Regular Paper

An Implementation of Exception Handling with Collateral Task Abortion

Tasuku Hiraishi[1,a) Shingo Okuno[2] Masahiro Yasugi[3]

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 `f0` 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) [*1], where the tasks spawned by worker 0 in

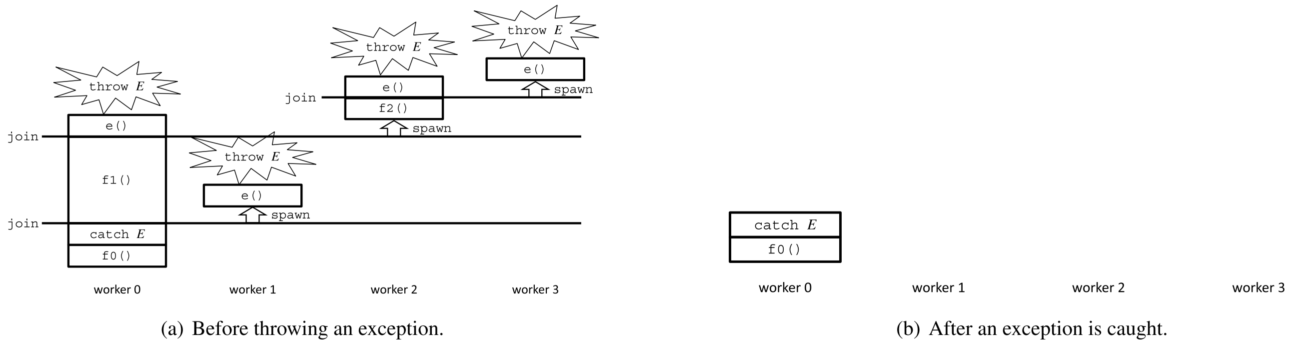
```
1  f0 () {
2    try {
3      join {
4        spawn e();
5        f1();
6      }
7    } catch (E) {
8      ...
9    }
10 }
11 f1 () {
12   join {
13     spawn f2();
14     e();
15   }
16 }
17 f2 () {
18   join {
19     spawn e();
20     e();
21   }
22 }
23 e () {
24   if (exceptional condition?) {
25     throw (E);
26   } else {
27     ...
28   }
29 }
```

Fig. 1 Example of task-parallel program with exceptions.

*1 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.

1 Academic Center for Computing and Media Studies, Kyoto University, Kyoto 606–8501, Japan
2 Graduate School of Informatics, Kyoto University, Kyoto 606–8501, Japan
3 Department of Artificial Intelligence, Kyushu Institute of Technology, Fukuoka 820–8502, Japan
a) tasuku@media.kyoto-u.ac.jp

© 2016 Information Processing Society of Japan



(a) Before throwing an exception. (b) After an exception is caught.

Fig. 2 Execution states in an execution of the program in Fig. 1.

```
1  binsearch (Node root) {
2    int found = 0;
3    try {
4      binsearch_r (Node root);
5    } catch (E) {
6      found = 1;
7    }
8    /* Returns 1 if a solution is found. */
9    return found;
10 }
11 binsearch_r (Node node) {
12   if (!node) { return; }
13   if (Solution found at node?) {
14     throw (E);
15   } else {
16     join {
17       spawn binsearch_r (node->right);
18       binsearch_r (node->left);
19     }
20   }
21 }
```

Fig. 3 Parallel binary tree search for one solution.

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` may or may not throw the exception E that would be caught by the catcher established in `f0`. One desired behavior when at least one of the workers has thrown the exception is that the execution stack becomes as shown in Fig. 2 (b) as soon as possible. For example, when only worker 1 has thrown the exception, not only the task assigned to worker 1 is 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. Our proposed implementation realizes such a behavior.

As explained in Ref. [6], 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 parallel binary tree search, the search can be terminated as soon as a solution is found simply by throwing an exception caught at the root of the search tree, as shown in **Fig. 3**.

