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 assigns them to the workers would cause a significant load imbalance as the sizes of the subtrees diverge.

In this research, we balance the workloads among workers using the dynamic load balancing mechanism of “work-stealing,” whereby an idle worker steals 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 task-parallel languages, such as Cilk [1] and Tascell [2], drastically eases the implementation difficulty. In most task-parallel 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) 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 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 Figure 2.6.

Among the several task-parallel languages, we chose Tascell as the baseline language for our implementation, because it is known that Tascell achieves particularly good performance for the implementation of parallel backtrack search algorithms. In the remainder of this chapter, we describe the mechanism of dynamic load balancing in Tascell.

3.1 Overview

The Tascell framework consists of a compiler for an extended C language, called the Tascell language\(^1\), and a runtime system for parallel computation.

Figure 3.1 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 has a TCP/IP connection with a relay server named the Tascell server.

---

\(^1\)The actual Tascell language has an S-expression-based syntax [10], but we write programs with a C-like syntax here for readers’ convenience.
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 the workload of computing nodes. As shown in Figure 3.1, a Tascell server can be connected to another Tascell server. Thus, computing nodes and Tascell servers generally form a tree, although performance evaluations with more than one Tascell server were not considered in this study.

### 3.2 Dynamic Load Balancing

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 3.2 illustrates how a task is spawned lazily, that is, only after a worker receives a task request. Suppose that \( n_v \) is the oldest task-spawnable point passed by worker \( w_p \). When \( w_v \) receives a task request from worker \( w_p \),
(1) it backtracks to $n_s$, performing undo operations to restore the state of its workspace at $n_s$, and then

(2) spawns a task to traverse some of right subtrees.

For example, $w_v$ creates part of the unexecuted iterations of a parallel loop at $n_v$ as a task. After sending the task to $w_t$,

(3) $w_v$ returns from the backtracking, using 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. The structure of this object is defined in a Tascell program by the user. It can be transferred by passing the pointer in shared memory environments, or by serialization via Tascell servers in distributed memory environments.

In contrast to Tascell, there is another task creation technique called Lazy Task Creation (LTC) [11], which is employed in “multithreaded languages” such as Cilk. In LTC, parallel execution units called “logical threads”\(^2\) 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 the logical threads that have already been created, the cost of a stealing operation in LTC is smaller than that of Tascell, which involves backtracking. On the other hand, the overhead of sequential execution in LTC in the absence of task requests is larger than that in Tascell, since logical threads that may not be assigned to another worker can be created as a task at the 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 [2]. As shown later in the performance evaluations, while the number of potential tasks (the number of visits to vertices during the search) ranges from hundreds of millions to tens of billions, the number of created tasks ranges from thousands to tens of thousands at most. Therefore, it is clear that Tascell has an advantage over LTC.

\(^2\)So called since they are executed under the management of a language system and differ from “physical threads,” which are executed under OS management and correspond to a worker assigned with the actual execution of tasks.
3.3 Work-Stealing Strategy

3.3.1 Task Request

A thief worker uses the following strategy to choose which victim worker 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 receives a task as a response to the request in (1), it executes the task. If it receives 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 receives 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. On receiving a task request, the Tascell server randomly chooses a computing node from those connected to the server, excluding the request sender, and forwards the request to it.

5. The computing node that receives this task request from the Tascell server checks the 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 any external computing nodes, it returns to (1) and retries the task-stealing process after halting for a short period.

3.3.2 Stealing Back

When a Tascell worker $w_1$ cannot process its running task without receiving the result of a task $t$ that has been sent to another worker $w_2$ as part of the running task, the worker $w_1$ tries to steal another task as a thief rather than becoming idle waiting for the result. In this case, the victim of the request to steal is not chosen using the strategy explained in Section 3.3.1, but the thief $w_1$ steals back a task from the worker $w_2$ to which the task $t$ causing the synchronization delay has been assigned. Using this technique, called Leapfrogging [12], we can guarantee that the maximum size of the workers’ execution stacks is at most a constant times as large as the stack-size watermark in a sequential execution.

3.4 Tascell Language

We explain the Tascell language using a backtrack search algorithm for finding all possible solutions to the Pentomino puzzle. A pentomino consists of five squares attached edge-to-edge. The Pentomino puzzle involves filling the $6 \times 10$ rectangular board with twelve pentominos. Figure 3.3 shows a sequential C program for this problem.
```c
int a[12]; // manage unused pieces
int b[70]; // the board, with (6+sentinel) x 10 cells
// Try from the j0-th piece to the 12th piece in a[].
// The i-th piece for i<j0 is already used.
// b[k] is the first empty cell in the board.
int search(int k, int j0)
{
    int s=0; // the number of solutions
    for(int p=j0; p<12; p++) { // iterate through unused pieces
        int ap=a[p];
        for(each possible direction d of the piece) {
            ... local variable definitions here ...
            if(Can the ap-th piece in the d-th direction be placed on the board b?);
            else continue;
            Set the ap-th piece onto the board b and update a.
            kk = the next empty cell;
            if(no empty cell?) s++; // a solution found
            else s += search(kk, j0+1); // try the next piece
            Backtrack, i.e., remove the ap-th piece from b and restore a.
        }
    }
    return s;
}
```

Figure 3.3: Program (pseudo-code written in C) that performs backtrack search for finding all possible solutions to the Pentomino puzzle.

Figure 3.4 illustrates a parallelized Tascell program for Pentomino based on the C code in Figure 3.3. We can write a worker program with the following constructs in Tascell, starting with an existing sequential program.

The top-level task declaration `task pentomino { · · · };` defines the structure of a task object named `pentomino`. Several fields with `in:` attribute are declared as the search input. The field `s` is declared for storing the result. A Tascell worker that receives a `pentomino` task executes `pentomino`'s `task_exec` body. In the `task_exec` body, the Tascell worker can refer to the received task object by the keyword `this`. In addition, we can call `worker` functions in the body.

A function that uses Tascell’s parallel constructs must be attributed by the keyword `worker`. The parallelized part of the `search` function employs Tascell’s task division constructs. A parallel `for` loop construct can be used to divide an iterative computation. This is syntactically denoted by:

```c
for(int identifier : expr_from, expr_to) statement_body
    handles task-name(int identifier_from, int identifier_to)
    { statement_put statement_get }.
```

This iterates `statement_body` over integers from `expr_from` (inclusive) to `expr_to` (exclusive). A worker performs iterations for a parallel `for` loop sequentially, unless it detects any task requests.
task pentomino {
    out: int s; // output
    in: int k, i0, i1, i2;
    in: int a[12]; // manage unused pieces
    in: int b[70]; // the board, with (6+sentinel) × 10 cells
};
task_exec pentomino {
    this.s = search(this.k, this.i0, this.i1, this.i2, &this);
}
worker int search(int k, int j0, int j1, int j2, task pentomino *tsk)
{
    int s=0; // the number of solutions
    // parallel for construct in Tascell
    for(int p : j1, j2) {
        int ap=tsk->a[p];
        for(each possible direction d of the piece) {
            ... local variable definitions here ...
            if(Can the ap-th piece in the d-th direction be placed on the board tsk->b?);
            else continue;
            dynamic_wind // construct for specifying undo/redo operations
            { // do/redo operation for dynamic_wind
                Set the ap-th piece onto the board tsk->b and update tsk->a.
            }
            { // body for dynamic_wind
                kk = the next empty cell;
                if(no empty cell?) s++; // a solution found
                else // try the next piece
                    s += search(kk, j0+1, j0+1, 12, tsk);
            }
            { // undo operation for dynamic_wind
                Backtrack, i.e., remove the ap-th piece from tsk->b and restore tsk->a.
            }
            } // end of dynamic_wind
    }
} //end of parallel for
handles pentomino(int i1, int i2)
// declaration of this and setting a range (i1-i2) is done implicitly
{
    // put part (performed before sending a task)
    { // put task inputs for upper half iterations
        copy_piece_info(this.a, tsk->a);
        copy_board(this.b, tsk->b);
        this.k=k; this.i0=j0; this.i1=i1; this.i2=i2;
    }
    // get part (performed after receiving the result)
    { s += this.s; }
} // end of parallel for
return s;

Figure 3.4: A Tascell program that performs a backtrack search for the Pentomino puzzle.
When the implicit task-request handler (available during the iterative execution of \texttt{statement}_{\text{body}}) is invoked, the upper part of the remaining iterations are spawned as a new task, the object of which is initialized by \texttt{statement}_{\text{put}}. In \texttt{statement}_{\text{put}}, the actual assigned range can be referred to by \texttt{identifier}_{\text{from}} and \texttt{identifier}_{\text{to}}. When all the remaining iterations are assigned to other workers as tasks, the worker waits for the results of these tasks, and then handles (merges) the results of the tasks by executing \texttt{statement}_{\text{get}}. As described in Section 3.3.2, the waiting worker steals back a task from one of the workers to which tasks for parts of this parallel \texttt{for} loop are assigned while waiting for the results.

Parallel \texttt{for} statements may be nested dynamically in their \texttt{statement}_{\text{body}}. Therefore, multiple task-request handlers may be available at the same time. Each worker attempts to detect a task request by polling every parallel \texttt{for} statement without heavy memory barrier (fence) instructions. When the worker detects a task request, it temporarily backtracks to spawn a larger task by invoking the oldest handler possible.

Tascell has a \texttt{dynamic\_wind} construct, as in the Scheme language [13], for specifying application-dependent undo/redo operations, e.g., removing/laying pieces in Pentomino. This is syntactically denoted by:

\begin{verbatim}
\texttt{dynamic\_wind statement\_before statement\_body statement\_after}.
\end{verbatim}

The Tascell worker basically executes \texttt{statement\_before} (“set a piece” in Figure 3.4 as “do”), \texttt{statement\_body}, and \texttt{statement\_after} (“remove the piece” in Figure 3.4 as “undo”) in this order. However, during the execution of \texttt{statement\_body}, \texttt{statement\_after} is also executed as an “undo” clause \texttt{before} an attempt to invoke an older task request handler. \texttt{statement\_before} is also executed as a “redo” clause \texttt{after} the attempt.

### 3.5 Implementation

The Tascell compiler is implemented as a translator of the C language in order to render the implementation portable. It is difficult to realize the temporary backtracking mechanism in “standard” C, because a stack walk operation is required, whereby the values of variables located below the current frame in the execution stack must be accessed. The implementation of Tascell exploits nested functions [14] to realize a stack walk.

#### 3.5.1 Nested Functions

A nested function is a function defined inside another function in locations where variable definitions are allowed, except at the top level. Its evaluation creates a lexical closure accompanying the creation-time environment, and indirect calls to a nested function provide legitimate stack access. Figure 3.5 shows an example of a program with nested functions.

When the function \texttt{bk\_exit1()} nested in \texttt{fib()} is (indirectly) called, the parameters \texttt{bk\_exit0} and \texttt{n}, and a local variable \texttt{s} located in the (older) frame, can be accessed. In addition, a nested function can jump to a label inherited from a containing function, provided the label is explicitly declared in the containing function. Such a jump returns instantly to the containing function, exiting the nested function that performed the \texttt{goto} as well as any intermediate functions. In the program in Figure 3.5, when \texttt{bk\_exit1()} is called with the argument \texttt{n0 = n}, \texttt{s} is set to \texttt{v} and the control returns to \texttt{fib()}, exiting \texttt{bk\_exit1()} and all the intermediate functions between \texttt{bk\_exit1()} and \texttt{fib()}. This capability is not used in the baseline Tas-
int fib(void (*bk_exit0)(int, int), int n)
{
    __label__ l_exit; // label should be declared explicitly
    int s = 0;
    /* nested function */
    void bk_exit1(int n0, int v) {
        if(n == n0) {
            s = v;
            goto l_exit; // jump to l_exit in fib()
        }
        bk_exit0(n0, v); // call caller’s nested function
        return;
    }
    if(found that fib(n0) = v) bk_exit1(n0, v);
    if(n <= 2) return 1;
    {
        int s1 = 0, s2 = 0;
        s1 = fib(bk_exit1, n - 1);
        s2 = fib(bk_exit1, n - 2);
        s = s1 + s2;
    }
    l_exit:
    return s;
}

Figure 3.5: Program with nested functions.

The best-known implementation of nested functions for C is the trampoline-based implementation in GCC [15, 16]. In addition, L-closure-based implementations of nested functions have been proposed to achieve low maintenance/creation costs by delaying the initialization of the closure until it is invoked and enabling register allocation. Two L-closure implementations exist: a translator of standard C, called LW-SC [17, 18], and an enhancement of GCC, called XC-cube [19]. However, XC-cube does not support goto commands that exit a nested function.

3.5.2 Translation to C with Nested Functions

Using nested functions, the program in Figure 3.4 can be translated into the program in Figure 3.6. Each worker function is translated to have an additional parameter _bk0 holding a nested function pointer corresponding to the newest handler for a parallel for or dynamic_wind statement. Each parallel for statement is translated into a piece of code that includes a definition of a nested function (_bk1_par_for in Figure 3.6) as the newest handler, which is called when a task request is detected by polling. The nested function first tries to spawn a larger task by calling a nested function (_bk0) corresponding to the second-newest handler (which calls another
int search(void (*_bk0)(void), struct thread_data *_thr, int k, int j0, int j1, int j2, struct pentomino *tsk) {
    int s = 0; // the number of solutions
    /*------------------ parallel for ------------------*/
    int p = j1; int p_end = j2;
    struct pentomino *pthis;
    int spawned = 0; // the number of spawned tasks

    void _bk1_par_for(void) { // nested function
        if(!spawned) _bk0(); // continue backtracking
        while(p + 1 < p_end &&
            task request exists?) {
            int i1 = (1 + p + p_end)/2, i2 = p_end; // the range for the sub-task
            p_end = i1; // shrink the range for itself
            pthis = malloc(sizeof(struct pentomino)); // allocate a workspace
            copy_piece_info(pthis->a, tsk->a);
            copy_board(pthis->b, tsk->b);
            pthis->k = k; pthis->i0 = j0;
            pthis->i1 = i1; pthis->i2 = i2;
        }
        spawned++;
        make_and_send_task(_thr, 0, pthis); // spawn
    }
    if(_thr->req) // check task requests
        _bk1_par_for(); // start backtracking for spawning tasks
    for(; p < p_end; p++) {
        int ap = tsk->a[p];
        for(each possible direction d of the piece) {
            // examine the "i-th" (piece, direction)
            ... local variable definitions here ...
            if(Can the ap-th piece in the d-th direction be placed on the board tsk->b?){
                else continue;
            }
        }
    }
    while(spawned-- > 0) {
        // get and integrate results of spawned tasks
        pthis = (struct pentomino *)wait_rslt(_thr);
        s += pthis->s; // statement
        free(pthis); }
    }
    return s;
}

Figure 3.6: Translation result from the worker function search for Pentomino in Figure 3.4, including translation of a parallel for statement and a dynamic_wind statement.
nested function for the third-newest handler, and so on). If there is a remaining task request, the worker calculates a range for a new task, updates a range for itself, and creates a new task to send to the requester. After sending the task, the worker returns from the nested function and resumes its own computation.

The translation for a `dynamic_wind` statement is also included in Figure 3.6. To perform the undo/redo operations described in Section 3.4, the `statement_body` employs a nested function (`_bk2_dwind` in Figure 3.6), which is composed of (a copy of) `statement_after` (as undo operations), a call to the second newest nested function, and (a copy of) `statement_before` (as redo operations).
Chapter 4

Parallel Implementation in Shared Memory Environments

4.1 Introduction

This chapter introduces implementations of the parallel COPINE algorithm using Tascell in shared memory environments. As stated in Section 2.4, Prunings 1 and 2 can be directly applied to the parallel search. However, to achieve Pruning 3 efficiently, workers need to share itemset table entries under the restriction described in Section 2.3.3. This sharing requires some consideration of a trade-off between the opportunity to share useful information among workers and the cost of doing so safely. In Section 4.2, we show four methods to share the information in the table with their own trade-off points, including one which we devised 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 tree. Then in Section 4.3, we discuss the advantage of Tascell over other LTC-based languages based on the implementation issues of the parallel COPINE including the methods of the itemset table sharing.

The effectiveness of Pruning 3 is also deeply affected by the task creation strategy of determining which portion of a subtrees, or, in other words, unscanned iterations of a parallel loop, is chosen for a task to be created. That is, while it is reasonable to split the subtree by half to give one portion to the thief and to keep the other for the victim from the viewpoint of load balancing, reducing the portion for the victim would improve the effectiveness of Pruning 3 because it gives the thief better chance to refer to up-to-date pruning results by the victim. Section 4.4 discusses this issue more deeply with several strategies of task creation, whose performance evaluations are presented in Section 4.5 with the variation of itemset table sharing methods.

4.2 Itemset Table Sharing

Considering the restriction described in Section 2.3.3 and the trade-off between the effectiveness of Pruning 3 and the cost for the effective pruning, we developed the following four methods of sharing the itemset table to examine the performance effect of them.
4.2.1 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 the search space remains huge. We implemented this method 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 graphs for which Pruning 3 does not work.

4.2.2 Sharing Method 1: Non-Sharing Method

Each task maintains 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. Because 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 other tasks executed by the worker itself).

4.2.3 Sharing Method 2: Replicating Method

As in the non-sharing method, each task maintains 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 can safely refer to itemsets in the table preserving the completeness of the search.

In addition to the costs of the non-sharing method, this method has the overhead of making a copy of the itemset table on each steal.

4.2.4 Sharing Method 3: Fully-Sharing Method

All workers share a single itemset table with a mutual exclusion lock for each table entry. We satisfy the restriction described in Section 2.3.3 by associating a task ID with each task and adding 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 only use an itemset in the table if the task ID of the itemset is not greater than that of the task being executed.

Although this method has some overheads associated with the locks and is difficult to implement in distributed memory environments, a worker can use itemsets registered by other workers immediately.

Since a task consists of a set of subtrees whose roots are children of a search tree node, a task can be identified by the leftmost subtree root. Therefore, by giving unique ID to each tree node by concatenating the ID of its parent and its ordinal among its sibling, we can give unique ID to each task as well. In addition, the lexicographical order of the node ID obviously corresponds to the order of left- and depth-first traversal of the search tree. However, the string for an ID can be arbitrarily long to make it hard to keep the table size and the cost of comparison between two
IDs reasonably small. Therefore, we represent an ID by a pair of integers as follows so as to keep the spatial and temporal cost small, while introducing a restriction to task creation.

- We associate a range \([\text{minID}, \text{maxID}]\) represented by the pair of 128-bit unsigned integers \(\text{minID} \leq \text{maxID}\) with each task as its task ID.
  - We associate \([0, 2^{128} - 1]\), the full range of 128-bit unsigned integers, with the root task, which is initially assigned to a certain worker at the beginning of a search.
  - When a task division occurs for task \([\text{minID}, \text{maxID}]\), we divide the range by an appropriate integer \(i\) within the range \([\text{minID}, \text{maxID})\) (in our current implementation, \(i = \lfloor(\text{minID} + \text{maxID})/2\rfloor\)), giving \([\text{minID}, i]\) and \([i + 1, \text{maxID}]\) to the tasks for the left and right portions of the subtree to be split. That is, the original task now has \([\text{minID}, i]\) and the new task has \([i + 1, \text{maxID}]\).

