An Implementation of Exception Handling with Collateral Task Abortion

TASUKU HIRAISHI1,a) SHINGO OKUNO2 MASAHIRO YASUGI3

Received: July 3, 2015, Accepted: September 30, 2015

Abstract: In this paper, an implementation of exception handling for task-parallel languages is proposed such that all running parallel tasks in a try block with an exception are automatically aborted as soon as possible. In parallel tree search, exception handling that allows such a collateral task abortion is useful when the objective is to complete the search as soon as one solution is found or to allow a worker to abort the traversal of a subtree that is found to be redundant by another worker, even when it has been initiated. However, few existing task-parallel languages, such as Cilk Plus and X10, have this capability. In this study, we enhanced a task-parallel language, Tascell, with this capability. Since the Tascell compiler is implemented as a translator to C code, techniques are required for implementing the non-local exit mechanism with cleanup code execution in the “finally” clauses. We achieved this implementation by exploiting nested functions, which are already used in the temporary backtracking mechanism of Tascell. We also modified the task scheduler provided by Tascell such that a worker can abort a task after it is started. When aborting a task, the scheduler also aborts all its descendant tasks. We evaluated our implementation in terms of overheads and time taken to abort tasks.

Keywords: task-parallel languages, exception handling, task abortion, nested functions

1. Introduction

Backtrack search algorithms are applied in many applications, such as graph mining, the satisfiability problem (SAT), and board games. In order to solve large-scale problems quickly, parallel algorithms that exploit the powerful computation power provided by the rapid increase in the number of processor cores in both cloud-type general purpose servers and high-performance computing (HPC) systems need to be designed and implemented.

Since actual search trees in backtrack search algorithms usually grow dynamically and thus unpredictably, dynamic load balancing should be applied to parallelized implementations. Applications having this property are often implemented using task-parallel languages, such as X10 [1], Cilk [2], Intel Cilk Plus [3], and Tascell [4], that allow tasks to be dynamically spawned such that they are automatically assigned to workers, that is, parallel threads and/or processes, so that a worker has an exclusive set of subtrees as its task set.

In this paper, an implementation of exception handling is proposed in which all the running parallel tasks in a try block with an exception are automatically aborted as soon as possible, as a language extension for the task-parallel languages mentioned above.

For example, suppose a task-parallel execution with four workers, named worker 0–3, in which the function $f_0$ in Fig. 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 Fig. 1 is illustrated in Fig. 2 (a)$^{a1}$, where the tasks spawned by worker 0 in

---

$a1$ In multithreaded languages based on Lazy Task Creation [5] 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 Fig. 2, is not strictly applicable for such a language.
Fig. 3
Parallel binary tree search for one solution.

```plaintext
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);
        );
    }
}
```

Fig. 4
Parallel binary tree search for all solutions where exceptions are used for reducing redundant search.

```plaintext
binsearch (Node node) {
    int xnode=0, s1=0, s2=0;
    try {
        if (node) { return 0; }
        if (Solution found at node?)
            throw (xnode);
        /* Prune the subtree the root of which is xnode. */
        prune (xnode);
        if (Is node included in a pruned subtree the root of which is xnode?)
            throw (xnode);
    }
    try {
        join (
            if (Solution found at node?)
                s1 = spawn binsearch (node->right);
                s2 = binsearch (node->left);
        )
        catch (node) {
            /* catch an exception the tag value of which equals node */
            ...
            /* Returns the number of solutions. */
            return s0=s1+s2;
        }
    }
```
terms of overheads and time taken to abort tasks.

The remainder of this paper is organized as follows. We summarize related work in Section 2. We introduce the baseline Tascell framework to which we added the exception handling features in Section 3. In Section 4, we provide the language extension to Tascell for the exception handling features. In Section 5, we present our implementation of the enhanced Tascell. We show the performance evaluations in Section 6. Finally, we conclude this paper in Section 7.

2. Related Work

In this section, we discuss exception handling and task abortion as supported in other task-parallel languages. As mentioned in Section 1, a semantics of a task-parallel language with collateral task abortion was proposed in Ref. [8] and our design of the enhanced Tascell is based on this semantics. However, no major task-parallel language exists that supports such features, as described below.