Furthermore, a more attractive use of this exception handling mechanism in parallel tree search was recently reported [7]. 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 4** shows pseudo code for a parallel binary search for all solutions using this technique. When a worker notices that no solutions exist in the subtree having the root `xnode`, it prunes the subtree (lines 4–8). Then, when another worker notices that the worker is traversing the pruned subtree,

```
1  binsearch (Node node) {
2    int s0=0, s1=0, s2=0;
3    if (!node) { return 0; }
4    if (Noticed that there are no solutions in the subtree the root of which
5        is xnode?) {
6      /* "Prune" the subtree the root of which is xnode. */
7      prune (xnode);
8    }
9    if (Is node included in a pruned subtree the root of which is xnode?) {
10     throw (xnode);
11   }
12   try {
13     join {
14       if (Solution found at node?) {
15         s0 = 1;
16       }
17       s1 = spawn binsearch (node->right);
18       s2 = binsearch (node->left);
19     }
20   } catch (node) {
21     /* catches an exception the tag value of which equals node */
22     ...
23   }
24   /* Returns the number of solutions. */
25   return s0+s1+s2;
26 }
```

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

it can abort the traversal simply by throwing an exception tagged with `xnode`, which is caught by the catcher (line 20) 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 a part of the pruned subtree is assigned to other workers. The implementation of these operations without exceptions is complicated: programmers need to implement not only the transferal of the control back to the root of the pruned subtree, but also the abortion of the involved 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 achieve search space reduction.

A semantics for such language features was proposed in Ref. [8]. However, no major language supports them. For example, Intel Cilk Plus supports throwing an exception beyond synchronization points, but the child tasks spawned in the try block are not aborted automatically. Therefore, we implemented exception handling features with collateral task abortion as an enhancement of the existing task-parallel language Tascell [4]. 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

© 2016 Information Processing Society of Japan

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

© 2016 Information Processing Society of Japan

```
int a[12];     // manage unused pieces
int b[70];     // the board, with (6+sentinel) × 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;
}
```

Fig. 5 Program (pseudo-code written in C) that performs backtrack search for finding all possible solutions to the Pentomino puzzle.

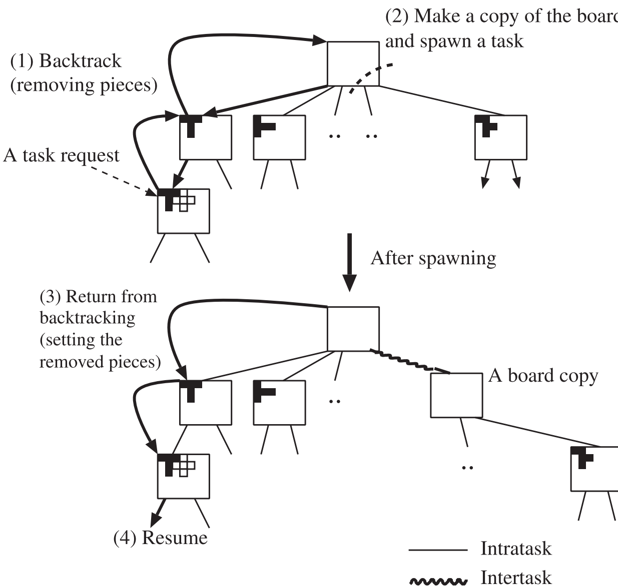


Fig. 6 Illustration of task spawning employing temporary backtracking. (1) The backtracking step includes undo operations (i.e., removing pieces). (2) The spawning-half-iterations step includes making a copy of the temporarily restored board. (3) The returning-from-backtracking step includes redo operations (i.e., setting pieces).

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

```
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
   }
  }
 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;
}
```

Fig. 7 A Tascell program that performs backtrack search for Pentomino.