- When a worker registers an itemset \(I\) to the itemset table, the 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 remain 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.

This management technique enables the order of tasks to be defined by the order of their minIDs. Since 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 loss of completeness.

The range \([\text{minID}, \text{maxID}]\) 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 recursively up to 128 times so that the resulting sequential tasks are sufficiently small.

4.3 Advantageous Feature of Tascell for Parallel COPINE

To implement our parallel search, other LTC-based multithreaded candidates besides Tascell, such as Cilk [1] and Intel Cilk Plus [20] are to be considered as alternatives. The following advantages of Tascell, however, justify our choice.

1) As mentioned in Chapter 3, 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), meaning that the cost of managing them is eliminated.

2) In a multithreaded language, each (logical) thread requires its own workspace; our COPINE implementations require 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). Since the workspace must be allocated for each
thread, a worker will use new workspace each time it takes a new thread not stolen by another worker, in the sequential execution of a series of threads. In contrast, since Tascell does not allocate the workspace for a task until it is stolen, a worker can reuse a single workspace while performing a sequential computation, thus improving the locality of references.

3) When we implement a backtrack search algorithm in a multithreaded language, each thread often needs its own copy of the workspace of its parent thread. In parallel COPINE, the workspace for the data structures shown in 2) must be copied from the parent thread each time a thread is created. In contrast, Tascell’s temporary backtracking mechanism allows workers to delay the copy operation until absolutely necessary.

4) Tascell enables us to realize dynamic load balancing more easily among computing nodes in distributed memory environments. By this feature, we implemented our distributed-memory version shown in Chapter 5 fairly quickly.

5) The replicating method explained in Section 4.2.3 is easily and efficiently implemented by allowing victim workers to suspend task execution and update a copy of its itemset table at the time a task request is received from a thief worker. It is difficult to implement the same mechanism in an LTC language, because workers steal tasks from the task queue of victims, and the victim that created a specific task is not concerned by 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, because information in 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 4.2.4, a victim worker divides the ID range of a task being executed when it receives a task request from another worker. Since this division only occurs when part of the task is actually stolen by another worker, the number of divisions remains 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 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 the ID range when creating a logical thread, rather than when having a task stolen, thus exhausting the task IDs rapidly and causing minID to become equal to maxID.

Note that advantages 1)–4) were listed in [2], whereas 5) and 6) were discovered during our research.

4.4 Task Creation Strategy

As explained in Section 3.2, a Tascell worker backtracks to the oldest task-spawnable point and spawns 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\(^1\) in Figure 2.6. The search function corresponding to ExploreCCIG() is called recursively in each loop. Therefore, the victim worker

\(^1\)Each of the while loops can be implemented as a for loop whose number of iterations is fixed at the beginning of the loop.
chooses the oldest loop\(^2\) from the parallel \texttt{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 is fixed to half-and-half by the conventional Tascell implementation, preventing us from changing it in our program.

In many parallel backtrack search algorithms, this task creation strategy works reasonably well in terms of increasing the size of each task and reducing the number of steals. 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 preceding nodes in the sequential search have been visited as possible. The half-and-half strategy deprives a thief worker of the chance 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.

Therefore, we have 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 that are stolen, the earlier the spawned task can traverse a subtree that has been 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. We need to set the number of iterations left for a victim in consideration of the trade-off between the effectiveness of the pruning and the traversal speed.

To enable a programmer to change the division ratio at the program level, we enhanced the parallel \texttt{for} construct of Tascell. As shown in Section 4.5.3, we conducted the performance evaluations with a variety of settings of the number of iterations left for a victim worker, which is determined by one of the following two factors; a specific number of iterations, and the ratio of the unexecuted iterations.

\subsection{Performance Evaluation}

\subsubsection{Evaluation Setup}

We evaluated the implementations described in Sections 4.2 and 4.4 by measuring their performance on a single computing node of Appro GreenBlade 8000. The evaluation environment is summarized in Table 4.1. We used a real protein network in which each vertex and edge represent a gene (protein) and a protein-protein interaction, respectively. A set of active conditions of a gene in the real protein network is represented as an itemset associated with a vertex. We used an existing database [21] and the results shown in [22] to create the network and the itemset, respectively. Table 4.2 shows the characteristics of this graph. We set the threshold \(\theta\) 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 \(\theta = 13\). We also compared the performance of each implementation with that of the sequential COPINE implementation written in C.

Note that we executed the programs three times for each measurement of elapsed time, and present the arithmetic mean of the measured results in the tables and charts in this section. There was no great variation among the three samples in each evaluation.

\(^2\)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 4.1: Evaluation environment on Appro GreenBlade 8000.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5-2670 2.6GHz 8-core × 2 (16 cores in total per node)</td>
</tr>
<tr>
<td>Memory</td>
<td>DDR3-1600 64GB</td>
</tr>
<tr>
<td>Network</td>
<td>InfiniBand FDR × 2</td>
</tr>
<tr>
<td>OS</td>
<td>Red Hat Enterprise Linux Server (Santiago)</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC 4.4.7 with -O3</td>
</tr>
<tr>
<td>Nested functions</td>
<td>Trampoline-based implementation in GCC</td>
</tr>
<tr>
<td>MPI</td>
<td>Intel MPI Library 5.1.3</td>
</tr>
<tr>
<td>Worker</td>
<td>Created by <code>pthread_create</code> with <code>PTHREAD_SCOPE_SYSTEM</code></td>
</tr>
<tr>
<td>Lock</td>
<td>A <code>pthread_mutex_t</code> lock is attached to each entry (for Sharing Method 3: Fully-Sharing Method)</td>
</tr>
<tr>
<td>Tascell Server</td>
<td>Implemented in Allegro Common Lisp 8.1 with (speed 3) (safety 1) (space 1)</td>
</tr>
<tr>
<td></td>
<td>Executed on one of the same computing nodes</td>
</tr>
</tbody>
</table>

Table 4.2: Characteristics of the real protein network.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>V</td>
</tr>
<tr>
<td>$</td>
<td>E</td>
</tr>
<tr>
<td>$</td>
<td>I</td>
</tr>
<tr>
<td>Average degree</td>
<td>29.6</td>
</tr>
<tr>
<td>Diameter of $G$</td>
<td>12</td>
</tr>
<tr>
<td>Average depth of search tree in the sequential search ($\theta = 3$)</td>
<td>82.9</td>
</tr>
<tr>
<td># of vertices in the largest connected component</td>
<td>15,061</td>
</tr>
<tr>
<td># of vertices in the smallest connected component</td>
<td>1</td>
</tr>
<tr>
<td>Average # of items in each vertex</td>
<td>9.42</td>
</tr>
</tbody>
</table>

4.5.2 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 number of iterations left for a victim when dividing a parallel loop is half the number of 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 on a single computing node of the Appro GreenBlade 8000. The evaluation results for non-registering method (method 0) is shown in Table 4.3, and those for other methods, i.e., non-sharing (method 1), replicating (method 2) and fully-sharing (method 3), are shown in Table 4.4. Figures 4.1–4.4 show the speedup relative to one-worker execution and the number of visits to vertices for each execution with a worker population. Note that, in Tables 4.3 and 4.4, “Task creation time (TCT)” is the accumulated total time required for initializing task objects and copying workspaces, and “Lock contention rate (LCR)” is the percentage of lock contentions for table entries in all acquisitions.
Table 4.3: Results of performance evaluation with the standard task creation strategy (method 0, $\theta = 13$).

<table>
<thead>
<tr>
<th>Impl.</th>
<th>$w$</th>
<th>ET</th>
<th>SU/C</th>
<th>SU/1</th>
<th>#V</th>
<th>#V/s</th>
<th>#TC</th>
<th>TCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>$5.93 \times 10^{-3}$</td>
<td>1</td>
<td>—</td>
<td>0.085</td>
<td>14.3</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>method 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>139</td>
<td>$0.43 \times 10^{-4}$</td>
<td>1</td>
<td>417</td>
<td>3.01</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>72.0</td>
<td>$0.82 \times 10^{-4}$</td>
<td>1.93</td>
<td>417</td>
<td>2.90</td>
<td>47</td>
<td>0.44</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>36.4</td>
<td>$1.63 \times 10^{-4}$</td>
<td>3.81</td>
<td>417</td>
<td>2.86</td>
<td>589</td>
<td>6.09</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>18.3</td>
<td>$3.24 \times 10^{-4}$</td>
<td>7.59</td>
<td>417</td>
<td>2.78</td>
<td>3.937</td>
<td>3.64</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>9.3</td>
<td>$6.39 \times 10^{-4}$</td>
<td>15.0</td>
<td>417</td>
<td>2.81</td>
<td>8.346</td>
<td>11.3</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: Results of performance evaluation with the standard task creation strategy (methods 1–3, $\theta = 5$).

<table>
<thead>
<tr>
<th>Impl.</th>
<th>$w$</th>
<th>ET</th>
<th>SU/C</th>
<th>SU/1</th>
<th>#V</th>
<th>#V/s</th>
<th>#TC</th>
<th>TCT</th>
<th>LCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>41.3</td>
<td>1</td>
<td>—</td>
<td>659</td>
<td>15.9</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>method 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>48.4</td>
<td>0.854</td>
<td>1</td>
<td>659</td>
<td>13.6</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>47.0</td>
<td>0.558</td>
<td>0.653</td>
<td>2,007</td>
<td>13.6</td>
<td>31</td>
<td>61.6</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>105</td>
<td>0.392</td>
<td>0.459</td>
<td>5,772</td>
<td>13.7</td>
<td>294</td>
<td>355</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>8</td>
<td>66.0</td>
<td>0.625</td>
<td>0.733</td>
<td>7,179</td>
<td>13.6</td>
<td>1,213</td>
<td>2,000</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>16</td>
<td>70.0</td>
<td>0.725</td>
<td>0.849</td>
<td>11,770</td>
<td>12.9</td>
<td>3,962</td>
<td>5,810</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>method 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>48.4</td>
<td>0.854</td>
<td>1</td>
<td>659</td>
<td>13.6</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>47.2</td>
<td>0.875</td>
<td>1.03</td>
<td>1,293</td>
<td>13.7</td>
<td>27</td>
<td>103</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>41.4</td>
<td>0.997</td>
<td>1.17</td>
<td>2,275</td>
<td>13.7</td>
<td>109</td>
<td>414</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>8</td>
<td>34.3</td>
<td>1.20</td>
<td>1.41</td>
<td>3,660</td>
<td>13.3</td>
<td>433</td>
<td>2,040</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>16</td>
<td>28.6</td>
<td>1.44</td>
<td>1.69</td>
<td>5,739</td>
<td>12.5</td>
<td>1,177</td>
<td>5,710</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>method 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>50.3</td>
<td>0.821</td>
<td>1</td>
<td>659</td>
<td>13.1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>59.0</td>
<td>0.700</td>
<td>0.853</td>
<td>1,273</td>
<td>10.8</td>
<td>33</td>
<td>0.54</td>
<td>0.02</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>42.0</td>
<td>0.984</td>
<td>1.12</td>
<td>1,962</td>
<td>11.7</td>
<td>435</td>
<td>4.31</td>
<td>0.09</td>
<td>—</td>
</tr>
<tr>
<td>8</td>
<td>29.2</td>
<td>1.42</td>
<td>1.72</td>
<td>2,483</td>
<td>10.6</td>
<td>675</td>
<td>9.60</td>
<td>0.28</td>
<td>—</td>
</tr>
<tr>
<td>16</td>
<td>21.1</td>
<td>1.96</td>
<td>2.38</td>
<td>2,753</td>
<td>8.15</td>
<td>2,113</td>
<td>4.40</td>
<td>0.81</td>
<td>—</td>
</tr>
</tbody>
</table>

$w$: # of workers; ET: Execution time [s]; SU/C: Speedup (vs. C)
SU/1: Speedup (vs. 1 worker); #V: # of visits to vertices [$\times 10^6$] (total of all workers)
#V/s: # of visits to vertices [$\times 10^6$/s (average among all workers)
#TC: # of task creations TCT: Task creation time [ms]
LCR: Lock contention rate [%]

33
Figure 4.1: Speedup and number of visits to vertices with the standard task creation strategy (non-registering method, $\theta = 13$).

Figure 4.2: Speedup and number of visits to vertices with the standard task creation strategy (non-sharing method, $\theta = 5$).
Figure 4.3: Speedup and number of visits to vertices with the standard task creation strategy (replicating method, $\theta = 5$).

Figure 4.4: Speedup and number of visits to vertices with the standard task creation strategy (fully-sharing method, $\theta = 5$).
Results with Non-Registering Method

Because Pruning 3 is not applied, the search space of the parallel search with multiple workers is identical to that of the sequential search done by one worker. Therefore, the speedup relative to one worker is almost ideal, and we can 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 inapplicable to the graph used in this evaluation.

Results with Non-Sharing Method

The execution time with one worker is approximately 17.2% worse (7.1 s longer) than that of the sequential implementation in C. This is mainly caused by the cost of the Tascell mechanisms such as polling. Because 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 a performance improvement.

From Tables 4.3 and 4.4, we can see that the per-task creation time (given by TCT/#TC) in the non-sharing method (method 1) is much 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. As 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.

Results with Replicating Method

In the replicating method, a victim worker makes a copy of its own table at each task creation so that the thief worker can use a part of the 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 achieve 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, which is insignificant in the execution time of 21.1 s.

Results with Fully-Sharing Method

The performance of this method is better than that of 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 (because it is not necessary to make a copy of an itemset table at each task creation), the traversal 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 referring to a table entry and the cost of acquiring a lock at each search step. We discuss this influence on the parallel performance in more detail in Section 5.4.

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

We evaluated the effect of the number of iterations left for a victim worker on the effectiveness of Pruning 3. We measured the performance of the replicating method (method 2) and the fully-sharing method (method 3) using Appro GreenBlade 8000 on a single computing node, as 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 number of iterations** 1, and 10–200 in steps of 10, and

**setting by 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 with 16 workers. In addition, we carried out the evaluation with 1, 2, 4 and 8 workers as well, for the setting in which only one iteration is left for the victim worker because it is most promising in the effectiveness of Pruning 3.

Figures 4.5 (replicating method) and 4.6 (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 Figures 4.7 and 4.8, and Table 4.5.

**Results with Replicating Method**

Comparing the results of the replicating method in Tables 4.4 and 4.5, it can be seen that the total number of visits to vertices is reduced by leaving only one iteration for a victim worker. In contrast, the traversal speed decreases considerably as the number of task creations increases. As 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. This has a negative and serious impact on the overall performance. For instance, the total task creation time in the replicating method with 16 workers in Table 4.5 (54.7 s) is 9.58 times that in Table 4.4 (5.81 s), resulting in a protraction of the overall execution time by 20 s (approximately 70%). The tendency whereby a reduction in the number of iterations worsens the overall performance while shrinking the search space is also visible in Figure 4.5, leading us to conclude that Tascell’s standard half-and-half strategy is appropriate for this method.

**Results with Fully-Sharing Method**

A comparison of the results for the fully-sharing method (Tables 4.4 and 4.5) leads to different conclusions from those for the replicating method. That is, by leaving only one iteration for a victim worker, we successfully reduce both the search space size and the overall execution time, resulting in the highest 4.07-fold 16-worker speedup among the experiments in this chapter. This is not only because the search space reduction is more efficient (2.28-fold for this method compared with 1.55-fold for the replicating method in the 16-worker case), but also because the reduction in traversal speed is less significant (1.33-fold for this method and 2.61-fold for the

---

3 This means that only the iteration being executed by a victim worker is left. That is, “unexecuted iterations” here include the iteration being executed.
Figure 4.5: Execution time and number of visits to vertices versus the number/ratio of iterations left for a victim worker (replicating method with 16 workers).
Figure 4.6: Execution time and number of visits to vertices versus the number/ratio of iterations left for a victim worker (fully-sharing method with 16 workers).
Figure 4.7: Speedup and number of visits to vertices with the setting that leaves one iteration for the victim worker on task creation (replicating method, $\theta = 5$).

Figure 4.8: Speedup and number of visits to vertices with the setting that leaves one iteration for the victim worker on task creation (fully-sharing method, $\theta = 5$).
Table 4.5: Results of performance evaluation with the setting that leaves one iteration for a victim worker on task creation ($\theta = 5$).

<table>
<thead>
<tr>
<th>Impl.</th>
<th>$w$</th>
<th>ET [s]</th>
<th>SU/C</th>
<th>SU/1</th>
<th>#V</th>
<th>#V/s</th>
<th>#TC</th>
<th>TCT [ms]</th>
<th>LCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>41.3</td>
<td>1</td>
<td>—</td>
<td>659</td>
<td>15.9</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>47.2</td>
<td>0.732</td>
<td>0.858</td>
<td>1226</td>
<td>10.9</td>
<td>4,140</td>
<td>12,800</td>
<td>—</td>
</tr>
<tr>
<td>method 2</td>
<td>4</td>
<td>43.4</td>
<td>0.946</td>
<td>1.11</td>
<td>1,936</td>
<td>11.1</td>
<td>2,506</td>
<td>8,040</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>53.4</td>
<td>0.773</td>
<td>0.906</td>
<td>2,549</td>
<td>5.97</td>
<td>10,085</td>
<td>3,200</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>48.2</td>
<td>0.856</td>
<td>1.00</td>
<td>3,693</td>
<td>4.79</td>
<td>14,575</td>
<td>54,700</td>
<td>—</td>
</tr>
<tr>
<td>method 3</td>
<td>1</td>
<td>50.3</td>
<td>0.821</td>
<td>1</td>
<td>659</td>
<td>13.1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>28.1</td>
<td>1.47</td>
<td>1.79</td>
<td>683</td>
<td>12.2</td>
<td>52</td>
<td>3.53</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>19.6</td>
<td>2.11</td>
<td>2.57</td>
<td>830</td>
<td>10.6</td>
<td>185</td>
<td>14.4</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>15.8</td>
<td>2.61</td>
<td>3.18</td>
<td>1,076</td>
<td>8.51</td>
<td>1,137</td>
<td>97.3</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>12.4</td>
<td>3.34</td>
<td>4.07</td>
<td>1,209</td>
<td>6.12</td>
<td>7,201</td>
<td>198</td>
<td>0.53</td>
</tr>
</tbody>
</table>

$w$: # of workers; ET: Execution time [s]; SU/C: Speedup (vs. C) SU/1: Speedup (vs. 1 worker); #V: # of visits to vertices [$\times 10^6$] (total of all workers) #V/s: # of visits to vertices [$\times 10^6$]/s (average among all workers) #TC: # of task creations TCT: Task creation time [ms] LCR: Lock contention rate [%]

replicating method). The latter is derived from the fact that a victim worker does not have to copy its itemset table, as required for every task creation in the replication method.

