Hierarchically Structured Synchronization and Exception Handling in Parallel Languages Using Dynamic Scope

Masahiro Yasugi
Department of Communications and Computer Engineering,
Graduate School of Informatics, Kyoto University
Sakyo Kyoto, JAPAN 606-8501
E-mail: yasugi@kuis.kyoto-u.ac.jp

In high-level parallel programming languages for practical parallel processing, it is desired to be able to describe practical parallel programs with irregular computation and/or side effect easily and safely as well as to execute them on parallel computers efficiently. Important properties for good description include the ease of writing programs, the support of describing a variety of parallel processing and exception handling, and restrictions on the use of harmful operations such as the ones corresponding to `goto` statements in some sequential languages. This paper shows the effectiveness of parallel languages with hierarchically structured synchronization and exception handling using dynamic scope to describe a variety of parallel processing easily and safely with a small set of language constructs. The structured synchronization and exception handling are expressed by syntactic constructs rather than individual operations. In structured languages using those constructs, the user can easily picture the configuration of the current execution context and can naturally handle an exception thrown in the course of parallel execution. The description length and the possibility of the presence of bugs are also reduced compared to the conventional approach that uses some synchronization operations. This paper shows the proposed language design and its effectiveness and also describes the language semantics and the implementation issues.

1 Introduction

High-level programming languages for parallel processing are quite useful to develop reliable, reusable and efficient applications on various parallel architecture including shared-memory architecture and distributed-memory architecture. In high-level parallel programming languages for practical parallel processing, it is desired to be able to describe practical parallel programs with irregular computation and/or side effect easily and safely as well as to execute them on parallel computers efficiently.

This paper shows the effectiveness of parallel languages with hierarchically structured synchronization and exception handling using dynamic scope to describe a variety of parallel processing easily and safely with a small set of language constructs. This paper also presents the design of an object-oriented parallel language based on the hierarchical structure, the semantics of a simplified structured language, the discussion on extending the synchronization construct to the degree of the exception handling construct, and the implementation issues.

A possible way to concisely describe parallel programs is to restrict types (or patterns) of parallel processing and employ a language construct for the restriction. An example is to restrict the types to data-parallel and employ a `forall` construct, which reduces the possibility of the presence of bugs compared to languages in which synchronization among threads must be described explicitly; such synchronization is implicit in the `forall` construct. Furthermore, an exception that is thrown during the execution of a `forall` statement can be properly handled as the exception of the `forall` statement itself. However, the restriction posed by `forall` is too strict to describe a variety of parallel processing including irregular computations.

In order to describe programs for irregular problems, a number of languages, in which operations for thread creation and thread synchronization can be described, have been designed. We admit that the explicit use of thread creation operations is necessary since the irregularity of the problem defers the detection of a task suitable for parallel execution until runtime. However, we do not recommend the explicit use of thread synchronization operations, since they make the programs difficult to understand and increase the description length, sometimes causing bugs. This is because they prevent programs from being hierarchically structured by connecting some distant program points in such a way that one cannot recognize the structure in a top-down manner; the similar situation occurs with the use of `goto` statements.

In the proposed language design, structured synchronization and exception handling are expressed by syntactic constructs (using dynamic scope) rather than individual operations. This is a useful restriction in parallel languages to prevent bugs; the similar restrictions in sequential languages include the use of a `while` construct rather than arbitrary control transfer by `goto` statements and the use of object-oriented encapsulation rather than arbitrary field access to simple records. That is to say, this study deals with the *structured programming* for parallel programming. The difference between "explicit operations" and "syntactic constructs", which this paper compares, can be explained with a loop example in C language as follows: with "explicit operations", the programmer describes a loop by putting a label and a conditional branch around the repeatedly-executed part (to arbitrarily connect some distant program points by spit-operations) but that part cannot be considered inside the loop in terms of the program structure, while the body of a "syntactic constructs"-based `while` statement can be considered inside the loop in terms of the program structure where the target of a `continue` statement can be automatically determined by the nesting level in the program structure.