`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* `:` $expr_{from}$ `,` $expr_{to}$`)` $statement_{body}$
`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. When the implicit task-request handler (available during the iterative execution of $statement_{body}$) is invoked, the *upper half* of the remaining iterations are spawned as a new *task-name* task, the object of which is initialized by $statement_{put}$. In $statement_{put}$, the actual assigned range can be referred to by $identifier_{from}$ and $identifier_{to}$. 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 $statement_{get}$. In order not to be idle, the worker requests and executes other tasks

© 2016 Information Processing Society of Japan

```
 1  int fib(void (*bk_exit0) (int, int), int n)
 2  {
 3    __label__ l_exit;   /* label should be declared explicitly */
 4    int s = 0;
 5    /* nested function */
 6    void bk_exit1 (int n0, int v) {
 7      if (n == n0) {
 8        s = v;
 9        goto l_exit;    /* jump to l_exit in fib */
10      }
11      bk_exit0(n0, v);  /* call caller's nested function */
12      return;
13    }
14    if (found that fib(n0)= v) bk_exit1(n0, v);
15    if (n <= 2) return 1;
16    {
17      int s1=0, s2=0;
18      s1 = fib(bk_exit1, n - 1);
19      s2 = fib(bk_exit1, n - 2);
20      s = s1 + s2;
21    }
22   l_exit:
23    return s;
24  }
```

Fig. 8 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 $statement_{body}$. 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 is 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*$_1$
  `catch` (*expression*) *compound-statement*$_2$
- `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.

© 2016 Information Processing Society of Japan

```
 1  int search (void(*_bk0) (void), struct thread_data *_thr,
 2              int k, int j0, int j1, int j2,
 3              struct pentomino *tsk)
 4  {
 5   int s = 0;                   // the number of solutions
 6   { /*------------------ parallel for ------------------*/
 7    int p = j1;   int p_end = j2;
 8    struct pentomino *pthis;
 9    int spawned = 0;              // the number of spawned tasks
10    void _bk1_par_for (void){   // nested function
11     if (!spawned) _bk0();      // continue backtracking
12     while (p + 1 < p_end && task request exists?) {
13      int i1 = (1 + p + p_end)/2,
14          i2 = p_end;           // the range for the sub-task
15      p_end = i1;               // shrink the range for itself
16      pthis = malloc(sizeof(struct pentomino));
17                                // allocate a workspace
18      { //statement_put
19        copy_piece_info(pthis->a, tsk->a);
20        copy_board(pthis->b, tsk->b);
21        pthis->k = k; pthis->i0 = j0;
22        pthis->i1 = i1; pthis->i2 = i2;
23      }
24      spawned++;
25      make_and_send_task(_thr, 0, pthis); // spawn
26     }
27    }
28    if (_thr->req)    //check task requests
29     _bk1_par_for(); //start backtracking for spawning tasks
30    for (; p < p_end; p++) {
31     int ap = tsk->a[p];
32     for (each possible direction d of the piece) {
33      // examine the "i-th" (piece, direction)
34      ...local variable definitions here ...
35      if(Can the ap-th piece in the d-th direction be placed
36         on the board tsk->b?);
37      else continue;
38      { /*------------ dynamic_wind ------------*/
39       { // do operation (statement_before)
40         Set the ap-th piece onto the board tsk->b and update tsk->a.
41       {
42        void _bk2_dwind (void) // nested function
43        {
44         { // undo operation (statement_after)
45           Backtrack, i.e., remove the ap-th piece from tsk->b
46           and restore tsk->a. }
47         _bk1_par_for(); // continue backtracking (call the
48                         // nested function defined above)
49         { // redo operation (statement_before)
50           Set the ap-th piece onto the board tsk->b
51           and update tsk->a. }
52        }
53        { // statement_body
54          kk = the next empty cell;
55          if (no empty cell?) s++;      // a solution found
56          else   // try the next piece
57           s += search (_bk2_dwind, _thr,
58                        kk, j0+1, j0+1, 12, tsk);
59        }
60       }
61       { // undo operation (statement_after)
62         Backtrack, i.e., remove the ap-th piece from tsk->b
63         and restore tsk->a. }
64      } /*------------ dynamic_wind ------------*/
65     }
66    }
67    while (spawned-- > 0) {
68     // Get and integrate results of spawned tasks
69     pthis = (struct pentomino *)wait_rslt(_thr);
70     s += pthis->s;          // statement_get
71     free(pthis); }
72   } /*------------------ parallel for ------------------*/
73   return s;
74  }
```

Fig. 9 Translation result from the worker function `search` for Pentomino in Fig. 7, including translation of a parallel `for` statement and a `dynamic_wind` statement.