The good speedup is also confirmed by comparing the search space size and traversal speed of one-worker and 16-worker executions with the half-and-half and only-one strategies. The rates at which the search space expands and the traversal speed decreases from one-worker to 16-worker are 1.83 ($= 1,209 \times 10^6/6.59 \times 10^6$) and 2.14 ($= 13.1 \times 10^6/6.12 \times 10^6$) with the only-one strategy, compared with 4.17 ($= 2,753 \times 10^6/6.59 \times 10^6$) and 1.61 ($= 13.1 \times 10^6/8.15 \times 10^6$) in the half-and-half case. As the product of the two rates determines the parallel speedup, where smaller values are better in this case, the 16-worker speedup is greatly improved from 2.38 ($= 16/4.17/1.61$) to 4.07 ($= 16/1.83/2.14$).

In addition, Figure 4.6 confirms that the only-one strategy is most appropriate for the fully-sharing method. Both charts in the figure clearly show that the smaller the number of iterations left for a victim, the better the speedup in the execution time.

### 4.5.4 Effect of Vertex Ordering

The shape of a search tree in the COPINE algorithm depends on the order of vertices, as described in Section 2.2 for Pruning 1. Because 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 vertex ordering. To evaluate this effect, we generated 100 random vertex orders in the graph used in this evaluation, and measured the execution time for each ordering.

The evaluations were conducted under the optimal conditions from the previous evaluations, that is, the fully-sharing method with the setting that leaves one iteration for a victim when divid-
Figure 4.9: Execution time and speedup with 16 workers versus vertex ordering (sorted by execution time with 16 workers).

We executed the Tascell program with 1 and 16 workers and the sequential C program for each ordering on a single computing node of the Appro GreenBlade 8000. We determined each order by generating uniform random numbers in the range $[1, |V|]$ by the Mersenne Twister [23] and assigning a unique number to each vertex.

The execution times are shown in Figure 4.9. As shown in the chart, the times of the one-worker execution and the sequential execution are almost stable because the search space size is virtually independent of the tree shape. The 16-worker execution times show a slightly larger deviation but remain, in 95 cases out of 100, within the range 11.1–16.4 s, resulting in a 3.10- to 4.52-fold speedup, similar to the results in Table 4.5. 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.

### 4.6 Conclusion

In this chapter, we implemented the parallel COPINE algorithm in shared memory environments using the task-parallel language Tascell. In this implementation, workers can only refer to an itemset registered by another worker if the registration-reference flow conforms to the sequential search order, thus preventing excessive pruning. To prune a branch corresponding to a subgraph having an already-visited supergraph with an identical itemset, workers need to share table in-
formation efficiently in such a manner that a worker can use as many itemsets registered by other workers as possible under the given restriction. We implemented the following sharing methods, in which workers are urged to use itemsets registered by other workers: (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 for comparison with 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.
Chapter 5

Parallel Implementation in Distributed Memory Environments

5.1 Introduction

This chapter describes a parallel implementation of COPINE in distributed memory environments as an enhancement of the implementations described in Chapter 4. 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 [2,8]. 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. In our implementation described in Chapter 4, a single table controlled by locks is shared among workers. This method is unrealistic in distributed memory environments, since it drastically increases the cost of internode communication. Therefore, in Section 5.2, 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 under the conventional Tascell work-stealing strategy, an idle worker (thief) targets another worker (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 previous evaluations using backtrack search algorithms [2, 8], serious performance degradation occurs in our parallel COPINE solver because of the high cost of work-stealing. As described in Section 5.3, 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 size of the 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 that is expected to steal only a small task waits for a certain time before attempting to steal.
while(1) {
    uint64_t send_buf[], recv_buf[];
    sleep(t_comm);
    // extract table updates and pack them into send_buf
    for(i=0; i<|V|; i++) {
        pthread_mutex_lock(&table_entry_lock[i]);
        Extract itemsets registered to the table entry v_i after the previous communication and append them to send_buf.
        pthread_mutex_unlock(&table_entry_lock[i]);
    }
    // exchange the sizes of send_buf in all nodes
    MPI_Allgatherv(send_buf, ...);
    // exchange the table updates in all nodes
    MPI_Allgatherv(send_buf, ..., recv_buf, ...);
    // unpack the received table updates from recv_buf
    for(k=0; k<# of itemsets in recv_buf; k++) {
        e = the k-th element in recv_buf;
        i = the vertex number in e;
        pthread_mutex_lock(&table_entry_lock[i]);
        Register the itemset in e to the table entry v_i.
        pthread_mutex_unlock(&table_entry_lock[i]);
    }
}

Figure 5.1: Flow of the communication thread for updates to an itemset table.

5.2 Itemset Table Sharing

A worker can immediately use itemsets registered by other workers by using the sharing method described in Section 4.2.4. However, this method is unrealistic to be used in distributed memory environments, because internode communication is required every time a worker registers an itemset to the itemset table. Therefore, we implemented the following method, which allows table information to be shared at a realistic cost while allowing some incompleteness in the sharing of 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 4.2.4. 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) extracts table updates from its computing node, (2) sends the updates to all other nodes and receives updates from them, and (3) registers the received updates to the table of its node.

Figure 5.1 shows the pseudo-code for the communication thread operations. The communication thread of each computing node extracts all itemsets (including the minIDs and maxIDs described in Section 4.2.4) registered to the table after the previous communication, and stores
them in the send buffer. The communication threads in all the computing nodes then 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 the minID and maxID associated with each received itemset, and performs the operations described in Section 4.2.4 to satisfy the restriction described in Section 2.3.3 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.

5.3 Improvement of Work-Stealing Strategy

As explained in Section 3.2, a victim worker that receives a task request backtracks to the oldest task-spawnable point to spawn a larger task. However, the thief’s strategy in choosing a victim, explained in Section 3.3.1, is oblivious to how large a task obtained from a candidate is. In addition, under the stealing-back mechanism mentioned in Section 3.3.2, the thief is often restricted to sending task requests to specific workers. This approach did not cause considerable performance problems in previous evaluations [2, 8]. However, the cost of a work-steal in our parallel COPINE implementation is high since the task objects in COPINE are large, containing a current subgraph from which a thief starts its search, a set of vertices adjacent to the subgraph, and so on. As such task objects are 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 have improved the strategy for choosing a victim and have attempted to reduce the number of tasks that are stolen back to allow workers to obtain larger tasks.

5.3.1 Strategy for Choosing a Victim

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 there exists a worker with a spawnable task in the same node. This means that, once a representative worker in a node has acquired a task, none of the workers in that node send task requests to external nodes until the first task has been completed (except for stealing back). This strategy is effective from the viewpoint of reducing the number of internode task stealing. However, the thief always sends task requests to workers in the same node, even if it could acquire a larger task from an external node. Therefore, if there only are small tasks in the node in which the thief worker resides, the worker completes the stolen task in a short time and then obtains another small task again. Such repetition continues until all tasks assigned to that node have been 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 encourages workers to request tasks from external nodes. When sending a task request, the thief omits step (1) in Section 3.3.1 to choose a victim in the node in which the thief resides, 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 $\tau$. The larger the value of $\tau$, the more frequently workers request tasks from external nodes. Note that this strategy is equivalent to the conventional strategy when $\tau = 1$.

**Choosing a Victim inside a Computing Node**

We also implemented a strategy where, at steps (1) and (5) in Section 3.3.1 for the choice of the victim of intra- and inter-node task requests, respectively, the thief estimates the size of the tasks being executed by other workers in the computing node and chooses the worker with the maximum estimated size as a victim from which to steal a larger task. Specifically, the following operations are performed.

- Each worker remembers the number of divisions of its running task counted from the root task. In the COPINE case, the root task corresponds to the task of traversing the whole search tree.
- Steps (1) and (5) in Section 3.3.1 are modified as follows:
  - The thief (Step (1)) or the Tascell server (Step (5)) randomly chooses $\kappa$ workers ($1 \leq \kappa \leq$ the number of workers in the node) from those in the node, and checks the numbers of task divisions of their running tasks.
  - The thief or the server chooses a worker with the smallest number of task divisions among the $\kappa$ 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 $\kappa$ should account for the trade-off between the expected sizes of the stolen tasks and the cost of checking the other workers. Note that this strategy is equivalent to the conventional strategy when $\kappa = 1$.

**5.3.2 Reducing the Number of Tasks Stolen Back 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 assigned with the task causing the synchronization delay, even if another worker can spawn a larger task. As stated in Section 5.3.1, 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 5.3.1, we cannot solve this problem by changing the destination of a request for a task that has been stolen back. Therefore, we attempted to alleviate the performance degradation caused by the stealing-back chain by reducing the number of tasks stolen back. Specifically, a worker that is waiting for the result of another task assigned to a worker in an external node pauses 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, as the thief would receive the result of the spawned task within the waiting time if the task was too small to be worth stealing.
Table 5.1: Evaluation environment on Cray XC40.

<table>
<thead>
<tr>
<th></th>
<th>Cray XC40 (1 node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon Phi 7250 (KNL) 1.4GHz 68-core</td>
</tr>
<tr>
<td>Memory</td>
<td>DDR4-2666 96GB</td>
</tr>
<tr>
<td>OS</td>
<td>SUSE Linux Enterprise Server 12</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC 6.1.0 with -O3</td>
</tr>
<tr>
<td>Nested functions</td>
<td>Trampoline-based implementation in GCC</td>
</tr>
<tr>
<td>Worker</td>
<td>Created by pthread_create with PTHREAD_SCOPE_SYSTEM</td>
</tr>
<tr>
<td>Lock</td>
<td>A pthread_mutex_t lock is attached to each entry</td>
</tr>
</tbody>
</table>

5.4 Performance Evaluation

5.4.1 Evaluation Setup

We measured the performance of our implementations in distributed memory environments using the Appro GreenBlade 8000 supercomputer, which was used to conduct the experiments in Section 4.5. For comparison, we also used a Cray XC40 supercomputer with Xeon Phi processors to measure the performance in shared memory environments with many workers. The evaluation environment in the Cray XC40 is summarized in Table 5.1. We used the same input data as given in Section 4.5. We set the threshold $\theta = 5$. We set the number of iterations left for a victim worker from unexecuted iterations as 2 in consideration of the trade-off between the effectiveness of Pruning 3 and the traversal speed. We also compared the performance of each implementation with that of the sequential COPINE implementation written in C.

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

5.4.2 Performance in Distributed Memory Environments

We evaluated the performance improvement attained by the work-stealing strategies proposed in Section 5.3 by measuring the performance with the Appro GreenBlade 8000 on multiple computing nodes with the following two parameter settings:

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

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

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

(b) We found the optimal values of $\tau$ and $t_{\text{sb}}$ by testing all combinations of $(\tau, t_{\text{sb}})$ that satisfy $\tau \in \{1, 2, 4, 8, 16\}$ and $t_{\text{sb}} \in \{0 \text{ s}, 1 \text{ ms}, 10 \text{ ms}, 100 \text{ ms}, 1 \text{ s}\}$ while fixing $t_{\text{comm}}$ and $\kappa$ to the values found in (a).
Table 5.2: Results of performance evaluation with multiple computing nodes.

<table>
<thead>
<tr>
<th>Impl.</th>
<th>n</th>
<th>w</th>
<th>ET</th>
<th>SU/C</th>
<th>SU/1</th>
<th>TER</th>
<th>#V</th>
<th>#V/s</th>
<th>#RI</th>
<th>#TC</th>
<th>LCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>30.3</td>
<td>1</td>
<td>—</td>
<td>100 ± 0</td>
<td>614</td>
<td>20.3</td>
<td>172</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>One</td>
<td>1</td>
<td>16</td>
<td>47.0</td>
<td>0.644</td>
<td>1</td>
<td>100 ± 0</td>
<td>614</td>
<td>13.1</td>
<td>172</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Conv.</td>
<td>1</td>
<td>16</td>
<td>11.5</td>
<td>2.64</td>
<td>4.09</td>
<td>84.2 ± 3.32</td>
<td>1,370</td>
<td>7.46</td>
<td>1,337</td>
<td>8,658</td>
<td>0.827</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16</td>
<td>10.9</td>
<td>2.79</td>
<td>4.33</td>
<td>57.3 ± 16.1</td>
<td>1,799</td>
<td>5.18</td>
<td>2,011</td>
<td>15,941</td>
<td>0.834</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>10.1</td>
<td>2.98</td>
<td>4.63</td>
<td>47.6 ± 20.3</td>
<td>2,627</td>
<td>4.05</td>
<td>3,209</td>
<td>14,935</td>
<td>0.928</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>10.5</td>
<td>2.89</td>
<td>4.49</td>
<td>29.8 ± 15.9</td>
<td>3,269</td>
<td>2.44</td>
<td>3,030</td>
<td>24,287</td>
<td>0.957</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>11.1</td>
<td>2.72</td>
<td>4.22</td>
<td>15.3 ± 10.2</td>
<td>3,721</td>
<td>1.31</td>
<td>5,083</td>
<td>32,561</td>
<td>0.958</td>
</tr>
<tr>
<td>Prop.</td>
<td>1</td>
<td>16</td>
<td>10.2</td>
<td>2.96</td>
<td>4.60</td>
<td>84.5 ± 6.30</td>
<td>1,270</td>
<td>7.77</td>
<td>1,211</td>
<td>8,656</td>
<td>0.817</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16</td>
<td>8.57</td>
<td>3.53</td>
<td>5.48</td>
<td>63.1 ± 11.5</td>
<td>1,525</td>
<td>5.56</td>
<td>1,660</td>
<td>9,536</td>
<td>0.782</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>7.83</td>
<td>3.86</td>
<td>6.00</td>
<td>55.1 ± 14.0</td>
<td>2,194</td>
<td>4.38</td>
<td>2,224</td>
<td>15,967</td>
<td>1.04</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>6.46</td>
<td>4.69</td>
<td>7.28</td>
<td>51.1 ± 11.3</td>
<td>3,085</td>
<td>3.73</td>
<td>5,579</td>
<td>21,902</td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>7.10</td>
<td>4.26</td>
<td>6.62</td>
<td>25.4 ± 13.8</td>
<td>3,548</td>
<td>1.95</td>
<td>6,298</td>
<td>26,669</td>
<td>1.08</td>
</tr>
</tbody>
</table>

Conventional: \( t_{comm} = 500 \text{ ms}, \tau = 1, \kappa = 1, t_{sb} = 0\text{ s} \)

Proposed: \( t_{comm} = 100 \text{ ms}, \tau = 2, \kappa = 16, t_{sb} = 100\text{ ms} \)

n: # of computing nodes; w: # of workers per node; ET: Execution time [s]

SU/C: Speedup (vs. C); SU/1: Speedup (vs. 1 worker); TER: Task execution rate [%]

#V: # of visits to vertices \( [\times10^6] \) (total of all workers)

#V/s: # of visits to vertices \( [\times10^6]/s \) (average among all workers)

#RI: # of referred itemsets in itemset table \( [\times10^6] \) (total of all workers)

#TC: # of task creations; LCR: Lock contention rate [%]

Table 5.2 contains the results of the evaluation. Note that, in Table 5.2, “Task execution rate (TER)” is the average and standard deviation of the percentage of time spent on task execution with respect to the total execution time among all workers. “Conventional” and “Proposed” are executions with the parameter settings in (1) and (2) above, respectively.

The “Conventional” results in Table 5.2 indicate that the task execution rates significantly decrease as the number of nodes increases, and we cannot obtain any performance speedup with multiple computing nodes. However, the “Proposed” strategy enables us to alleviate this degradation in task execution rates. The improvement in the execution rates enhances the degree of parallelism. This usually leads to some degradation in the completeness of Pruning 3, or an increase in the number of visits to vertices\(^1\). However, the number of visits to vertices in the “Proposed” scheme is similar to that for the “Conventional” case. This is because the communication interval for exchanging table updates \( t_{comm} \) in “Proposed” is shorter than that in “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 7.28-fold speedup compared to the one-worker execution, which is a 4.69-fold speedup compared to C and 1.78-fold faster (= 7.28/4.09) than the execution by 1 node × 16 workers. However, we were unable to achieve good performance in the 16-node execution of the “Proposed” scheme because the task execution rate decreases significantly. Future work will focus on improving the performance with additional computing nodes.

\(^1\)This problem also occurs in shared memory environments and is difficult to solve, as discussed in Chapter 4.
We evaluated the effect of changing the values of $t_{\text{comm}}$, $\tau$, $\kappa$, and $t_{\text{sb}}$ by measuring the performance with 8 nodes $\times$ 16 workers when varying the values of these parameters independently while fixing the other parameters to the settings of “Conventional.” Figures 5.2–5.5 show the measurement results. The error bars of the task execution rates show the standard deviation among workers. These results show the following.

- Reducing $t_{\text{comm}}$ enables us to reduce the total number of visits to vertices and improve the overall performance. In Table 5.3, we present the cumulative processing time of the communication thread in the process with MPI rank 0 when $t_{\text{comm}}$ is set to 100 ms, 500 ms, and 1 s. We can see that the increase in the cost of communication threads when reducing $t_{\text{comm}}$ 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 nodes by increasing $\tau$, the task execution rate decreases. This is because the increase in the amount of work obtained by internode steals results in an increase in the number of internode steal-back of tasks. Note that we obtained a slight performance improvement in spite of the degradation in task execution rates when $\tau$ was set to 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 $\kappa$ increases, there is no great change in the task execution rate, but the overall performance improves slightly.

- There is no correlation between $t_{\text{sb}}$ and the performance or the task execution rate.

From these results, we can conclude that the task execution rates and overall performance cannot be improved by varying only one of $\tau$ and $t_{\text{sb}}$. We evaluated the effect of varying both $\tau$ and $t_{\text{sb}}$ in detail by rechecking the results of the performance measurements conducted to find the optimal settings of “Proposed,” that is, the measurements when varying $\tau$ and $t_{\text{sb}}$ at the same time while fixing $t_{\text{comm}}$ and $\kappa$ to 100 ms and 16, respectively. Table 5.4 shows the execution times and the task execution rates with these settings. It is clear that the task execution rate considerably improves as $t_{\text{sb}}$ increases when $\tau$ is set to 2 or 4. One possible reason for these performance improvements is that, as the number of internode-stolen tasks increases with $\tau$, the increase in the number of internode steal-back of tasks is suppressed by increasing $t_{\text{sb}}$.

### 5.4.3 Comparison with Performance in Shared Memory Environments

To compare the performance between shared and distributed memory environments, we measured the performance of each environment with 1 node $\times$ 1–16 workers and 1–16 nodes $\times$ 1 worker on the Appro GreenBlade 8000. We set the parameters to the values of “Conventional” value in Table 5.2. Figure 5.6 shows the measurement results.