In the structured languages using those syntactic constructs, the user can easily picture a configuration of the current parallel-execution context and can naturally handle an exception thrown in the course of parallel execution. The type of parallel processing that our language syntax supports is fork-join parallel in which partitioning of a given task into subtasks is irregular but the completion of the given task can be achieved by the synchronization of the subtask completions. Of course, we do not argue that all patterns of parallel processing belong to fork-join type, but many patterns which have a definite goal, such as "to get the answer" or "to be completed for the next step", belong to this type.

The rest of the paper is organized as follows. In Section 2, we point out issues on the description of parallel programs. In Section 3, we describe the effectiveness of parallel languages with hierarchically structured synchronization and exception handling using dynamic scope, and the design of an object-oriented parallel language based on the hierarchical structure. In Section 4, semantics of a simplified structured language is examined. In Section 5, we discuss an extended synchronization construct to the degree of the exception handling construct. Implementation issues are presented in Section 6. In Section 7, we discuss the related work.

2 Issues on the description of parallel programs

In this section, we discuss expressiveness of various constructs for the description of parallel programs in terms of (1) conciseness of the description,[a] (2) describable types of parallel processing, and (3) the ease of handling exceptions. We use a base sequential language with the following assumptions:

- Constructs for parallel processing are explicit.
- Side-effects are permitted. (such as in C and Java)
- Fine granularity of parallel processing is permitted by language systems.

2.1 Description of parallel execution by syntactic constructs

Let us review the `forall` construct in terms of the three criteria. The `forall` construct can be used to describe parallel processing as follows:

`forall(i=1 to` N`)` *stat*

where N threads are created and they execute *stat* in parallel, and their completions are automatically synchronized. Concurrent Pascal-style `cobegin`

[a]We will focus on the structure and contents of programs; thus, we will ignore redundancy which simply helps the readability and error detection.

...`coend` construct can be used to specify a distinct statement for each thread as follows:

`cobegin` $stat_1$`;` $stat_2$`;` ... $stat_N$ `coend`

where N threads are created and each executes the corresponding $stat_i$ and their completions are automatically synchronized.

The description by these constructs has the following features. (1) It is concise. (2) It only supports a simple type of fork-join parallel and do not support a variety of parallel processing including irregular computations except for hierarchical structures that are formed by describing `forall` or `cobegin ... coend` in *stat* recursively. (3) An exception that is thrown during the execution of a `forall` statement (including parallel execution of *stat*) can be properly handled as the exception of the `forall` statement itself. We can define the language semantics so that, if the `forall` statement is wrapped by a `try-catch` statement, the exception thrown during the parallel execution can be caught by the exception handler of the `try-catch` statement and the whole parallel execution is stopped without describing individual `stop` operations.

2.2 Thread creation and thread synchronization by operations

In order to describe a variety of parallel processing including irregular computations, a number of languages, in which operations for thread creation and thread synchronization (including operations on locations for synchronization) can be described, have been designed. For example, in Java language, [1] a thread can be created at runtime and various operations with the reference to the thread are supported. Here, we consider the following operations:

```
thr = spawn stat;
```

where a thread executing *stat* is created and the reference to the thread is obtained in the variable `thr`, and

```
join(thr);
```

where the completion of the thread referred to by `thr` is waited for. The combination of these operations enable the following description of parallel processing where the number of created threads is not fixed:

```
{
  int i, n = 0;
  thread_t thr[N];
  for(i=0;i<N;i++)
    if(...) thr[n++] = spawn stat;
  for(i=0;i<n;i++) join(thr[i]);
}
```

where a thread executing *stat* is created only when the condition is met and the synchronization of the completion of every thread is expressed explicitly by the `join` statement.