2.1 Intel Cilk Plus

Exception handling in Intel Cilk Plus [3] 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.

2.2 X10

In X10 [1], when an exception is thrown, try-catch blocks inside the same activity attempt to catch it. If the exception is not caught, the activity is aborted. However, the 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 it are confirmed. Therefore, by surrounding a finish statement by a try-catch block, we can catch an exception thrown by a child activity spawned in the statement. 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 [9].

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

2.3 Cilk

Cilk [2] provides the abort statement, which aborts all the already-spawned children of the procedure that has called the abort [6]. It can be used only 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. In order to transfer control straight back to an ancestor procedure, such operations must be implemented explicitly.

2.4 Java

Java Fork/Join Framework [10] was added to Concurrency Utilities in Java SE 7 for natural descriptions of fine-grained parallel processing. 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. When a parallel loop is exited abnormally, all running tasks spawned at the loop are not aborted automatically. In order to abort such tasks, programmers need to explicitly call the cancel method that can cancel another task.

2.5 OpenMP

The cancel and the cancellation point constructs are introduced in OpenMP 4.0 [11]. 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 restrictions.

3. Tascell Framework

3.1 Overview

The Tascell framework [4], [12] consists of a compiler for an extended C language, called the Tascell language, and a runtime system for parallel computations.

In Tascell, computations are accomplished by the Tascell workers that execute tasks. A task is a data object that is necessary for accomplishing a certain computation. Its structure is defined in a Tascell program by the user. A task is associated with a specific function. When a worker receives a task, it invokes the associated function and completes its work on the given task object. Tascell employs a randomized work-stealing strategy to achieve dynamic load balancing among the workers. In Tascell, an idle worker (thief) can request a task from a loaded worker (victim). When receiving a task request, the victim worker creates a new task by dividing its own task and returns it to the thief worker. Then, the thief worker performs the received task and returns its result to the victim worker.

A Tascell worker spawns a task by temporarily backtracking and restoring its oldest task-spawnable state. That is, when a worker receives a task request, it:

1. Temporarily backtracks (goes back to the past),
2. Spawns a task (and changes the execution path to receive the result of the task),
3. Returns from the backtracking, and
4. Resumes its own task.

A Tascell worker always chooses not to spawn a task at first and performs sequential computations. However, when a worker receives a task request, it spawns a task as if it changed the past choice. Figure 6 shows the manner in which backtracking-based task spawning occurs when a Tascell worker performs backtrack...
search based on the C code in Fig. 5.

In general, a larger task can be spawned by backtracking to the oldest task-spawnable state. Because no logical threads are created as potential stealable tasks, the cost of managing a queue for them, as required in multithreaded languages based on Lazy Task Creation [5] such as Cilk, can be eliminated in Tascell.

### 3.2 Backtrack Search in Tascell

Figure 7 illustrates a parallelized Tascell program that performs a backtrack search for finding all the possible solutions to the Pentomino puzzle based on the C code in Fig. 5.

We defined a task object named **pentomino**. Several fields 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`.

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 for dividing an iterative computation. It is syntactically denoted by

```
for(int identifier : exprfrom, exprto) statementbody
```

This iterates `statementbody` over integers from `exprfrom` (inclusive) to `exprto` (exclusive). A worker performs iterations for a parallel `for` loop sequentially, unless it detects any task requests. When the implicit task-request handler (available during the iterative execution of `statementbody`) is invoked, the upper half of the remaining iterations are spawned as a new task, the object of which is initialized by `statementput`. In `statementput`, the actual assigned range can be referred to by `identifierfrom` and `identifierto`.