A comparison of the results in Figure 5.6 indicates that the difference between the performance in shared and distributed memory environments widens as the total 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 ± 4.20% and 32.5 ± 13.1%, respectively, when $n = 16$. The task execution rate decreases in distributed memory environments because
Figure 5.2: Performance when changing the value of $t_{\text{comm}}$ (communication interval for broadcasting table updates) using 8 nodes $\times$ 16 workers.

Figure 5.3: Performance when changing the value of $\tau$ (threshold determining whether a task request is sent to external node) using 8 nodes $\times$ 16 workers.
vs. 1 worker

Figure 5.4: Performance when changing the value of $\kappa$ (# of workers where # of task divisions is queried) using 8 nodes $\times$ 16 workers.

Figure 5.5: Performance when changing the value of $t_{sb}$ (waiting time before stealing back) using 8 nodes $\times$ 16 workers.
Table 5.3: Cumulative processing time of a communication thread.

<table>
<thead>
<tr>
<th>( t_{\text{comm}} )</th>
<th>Pack [s]</th>
<th>MPI comm. [s]</th>
<th>Unpack [s]</th>
<th>Total [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 ms</td>
<td>0.0362</td>
<td>1.46</td>
<td>1.30</td>
<td>2.80</td>
</tr>
<tr>
<td>500 ms</td>
<td>0.0354</td>
<td>1.27</td>
<td>0.884</td>
<td>2.19</td>
</tr>
<tr>
<td>1 s</td>
<td>0.0249</td>
<td>0.883</td>
<td>0.599</td>
<td>1.50</td>
</tr>
</tbody>
</table>

Table 5.4: Performance when changing the values of \( \tau \) and \( t_{sb} \) (using 8 nodes \( \times \) 16 workers).

(Average task execution rate among workers \( \pm \) S.D. [%])

<table>
<thead>
<tr>
<th>( \tau )</th>
<th>( t_{sb} )</th>
<th>0 s</th>
<th>1 ms</th>
<th>10 ms</th>
<th>100 ms</th>
<th>1 s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>8.62</td>
<td>8.24</td>
<td>8.23</td>
<td>7.99</td>
<td>7.23</td>
</tr>
<tr>
<td>1</td>
<td>(32.9 ( \pm ) 12.3)</td>
<td>(24.6 ( \pm ) 19.4)</td>
<td>(30.3 ( \pm ) 14.0)</td>
<td>(37.9 ( \pm ) 9.27)</td>
<td>(37.1 ( \pm ) 8.44)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.97</td>
<td>7.89</td>
<td>7.51</td>
<td>6.46</td>
<td>6.68</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(29.2 ( \pm ) 16.5)</td>
<td>(33.0 ( \pm ) 15.6)</td>
<td>(36.4 ( \pm ) 12.9)</td>
<td>(51.1 ( \pm ) 11.3)</td>
<td>(38.4 ( \pm ) 9.46)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.76</td>
<td>7.66</td>
<td>7.50</td>
<td>7.11</td>
<td>7.75</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(24.8 ( \pm ) 13.7)</td>
<td>(32.5 ( \pm ) 13.6)</td>
<td>(33.1 ( \pm ) 10.4)</td>
<td>(41.2 ( \pm ) 13.2)</td>
<td>(34.3 ( \pm ) 13.1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12.9</td>
<td>12.7</td>
<td>12.7</td>
<td>11.4</td>
<td>10.8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(14.3 ( \pm ) 9.16)</td>
<td>(13.1 ( \pm ) 8.22)</td>
<td>(17.8 ( \pm ) 11.9)</td>
<td>(17.0 ( \pm ) 9.11)</td>
<td>(20.3 ( \pm ) 9.91)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.3</td>
<td>21.1</td>
<td>23.2</td>
<td>20.8</td>
<td>19.5</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>(4.83 ( \pm ) 3.28)</td>
<td>(4.91 ( \pm ) 3.30)</td>
<td>(5.90 ( \pm ) 3.86)</td>
<td>(6.86 ( \pm ) 4.04)</td>
<td>(6.45 ( \pm ) 4.07)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.6: Performance comparison between shared and distributed memory environments.
Table 5.5: Results of performance evaluation on the Xeon Phi processor.

<table>
<thead>
<tr>
<th>Impl.</th>
<th>$w$</th>
<th>ET</th>
<th>SU/C</th>
<th>SU/1</th>
<th>TER</th>
<th>#V</th>
<th>#V/s</th>
<th>#RI</th>
<th>#TC</th>
<th>LCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>140</td>
<td>0.815</td>
<td>1</td>
<td>100</td>
<td>± 0</td>
<td>614</td>
<td>5.38</td>
<td>172</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>84.2</td>
<td>1.35</td>
<td>1.66</td>
<td>99.1</td>
<td>± 0.763</td>
<td>680</td>
<td>4.04</td>
<td>272</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>57.0</td>
<td>2.00</td>
<td>2.46</td>
<td>97.8</td>
<td>± 1.02</td>
<td>814</td>
<td>3.57</td>
<td>435</td>
<td>0.135</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>44.2</td>
<td>2.58</td>
<td>3.17</td>
<td>95.1</td>
<td>± 1.10</td>
<td>1,131</td>
<td>3.20</td>
<td>797</td>
<td>3.398</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>35.3</td>
<td>3.23</td>
<td>3.96</td>
<td>85.1</td>
<td>± 3.90</td>
<td>1,426</td>
<td>2.52</td>
<td>1,352</td>
<td>18,671</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>30.0</td>
<td>3.80</td>
<td>4.66</td>
<td>72.4</td>
<td>± 8.92</td>
<td>1,634</td>
<td>1.70</td>
<td>2,160</td>
<td>33,562</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>28.8</td>
<td>3.97</td>
<td>4.87</td>
<td>67.9</td>
<td>± 7.57</td>
<td>2,040</td>
<td>1.11</td>
<td>3,727</td>
<td>19,569</td>
</tr>
<tr>
<td></td>
<td>68</td>
<td>28.5</td>
<td>4.01</td>
<td>4.92</td>
<td>66.5</td>
<td>± 5.18</td>
<td>2,069</td>
<td>1.07</td>
<td>3,777</td>
<td>19,423</td>
</tr>
</tbody>
</table>

$w$: # of workers; ET: Execution time [s]; SU/C: Speedup (vs. C)
SU/1: Speedup (vs. 1 worker); TER: Task execution rate [%]
#V: # of visits to vertices [$\times 10^6$] (total of all workers)
#V/s: # of visits to vertices [$\times 10^6$]/s (average among all workers)
#RI: # of referred itemsets in itemset table [$\times 10^6$] (total of all workers)
#TC: # of task creations; LCR: Lock contention rate [%]

of the large cost of internode steal of work. Note that the number of visits to vertices in distributed memory environments is smaller than that in shared memory environments because of the decrease in the degree of parallelism caused by the degradation of the task execution rates. Thus, from these results, it is difficult to evaluate the extent to which the incompleteness of table information shared across computing nodes affects the performance.

We also measured the performance in shared memory environments with up to 68 workers on the Xeon Phi processor and compared the results with those from the “Proposed” strategy (Table 5.2) using 1–4 nodes × 16 workers on the Appro GreenBlade 8000. The detailed results from this evaluation are presented in Table 5.5. Figure 5.7 illustrates the comparison result.

In the executions on the Xeon Phi, a worker can immediately refer to itemsets registered by other workers, and the cost of work stealing is smaller than that in distributed memory environments. Nevertheless, we achieved only a 4.92-fold speedup with 68 workers, because the search space enlarges as the number of workers increases. In addition, the task execution rate decreases in the many-worker executions, that is, the stealing-back mechanism also affects the performance in shared memory environments with many workers. The speedup with 4 nodes × 16 workers on the Appro GreenBlade 8000 (6.00-fold relative to the one-worker execution) is comparable to that in shared memory environments with approximately the same number of workers.

One of the reasons why the performance in distributed memory environments is better than that on the Xeon Phi concerns the assignment of the itemset table for each computing node. From the “Lock contention rate” column in Tables 5.2 and 5.5, we can see that the cost of acquiring a lock in distributed memory environments is almost the same as that in shared memory environments with 16 workers, although that on the Xeon Phi rises as the number of workers increases. This is because the number of workers per computing node is 16 at most.

However, in both shared and distributed memory environments, workers in multiple-worker executions refer to more itemsets than in the one-worker execution. For instance, in Table 5.5,
the number of referred itemsets in the 68-worker execution is 21.9 times that in the one-worker execution. If we consider the rate of search space expansion (3.37 times), this is an increase of a factor of $6.50 = 21.9/3.37$. This enlargement is also observed in distributed memory environments, as we can see from Table 5.2. We need to solve this problem to attain further performance improvement.

5.5 Conclusion

In this chapter, we have proposed a parallel implementation of the COPINE algorithm 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 using the MPI collective communication functions.

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 inefficient 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 size of tasks created by victim candidates to obtain larger tasks.

As a result of these improvements, we achieved a 7.28-fold speedup with 8 nodes x 16 workers compared to the one-worker execution in a performance evaluation using a real protein network.
Chapter 6

Enhancement of Tascell for Abort Operation

6.1 Introduction

This chapter introduces a language extension for the task-parallel language Tascell. This implementation enables a form of exception handling in which all the running parallel tasks in a try block with an exception are automatically aborted as soon as possible.

We first describe an example of the exception handling mechanism. Consider a task-parallel execution with four workers, named workers 0–3, in which the function $f0$ in Figure 6.1 is called by worker 0. In this program, spawn $S$ is used to spawn a task to execute $S$ asynchronously that can be assigned to another worker, and join $S$ is used to synchronize all the tasks spawned during the execution of $S$. The execution context in such a task-parallel execution forms a data structure called a cactus stack. One possible context in the execution of the program shown in Figure 6.1 is illustrated in Figure 6.2(a), where the tasks spawned by worker 0 in lines 4 and 13 have been assigned to worker 1 and worker 2, respectively, and the task spawned by worker 2 in line 19 has been assigned to worker 3. Then, each worker executing $e(\cdot)$ may or may not throw the exception $E$ that would be caught by the catcher established in $f0(\cdot)$. When at least one of the workers has thrown the exception, one desired behavior is that the execution stack becomes as shown in Figure 6.2(b) as soon as possible. For example, when only worker 1 has thrown the exception, not only the task assigned to worker 1 completed with the return of the exception, but also the tasks assigned to workers 2 and 3 are aborted “collaterally,” and the control of worker 0 returns to the catch block in lines 7–9.

Such an exception handling mechanism with collateral task abortion is useful when the objective is to complete the search as soon as one solution has been found. For example, in a parallel binary tree search, the search can be terminated as soon as a solution is found by simply throwing an exception caught at the root of the search tree, as shown in Figure 6.3.

Furthermore, exceptions are used to allow a worker to abort the traversal of a subtree that is found to be redundant by another worker, even after it has initiated the traversal. Figure 6.4 shows the pseudo-code for a parallel binary search for all solutions using this technique. When a

---

1In multithreaded languages based on Lazy Task Creation, such as Cilk, a spawned task is immediately executed by the worker that spawned the task, and another idle worker steals the continuation at this point as a task. Thus, part of the following discussion in this section, including the execution contexts shown in Figure 6.2, is not strictly applicable for such a language.
Figure 6.1: Example of task-parallel program with exceptions.

```c
f0() {  
    try {
        join {
            spawn e();
            f1();
        }
    } catch(E) {
    }
}
f1() {
    join {
        spawn f2();
        e();
    }
}
```

Figure 6.2: Execution states in an exception of the program in Figure 6.1.

(a) Before throwing an exception

```c
f2() {
    join {
        spawn e();
        e();
    }
}
e() {
    if(exceptional condition?) {
        throw(E);
    } else {
        ...
    }
}
```

(b) After an exception is caught
binsearch(Node root) {
    int found = 0;
    try {
        binsearch_r(Node root);
    } catch (E) {
        found = 1;
    }
    // return 1 if a solution is found
    return found;
}
binsearch_r(Node node) {
    if(!node) return;
    if(Solution found at node?) {
        throw (E);
    } else {
        join {
            spawn binsearch_r(node->right);
            binsearch_r(node->left);
        }
    }
}

Figure 6.3: Parallel binary tree search for one solution.

binsearch(Node node) {
    int s0 = 0, s1 = 0, s2 = 0;
    if(!node) return 0;
    if (Noticed that there are no solutions in the subtree the root of which is xnode?) {
        // “prune” the subtree that has the root xnode
        prune(xnode);
    }
    if(Is node included in a pruned subtree that has the root xnode?) {
        throw (xnode);
    }
    try {
        join {
            if(Solution found at node?) {
                s0 = 1;
                s1 = spawn binsearch(node->right);
                s2 = binsearch(node->left);
            }
        }
    } catch (node) {
        // catch an exception that has the tag value node
        ...
    }
    // return the number of solutions
    return s0+s1+s2;
}

Figure 6.4: Parallel binary tree search for all solutions where exceptions are used to reduce redundant search.
worker notices that no solutions exist in the subtree that has the root `xnode`, it prunes the subtree (lines 4–7). Then, when another worker notices that the worker is traversing the pruned subtree, it can abort that traversal by simply throwing an exception tagged with `xnode`, which is caught by the catcher (lines 19–22) established when `xnode` is visited as `node`. Note that, because of the collateral task abortion, the traversal of the subtree is completely aborted, even if part of the pruned subtree has been assigned to other workers. The implementation of these operations without exceptions is complicated: programmers need to implement the transfer of control back to the root of the pruned subtree, as well as the abortion of the relevant tasks, following their parent-child relationship upward and downward. Thus, it is expected that this exception handling mechanism will be useful for many backtrack search algorithms in practical applications, which often prune redundant subtrees to reduce the search space.

A semantics for such language features was proposed in [24]. In addition, Danaher et al. developed an LTC-based multithreaded language named JCilk [3] that offers exception handling with collateral task abortion. However, it is difficult to implement the parallel COPINE algorithm (even without abort operations) in an LTC-based language, as described in Section 4.3. Therefore, we implemented exception handling features with collateral task abortion as an enhancement of the existing task-parallel language Tascell [2]. That is, we designed an enhanced Tascell language by adding the try-catch and throw constructs to the baseline Tascell, and implemented this enhanced Tascell by modifying the Tascell compiler and the task scheduler. In addition, we evaluated our implementation in terms of overheads and time taken to abort tasks.

### 6.2 Language Extension

We added the try-catch and throw constructs to Tascell as statements. The syntax for these constructs is as follows:

- `try compound-statement_1
  catch (expression) compound-statement_2`

- `throw (expression);`

The try construct above does not have the “finally” clause for the finalization of the execution of try-catch construct no matter whether exceptions are raised, since Tascell already has the `dynamic_wind` construct for the finalization, as described in Section 3.4.

A try-catch construct indicates that an exception could be thrown during the execution of `compound-statement_1`. The evaluation steps of a try-catch construct are as follows. First, `expression` is evaluated and an exception catcher tagged with the resulting value (cast to `size_t`) is established. Then, `compound-statement_1` is executed. The catcher is disestablished when a worker exits this statement normally or abnormally.

A throw statement creates and throws an exception tagged with the value of `expression` (cast to `size_t`). When an exception tagged with `tag` is thrown, the most recent try block whose catcher is tagged with `tag` is forced to exit, and then `compound-statement_2` is executed. If no catcher tagged with `tag` has been established in the task being executed, the task terminates its execution, returning the exception as the result. This exceptional return is notified to the victim worker of the aborted task, and a partial cancellation flag with `tag` is attached to the corresponding parallel for statement in the victim. The victim then notices that its task has a
parallel `for` with the flag raised, and raises an exception as if the stolen iterations of the parallel `for` had been replaced with `throw(tag)`. In our current implementation, the existence of such a parallel `for` is checked at every entry point of any (other) parallel `for`, so that the victim notices this as soon as possible before becoming idle waiting for the completion of the parallel `for` in question, and eventually receives the exceptional return. Furthermore, we can guarantee that a worker does not abort a task in the middle of an atomic operation by limiting the task cancellation points to those entry points.

In addition, if there are unfinished parallel `for` statements in the dynamic scope of the try block to be aborted, cancellation messages are sent to thieves of unfinished tasks that were spawned from such parallel `for` statements. Each thief notices the message regarding the stolen task it is executing, by polling again for immediate abortion, and then aborts the task, possibly forwarding the message to other workers that have since stolen part of the task.

If uncompleted `dynamic_wind` statements exist in the dynamic scope of an exception catcher to be disestablished, cleanup operations defined as `statement` after are executed (in an innermost to outermost order, if `dynamic_winds` are nested) before exiting the corresponding `try` block, as in the temporary backtracking described in Section 3.4.

Figure 6.5 shows an example of a Tascell program using exceptions. In this program, a backtrack search is performed for Pentomino as in Figure 3.4, and the search is terminated as soon as a worker finds that the number of solutions is larger than `THRESHOLD`. Examples of exception handling are presented in Figure 6.6, which shows a cactus stack representing an execution context of the program in Figure 6.5 with four workers, assuming the following two cases.

a) When an exception tagged with `STOP`, which would be caught at `catch STOP` in task 0-0, is thrown by worker 0, cancellation flags are set to task 1-0, task 2-0, and its descendants, that is, tasks 3-0 and 2-1. Workers 1–3 notice the flags set to tasks 1-0, 2-1, and 3-0, respectively, and abort these tasks. After task 2-1 has been aborted, worker 2 resumes task 2-0, but immediately aborts it, as the worker notices the cancellation flag. The control of worker 0 returns to `catch STOP` after the cleanup operations at (1), (2), and (3) have been executed in that order.

b) When an exception tagged with `STOP` is thrown by worker 1 and not caught inside task 1-0, the exception return is notified to worker 0, raising the partial cancellation flag for the parallel `for` statement at join-a. After noticing the flag, worker 0 goes back to join-a and performs a throw operation for the returned tag `STOP` as if task 1-0 has been replaced with the throw. This exception is then caught at `catch STOP`. Because the parallel `for` corresponding to join-b has exited, tasks 2-0, 2-1, and 3-0 are aborted in the same manner as in Case a). Before the control of worker 0 returns to `catch STOP`, the cleanup operations at (1), (2), and (3) are executed (operations at (3) are executed after operations at (1) and (2)).

Note that a parallel `for` may have two or more exceptions at the time when the worker responsible for it notices the exceptions. If this occurs, one of them is chosen arbitrarily, with the others ignored because the parallel execution semantics allow us to do so.