A `try-catch` statement indicates that an exception could be thrown during the execution of *compound-statement*$_1$. The evaluation steps of a `try-catch` statement 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 an exception tagged with the value of *expression* (cast to `size_t`) and throws it. When an exception tagged with *tag* is thrown, the most recent try block having a catcher 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 exception return is notified to the victim worker of the aborted task so that a flag named *partial cancellation flag* with *tag* is attached to the corresponding parallel `for` statement in the victim. Then, the victim notices that its task has a parallel `for` with the flag raised, and then, raises an exception as if the stolen iterations of parallel `for` were replaced with `throw` *tag*`;`. In our current implementation, the check for the existence of such a parallel `for` is executed at the same time of the check for a task request, that is, at every entry point of any (other) parallel `for`, so that the victim notices it as soon as possible before it becomes idle waiting for the completion of the parallel `for` and eventually receives the exception 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 uncompleted parallel `for` statements exist in the dynamic scope of an exception catcher to be disestablished, *cancellation flags* are set to the uncompleted tasks spawned from such parallel `for` statements and all their descendant tasks. Then, each thief notices the message, by polling again for immediate abortion, for the stolen task that it is executing, and aborts 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_wind`s are nested) before exiting the corresponding `try` block, as in temporary backtracking described in Section 3.2.

Figure 10 shows an example of a Tascell program using exceptions, which performs a backtrack search for Pentomino as the program in Fig. 10 and terminates the search as soon as a worker finds that the number of solutions is larger than `THRESHOLD`[*2]. We show examples of exception handling using **Fig. 11**, which shows a cactus stack representing an execution context of the program in Fig. 10 with four workers, supposing the following two cases.

a) When an exception tagged with 1, which would be caught at `catch 1` 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 is aborted, worker 2 resumes task 2-0, but immediately aborts it, since worker 2 notices the cancellation flag. The control of worker 0 returns to `catch 1` after the cleanup operations at (1), (2), and (3) are executed in that order.

[*2] This program is used only as an example and for performance evaluations in Section 6. Obviously, this algorithm is not efficient because a worker cannot count the number of solutions found by other tasks until it merges their results.

© 2016 Information Processing Society of Japan

```
1  task pentomino {
2    The same code as in Fig. 7
3  };
4  task_exec pentomino {
5    this.s = 0;
6    if (The root of the search?) {
7      try {
8        search (this.k, this.i0, this.i1, this.i2,
9                &this);
10     } catch (1) {
11       this.s = 1;
12     }
13   } else {
14     this.s = search (this.k, this.i0, this.i1, this.i2,
15                      &this);
16   }
17 }
18 worker int search (int k, int j0, int j1, int j2,
19                    task pentomino *tsk)
20 {
21  int s=0; // the number of solutions
22  // parallel for
23  for (int p : j1, j2)
24   { The same code as in Fig. 7 }
25  handles pentomino (int i1, int i2)
26   { The same code as in Fig. 7 }
27  // Throw an exception if s exceeds THRESHOLD.
28  if (s > THRESHOLD) { throw 1; }
29  return s;
30 }
```

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.

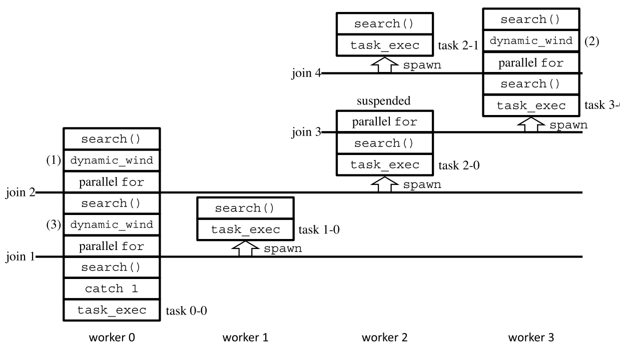


Fig. 11 Execution context of the program in Fig. 10.