When all the remaining iterations are assigned to other workers as tasks, the worker waits for the results of the tasks, and then, handles (merges) the results of the tasks by executing `statementget`. In order not to be idle, the worker requests and executes other tasks.
int fib(void (*bk_exit1) (int, int), int n) 
{
    _label_.1_exit: /* label should be declared explicitly */
    int s = 0;
    /* nested function */
    void bk_exit1 (int n0, int v) {
        if (n == n0) {
            s = v;
            goto _label_.1_exit; /* jump to l_exit in fib */
        }
        bk_exit1(n0, v); /* call caller's nested function */
        return;
    }
    if (!found that fib(n)=v) bk_exit1(n, s);
    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;
    
    _label_.1_exit:
    return s;
}

Program with nested functions.

While waiting for the results. At this time, in order to reduce the execution stack size of the worker, it “steals back” a task from one of the workers to which tasks for parts of this parallel for loop are assigned. This technique is called Leapfrogging [13].

Parallel for statements may be nested dynamically in their statements.body. Therefore, multiple task-request handlers may be available at the same time. Each worker attempts to detect a task request by polling at every parallel for statement without heavy memory barrier (fence) instructions. When the worker detects a task request, it performs temporary backtracking in order to spawn a larger task by invoking as old a handler as possible.

Tascell has a dynamic_wind construct, as in the Scheme language [14] for specifying application-dependent undo/redo operations, e.g., removing/putting pieces in Pentomino, syntactically denoted by

dynamic_wind statement_before statement_body statement_after.

The worker basically executes statement_before ("set a piece" in Fig. 7 as “do”), statement_body, and statement_after ("remove the piece" in Fig. 7 as “undo”) in this order. However, during the execution of statement_body, statement_after is also executed as an "undo" clause before an attempt to invoke an older task request handler. statement_before is also executed as a "redo" clause after the attempt.

3.3 Implementation

The Tascell compiler is implemented as a translator to the C language in order to render the implementation portable. It is difficult to realize the temporary backtracking mechanism in “standard” C, because it needs stack walk, accessing variables the values of which are located below the current frame in the execution stack. This implementation exploits nested functions [15] to realize stack walk.

3.3.1 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 it provide legitimate stack access. 

Figure 8 shows an example of a program with nested functions.

When the function bk_exit1 nested in fib is (indirectly) called, a parameter bk_exit0 and n, and a local variable 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 goto and any intermediate functions as well. In the program in Fig. 8, when bk_exit1 is called with the argument n0=n, s is set to v and the control goes back to fib, exiting bk_exit1 and all the intermediate functions between bk_exit1 and fib. This capability is not used in the baseline Tascell implementation, but is used to implement the non-local exit mechanism in the enhanced Tascell.

The most well-known implementation of nested functions for C is the trampoline-based implementation in GCC [16], [17]. In addition, L-closure-based implementations of nested functions are proposed for achieving low maintenance/creation costs by delaying the initialization of the closure until it is invoked and enabling register allocation. Two versions of L-closure implementations exist: a translator to standard C, called LW-SC [18], [19], and an enhancement of GCC, called XC-cube [20]. However, XC-cube does not support goto that exits a nested function.

3.3.2 Translation to C with Nested Functions

The program in Fig. 7 is translated to the program in Fig. 9 with nested functions. 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 Fig. 9) 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) that corresponds to the second newest handler (which calls another nested function for the third newest handler and so on). Only if a task request still remains, the worker calculates a range for a new task, updates a range for itself, and creates a new task and sent to the requester. After sending a task, the worker returns from the nested function and resumes its own computation. Translation for a dynamic_wind statement is also included in Fig. 9. As you can see, statement_body employs a nested function (_bk2_dwind in Fig. 9), 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), in order to perform undo/redo operations as described in Section 3.2.

4. Language Extension to Tascell

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

- try compound-statement
  catch (expression) compound-statement
- throw expression;

The try construct above does not have the “finally” clause, because Tascell already has the dynamic_wind construct, as described in Section 3.2.
A try-catch statement indicates that an exception could be thrown during the execution of compound-statement1. The evaluation steps of a try-catch statement are as follows. First, expression is evaluated and an exception is caught if an exception occurred during the evaluation. Then, compound-statement1 is executed. The catch is disestablished when a worker exits this statement normally or abnormally.

A throw statement creates an exception tagged with the value of expression. The expression must be of type void and the exception becomes a throwable exception. The try block must have a catch block that catches the exception, and the throw statement must be within the try block.