If an exception is thrown during the execution of cleanup operations, the new exception is propagated from there, and the old one (if any) is discarded, as in Java [25]. Furthermore, a

---

2This program is used only as an example and for performance evaluations in Section 6.4. Obviously, this algorithm is not efficient, because a worker cannot count the number of solutions found by other tasks until it merges their results.
task pentomino {
  The same code as in Figure 3.4 lines 2–5
};
task_exec pentomino {
  this.s = 0;
  if(The root of the search?) {
    try {
      search(this.k, this.i0, this.i1, this.i2, &this);
    } catch(i) {
      this.s = 1;
    }
  } else {
    this.s = search(this.k, this.i0, this.i1, this.i2, &this);
  }
}
task_exec worker int search(int k, int j0, int j1, int j2, task pentomino *tsk) {
  int s=0; // the number of solutions
  // parallel for
  for(int p : j1, j2) {
    The same code as in Figure 3.4 lines 15–33
  } handle pentomino (int i1, int i2) {
    The same code as in Figure 3.4 lines 38–47
  }
  // throw an exception if s exceeds THRESHOLD
  if(s > THRESHOLD) throw 1;
  return s;
}

Figure 6.5: Tascell program that performs backtrack search for Pentomino and terminates the search as soon as a worker finds that the number of solutions is larger than THRESHOLD.

Figure 6.6: Execution context of the program in Figure 6.5.
task may not be aborted (partially) because of a (partial) cancellation flag during the execution of cleanup operations. Note that, in our current implementation, such an abortion does not occur because we limit the cancellation points to entry points of parallel for statements and Tascell does not allow a parallel for statement to be executed during the execution of cleanup operations.

6.3 Implementation of Exception Handling

We implemented the Tascell exception handling mechanism presented in Section 6.2 by modifying the Tascell compiler and the task scheduler provided by Tascell, which are presented in Sections 6.3.1 and 6.3.2, respectively.

6.3.1 Tascell Compiler

Since the Tascell compiler is implemented as a translator for C code, we require some techniques to implement the non-local exit mechanism with cleanup code execution in the “finally” clauses. There are various techniques for implementing such a mechanism as a translator to C, such as the setjmp method and two return values method [26]. However, to minimize the implementation cost and additional overheads to the baseline Tascell, we exploited nested functions, which are already used for the temporary backtracking mechanism.

The functions task_exec and search in Figure 6.5 are translated to the programs in Figures 6.7 and 6.8, respectively. Each try-catch statement is translated into a piece of code that includes a definition of a nested function (lines 19–42 in Figure 6.7), as well as parallel for and dynamic_wind statements. These nested functions are called in the order of newest to oldest for propagating an exception (line 54 in Figure 6.8), aborting a task (line 36), or spawning a task (line 40). Temporary backtracking for spawning tasks is executed in the same manner as in the baseline Tascell, as explained in Section 3.5.2. During backtracking for propagating an exception, a worker executes cleanup operations in nested functions derived from dynamic_winds, aborts, and waits for tasks spawned at parallel for statements (lines 15–22 in Figure 6.8). When the worker reaches a nested function derived from a try-catch statement, the catcher tag of which is equal to the tag of the thrown exception, it exits the try block by exiting the nested function using goto (see Section 3.5.1 for this capability of nested functions). If the exception is not caught in the task being executed, the nested function located at the termination of the backtracking (lines 4–14 in Figure 6.7) is called to exit pentomino_task_exec. After exiting pentomino_task_exec, the scheduler notices that the task has exited with an exception return from the fact that _thr->backtrack_rsn is EXCEPTION. Backtracking for aborting a task is conducted in a similar manner as for propagating an exception, except that the check for the exception tag value is unnecessary in a try-catch statement (lines 35 and 36 in Figure 6.7), and a worker retrieves and rethrows an exception if a task spawned at a parallel for statement has returned the exception (lines 17–20 in Figure 6.8).

6.3.2 Task Scheduler

To support the exception handling mechanism, we enhanced the task scheduler of Tascell as follows:
void pentomino_task_exec(struct thread_data *thr, struct pentomino *pthis)
{
    __label__ pentomino_abort;

    void _bk(void) {
        switch(_thr->backtrack_rsn) {
            EXCEPTION:
                goto pentomino_abort; // exit the task
            CANCEL:
                // the exception is not caught in the task, or the task is aborted by a cancellation flag
                goto pentomino_abort; // exit the task
            SPAWN:
                // _bk is called for spawning tasks (temporary backtracking)
                return; // return from the backtracking.
        }
    }

    pthis->s = 0;
    if(The root of the search?) {
        /* ---------------------- try-catch ----------------------*/
        __label__ catch_exit;
        void catch_bk(void) {
            switch(_thr->backtrack_rsn) {
                EXCEPTION:
                    // check if the exception tag is equal to the catcher tag
                    if(_thr->excep_tag == 1) {
                        _thr->backtrack_rsn = NOT_BACKTRACKING;
                        // execute the code in the catch block
                        pthis->s = 1;
                    }
                    goto catch_exit; // exit the try block
                } else {
                    // continue backtracking since the exception is not caught
                    _bk(); // never returns
                    system_error();
                }
                break;
            CANCEL:
                // continue backtracking (never returns)
                system_error();
            SPAWN:
                _bk();
                return;
            }
        }
        /* ---------------------- try-catch ----------------------*/
        search(catch_bk, _thr, pthis->k, pthis->i0, pthis->i1, pthis->i2, pthis);
        catch_exit;;
    } else {
        pthis->s = search(_bk, _thr, pthis->k, pthis->i0, pthis->i1, pthis->i2, pthis);
    }
    pentomino_abort;;
}
```c
int search(void(*_bk0)(void), struct thread_data *_thr, int k, int j0, int j1, int j2,
           struct pentomino *tsk)
{
    int s = 0; // the number of solutions
    /*------------------------ parallel for ------------------------*/
    int p = j1; int p_end = j2;
    struct pentomino *pthis;
    int spawned = 0; // the number of spawned tasks
    void _bk1_par_for(void) { // nested function
        switch(_thr->backtrack_rsn) {
        EXCEPTION:
            CANCEL:
                // the reason for backtracking is an exception or a cancellation flag
                // abort and synchronize with all the tasks spawned at this parallel for
                while(spawned-- > 0) {
                    pthis = (struct pentomino *)abort_and_wait(_thr);
                    if(pthis has a returned exception?) {
                        _thr->backtrack_rsn = EXCEPTION;
                        _thr->excep_tag = pthis->excep_tag;
                    }
                    free(pthis);
                }
                _bk0(); // continue backtracking (never returns)
                system_error();
        SPAWN:
            // the reason for backtracking is a task request
            if(!spawned) _bk0(); // continue backtracking
            while(p + 1 < p_end &&
                   task request exists?) {
                (The same code as Figure 3.6 lines 12–22 (make and spawn a task))
            }
            return;
        }
    }
    if(_thr->task_top->cncl) { // check whether (partial) cancellation flags are set
        _thr->backtrack_rsn = CANCEL;
        _bk1_par_for(); // start backtracking for aborting the task
    }
    if(_thr->req) { // check task requests
        _thr->backtrack_rsn = SPAWN;
        _bk1_par_for(); // start backtracking for spawning tasks
    }
    for(; p < p_end; p++) {
        (The same code as Figure 3.6 lines 28–55 (loop body))
    }
    while(spawned-- > 0) {
        (The same code as Figure 3.6 lines 58–61)
        (Get and integrate results of spawned tasks)
    }
    /*------------------------ parallel for ------------------------*/
    if(s > THRESHOLD) {
        /*------------------------ throw ------------------------*/
        _thr->backtrack_rsn = EXCEPTION;
        _thr->excep_tag = 1; // set the exception tag value
        _bk1_par_for(); // start backtracking for propagating the exception
    }
    /*------------------------ throw ------------------------*/
    return s;
}
```

Figure 6.8: Translation result from the worker function search for Pentomino in Figure 6.5, including translation of a parallel for statement and a throw statement.
• We implemented task cancellation and partial cancellation flags in parallel for statements, and

• enhanced the message handler among workers so that a worker can return an exception as the result of a task when the exception is not caught inside the task, and can notify the abortion of a task for which a cancellation flag is set.

When a worker returns an exception as a task result, a partial cancellation flag is set in the parallel for statement at which the task was spawned. In addition, cancellation flags are set in all the tasks that were spawned at the parallel for statement and all the parallel for statements dynamically enclosed by it.

Cancellation flags are also set when a worker backtracking to propagate an exception reaches a parallel for statement; flags are set in all the tasks spawned at that statement (line 16 in Figure 6.8). In addition, when a cancellation flag is set in a task, flags are set in all the tasks spawned during its recursive execution.

At every entry point of a parallel for statement, a worker checks whether a cancellation flag is set in a task being executed and partial cancellation flags are set in parallel for statements in the task. If any flags exist, the worker (partially) aborts the task by calling nested functions (lines 34–37 in Figure 6.8).

Note that a task for which a (partial) cancellation flag is set may be suspended, such as task 2-0 in Figure 6.6, but workers cannot notice such a flag only by checking for a task being executed. However, we can guarantee that such a task will become active immediately, after other active tasks have been aborted. This is because, as a result of the Leapfrogging employed by the current Tascell implementation (Section 3.3.2), a suspended task is always an ancestor of a task being executed, and we implemented the scheduler so that, when a flag is set to a task, flags are automatically set to all its descendant tasks. Thus, it is only necessary to allow each worker to check for a task being executed.

6.4 Performance Evaluation

6.4.1 Evaluation Setup

We evaluated our implementation of the enhanced Tascell using the following programs:

\textbf{Fib}(n) \text{ recursively computes the } n\text{-th Fibonacci number.}

\textbf{Nq}(n) \text{ finds all solutions to the } n\text{-queens problem. In Tascell, this is coded with a combination of a parallel for and a dynamic wind in the same way as for Pentomino.}

\textbf{Pen}(n) \text{ finds all solutions to the Pentomino problem with } n\text{ pieces, using additional pieces and an expanded board for } n > 12.

The evaluation environment is the same as that in Table 4.1.

\footnote{To avoid the costly operations of checking all the parallel for statements periodically, we implement a task object that has a counter for partial cancellation flags.}
Table 6.1: Comparison of execution time and relative time to sequential C programs with one worker between the baseline and the enhanced Tascell.

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>Tascell (baseline)</th>
<th>Tascell (enhanced)</th>
<th>Tascell (enhanced, w/ try)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib(51)</td>
<td>54.3 (1.00)</td>
<td>208 (3.82)</td>
<td>203 (3.74)</td>
<td>432 (7.97)</td>
</tr>
<tr>
<td>Nq(17)</td>
<td>464 (1.00)</td>
<td>476 (1.03)</td>
<td>489 (1.05)</td>
<td>540 (1.16)</td>
</tr>
<tr>
<td>Pen(15)</td>
<td>685 (1.00)</td>
<td>640 (0.933)</td>
<td>630 (0.920)</td>
<td>738 (1.07)</td>
</tr>
</tbody>
</table>

### 6.4.2 Overheads

We evaluated the overheads of the exception handling mechanism by measuring the performance of the baseline and enhanced implementations of Tascell using Fib(n), Nq(n), and Pen(n). In addition, to evaluate the cost of the exception handlers, we measured the performance of programs that perform the same computations as Fib(n), Nq(n), and Pen(n), respectively, but where the entire body of each recursive function is enclosed by an unused try block. We also compared the performance of each implementation with that of the sequential programs written in C.

The measurement results are presented in Table 6.1 (sequential executions) and Figure 6.9 (parallel executions). We can see that the overheads of the exception handling mechanism itself, including checking cancellation flags and additional operations in nested functions called when spawning tasks, e.g., checking the reason for backtracking, are less than 6.2% for all measurement conditions. Note that the overheads are very small, even for Fib(51), which performs the checking for cancellation flags very frequently. Furthermore, except for Fib(51), the cost of the try blocks is relatively small: the performance degradation as compared to the baseline Tascell is less than 16% for Nq(17) and Pen(15). According to these results, we expect that the technique could be used to abort the redundant search shown in Figure 6.4, which requires that an exception handler be created at every search step, without excessive overheads.

For Pen(15), the performance of the C program is worse than that of Tascell. Although we cannot be certain, this may have been caused by a bad optimization of GCC.

### 6.4.3 Task Abortion Time

We evaluated the time taken to abort tasks by measuring the performance of programs that perform the same computation as Fib(n), Nq(n), and Pen(n), respectively, but terminate the computation by throwing an exception as soon as a worker finds that the answer is larger than a threshold, as shown in Figure 6.5 for Pen(n). The threshold is set to $\alpha \cdot A$, where $A$ is the true answer of the computation and $\alpha$ is set to 0–0.3 in units of 0.01 and 0.4–1 in units of 0.1 (the computation terminates without exceptions when $\alpha = 1$). We executed these programs using 2 and 16 workers, and measured the total elapsed time ($T$) and the elapsed time before the first exception was thrown ($T_{\text{throw}}$).

The measurement results are shown in Figure 6.10. In addition, for the 16-worker executions,
Figure 6.9: Comparison of speedup relative to C between the baseline and the enhanced Tascell.
Figure 6.10: Elapsed time between the time when an exception is thrown and the termination of the computation.

the elapsed time between the first throw operation and the termination of the program execution ($T - T_{\text{throw}}$) and the number of aborted tasks are shown in Figure 6.11. The number of aborted tasks includes those that terminated by returning exceptions and those aborted collaterally by cancellation messages from their parents.

We can see that the abortion time increases in proportion to the number of aborted tasks, but is very short (less than 500 $\mu$s in all executions), even when an exception is thrown in the middle of the execution and tens of tasks are aborted collaterally.

6.5 Conclusion

In this chapter, we extended the task-parallel language Tascell by adding exception handling features in which all running parallel tasks in a try block with an exception are collaterally aborted as soon as possible. We implemented the non-local exit mechanism by exploiting nested
Figure 6.11: Elapsed time between the first throw operation and the termination of the program execution, and the number of aborted tasks (16-worker executions).
functions, which are already used for the temporary backtracking mechanism of Tascell. We also modified the task scheduler provided by Tascell so that a worker can abort a task that is being executed. The implementation achieved an exception mechanism with low overheads and short task abortion time.

In Chapter 7, we attempt to improve the performance of the parallel COPINE algorithm using the exception handling features to reduce the number of redundant searches.
Chapter 7
Abortion of Redundant Search

7.1 Introduction
This chapter describes a mechanism to abort redundant searches in COPINE by exploiting exceptions. In parallel implementations, it is virtually impossible to perform pruning perfectly, because one worker may traverse a search space that is later pruned by another worker. Thus, the search space in the parallel search becomes larger than that in the sequential search, even with our attempts to minimize the possibility of such traversals starting before the subtrees have been pruned. In fact, the search space size with 16 workers was found to be approximately twice as large as that with one worker (see Chapter 4). This expansion seriously affects performance.

To improve the performance of our parallel solver, it is necessary to reduce the number of redundant searches more aggressively. One possible approach is to abort the execution of a worker traversing a subtree pruned by another worker. These observations motivated us to introduce the abortion of redundant searches into our baseline implementation.

7.2 Implementation of Abort Operations
To implement abort operations to reduce redundant searches, we first need to detect that a worker is performing a redundant search. A worker $w_0$ executing a task $t$ can detect that another worker $w_1$ executing a task $t'$ is performing a redundant search when $w_0$ registers an itemset $I_i$ to the entry corresponding to a vertex $v_i$ in the itemset table if all of the following conditions are satisfied:

1. a subset $I'_i$ of $I_i$ is already registered in the entry,
2. minID of the task $t'$ attached to $I'_i$ is greater than minID of $t$, and
3. the worker $w_1$ executing $t'$ is traversing the subtree whose root is $v_i$.

To enable $w_0$ to recognize condition (3), when the worker $w_1$ visits the vertex $v_i$ and registers the itemset $I'_i$ to the itemset table entry corresponding to $v_i$, it attaches a flag to the registry object to indicate that the worker is traversing the subtree whose root is $v_i$. This flag is removed when $w_1$ completes the traversal of the subtree.

When a worker performing a redundant search is detected, it is notified to abort the search. Figure 7.1 illustrates the worker $w_1$ executing the task $t'$ by traversing the subtree whose root is $v_i$, for which the common itemset $I'_i$ was registered. When the worker $w_0$ executing a task $t$ registers an itemset $I_i$ to the entry corresponding to $v_i$ in the itemset table, it detects that $w_1$ is performing a redundant search by checking the three conditions above (Step 1 in Figure 7.1). $w_0$
then notifies \( w_1 \) to abort the search by inserting \( v_i \) into the exception queue for \( w_1 \) (Step 2).

Each worker has its own exception queue, and checks whether the queue contains any elements at every visit to a vertex. When \( w_1 \) finds an element \( v_i \) in the queue, it recognizes that it is executing a redundant search (Step 3) and aborts the search by transferring its control back to the search tree node from which \( v_i \) was last visited, i.e., the root of the redundant subtree (Step 4). After this abort operation, \( w_1 \) resumes its search from the parent of the root of the redundant subtree (Step 5).

Figure 7.2 shows the pseudo-code for COPINE with abort operations (\( EnumCCIG() \) has been omitted). The variable \( w_c \) has the worker executing the function, and \( w_c.Q_e \) denotes the exception queue of a worker \( w \). The function \( index(q_j, T) \) returns the index number of \( q_j \), being the root vertex of a subtree whose search is to be aborted, in the vertex sequence \( T \) of CST formed at the function is called. By exploiting exception handling, we can write this algorithm by making only small changes to Figure 2.6.

In line 21, the worker \( w_c \) raises the flag \( traversing \) to indicate that the subtree rooted by \( c \) is being traversed by \( w_c \) after registering \( I_c \) for \( c \). This flag is removed in the “finally” clause on line 25 to ensure its removal even in the case of abnormal exits of the try block. This raised flag may be seen by another worker \( w'_c \) in its execution of lines 15–17, in which it is examined whether there are any workers traversing redundant subtrees rooted by \( c \) satisfying conditions (1)–(3) described above. If the worker \( w_c \) is among them, \( c \) is added to its exception queue as the notification of the fact that \( w_c \) is traversing a redundant subtree. The worker \( w_c \) then checks its exception queue to find the exception (line 2). If there are one or more elements in the queue, \( w_c \) chooses the element closest to the root, i.e., the topmost one in the sequence \( T \), from the queue and throws an exception tagged with the vertex (lines 3 and 5), removing all elements in the
function ExploreCCIG(v, T, I, V, E, C, L) begin
if $|w_c.Q_c| > 0$ then
q ← q_l s.t. $q_l \in w_c.Q_c$, $i = \arg \min_j \{\text{index}(q_j, T) \mid \forall q_j \in w_c.Q_c\}$;

$w_c.Q_c \leftarrow \emptyset$;
throw(q);
closed ← true;
N ← sort(neighbors(v, E));
C ← $N \cdot C$;
while C ≠ ∅ do begin

c ← car(C);
C ← cdr(C);
if c /∈ V then continue;
V ← V − {c};
$I_c \leftarrow I \cap I(c)$;
if $|I_c| < \theta$ then continue;
if \exists $I'$ s.t. $I' \in c.I$ \and $I' \supseteq I_c \and \text{minID} \geq I'.\text{minID}$ then continue;
for $I' \in c.I$ s.t. $I' \subseteq I_c \and \text{minID} < I'.\text{minID} \and I'.\text{traversing} \neq \epsilon$ do begin

w ← I'.traversing; $w.Q_c \leftarrow w.Q_c \cup \{c\};$
end

$c.I \leftarrow c.I \cup \{I_c\}$;
if $I_c = I$ then closed ← false;
try {
c.I..traversing ← $w_c$;
L ← ExploreCCIG(c, $T \cdot \langle c \rangle$, $I_c$, V, E, C, L);
} catch c {
} finally {
$c.I..traversing \leftarrow \epsilon$;
$w_c.Q_c \leftarrow w_c.Q_c \setminus \{c\}$;
}
end
if closed then L ← L ∪ \{T\};
return(L);
end

Figure 7.2: Pseudo-code for the COPINE algorithm with abort operations.

queue (line 4). In addition, the exception is also removed from the queue in the “finally” clause (line 26) so as to ensure the removal even when the exception is inserted at the very last phase of the try block execution, missing the chance of a corresponding throw. If the exception queue element for $c$ caused the throw, the corresponding try block with the catch tag $c$ exits abnormally to abort the traversal of the subtree rooted by $c$.