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_wind`s, 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:

© 2016 Information Processing Society of Japan

```
1  void pentomino_task_exec (struct thread_data *_thr,
2                            struct pentomino *pthis)
3  {
4    __label__ pentomino_abort;
5    void _bk (void)
6    {
7      switch (_thr->backtrack_rsn) {
8       EXCEPTION:
9       CANCEL:
10      // The exception is not caught in the task,
11      // or the task is aborted by a cancellation flag.
12       goto pentomino_abort; // exit the task
13      SPAWN:
14      // _bk is called for spawning tasks (temporary backtracking).
15       return;    // return from the backtracking.
16     }
17   }
18   pthis->s = 0;
19   if (The root of the search?) {
20     { /*------------------ try-catch ------------------*/
21       __label__ catch_exit;
22       void catch_bk (void) {
23         switch (_thr->backtrack_rsn) {
24          EXCEPTION:
25           // Check if the exception tag is equal to the catcher tag
26           if (_thr->excep_tag == 1) {
27             _thr->backtrack_rsn = NOT_BACKTRACKING;
28             { // Execute the code in the catch block
29               pthis->s = 1;
30             }
31             goto catch_exit;  // exit the try block
32           } else {
33             // Continue backtracking since the exception is not caught
34             _bk (); // never returns
35             system_error();
36           }
37           break;
38          CANCEL:
39           _bk (); // continue backtracking (never returns)
40           system_error();
41          SPAWN:
42           _bk ();
43           return;
44         }
45       }
46       { // Execute the code in the try block
47         search (catch_bk, _thr,
48                 pthis->k, pthis->i0, pthis->i1,
49                 pthis->i2, pthis);
50       }
51       catch_exit:;
52     } /*------------------ try-catch ------------------*/
53   } else {
54     pthis->s = search (_bk, _thr,
55                        pthis->k, pthis->i0, pthis->i1,
56                        pthis->i2, pthis);
57   }
58   pentomino_abort:;
59 }
```

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

```
1  int search (void(*_bk0) (void), struct thread_data *_thr,
2              int k, int j0, int j1, int j2,
3              struct pentomino *tsk)
4  {
5   int s = 0;                 // the number of solutions
6   { /*------------------ parallel for ------------------*/
7    int p = j1;   int p_end = j2;
8    struct pentomino *pthis;
9    int spawned = 0;           // the number of spawned tasks
10   void _bk1_par_for (void) { // nested function
11    switch (_thr->backtrack_rsn) {
12     EXCEPTION:
13     CANCEL:
14     // The reason for backtracking is an exception or a cancellation flag.
15     // Abort and synchronize with all the tasks spawned at this parallel for.
16      while (spawned-- > 0) {
17       pthis = (struct pentomino *) abort_and_wait (_thr);
18       if (pthis has a returned exception?) {
19         _thr->backtrack_rsn = EXCEPTION;
20         _thr->excep_tag = pthis->excep_tag;
21       }
22       free (pthis);
23      }
24      _bk0(); // continue backtracking (never returns)
25      system_error();
26     SPAWN:
27      // The reason for backtracking is a task request.
28      if (!spawned) _bk0();      // continue backtracking
29      while (p + 1 < p_end && task request exists?) {
30        The same code as Fig. 9 lines 13–25 (make and spawn a task)
31      }
32      return;
33    }
34   }
35   if (_thr->task_top->cncl)
36   {                  // check whether (partial) cancellation flags are set
37     _thr->backtrack_rsn = CANCEL;
38     _bk1_par_for(); // start backtracking for aborting the task
39   }
40   if (_thr->req) {  //check task requests
41     _thr->backtrack_rsn = SPAWN;
42     _bk1_par_for(); //start backtracking for spawning tasks
43   }
44   for (; p < p_end; p++) {
45     The same code as Fig. 9 lines 31–65 (loop body)
46   }
47   while (spawned-- > 0) {
48     The same code as Fig. 9 lines 68–71
49     (Get and integrate results of spawned tasks)
50   }
51  } /*------------------ parallel for ------------------*/
52  if (s > THRESHOLD) {
53   { /*------------------ throw ------------------*/
54     _thr->backtrack_rsn = EXCEPTION;
55     _thr->excep_tag = 1; // set the exception tag value
56     _bk1_par_for(); //start backtracking for propagating the exception
57   } /*------------------ throw ------------------*/
58  }
59  return s;
60 }
```

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.