A try-catch statement indicates that an exception could be thrown during the execution of compound-statement1. The evaluation steps of a try-catch statement are as follows. First, expression is evaluated and an exception is caught if an exception occurred during the evaluation. Then, compound-statement1 is executed. The catch is disestablished when a worker exits this statement normally or abnormally.

A throw statement creates an exception tagged with the value of expression. The expression must be of type void and the exception becomes a throwable exception. The try block must have a catch block that catches the exception, and the throw statement must be within the try block.
b) When an exception tagged with 1 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 1. After noticing the flag, worker 0 goes back to join 1 and performs a throw operation for the returned tag 1 as if task 1-0 is replaced with the throw. This exception is then caught at catch 1. Since the parallel for corresponding to join 2 is 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 1, 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, and the others ignored. We cannot simply employ the same semantics as Cilk Plus or X10 presented in Sections 2.1 and 2.2, because a task may be aborted without returning a result or an exception.

If an exception is thrown during the execution of cleanup operations, the new exception is propagated from there, discarding the old one (if any), as in Java[21]. Furthermore, a 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 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.

5. Implementation of Exception Handling

We implemented the exception handling mechanism for Tascell presented in Section 4 by modifying the Tascell compiler and the task scheduler provided by Tascell, which are presented in Section 5.1 and Section 5.2, respectively.

5.1 Tascell Compiler

Since the Tascell compiler is implemented as a translator to C code, techniques are required to implement the non-local exit mechanism with cleanup code execution in finally clauses. Although the setjmp method and two return values method are well known as techniques for implementing such a mechanism as a translator to C[22], we implemented it by exploiting nested functions, which are already used for the temporary backtracking mechanism of Tascell, in order to minimize the implementation cost and additional overheads to the baseline Tascell.

The functions task_exec and search in Fig. 10 are translated to the programs in Fig. 12 and Fig. 13, respectively. Each try-catch statement is translated into a piece of code that includes a definition of a nested function (lines 22–45 in Fig. 12), 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 56 in Fig. 13), aborting a task (line 38), or spawning tasks (line 42). Temporary backtracking for spawning tasks is executed in the same manner as in the baseline Tascell, as explained in Section 3.3.2. 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 16–23). 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 3.3.1 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 5–17 in Fig. 12) is called to exit pentomino_task_exec. After exiting pentomino_task_exec, the scheduler notices that the task is exited with an exception return from the fact that _thr->backtrack_rsn is EXCEPTION. Backtracking for aborting a task is done 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 38 and 39 in Fig. 12), and a worker retrieves an exception and rethrows it if a task spawned at a parallel for statement has returned the exception (lines 18–21 in Fig. 13).

5.2 Task Scheduler

In order to support the exception handling mechanism, we enhanced the task scheduler of Tascell as follows:

Fig. 10 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.
We implemented cancellation flags of tasks and partial cancellation flags of parallel for statements, and
We 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 notify the abortion of a task to which a cancellation flag is set. When a worker returns an exception as a task result, a partial cancellation flag is set to the parallel for statement at which the task is spawned. In addition, cancellation flags are set to all the tasks that are spawned at the parallel for statement and all the parallel for statements dynamically enclosed by it.

Cancellation flags are set also when a worker performing backtracking for propagating an exception reaches a parallel for statement; flags are set to all the tasks spawned at the statement (line 17 in Fig. 13). In addition, when a cancellation flag is set to a task, flags are set to all the tasks spawned during its recursive execution.

At every entry point of parallel for statement, a worker checks whether a (partial) cancellation flags are set (lines 35–39 in Fig. 13), and if any flags exist, the worker (partially) aborts the task by calling nested functions.

Note that a (partial) cancellation flag may be suspended, such as task 2-0 in Fig. 11, and a worker cannot notice such a flag only by the check for a task being executed. However, we can guarantee that such a task will become active immediately, after other active tasks are aborted. This is because, as a result of the Leapfrogging employed by the current Tascell implementation (Section 3.2), a suspended task is always an ancestor of a task being executed, and we implemented the sched-

---

**Fig. 12** Translation result from the function task.exec for Pentomino in Fig. 10, including translation of a try-catch statement.