Similarly, in some languages which employ data-flow synchronization (i.e., a thread which tries to extract the value from a location is suspended until the value of the location is determined), the following operations would be used:

```
ch = future exp;
```

where a thread evaluating *exp* in parallel is created and the reference to the location into which the result value will be stored is obtained in the variable `ch`, and:

```
val = touch(ch);
```

where the value stored in the location referred to by `ch` is extracted to `val` after the necessary suspension. The following description of parallel processing where the number of created threads is not fixed is similar to that using `spawn` and `join`:

```
{
  int i, n = 0, sum = 0;
  int_channel ch[N];
  for(i=0;i<N;i++)
    if(...) ch[n++] = future exp;
  for(i=0;i<n;i++) sum += touch(ch[i]);
}
```

where a thread evaluating *exp* is created only when the condition is met and the synchronization of the completion of every thread is expressed explicitly by the `touch` expression.

The description by these operations for thread creation (such as `spawn`, `future`) and thread synchronization (such as `join`, `touch`) has the following features. (1) It is not considered concise. In particular, the synchronization code (the loops with `join` or `touch` in the above examples) and thread management code (the array operations with `thr` or `ch` in the above examples) are required for the correct synchronization. The possibility of introducing bugs increases with the too specific description.[b] (2) It supports a variety of parallel processing including irregular computations. (3) The handling of an exception that is thrown during the execution of *stat* (or *exp*) is subtle because the way how the exception can be propagated outside *stat* is not trivial. In order to properly handle the exception, the language has to prepare operations for propagating the exception to the parent thread and for stopping a thread

[b] I experienced that, if the description of synchronization is incorrect, a serious symptom where the bug identification is difficult may be led by a zombie thread (i.e., the programmer considers it dead at some point while it continues its execution in practice).

whose result is no longer needed, then the programmer has to describe the exception handling explicitly and carefully with the timing consideration. In Java, operations for stopping multiple threads can be briefly described using a `ThreadGroup` object to manage related threads, but the operation itself cannot be omitted. In some language designs, [2] when a thread performs a `join` operation to another thread in which an exception is thrown, it receives the exception automatically, and when a thread performs the `touch` operation to a location to which an exception is propagated, it receives the exception automatically; however, operations for stopping threads are still required and also propagation of an exception is deferred until the corresponding synchronization operations are performed. Furthermore, if the language design employs explicit operations for storing a value to a synchronization location, the operation itself will not sometimes be executed due to an exception; thus, propagation of an exception to a location is sometimes impossible.

In languages where thread creation and thread synchronization are described with explicit operations, the user cannot easily picture a configuration of the current parallel-execution context. This is because the synchronization point is not known until the synchronization operation is actually performed; such a synchronization point is the point where the result of the thread execution is necessary and should be known for the user to know the goal why the thread is being executed.

2.3 Thread synchronization by syntactic constructs

To reduce the description for thread management, a syntactic construct is useful. For example, in Cilk [3] which is a parallel C dialect, the `cilk` construct can be used to define a `cilk` function, which automatically manages threads created during the execution of the function body:

```
cilk void foo(...) {
  int i;
  for(i=0;i<N;i++)
    if(...) spawn funcall;
  sync;
}
```

where a thread executing *funcall* is created only when the condition is met and the synchronization of the completions of multiple threads is expressed explicitly by the `sync` statement. The threads created within the lexical scope (i.e. the `cilk` function body) are automatically managed, and the `sync` statement expresses the synchronization of the completions of all threads which have been created before the `sync` statement is performed. In Cilk, thread creation is permitted only within `cilk` function bodies, and the same synchronization as the `sync` statement is implicitly performed when returning from `cilk` functions.

Compared to the description by operations for thread creation and thread synchronization, the description by the `cilk` construct and the `sync` operation has the following features. (1) It is more concise. Thread management code is eliminated and synchronization code is also reduced to a single `sync` statement. The possibility of introducing bugs decreases with the description length. (2) It supports a variety of parallel processing including irregular computations to some degree with the restriction that the programmer cannot directly specify threads involved in some synchronization and that one cannot make a thread to survive across function-call boundaries which means that one cannot define an independent function to abstract several thread creations. (3) The handling of an exception that is thrown during the execution of *funcall* is subtle since the `sync` statement is still an operation. In Cilk, an `abort` statement can be used to stop the threads that are automatically managed by the `cilk` construct but the propagation of the exception is not supported. (In practice, we can improve Cilk to support deferred propagation.)