- 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 backtrack 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 cancellation flag is set to a task being executed and partial cancellation flags to parallel `for` statements in the task [*3]. If any flags exist, the worker (partially) aborts the task by calling nested functions (lines 35–39 in Fig. 13).

Note that a task to which a (partial) cancellation flag is set 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-

[*3] In order to avoid costly operations for checking all the parallel `for` statements periodically, we implement a task object having a counter of partial cancellation flags.

© 2016 Information Processing Society of Japan

uler 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:

- Fib(n): recursively computes the n-th Fibonacci number.
- 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.
- 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 Fib(n), Nq(n), and 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 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 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 Fib(51), which performs the checking for cancellation flags very frequently. Furthermore, except for 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 Nq(17) and 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 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.

Table 1 Evaluation environment.

	Appro Green Blade 8000 (1 node)
CPU	Intel Xeon E5-2670 2.3 GHz 8-core × 2 (16 cores in total)
Memory	DDR3-1600 64 GB
OS	Red Hat Enterprise Linux Server release 6.6 (Santiago)
Compiler	GCC 4.4.7 with `-O3` option
Nested functions	Trampoline-based implementation in GCC
Worker	Created by `pthread_create` with `PTHREAD_SCOPE_SYSTEM`

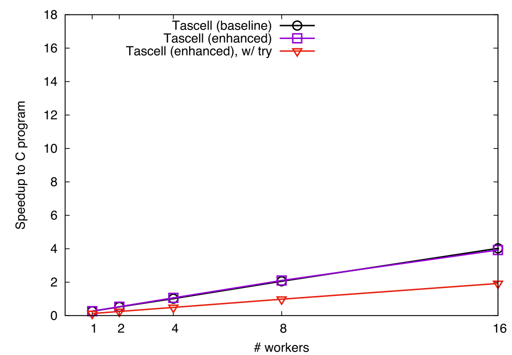
Table 2 Execution time and relative time to sequential C programs with one worker.

Elapsed time in seconds (relative time to plain C)

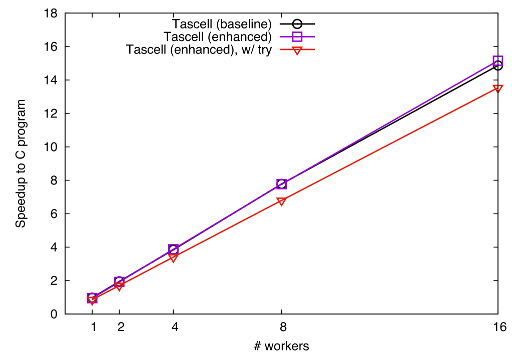
	C	Tascell (baseline)	Tascell (enhanced)	Tascell (enhanced, w/ `try`)
Fib(51)	54.3 (1.00)	208 (3.82)	203 (3.74)	432 (7.97)
Nq(17)	464 (1.00)	476 (1.03)	489 (1.05)	540 (1.16)
Pen(15)	685 (1.00)	640 (0.933)	630 (0.920)	738 (1.07)

6.2 Task Abortion Time

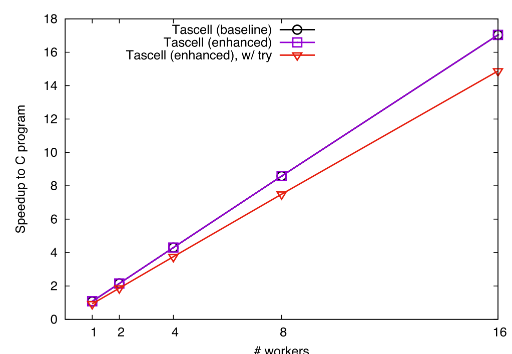
In order to evaluate the time taken to abort tasks, we measured the performance of the programs that perform the same computation as Fib(n), Nq(n), and Pen(n), respectively, but terminate the computation by throwing an exception as soon as a worker finds that the answer is larger than a threshold, as shown in Fig. 10 for Pen(n). The threshold θ is set to $\theta = \alpha \cdot A$, where A is the true answer of the computation and α is set to 0–0.3 in units of 0.01 and 0.4–1 in units of 0.1 (computation terminates without exceptions