Note that an execution of the parallel while loop (lines 9–28) may be terminated abnormally by exceptions because the loop is in the try block of the caller. If this happens, some iterations of the loop may have been assigned to other workers as their tasks, meaning that some portions

---

1The current implementation of Tascell ensures that a worker has a set of tasks mutually enclosed within their dynamic scope. Therefore, we may simply choose the topmost vertex in $T$, ignoring all others. If this assurance is not provided, we must retain queue elements that are not in $T$ and, if there are no elements in $T$, we have to skip the throw operation.
while(1) {
    uint64_t send_buf[], recv_buf[];
    sleep(t_comm);
    // The same code as Figure 5.1 lines 4–13 (extract and exchange table updates)
    for(k=0; k<# of itemsets in recv_buf; k++) {
        e = the k-th element in recv_buf;
        i = the vertex number in e;
        pthread_mutex_lock(&table_entry_lock[i]);
        for(l=0; l<# of itemsets in vi.I; l++) {
            l' = the l-th itemset in vi.I;
            if(l' ⊆ e.I && e.minID < l'.minID && l'.traversing ≠ e) {
                Insert vi into the exception queue for w.
            }
        }
        pthread_mutex_unlock(&table_entry_lock[i]);
    }
}

Figure 7.3: Flow of the communication thread for the detection of workers performing a redundant search.

of the redundant subtree have been assigned to these workers. Therefore, such tasks should be aborted as soon as possible. Furthermore, it may be the case that the worker \( w_c \) throws the exception in its execution of a child task assigned by another worker \( w_p \) and within the dynamic scope of the try block to be aborted. In this case, the worker \( w_c \) must not only abort the task, but also notify \( w_p \) that its parent task should be aborted as well, because the topmost try block of the child task in question cannot catch the exception.

The exception handling features in Tascell discussed in Chapter 6 support a mechanism to perform such collateral abort operations automatically. Without this mechanism, we would have to transfer control back to the root of a redundant subtree by repetitive function returns that consider any necessary undo operations. This coding is not especially difficult, but reduces the readability of the code. Furthermore, it is difficult to implement collateral task abortions at the program level, because implementing such operations requires the modification of the task scheduler provided by a task-parallel language. Tascell simplifies the description of cleaning up operations in the “finally” clause executed at collateral task abortions, as it has a dynamic\_wind construct similar to that in the Scheme language [13].

In distributed memory environments, we exploit the sharing method described in Section 5.2 to implement abort operations. As shown in Figure 7.3, the communication thread in each computing node detects a worker traversing a subtree pruned by another worker in a different computing node using received table updates (lines 10–16). The thread inserts the roots of the redundant subtrees into exception queues for the detected workers (line 14).
Table 7.1: Evaluation environment on Cray XC30.

<table>
<thead>
<tr>
<th></th>
<th>Cray XC30 (1 node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5-2695v3 (Haswell) 2.3GHz 14-core × 2 (28 cores in total per node)</td>
</tr>
<tr>
<td>Memory</td>
<td>DDR4-2133 64GB</td>
</tr>
<tr>
<td>OS</td>
<td>SUSE Linux Enterprise Server 11</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC 4.3.4 with -O3</td>
</tr>
<tr>
<td>Nested functions</td>
<td>Trampoline-based implementation in GCC</td>
</tr>
<tr>
<td>Worker</td>
<td>Created by pthread_create with PTHREAD_SCOPE_SYSTEM</td>
</tr>
<tr>
<td>Lock</td>
<td>A pthread_mutex_t lock is attached to each entry</td>
</tr>
</tbody>
</table>

7.3 Performance Evaluation

7.3.1 Evaluation Setup

We evaluated the improvement in the effectiveness of Pruning 3 with abort operations by measuring the performance of the baseline and the new implementation described in Section 7.2. We used a Cray XC30 and the Appro GreenBlade 8000 supercomputers for the evaluations. The evaluation environment in the Cray XC30 is summarized in Table 7.1. The environment for the Appro GreenBlade is the same as that in Table 4.1.

7.3.2 Performance in Shared Memory Environments

We evaluated the performance of each implementation on a single computing node of the Cray XC30 using the following three types of artificial graphs and the real protein network.

Random($N, M$) is a graph that has $N$ vertices and approximately $2MN$ edges, where each vertex picks $M$ neighbors uniformly at random.

Hypercube($N$) is a hypercube graph that has $2^N$ vertices with degree $N$.

2D-torus($N$) is a two-dimensional torus that has $N \times N$ vertices with degree 4.

Protein is the real protein network used for the evaluations in Sections 4.5 and 5.4.

Table 7.2 shows the characteristics of the artificial graphs. We associated each artificial graph with a 32-component itemset. For each pair of a vertex $v$ and an item $i$, we randomly generated 0 or 1 with equal probability, and associated $i$ with $v$ if a 1 was generated. The threshold $\theta$ was set to $\theta = 3$. We set the number of iterations left for a victim worker from unexecuted iterations as 100 for Random($1500, 12$) and Hypercube($13$), half of the unexecuted iterations for 2D-torus($500$), and 2 for Protein, in consideration of the trade-off between the effectiveness of Pruning 3 and the traversal speed. We also compared the performance of each implementation with that of the sequential COPINE implementation written in C.

Figures 7.4–7.7 illustrate the execution time and number of visits to vertices for each of the implementations and executions with varying numbers of workers. The detailed results obtained from the evaluation using Protein are presented in Table 7.3. Note that, in Table 7.3, the
Table 7.2: Characteristics of the artificial graphs.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Random(1500, 12)</th>
<th>Hypercube(13)</th>
<th>2D-torus(500)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>V</td>
<td>$</td>
<td>1,500</td>
</tr>
<tr>
<td>$</td>
<td>E</td>
<td>$</td>
<td>35,721</td>
</tr>
<tr>
<td>$</td>
<td>I</td>
<td>$</td>
<td>32</td>
</tr>
<tr>
<td>Average degree</td>
<td>23.8</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>Diameter</td>
<td>4</td>
<td>13</td>
<td>500</td>
</tr>
<tr>
<td>ADST ($\theta = 3$)</td>
<td>37.9</td>
<td>157</td>
<td>3.81</td>
</tr>
<tr>
<td>#V in LCC</td>
<td>1,500</td>
<td>8,192</td>
<td>250,000</td>
</tr>
<tr>
<td>#V in SCC</td>
<td>1,500</td>
<td>8,192</td>
<td>250,000</td>
</tr>
<tr>
<td>Average # of items in each vertex</td>
<td>16.0</td>
<td>16.1</td>
<td>16.0</td>
</tr>
</tbody>
</table>

ADST: Average depth of search tree in the sequential search
#V in LCC: # of vertices in the largest connected component
#V in SCC: # of vertices in the smallest connected component

aborted tasks include those that terminate returning exceptions and tasks aborted by cancellation messages from their parents.

For **Protein**, the one-worker execution time without abort operations was approximately 51.5% worse (40.8 s longer) than that of the sequential implementation in C. This is mainly due to the cost of the Tascell mechanisms. As for the execution with abort operations, further time overheads are incurred for the additional information attached to the itemset table for abortion, although no tasks were aborted.

The most fundamental feature of the parallel executions is how the load is balanced among the workers. From the $(t_{max} - \bar{t})/\bar{t}$ results in Table 7.3, we can see that good load balancing was achieved both with and without abort operations.

The traversal speeds in the #V/s column of Table 7.3 in executions with abort operations are lower than those without abort operations. This is due to the costs associated with abort operations, such as the management of exception queues, and the cost of adding information to the itemset table. Of these costs, we have confirmed that the cost of abort operations is not serious. For instance, in the 28-worker execution with abort operations, the cumulative time of the abort operations of all workers was 0.343 s, which is 1.49% of the cumulative total execution time ($= 28 \times 5.87s$).

Comparing the results with and without abort operations, we can see that the total number of visits to vertices is greatly reduced by abort operations, confirming the improved effect of Pruning 3 for **Random**(1500, 12), **Hypercube**(13), and **Protein**. Figures 7.4, 7.5, and 7.7 show that the search space has a tendency to enlarge as the number of workers increases, but this enlargement is alleviated by the use of abort operations. For instance, in the 28-worker executions, we reduced the search space by 27.1%, 17.9%, and 31.9% for **Random**(1500, 12), **Hypercube**(13), and **Protein**, respectively.

In contrast, Figure 7.6 shows that, for **2D-torus**(500), the search space only becomes slightly larger as the number of workers increases, and thus the abort operations have little effect. Considering Table 7.2, we can see that this is because the small degree of this graph means that the depth tends to be shallow, and thus the number of redundant searches is very small, even in the parallel search.
Figure 7.4: Execution time and number of visits to vertices for Random\(1500, 12\) with and without abort operations.

Figure 7.5: Execution time and number of visits to vertices for Hypercube(13) with and without abort operations.
Figure 7.6: Execution time and number of visits to vertices for 2D-torus (500) with and without abort operations.

Figure 7.7: Execution time and number of visits to vertices for Protein with and without abort operations.
Table 7.3: Results of performance evaluation for the real protein network with and without abort operations.

<table>
<thead>
<tr>
<th>Impl.</th>
<th>$w$</th>
<th>ET</th>
<th>SU/C</th>
<th>SU/1</th>
<th>$\frac{t_{\text{max}} - \bar{t}}{\bar{t}}$</th>
<th>#V</th>
<th>#V/s</th>
<th>#TC</th>
<th>#TE</th>
<th>#AT</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>79.2</td>
<td>1</td>
<td>—</td>
<td>0</td>
<td>1,795</td>
<td>22.7</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>68.0</td>
<td>1.15</td>
<td>1.74</td>
<td>0.008</td>
<td>2,023</td>
<td>14.9</td>
<td>116</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>48.3</td>
<td>1.64</td>
<td>2.48</td>
<td>0.01</td>
<td>2,434</td>
<td>12.6</td>
<td>743</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>w/o abort op.</td>
<td>8</td>
<td>40.5</td>
<td>1.96</td>
<td>2.96</td>
<td>0.01</td>
<td>3,472</td>
<td>10.7</td>
<td>2,055</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>37.5</td>
<td>2.11</td>
<td>3.20</td>
<td>0.05</td>
<td>3,994</td>
<td>8.86</td>
<td>39,666</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>35.2</td>
<td>2.25</td>
<td>3.41</td>
<td>0.07</td>
<td>4,271</td>
<td>7.59</td>
<td>68,673</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>34.4</td>
<td>2.30</td>
<td>3.49</td>
<td>0.03</td>
<td>5,575</td>
<td>8.11</td>
<td>24,665</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>32.0</td>
<td>2.48</td>
<td>3.75</td>
<td>0.05</td>
<td>5,777</td>
<td>7.52</td>
<td>24,664</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>31.7</td>
<td>2.50</td>
<td>3.79</td>
<td>0.1</td>
<td>6,070</td>
<td>6.83</td>
<td>46,434</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>w/ abort op.</td>
<td>1</td>
<td>135</td>
<td>0.587</td>
<td>1</td>
<td>0</td>
<td>1,795</td>
<td>13.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>80.2</td>
<td>0.988</td>
<td>1.68</td>
<td>0.007</td>
<td>3,013</td>
<td>12.6</td>
<td>116</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>52.7</td>
<td>1.50</td>
<td>2.56</td>
<td>0.006</td>
<td>2,341</td>
<td>11.1</td>
<td>935</td>
<td>321</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>39.1</td>
<td>2.03</td>
<td>3.45</td>
<td>0.02</td>
<td>2,955</td>
<td>9.44</td>
<td>8,305</td>
<td>3,064</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>35.1</td>
<td>2.26</td>
<td>3.85</td>
<td>0.04</td>
<td>3,532</td>
<td>8.39</td>
<td>14,246</td>
<td>5,641</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>31.3</td>
<td>2.53</td>
<td>4.31</td>
<td>0.04</td>
<td>3,682</td>
<td>7.36</td>
<td>20,260</td>
<td>7,185</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>28.5</td>
<td>2.78</td>
<td>4.74</td>
<td>0.09</td>
<td>3,808</td>
<td>6.68</td>
<td>39,564</td>
<td>9,279</td>
<td>413</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>25.0</td>
<td>3.17</td>
<td>5.40</td>
<td>0.07</td>
<td>3,996</td>
<td>6.70</td>
<td>29,765</td>
<td>9,501</td>
<td>264</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>23.0</td>
<td>3.44</td>
<td>5.87</td>
<td>0.04</td>
<td>4,131</td>
<td>6.40</td>
<td>25,723</td>
<td>7,950</td>
<td>222</td>
</tr>
</tbody>
</table>

$w$: # of workers; ET: Execution time [s]; SU/C: Speedup (vs. C) 
SU/1: Speedup (vs. 1 worker) 
$t_{\text{max}}$: the maximum cumulative time spent in task executions among all workers 
$\bar{t}$: the average cumulative time spent in task executions among all workers 
#V: # of visits to vertices [$\times 10^6$] (total of all workers) 
#V/s: # of visits to vertices [$\times 10^6$]/s (average among all workers) 
#TC: # of task creations; #TE: # of thrown exceptions; #AT: # of aborted tasks

In summary, the execution time is considerably improved by parallel executions with abort operations, except for sparse graphs, because the reduction in the search space outweighs the cost of the abort operations. However, the degree of speedup is limited by the number of remaining redundant searches and the increase in the cost of task creations and mutual exclusion locks for itemset tables, which are reflected by the decrease in #V/s (see Table 7.3). In fact, the execution time with 28 workers is reduced by 27.4% for Protein. Furthermore, note that the difference between the execution time with and without abort operations becomes greater as the number of workers increases.
Table 7.4: Performance for Protein ($\theta = 2$) with abort operations when changing the values of $\tau$ and $t_{sb}$ (using 8 nodes $\times$ 16 workers).

<table>
<thead>
<tr>
<th>$t_{sb}$</th>
<th>0 s</th>
<th>1 ms</th>
<th>10 ms</th>
<th>100 ms</th>
<th>1 s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>56.1</td>
<td>52.4</td>
<td>51.7</td>
<td>52.8</td>
<td>50.6</td>
</tr>
<tr>
<td></td>
<td>(50.8 ± 16.5)</td>
<td>(52.7 ± 10.1)</td>
<td>(50.8 ± 9.28)</td>
<td>(53.3 ± 10.9)</td>
<td>(50.8 ± 21.3)</td>
</tr>
<tr>
<td>2</td>
<td>47.9</td>
<td>47.8</td>
<td>45.2</td>
<td>45.4</td>
<td>44.4</td>
</tr>
<tr>
<td></td>
<td>(62.2 ± 11.6)</td>
<td>(58.5 ± 12.8)</td>
<td>(57.4 ± 11.1)</td>
<td>(59.1 ± 14.0)</td>
<td>(62.7 ± 9.74)</td>
</tr>
<tr>
<td>4</td>
<td>63.3</td>
<td>61.8</td>
<td>59.2</td>
<td>57.5</td>
<td>60.5</td>
</tr>
<tr>
<td></td>
<td>(57.9 ± 13.4)</td>
<td>(60.2 ± 14.3)</td>
<td>(66.1 ± 8.99)</td>
<td>(62.4 ± 12.4)</td>
<td>(56.1 ± 11.7)</td>
</tr>
<tr>
<td>8</td>
<td>73.9</td>
<td>73.4</td>
<td>70.1</td>
<td>69.8</td>
<td>71.9</td>
</tr>
<tr>
<td></td>
<td>(48.2 ± 11.9)</td>
<td>(46.0 ± 12.9)</td>
<td>(45.8 ± 10.3)</td>
<td>(55.8 ± 12.7)</td>
<td>(49.7 ± 14.0)</td>
</tr>
<tr>
<td>16</td>
<td>90.6</td>
<td>86.7</td>
<td>85.5</td>
<td>80.1</td>
<td>83.7</td>
</tr>
<tr>
<td></td>
<td>(32.5 ± 10.0)</td>
<td>(30.3 ± 9.80)</td>
<td>(37.7 ± 9.52)</td>
<td>(41.1 ± 11.7)</td>
<td>(35.9 ± 12.5)</td>
</tr>
</tbody>
</table>

### 7.3.3 Performance in Distributed Memory Environments

We also evaluated the implementation with abort operations by measuring the performance for Protein ($\theta = 2$) with the Appro GreenBlade 8000 on multiple computing nodes. We found the optimal performance values of $\tau$ and $t_{sb}$, which have a significant influence on the parallel performance in distributed memory environments, by the same optimal parameter search described in Section 5.4.2 using 8 nodes $\times$ 16 workers. That is, we tried all combinations of ($\tau, t_{sb}$) that satisfy $\tau \in \{1, 2, 4, 8, 16\}$ and $t_{sb} \in \{0 \text{ s, 1 ms, 10 ms, 100 ms, 1 s}\}$, while fixing $t_{comm}$ and $\kappa$ to 100 ms and 16, respectively.

Table 7.4 lists the execution times, rates of search space expansion, and task execution rates with these settings. We obtained the shortest execution time when $\tau = 2$ and $t_{sb} = 1$ s while suppressing the search space expansion and improving the task execution rate. Therefore, we set $\tau$ and $t_{sb}$ to 2 and 1 s for this evaluation, respectively.

The evaluation results are presented in Table 7.5. We can see that the total number of visits to vertices has also been reduced by the abort operations in distributed memory environments. However, the degree of reduction decreases with multiple computing nodes. The degradation of the task execution rate, which is observed in the measurements both with and without abort operations, reduces the performance improvement given by the abort operations. That is, it decreases the degree of parallelism, and thus improves the completeness of Pruning 3.

As a result, we could not improve the execution time by using abort operations in distributed memory environments. For further performance improvement, we need to devise a method to improve the task execution rate.
Table 7.5: Results of performance evaluation for Protein ($\theta = 2$) with abort operations in distributed memory environments.