**Fig. 13** Translation result from the worker function search for Pentomino in Fig. 10, including translation of a parallel for statement and a throw statement.
ter so that, when a flag is set to a task, flags are also set to all its descendant tasks automatically. Thus, it is necessary only to allow each worker to check for a task being executed.

6. Performance Evaluation

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

- $\text{Fib}(n)$: recursively computes the $n$-th Fibonacci number.
- $\text{Nq}(n)$: finds all solutions to the $n$-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.
- $\text{Pen}(n)$: finds all solutions to the Pentomino problem with $n$ pieces, using additional pieces and an expanded board for $n > 12$.

The evaluation environment is summarized in Table 1.

6.1 Overheads

In order to evaluate the overheads of the exception handling mechanism, we measured the performance of the baseline and enhanced implementations of Tascell using $\text{Fib}(n)$, $\text{Nq}(n)$, and $\text{Pen}(n)$. In addition, in order to evaluate the cost of the exception handlers, we measured the performance of the programs that perform the same computation to $\text{Fib}(n)$, $\text{Nq}(n)$, and $\text{Pen}(n)$, 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 shown in Table 2 (sequential executions) and Fig. 14 (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 the measurement conditions. Note that the overheads are very small even for $\text{Fib}(51)$, which performs the checking for cancellation flags very frequently. Furthermore, except for $\text{Fib}(51)$, which creates exception handlers very frequently, the cost of try blocks is relatively small: the performance degradation as compared to the baseline Tascell is less than 16% for $\text{Nq}(17)$ and $\text{Pen}(15)$. According to these results, we can expect that the technique can be used to abort the redundant search shown in Fig. 4, which requires that an exception handler be created at every search step, without large costs.

For $\text{Pen}(15)$, the performance of the C program is worse than that of Tascell. Although not certain, a bad optimization of GCC could have caused the performance degradation.

6.2 Task Abortion Time

In order to evaluate the time taken to abort tasks, we measured the performance of the programs that perform the same computation as $\text{Fib}(n)$, $\text{Nq}(n)$, and $\text{Pen}(n)$, 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 Fig. 10 for $\text{Pen}(n)$. The threshold $\theta$ is set to $\theta = \alpha \cdot A$, 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 (computation terminates without exceptions

![Fig. 14](https://i.imgur.com/3.png)
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 is thrown ($T_{\text{throw}}$).

The measurement results are shown in Fig. 15. In addition, as for the 16-worker executions, 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 Fig. 16. The number of aborted tasks here includes tasks that terminate returning exceptions and tasks 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 the executions), even when an exception is thrown in the middle of the execution and tens of tasks are aborted collaterally.

7. Conclusion and Future Work

We proposed an implementation of exception handling such that all running parallel tasks in a try block with an exception are collaterally aborted as soon as possible, as an enhancement of the existing task-parallel language, Tascell. We implemented the non-local exit mechanism by exploiting nested 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. Our implementation achieved an exception mechanism with low overheads and short task abortion time.

Future work will include the implementation and evaluation of the proposed exception handling mechanism in distributed memory environments and for other task-parallel languages, such as Cilk. We will also attempt to improve the performance of parallel search for various practical applications, such as graph mining, using exceptions to reduce redundant search.

The proposed Tascell implementation is available at https://bitbucket.org/tasuku/sc-tascell.

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

The authors wish to thank Prof. H. Nakashima of Kyoto University for his helpful discussions.

References
Shingo Okuno was born in 1989. He received his B.E. in Information Systems Engineering from Osaka University in 2012, and his M.E. in Informatics from Kyoto University in 2014. He has been a Ph.D. student at Department of Systems Science, Graduate School of Informatics, Kyoto University since 2014. His research interests include parallel programming languages and high performance computing. He won the IPSJ Best Paper Award in 2010. He is a member of IPSJ and the Japan Society for Software Science and Technology (JSST).

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