(a) Fib(51)



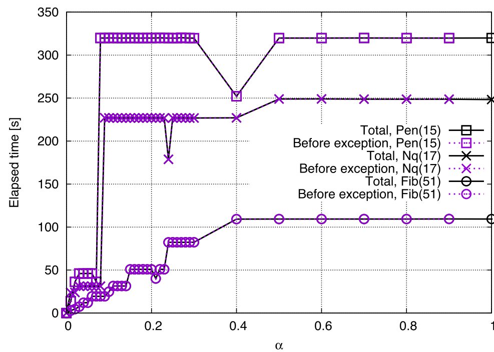
(b) Nq(17)



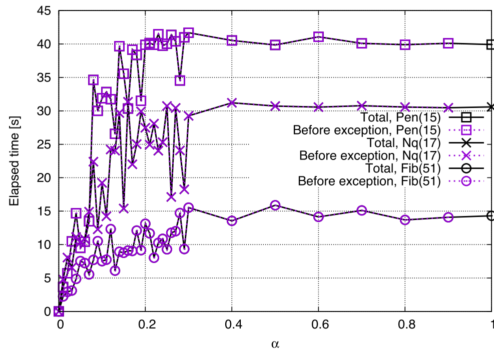
(c) Pen(15)

Fig. 14 Speedups relative to C with multiple workers.

© 2016 Information Processing Society of Japan



(a) 2 workers



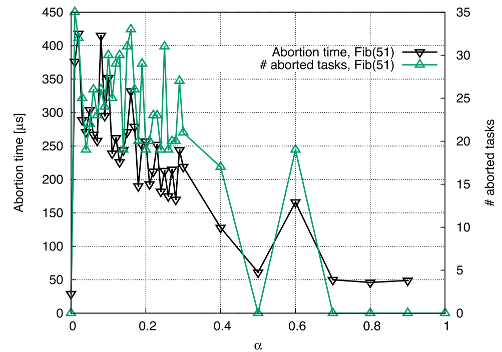
(b) 16 workers

Fig. 15 Elapsed time between the time when an exception is thrown and the termination of the computation.

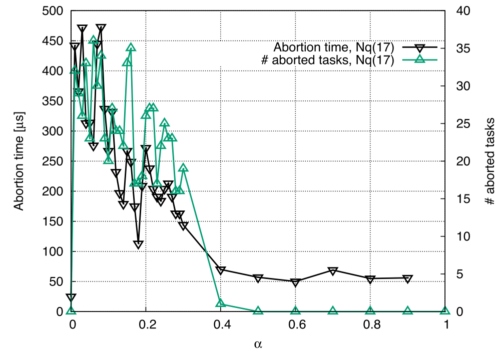
when $\alpha = 1$). We executed these programs using 2 and 16 workers, and measured the total elapsed time (T) and the elapsed time before the first exception is thrown (T_{throw}).

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.

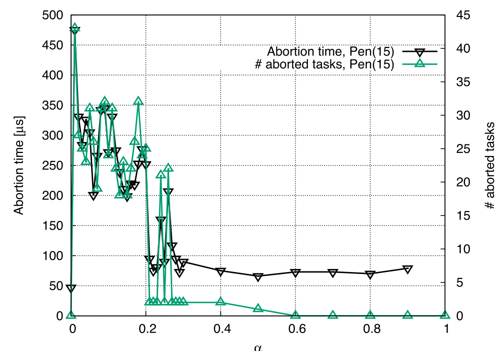
We can see that the abortion time increases in proportion to the number of aborted tasks, but is very short (less than 500 μs in all the executions), even when an exception is thrown in the middle of the execution and tens of tasks are aborted collaterally.



(a) Fib(51)



(b) Nq(17)



(c) Pen(15)

Fig. 16 Elapsed time between the first throw operation and the termination of the program execution, and the number of aborted tasks (16-worker executions).

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

[1] IBM Corporation: X10: Performance and Productivity at Scale, available from ⟨http://x10-lang.org⟩.
[2] Frigo, M., Leiserson, C.E. and Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices* (*PLDI '98*), Vol.33, No.5, pp.212–223 (1998).
[3] Intel Corporation: A quick, easy and reliable way to improve threaded performance—Intel Cilk Plus, available from ⟨https://software.intel.