<table>
<thead>
<tr>
<th>Impl.</th>
<th>$n$</th>
<th>$w$</th>
<th>ET</th>
<th>SU/C</th>
<th>SU/1</th>
<th>TER</th>
<th>#V</th>
<th>#V/s</th>
<th>#RI</th>
<th>#TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>147</td>
<td>1</td>
<td>—</td>
<td>100</td>
<td>±</td>
<td>2,100</td>
<td>14.3</td>
<td>1,268</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10.4</td>
<td>5.08</td>
<td>9,760</td>
<td>18,266</td>
<td></td>
</tr>
</tbody>
</table>

| w/o abort op. | | | | | | | | | | |
| 1             | 16  | 60.0 | 2.45 | 3.36 | 86.1 ± 3.56 | 4,874 | 5.08 | 9,760 | 18,266|
| 2             | 16  | 52.5 | 2.80 | 3.84 | 75.2 ± 11.2 | 6,757 | 4.02 | 16,099| 9,771 |
| 4             | 16  | 48.9 | 3.00 | 4.12 | 63.8 ± 10.9 | 9,530 | 3.04 | 28,49 | 27,658|
| 8             | 16  | 46.2 | 3.18 | 4.37 | 60.2 ± 10.3 | 13,588| 2.30 | 57,636| 48,984|
| 16            | 16  | 52.4 | 2.80 | 3.85 | 48.1 ± 12.0 | 21,703| 1.62 | 109,527| 73,691|

| w/ abort op. | | | | | | | | | | |
| 1             | 16  | 62.3 | 2.36 | 3.99 | 91.6 ± 3.39 | 4,599 | 4.61 | 10,504| 7,326 |
| 2             | 16  | 52.5 | 2.80 | 4.74 | 73.2 ± 14.6 | 5,784 | 3.45 | 14,204| 14,908|
| 4             | 16  | 46.8 | 3.14 | 5.31 | 68.8 ± 7.88 | 8,386 | 2.80 | 25,740| 36,982|
| 8             | 16  | 44.4 | 3.30 | 5.59 | 62.7 ± 9.74 | 10,617| 1.87 | 45,598| 43,990|
| 16            | 16  | 50.8 | 2.89 | 4.89 | 53.1 ± 10.1 | 16,354| 1.26 | 99,838 | 143,712|

$n$: # of computing nodes; $w$: # of workers per node; ET: Execution time [s]
SU/C: Speedup (vs. C); SU/1: Speedup (vs. 1 worker); TER: Task execution rate [%]
#V: # of visits to vertices [$\times 10^6$] (total of all workers)
#V/s: # of visits to vertices [$\times 10^6$]/s (average among all workers)
#RI: # of referred itemsets in itemset table [$\times 10^6$] (total of all workers)
#TC: # of task creations

<table>
<thead>
<tr>
<th>Impl.</th>
<th>$n$</th>
<th>$w$</th>
<th>LCRT</th>
<th>LCRQ</th>
<th>#TE</th>
<th>#AT</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w/o abort op.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>0.333</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>0.325</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>0.581</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>0.615</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>0.636</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

| w/ abort op. | | | | | | |
| 1             | 16  | 0.318 | 0.084 | 4,250 | 30 |
| 2             | 16  | 0.593 | 0.260 | 11,654| 120 |
| 4             | 16  | 0.588 | 0.207 | 30,519| 362 |
| 8             | 16  | 0.591 | 0.264 | 46,071| 684 |
| 16            | 16  | 0.693 | 0.207 | 117,866| 2,283 |

LCRT: the percentage of lock contentions for table entries in all acquisitions
LCRQ: the percentage of lock contentions for exception queues in all acquisitions
#TE: # of thrown exceptions; #AT: # of aborted tasks

81
7.4 Conclusion

In this chapter, we have improved the parallel COPINE algorithm by exploiting exceptions in a task-parallel language. In a parallel search, a worker could traverse redundant subtrees in the search tree pruned by another worker. We implemented the abortion of such redundant traversals by throwing exceptions. Experimental results show that this abort mechanism reduces the search space by 31.9% and the execution time by 27.4% in the 28-worker analysis of a real protein network. This mechanism can be implemented by making only small changes to the existing code.
Chapter 8

Related Work

This chapter introduces related work of parallel backtrack search implementations, task-parallel languages, and non-backtracking based data mining.

8.1 Parallel Backtrack Search Implementation

8.1.1 Parallel Game Tree Search

Game tree searches are important applications of backtrack search algorithms with pruning. To play a game advantageously, we need to search the game positions resulting from possible moves as widely and deeply as possible to find the best move. However, as wider and deeper positions are explored, the search space becomes exponentially larger.

Therefore, pruning is used to improve the search efficiency when we solve large-scale practical problems in games such as chess and shogi (Japanese chess). Alpha-beta pruning [27] is the most representative use of the minimax algorithm. This pruning method prevents redundant searches using an evaluation metric of the already-searched nodes. We can obtain the maximal pruning effect by move ordering, which controls the search order so that a node having the highest evaluation value is searched earlier. There are various heuristics for move ordering, including specific methods for different games.

The Young Brothers Wait Concept (YBWC) [28] is one parallelization technique for alpha-beta pruning. In YBWC, when the children $n_0, n_1, \cdots, n_k$ of a node are searched, the eldest son $n_0$ is evaluated first. After we have obtained the evaluation value of $n_0$, younger brothers $n_1, n_2, \cdots, n_k$ are then searched in parallel. Therefore, if move ordering works perfectly, we can completely avoid redundant searches, because $n_0$ gives us the best evaluation value with which to prune all younger siblings.

However, move ordering is inevitably imperfect, and so some redundant searches are unavoidable. In chess programs, therefore, there are implementations that support a mechanism to abort redundant partial searches. For example, *Socrates [4] uses the Jamboree algorithm (an improved version of YBWC). Its abort operation is implemented at the program level, as *Socrates is written in Cilk 1.0, which has no mechanism for aborting a task. Note that *Socrates was subsequently rewritten as Cilkchess using Cilk 5.0, exploiting the later version’s mechanism for aborting tasks [29]. For shogi, Bonanza [5] incorporates a mechanism to abort redundant partial searches, but the abortion is implemented at the program level. These complicated implementations could be simplified by the exception handling features of the enhanced Tascell.
Recently, the massive parallelization of game tree searches using multiple computing nodes has become a popular way to improve 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 (a hash table with the positions as the keys), which is used to avoid duplicate searches of identical positions. To manage such tables in distributed memory environments, there are various mechanisms to alleviate the increase in the search space caused by the incompleteness of information sharing. For example, Transposition table Driven work Scheduling (TDS) [30] uses a distributed hash table. When a task is created, a hash value is calculated for the task, and the task is sent to the computing node assigned with that hash value. In TDS, internode communication is required to send tasks to external nodes, but all table accesses are local. However, the performance of TDS depends on the quality of the hash function; redundant searches are performed when a task is sent to an inappropriate computing node.

8.1.2 Parallel SAT Solver

There have been a number of studies regarding the parallelization of the Satisfiability Problem (SAT) as a major application of the backtrack search with pruning [31, 32].

In these works, sharing pruning information is regarded as an important issue [33], but, unlike our problem, the pruning information can be freely referred to by any worker, regardless of the locations of the registration and reference in the search space. MiraXT [34], PaSAT [35], and ySAT [36] are SAT implementations in shared memory environments. To share table information, MiraXT employs a strategy similar to our fully-sharing method described in Section 4.2.4. In PaSAT and ySAT, each worker has its own table and exchanges the contents of this table with other workers periodically to update the table. This sharing method is similar to our implementation in distributed memory environments.

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

8.1.3 Task Scheduling in Parallel Tree Search

Tascell and LTC-based multithreaded languages such as Cilk share the oldest-first task scheduling policy that allows a worker to 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 the victim is traversing, as in the Parallel Depth First (PDF) scheduling [37] for left- and depth-first searches. In PDF scheduling, the leftmost unassigned branch of such a node is the task for the thief, achieving good utilization of on-chip cache in chip multiprocessors and better performance than the oldest-first for some applications [38, 39].

This thesis has presented a variation of the oldest-first strategy in which a stolen task consists of the leftmost and subsequent unassigned branches. The advantage of this variation over the conventional half-and-half splitting was demonstrated in terms of the effectiveness of sequentially dependent pruning.
8.2 Parallel Programming Language

The following discussion of parallel programming languages focuses on exception handling, task abortion, and distributed memory environment support.

8.2.1 Cilk

Cilk [1] provides an `abort` statement that aborts all already-spawned children of the procedure that has called the `abort` [40]. This statement can only be used inside an `inlet`, which is a handler invoked at the termination of the spawned procedure with the returned value. Cilk does not support non-local exit operations, such as throwing exceptions. To transfer control straight back to an ancestor procedure, such operations must be implemented explicitly.

There are many task-parallel languages derived from Cilk. For example, Distributed Cilk [41] and SilkRoad [42] support distributed memory environments. JCilk [3] has a similar capability to handle exceptions with collateral task abortion as Tascell. The methodology for aborting speculative searches in simple backtrack algorithms such as the N-queens problem and parallel alpha-beta search is discussed in [3]. We applied this methodology to COPINE, which is more complex and used for real-world applications. As mentioned in Section 4.3, it is difficult to implement the parallel COPINE algorithm without our backtracking mechanism.

8.2.2 Intel Cilk Plus

Exception handling in Intel Cilk Plus [20] has the same semantics as that in C++, i.e., the try-catch-finally mechanism. If a thrown exception is not caught inside a spawned function, the exception propagates from the point of the corresponding synchronization point. When several exceptions are asynchronously thrown and reach the synchronization point, the exception that would have occurred first in the serial execution is chosen and later exceptions are destroyed. When an exception is not caught inside a task, no other tasks spawned at the corresponding synchronization point are terminated early.

8.2.3 OpenMP

OpenMP is the de-facto standard for parallel programming on shared memory environments. It enables programmers to describe a task-parallel program by the `task` construct, which generates an explicit task.

The `cancel` and `cancellation point` constructs are introduced in OpenMP 4.0 [43]. The former activates a cancellation and the latter adds an explicit cancellation point to the user code.

An exception thrown inside a parallel region, such as `parallel`, `for`, `sections`, or `task`, must be caught within the same region. In addition, an exception must be caught by the same thread that threw it. That is, the propagation of an exception among threads must be implemented manually because of this restriction.

8.2.4 Java

The Java Fork/Join Framework [44] was added to the Concurrency Utilities in Java SE 7 for a natural description of fine-grained parallel processing. The `java.util.concurrent` package
defines the ForkJoinTask class [45] that executes tasks in parallel. If a thrown exception is not caught in a task, it is rethrown to the task attempting to join it. A rethrown exception is handled in the same way as a regular exception.

The ForkJoinTask class has the cancel method for canceling other tasks. If the method succeeds, calling join() and related methods will result in a CancellationException. Note that the method cannot cancel the execution of the current task and other running tasks in the default implementation. The completeExceptionally( Throwable ex) can abort the current task. If the method succeeds, a given exception will be thrown when join() and related methods are called.

8.2.5 X10

X10 [46] has the async-finish construct to support task parallelism. The async statement spawns an asynchronous activity, and the finish statement synchronizes all activities spawned inside the statement. In addition, X10 provides dynamic load balancing across computing nodes by using the Global Load Balancing (GLB) framework [47], which is included in the standard library in X10.

In X10, when an exception is thrown in the execution of an activity, a nest of try-catch blocks in the activity attempts to catch the exception if the thrower is in a try-construct in the nest. The activity is aborted if the exception is not caught by any try-catch blocks in it. Such an uncaught exception raised in an activity can be forwarded to its parent if the activity is spawned in a finish statement by which normal and abnormal completions of all activities spawned in that statement are confirmed. Therefore, surrounding a finish statement by a try-catch block allows exceptions thrown by a child activity spawned in the statement to be caught. If two or more child activities in a finish raise exceptions asynchronously, these exceptions are wrapped into a single object of x10.lang.MultipleExceptions to conform to the rooted exception model [48].

The finish statement is not capable of aborting activities other than those raising exceptions, but simply waits for their normal completion, as in Intel Cilk Plus. Therefore, a user-level implementation is required to abort such activities.

8.2.6 Manticore

Manticore [49, 50] is a functional language for parallel programming. It has explicit- and implicit-concurrency mechanisms. The former is based on Concurrent ML (CML) [51], and supports coarse-grained parallel tasks by threads explicitly created using the spawn primitive. A thread may terminate if a raised exception is uncaught. The uncaught exception is not propagated to its parent thread. The latter allows fine-grained parallel computation using constructs to inform the compiler which parts of a computation can be evaluated in parallel. These constructs enable the parallel execution of sequential semantics. That is, when raised in a sub-computation, the propagation of an exception must be delayed until it has been verified that all sequentially prior computations have terminated. All the posterior computations should be aborted as soon as possible.
8.2.7 Uni-Address Threads

One of the challenges in internode work-stealing is to migrate threads among computing nodes. Adaptive MPI [52] and Charm++ [53] use an iso-address approach in the PM2 runtime system [54], which requires virtual memory proportional to the number of workers in each node for thread stacks. That is, they cannot exploit Remote Direct Memory Access (RDMA) operations for thread migration, resulting in a scalability limitation.

Uni-address threads [55], which is a lightweight multithreaded library designed for large-scale distributed memory environments, reduces the virtual memory usage for thread migration. This enables the implementation of RDMA-based work-stealing. It has been confirmed that the performance scales well to 3,840 cores in three benchmarks: Binary Task Creation, Unbalanced Tree Search, and N-queens problems.

8.3 Non-Backtracking Based Data Mining

8.3.1 Graph Mining using Mathematical Calculation

A group of graph mining algorithms, including COPINE [6, 7] and gSpan [9], conducts a back-track search in the analysis of a graph to, for example, enumerate its subgraphs with or without restrictions to be satisfied. In contrast, a number of graph mining algorithms use vector calculation. For example, PageRank [56], which is a core technology of Google Search, determines the relative importance of each website by calculating eigenvectors of the adjacency matrix. For another example, many clustering algorithms group the vertices of a graph into appropriate clusters based on the distance between them. In particular, \( k \)-means clustering achieves cluster analysis for large-scale graphs by minimizing an evaluation function using hill climbing.

One of the advantages of graph mining algorithms using mathematical calculations is their scalability. However, we should treat the analysis results of algorithms such as \( k \)-means clustering carefully, because they are local optimum solutions. In addition, it is difficult to analyze graphs with external data associated with their vertices and/or edges using such algorithms.

8.3.2 Frameworks for Graph Mining

As the number of opportunities to use graph-structured data increases, various frameworks for graph processing are being developed. For example, Pregel [57] has a vertex-centric approach in which each worker is given a unique partition of a graph and executes a computation for the partition. This method is inspired by the Bulk Synchronous Parallel (BSP) model [58], in which computations consist of a sequence of iterations separated by barriers named supersteps. At each superstep, workers execute a computation using local data and the messages received at the previous superstep, and send messages to the other workers. That is, using Pregel, programmers can describe the processing of vertices as the main component of a program without worrying about deadlocks.

There are also programming languages that allow programmers to describe a program using graph structures, such as LMNtal [59]. Applications of these frameworks to graph algorithm problems such as the shortest path problem are discussed in [57, 59]. However, there have been no previous attempts to parallelize graph mining applications that require knowledge sharing for pruning using such frameworks.
8.3.3 Frequent Itemset Mining

There is a problem named Frequent Itemset Mining (FIM) [60, 61], which is similar to the CCIG enumeration problem. 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 \( I(t) \subseteq I \) \( (t \in T) \), and a threshold \( \theta \), FIM is to extract all itemsets whose members are commonly contained by \( \theta \) or more transactions. Many implementations of FIM solvers have been proposed [62]. Among them, the Linear time Closed itemset Miner (LCM) [63–65] realizes efficient enumeration of closed itemsets\(^1\) by defining and utilizing a 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.

---

\(^1\)Let \( I(S) = \bigcap_{t \in S} I(t) \) for a \( S \subseteq T \), and \( T(H) = \{t \in T | H \subseteq I(t)\} \) for a \( H \subseteq I \). An itemset \( H \) is said to be closed iff \( I(T(H)) = H \) [63].
Chapter 9

Conclusion and Future Work

In this thesis, we proposed a parallel algorithm and implementations for graph mining that extract all connected subgraphs, each of which shares a common itemset whose size is not less than a given threshold. In addition, we proposed capabilities of task-parallel languages required for implementing such parallel backtrack search algorithms with pruning.

An efficient sequential algorithm named COPINE has already been proposed for this problem. It ensures efficient subgraph extraction using a pruning mechanism that contains sequential dependencies, but its straightforward parallelization results in excessive pruning. To avoid such excessive pruning, in Chapter 2, we designed a parallel version of COPINE by introducing the restriction that workers can only refer to knowledge acquired during the search for the pruning if the acquirement-reference flow matches the dependency.

The parallel COPINE algorithm requires knowledge sharing among workers to achieve the pruning in a parallel search. We implemented a sharing method in which all workers in each computing node share a single table controlled by locks in Chapter 4. In addition, we implemented a communication thread in all the nodes that call MPI collective communication functions at regular time intervals to exchange the table updates synchronously in Chapter 5.

The conventional strategies in the task-parallel language Tascell, in which we implemented the parallel COPINE algorithm, are not optimal for COPINE from the viewpoint of the effectiveness of the pruning and the high cost of work-stealing. We employed a new task creation strategy that makes the pruning effective in a parallel search in Chapter 4. In Chapter 5, we also developed new work-stealing strategies that reduce the total cost of work-steals among workers, especially in distributed memory environments.

Despite our attempt to use many workers to obtain sufficient speedup, this was not very successful in shared or distributed memory environments. This is because a worker traversing a redundant subtree that has been pruned by another worker results in the search space enlarging as the number of workers increases. To alleviate this enlargement, in Chapter 7, we improved the parallel COPINE algorithm by adding a mechanism to abort redundant searches, and implemented it in both shared and distributed memory environments.

To implement the parallel COPINE algorithm elegantly and efficiently, we also added the following capabilities to Tascell; the capabilities to allow programmers to control the division ratio of a parallel loop (Chapter 4) and the strategy for choosing a victim of a work-steal (Chapter 5), and the exception handling features by which all tasks enclosed in the dynamic scope of a try block with an exception are automatically aborted (Chapter 6). The first and the second capabilities allow programmers to tune the performance, and the third one realizes operations
for aborting a traverse for a redundant subtree preserving productivity. We believe that these capabilities and our experience of developing COPINE using them are also useful when implementing other practical backtrack search algorithms.

Our future work will focus on improving performance in more highly parallel computing environments. We were unable to obtain sufficient accelerated performance in executions with many workers. We observed that the task execution rate significantly decreases as the number of workers increases. This indicates that workers could not obtain larger tasks by work-stealing, even though we employed strategies to reduce the total cost of work-steals. This degradation is mainly caused by the chains by which tasks are stolen back. We aim to solve this problem by further improving the strategies for choosing victims and assigning subtrees. We also need to improve the implementation of Tascell itself, including the stealing-back mechanism, which is a major factor in restricting how workers choose victims.