© 2016 Information Processing Society of Japan

com/en-us/intel-cilk-plus⟩.
[4] Hiraishi, T., Yasugi, M., Umatani, S. and Yuasa, T.: Backtracking-based Load Balancing, *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP 2009*), pp.55–64 (2009).
[5] Feeley, M.: A Message Passing Implementation of Lazy Task Creation, *Proc. International Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, Lecture Notes in Computer Science, No.748, pp.94–107, Springer-Verlag (1993).
[6] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science: *Cilk 5.4.6 Reference Manual* (2008).
[7] Okuno, S., Hiraishi, T., Nakashima, H., Yasugi, M. and Sese, J.: Reducing Redundant Search using Exception Handling in a Task-Parallel Language, *Annual Meeting on Advanced Computing System and Infrastructure* (*ACSI*) *2015* (2015).
[8] Yasugi, M.: Hierarchically Structured Synchronization and Exception Handling in Parallel Languages Using Dynamic Scope, *Proc. International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications* (1999).
[9] Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O. and Grove, D.: *X10 Language Specification Version 2.5* (2014), available from ⟨http://x10.sourceforge.net/documentation/languagespec/x10-251.pdf⟩.
[10] Lea, D.: A Java Fork/Join Framework, *Proc. ACM JAVA '00*, pp.36–43 (2000).
[11] OpenMP Architecture Review Board: *OpenMP Application Program Interface Version 4.0* (2013), available from ⟨http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf⟩.
[12] Yasugi, M., Hiraishi, T., Umatani, S. and Yuasa, T.: Parallel Graph Traversals using Work-Stealing Frameworks for Many-core Platforms, *Journal of Information Processing*, Vol.20, No.1, pp.128–139 (online), DOI: 10.2197/ipsjjip.20.128 (2012).
[13] Wagner, D.B. and Calder, B.G.: Leapfrogging: A Portable Technique for Implementing Efficient Futures, *Proc. Principles and Practice of Parallel Programming* (*PPoPP'93*), pp.208–217 (1993).
[14] Kelsey, R., Clinger, W. and Rees, J.: Revised^5 Report on the Algorithmic Language Scheme, *ACM SIGPLAN Notices*, Vol.33, No.9, pp.26–76 (1998).
[15] Free Software Foundation, Inc.: Nested Functions: Using the GNU Compiler Collection (GCC), available from ⟨https://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html⟩.
[16] Stallman, R.M.: Using and Porting GNU Compiler Collection (1999).
[17] Breuel, T.M.: Lexical Closures for C++, *Usenix Proceedings, C++ Conference* (1988).
[18] Hiraishi, T., Yasugi, M. and Yuasa, T.: A Transformation-Based Implementation of Lightweight Nested Functions, *IPSJ Digital Courier*, Vol.2, pp.262–279, *IPSJ Trans. Programming*, Vol.47, No.SIG 6(PRO 29), pp.50–67 (2006).
[19] Tazuke, M., Yasugi, M., Hiraishi, T. and Umatani, S.: Reducing Invocation Costs of L-Closures, *IPSJ Trans. Programming*, Vol.6, No.2, pp.13–32 (2013). (in Japanese).
[20] Yasugi, M., Hiraishi, T. and Yuasa, T.: Lightweight Lexical Closures for Legitimate Execution Stack Access, *Proc. 15th International Conference on Compiler Construction* (*CC2006*), Lecture Notes in Computer Science, No.3923, pp.170–184, Springer-Verlag (2006).
[21] Gosling, J., Joy, B., , Steele, G., Bracha, G. and Buckley, A.: The Java® Language Specification Java SE 8 Edition (2015), available from ⟨https://docs.oracle.com/javase/specs/jls/se7/jls8.pdf⟩.
[22] Chiba, Y.: Implementations of Exception Handling in a Java2C Translator, *IPSJ Trans. Programming*, Vol.42, No.SIG 11(PRO 12), pp.14–24 (2001). (in Japanese).

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

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

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.

© 2016 Information Processing Society of Japan