We will attempt to implement other backtrack search algorithms using our proposed mechanism. For example, we can apply our abort mechanism to algorithms for game playing since they have similar pruning mechanism as described in Section 8.1.1. We will also implement the parallel COPINE algorithm in other programming languages such as Cilk, and compare it to the Tascell version in terms of performance and productivity.
Appendix A

Proof of Correctness of COPINE

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 \( T(U) = \langle u_1, \ldots, u_n \rangle \), all of the following hold for all \( i, j, \) and \( k \) such that \( 1 \leq i, j, k, \leq n \).

1. \( u_i \in N^*(U_j) \rightarrow u_i \in N(U_j) \)
2. \( u_i \in N^*(U_j) - U_j \rightarrow u_i \in N(U_j) - U_j \)
3. \( \text{prev}^*(u_i, U_j) = \text{prev}(u_i, U_j) \)
4. \( u_i \stackrel{U_k}{\leftrightarrow} u_j \rightarrow u_i \stackrel{U_k}{\leftarrow} u_j \)

**Proof:**

1. Since \( u_i \in N^*(U_j) \), there must exist \( u_k \) such that \( (u_k, u_i) \in E \) and \( k \leq j \) by Definition 5 (1). Since \( u_k, u_i \in U \), it holds that \( (u_k, u_i) \in E(U) \) and thus \( u_i \in E(U_j) \) as well by Definition 4 (2).

2. Since \( u_i \in N^*(U_j) - U_j \), it holds that \( u_i \in N^*(U_j) \) and \( u_i \not\in U_j \). Therefore, \( u_i \in N(U_j) \) by (1) and thus \( u_i \in N(U_j) - U_j \).

3. Since \( u_i \in U \), it holds that \( \{u_k \mid (u_k, u_i) \in E, \ 1 \leq k \leq j\} = \{u_k \mid (u_k, u_i) \in E(U), \ 1 \leq k \leq j\} \). Therefore;

\[
\text{prev}^*(u_i, U_j) = \arg \max_k [u_k \mid (u_k, u_i) \in E, \ 1 \leq k \leq j] \quad \text{(Definition 5 (2))}
\]

\[
= \arg \max_k [u_k \mid (u_k, u) \in E(U), \ 1 \leq k \leq j]
\]

\[
= \text{prev}(u_i, U_j) \quad \text{(Definition 4 (3))}
\]
Proof:

Let \( \text{Lemma 1} \) be the help of Lemma 1, we can prove Theorem 1 as follows.

\( \textbf{Theorem 1 (Direct Successor of a Subgraph)} \)

Let \( \{G_1, \cdots, G_N\} \) be the set of all connected subgraphs of \( G \) such that \( G_1 < \cdots < G_N \). For \( G_l = (V_l, E(V_l)) \) (1 \( \leq l < N \)) where \( T(V_l) = \langle u_1, \cdots, u_n \rangle \), \( T(V_{l+1}) \) is given by:

\[
T(V_{l+1}) = \langle u_1, \cdots, u_k \rangle \cdot \langle \min\left( \frac{U_{1,k}}{U_{1,k}} \right) (C_k) \rangle
\]

where \( k \) and \( C_k \) are defined as follows with \( U_i = \{u_1, \cdots, u_i\} \) (1 \( \leq i \leq n \).

1. \( P_i = \{v \mid v \stackrel{U_{i+1}}{\leftarrow} u_i \} \cup \{u_i\} \)
2. \( D_i = \bigcup_{j=1}^{i} P_j \)
3. \( C_j = N^*(U_j) \cap \{v \mid D_{i+1} \cap D_n \} \)
4. \( k = \arg\max \{C_j \mid C_j \neq \emptyset, 0 \leq j \leq n \}\)

Proof: Let \( u = \min(\frac{U_{1,k}}{U_{1,k}})(C_k) \) and \( V_{l+1} = U' = \{u'_1, \cdots, u'_{k+1}\} \) where \( u'_j = u_j \) for all \( j \) such that 1 \( \leq j \leq k \) and \( u'_{k+1} = u \).

1. \( T(V_{l+1}) \) is canonical because of the following, where \( N(U_j) \) for 1 \( \leq j \leq k \) is defined with respect to \( U = \{u_1, \cdots, u_k\} \), while \( N(U'_j) \) for 1 \( \leq j \leq k + 1 \) is defined with respect to \( U' = U \cup \{u\} \). Therefore, it holds that \( N(U'_j) \subseteq N(U_j) \cup \{u\} \).

(a) If \( k = 0, T(V_{l+1}) = \langle u \rangle \) is canonical by definition. For the case of \( k > 0 \) we have the following (b)–(h) to prove the canonicity of \( T(V_{l+1}) \).

(b) Since \( T(V_l) \) is canonical and so is \( T(U_k) \), it holds that \( u'_1 = u_1 = \min(\langle\rangle)(U_k) = \min(\langle\rangle)(U'_k) \). Since \( P_1 = \{v \mid v \stackrel{U_{1,k}}{\leftarrow} u_1 \} \cup \{u_1\} = \{v \mid v < u_1 \} \cup \{u_1\} = \{v \mid v \leq u_1 \} \), it holds that \( D_1 = P_1 = \{v \mid v \leq u_1 \} \). Therefore, \( D_k \supseteq \{v \mid v \leq u_1 \} \) and thus \( C_k \cap \{v \mid v \leq u_1 \} = \emptyset \), to mean \( u'_{k+1} \not\in \{v \mid v \leq u_1 \} \) because \( u'_{k+1} = u \in C_k \), and thus \( u'_1 = u_1 < u'_{k+1} \) to hold \( u'_1 = u_1 = \min(\langle\rangle)(U') \) satisfying the canonicity requirement for \( u'_1 \).
(c) Since \(u_j \in P_j\) and \(P_j \subseteq D_j\) by definition, for all \(j\) such that \(1 \leq j \leq k\) it holds that \(u_j \in D_j \subseteq D_k\). Since \(u \in C_k\) to mean \(u \notin D_k\), for all \(j\) such that \(1 \leq j \leq k\) it holds that \(u \neq u_j\), and thus \(u \notin U_j = U_j'\).

(d) Since \(u \in C_k \subseteq N'(U_k)\), there must be \(i\) such that \(1 \leq i \leq k\), with which it is satisfied that \(u \notin N'(U_j) = N'(U_j')\) for all \(j\) such that \(1 \leq j < i\), and \(u \in N'(U_j) = N'(U_j')\) for all \(j\) such that \(i \leq j \leq k\).

(e) For \(j < i\), \(u \notin N'(U_j')\) means \(u \notin N(U_j')\) because \(N(U_j') \subseteq N'(U_j')\) and thus \(N(U_j') = N(U_j)\). Therefore we have \(N(U_j) - U_j = N(U_j') - U_j'\). Combining this equality with \(u_{j+1}' = u_{j+1} = \min[U_j](N(U_j) - U_j)\) because of the canonicity of \(T(U)\) derived from that of \(T(V_i)\), we conclude that \(u_{j+1}' = \min[U_j](N(U_j') - U_j')\) to satisfy the canonicity requirement for \(u_{j+1}'\).

(f) For \(j\) such that \(i \leq j < k\), \(u \in N'(U_j')\) means \(u \in N(U_j') = N(U_j) \cup \{u\}\) because \(u \in U'\) by Lemma 1 (1). Since \(u \in C_k\), and thus \(u \notin D_{j+1}\), it holds that \(u \notin P_{j+1}\). This means that \(u \not\in U'_{j+1}\) does not hold, while \(u \in N'(U_j') - U_j'\) because \(u \in N'(U_j')\) and \(u \notin U_j'\) by (c) above, to lead us to \(u_{j+1}' = \min[U_j](N(U_j) - U_j)\). Therefore by Lemma 1 (4), we have \(u_{j+1}' = \min[U_j](N(U_j) - U_j)\). On the other hand, the canonicity of \(T(U)\) requires that \(u_{j+1}' = u_{j+1} = \min[U_j](N(U_j) - U_j)\). Therefore the minimality of \(u_{j+1}'\) still holds with \(N(U_j') - U_j'\) to lead us to \(u_{j+1}' = \min[U_j](N(U_j') - U_j')\) to satisfy the canonicity requirement for \(u_{j+1}'\).

(g) Since \(u \in N'(U_k')\) and \(u \in U'\), it holds that \(u \in N(U_k') = N(U_k) \cup \{u\}\) by Lemma 1 (1). On the other hand, the graph \((U, E(U)) = (U_k, E(U_k))\) is a connected subgraph of \(G\), it is obvious that \(N(U_k) = U_k\). Therefore, we have \(N(U_k') = N(U_k) \cup \{u\} = U_k \cup \{u\} = U_k' \cap \{u\}\). From this equality and \(u \notin U_k'\) by (c), we have \(\min[U_k'](N(U_k') - U_k') = \min[U_k'](|u\}) = u = u_{k+1}'\) to satisfy the canonicity requirement for \(u_{k+1}'\).

(h) From (b) for \(u_1'\), (e) for \(u_{j+1}' (1 \leq j < i)\), (f) for \(u_{j+1}' (i \leq j < k)\), and (g) for \(u_{k+1}'\), we have \(u_j' = \min[U_j'](N(U_j') - U_j')\) for all \(j\) such that \(1 \leq j \leq k + 1\) to prove that \(T(U') = T(V_{i+1})\) is canonical.

(2) \(T(V_i) < T(V_{i+1})\) because of the following and the obvious fact that \(k = |prefix(T(V_i)), T(V_{i+1})|\).

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

(b) If \(k < n\) and thus \(u \notin D_{k+1}\) to mean that \(u \notin P_{k+1} \subseteq D_{k+1}\). Therefore, \(u = \min[U_k'] u_{k+1}'\) does not hold, while \(u \in N'(U_k) - U_k\) and \(u_{k+1} \in N'(U_k) - U_k\) obviously. Thus we conclude \(u_{k+1} u_{k+1}' = u = u_{k+1}'\) to prove the proposition \(T(V_i) < T(V_{i+1})\).

(3) Any spanning tree \(t\) such that \(T(V_i) < t < T(V_{i+1})\) is not canonical because of the following. Note that such \(t\) should satisfy \(|t| > k\) and \(|prefix(T(V_i), t)| = k' \geq k\) because of \(|prefix(T(V_i), T(V_{i+1})|) = k\). Let \(t = \langle u'_1, \ldots, u'_n \rangle\) and \(U'' = \langle u'_1, \ldots, u'_n \rangle\) 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 \(w = u_{k}'\) because \(t < T(V_{i+1})\). Since \(u = \min[U_k'](C_k)\), however, \(w \notin C_k\) but definitely \(w \in N'(U_k)\) to mean that there exists \(i\) such that \(w = D_i\) and \(i \leq k + 1\) if \(n > k\) or \(i \leq k\) 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 $i$ such that $w \in D_i$ and $i \leq k' + 1$ if $n > k'$ or $i \leq k'$ if $n = k'$.

(c) Existence of $i$ such that $w \in D_i$ shown in (a) and (b) above means the existence of $m$ such that $w \in P_m$, to mean $w \overset{U_m}{\sim} u_m$, and $m \leq i$. If $m = i = k' + 1$, we have $w \overset{U_{k'+1}}{\sim} u_{k'+1}$ but it contradicts $u_{k'+1} \overset{U_{k'+1}}{\sim} w$ which should be derived from $T(V_i) < t$. Therefore, $m \leq k'$ must be true.

(d) Since $m \leq k'$ as given by (c), $w \overset{U_m}{\sim} u_m$ means that $w \overset{U_m}{\sim} u''_m$. If $m = 1$, this means that $w \overset{U_m}{\sim} u''_1$ and thus $w < u''_1$ violating $u''_1 = \text{min}[<](U'')$ to be satisfied if $t$ is canonical.

If $m > 1$, $w \overset{U_m}{\sim} u''_m$ means that $w \overset{U_m}{\sim} u''_m$ since $w \in U''$ by Lemma 1 (4), violating $u''_m = \text{min}[<](U'_m - U''_{m-1})$ to be satisfied if $t$ is canonical. Therefore, $t$ is not canonical.

\[\square\]

A.2 Proof of Theorem 3

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

Lemma 2 (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. $G^2 < G^1$

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

\[\square\]

Lemma 3 (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 $G_1 < G_2$, and let $T(U^1) = \langle u^1_1, \ldots, u^1_m \rangle$ and $T(U^2) = \langle u^2_1, \ldots, u^2_n \rangle$. If their tails are common, i.e., $\text{tail}(U^1) = u^1_m = \text{tail}(U^2) = u^2_n$, all of the following hold.

1. $G^1 \cup G^2 < G^2$

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

94
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 \( \text{tail}(U^1) = \text{tail}(U^2) \). Since \( G^1 < G^2 \) means \( G^1 \not\subseteq G^2 \) 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, for \( k = |\text{prefix}(T(U^1), T(U^2))| \), both of \( k < |T(U^1)| \) and \( k < |T(U^2)| \) hold. Since \( G^1 < G^2 \) and \( G^2 \) is not a descendant of \( G^1 \), it holds that \( G^1 \not\subseteq G^2 \) by Lemma 2 and thus \( G^2 \subset G^\cup \subset G' \).

(1) \( G^2 \subset G^\cup \) means \( G^\cup < G^2 \) 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 \( |\text{prefix}(T(U^1), T(U^\cup))| = |\text{prefix}(T(U^1), T(U^2))| = k \). Let \( T(U^1) = \langle u_1^1, \cdots, u_{n^1}^1 \rangle \), \( T(U^2) = \langle u_1^2, \cdots, \rangle \), and \( T(U^\cup) = \langle u_1^\cup, \cdots, \rangle \). Since \( G^1 < G^2 \) and \( G^\cup \) is a descendant of \( G^2 \), it holds that \( u = u_{k+1}^1 < u_{k+1}^2 = u_{k+1}^\cup = w \). However, since \( G^1 \not\subseteq G^\cup \), it must be \( u \in N(U_k^\cup) - U_k^\cup \). Therefore, \( u < w \) violates \( w = \min \left\{ \frac{U_k^\cup}{npropersubset} \right\} \) for the canonicity of \( T(U^\cup) \) leading us to a contradiction. Therefore, \( G^\cup \) cannot be a descendant of \( G^2 \) and thus \( G^\cup < G^2 \).

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

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

Theorem 3 (Order of CIGs and their Closeness) Let \( G^1 = (U^1, E(U^1)) \) and \( G^2 = (U^2, E(U^2)) \) be connected subgraphs of a graph \( G = (V, E) \) such that \( G^1 < G^2 \). If the tails of \( T(U^1) = \langle u_1^1, \cdots, u_{n^1}^1 \rangle \) and \( T(U^2) = \langle u_1^2, \cdots, u_{n^2}^2 \rangle \) are common, i.e., \( \text{tail}(U^1) = u_{n^1}^1 = \text{tail}(U^2) = u_{n^2}^2 \), and \( \mathcal{X}(G^1) \supseteq \mathcal{X}(G^2) \), all of the following criteria hold.

(1) \( G^2 \) is not closed with respect to \( \mathcal{X}(G^2) \).

(2) Any descendant \( G' = (U', E(U')) \) of \( G^2 \) such that \( T(U') = T(U^2) \cdot T' \) is not closed with respect to \( \mathcal{X}(G') \).

Proof:

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

(2) Let \( T(U') = T(U^2) \cdot \langle w_1, \cdots, w_{n^2} \rangle \), \( I_w = \bigcap_{i=1}^{n^2} \mathcal{X}(w_i) \), and \( G^+ = G^1 \cup G' \). Since \( \mathcal{X}(G^+) = \mathcal{X}(G^1) \cap \mathcal{X}(G^2) \cap I_w = \mathcal{X}(G^1) \cap I_w = \mathcal{X}(G') \), the subgraph \( G' \) is not closed with respect to \( \mathcal{X}(G^') \) unless \( G' = G^+ \). However, since \( G^+ < G^2 \) by Lemma 3 and definitely \( G^2 < G' \) because \( G' \) is a descendant of \( G^2 \), we have \( G^+ < G' \) and thus \( G' \neq G^+ \). Therefore, \( G' \) is not closed with respect to \( \mathcal{X}(G^') \).
A.3 Proof of Theorem 4

Theorem 4 (CCIG) A connected subgraph $G' = (U, E(U))$ of a graph $G = (V, E)$ is closed with respect to $\mathcal{I}(G')$ iff all of the following conditions hold.

(1) For any direct descendant $G^d = (U^d, E(U^d))$ of $G'$, $\mathcal{I}(G') \neq \mathcal{I}(G^d)$.

(2) For any connected subgraph $G^p = (U^p, E(U^p))$ of $G$ such that $G^p < G'$ and $\text{tail}(U^p) = \text{tail}(U)$, $\mathcal{I}(G^p) \notin \mathcal{I}(G')$.

Proof:

Necessity: Obviously (1) is a necessary condition of the closeness of $G'$ with respect to $\mathcal{I}(G')$, 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 $G^s < G'$. 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 descendant of $G^d$. Since $\mathcal{I}(G') \neq \mathcal{I}(G^d)$ but clearly $\mathcal{I}(G^s) \supset \mathcal{I}(G^d) \supset \mathcal{I}(G^d)$, it must hold that $\mathcal{I}(G^s) \supset \mathcal{I}(G^d)$ and thus $\mathcal{I}(G^s) \supset \mathcal{I}(G')$. If $G^s < G'$ on the other hand, $T(U^s) = T(U^p) \cdot T'$ with some $G^p$ (possibly $G^s$ itself) such that $G^p < G'$ and $\text{tail}(U^p) = \text{tail}(U)$ because $G^s \supset G'$ and thus $\text{tail}(U) \in U^s$. Since $\mathcal{I}(G^p) \notin \mathcal{I}(G')$, we have $\mathcal{I}(G^s) \subset \mathcal{I}(G')$ because $\mathcal{I}(G^s) \subset \mathcal{I}(G^p)$ and $\mathcal{I}(G^p) \subset \mathcal{I}(G')$. Therefore, $\mathcal{I}(G^s) \subset \mathcal{I}(G')$ for any $G^s$ such that $G^s \supset G'$, and thus the subgraph $G'$ is closed with respect to $\mathcal{I}(G')$ by definition, proving that (1) and (2) are sufficient conditions.

$\square$
Bibliography


List of Publications

Journal Papers


International Conference Papers


International Conference Posters


Awards


Publications used in the Thesis

Part of this thesis are based on authors’s publications as follows.

**Chapters 2 & 4**

Permalink: http://id.nii.ac.jp/1001/00102154/
© 2014 Information Processing Society of Japan

**Chapter 5**

doi: http://doi.org/10.2197/ipsjjip.25.256
© 2017 Information Processing Society of Japan

**Chapter 6**

doi: http://doi.org/10.2197/ipsjjip.24.439
© 2016 Information Processing Society of Japan

**Chapter 7**

doi: http://doi.ieeecomputersociety.org/10.1109/IPDPSW.2016.136
© 2016 IEEE