Here we consider a new syntactic construct `waitfor` which expresses both thread management (rather than by `cilk`) and thread synchronization (rather than by `sync` operations) as follows:

```
waitfor stat;
```

where the completions of the threads created by `spawn` within the lexical scope (i.e. the `waitfor` body similar to the `cilk` function body) are synchronized. An example is as follows:

```
{
  int i;
  waitfor for(i=0;i<N;i++)
            if(...) spawn funcall;
}
```

where a thread executing *funcall* is created only when the condition is met and the synchronization of the completions of the threads created within the scope is expressed by the `waitfor` construct without explicit operations (other than `waitfor` construct itself).

Compared to the description in Cilk, the description by the `waitfor` construct has the following features. (1) It is more concise since `sync` operations are perfectly removed, and the possibility of introducing bugs decreases (2) It adds the restriction that the programmer cannot change the synchronization point at runtime. (3) An exception that is thrown during the execution of a `waitfor` statement (including parallel execution of *funcall*) can be properly handled as the exception of the `waitfor` statement itself just like a `forall` statement. We can define the language semantics so that, if the `waitfor` statement is wrapped by a `try-catch` statement, the exception thrown during the parallel execution can be caught by the exception handler of the `try-catch` statement and the whole parallel execution is stopped without describing individual `stop` or `abort` operations.

3 Synchronization construct and exception handling construct using dynamic scope

Dynamic scope means indefinite scope and dynamic extent, where references to an established entity may occur anywhere and at any time in the interval between establishment of the entity and the explicit disestablishment of the entity. In this section, we first describe that exception handlers (i.e. catchers) for the exception handling based on `catch-throw` have dynamic scope, then we propose and examine the `fork-join` style synchronization where join targets (i.e. synchronizers) have dynamic scope.

3.1 Exception handling based on catch-throw

We first explain the description and the meaning of exception handling in Java language. An exception can be thrown by a `throw` statement:

`throw` *exp* ;

where, the value of *exp* must be (a reference to) an object representing the exception. Only objects that are instances of the `Throwable` class (or of one of its subclasses) can be thrown by the Java throw statement. (Some exceptions are thrown by the Java Virtual Machine.) The exception stops the current execution and the control is transferred to the exception handler which is referred to using dynamic scope.

The `try-catch-finally` construct is prepared for exception handling. Exception handlers for an exception thrown during the execution of a `try` block are described as `catch` clauses:

```
try {
  ... // an exception may be thrown.
}catch(Exception1 ex1){
  ... // may be executed for Exception1
}catch(Exception2 ex2){
  ... // may be executed for Exception2
}finally{
  ... // always executed
}
```

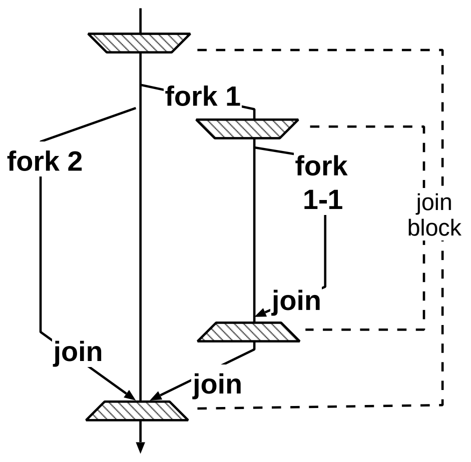

Figure 1: Structured synchronization

A `try` statement executes a `try` block. The `catch` clauses will not be executed when no exception is thrown. If an exception is thrown and the `try` statement has one or more `catch` clauses that can catch it, then control will be transferred to the first such `catch` clause. A `catch` clause can catch and handle an exception (object) of the specified class (or of one of its subclasses). If an exception is not caught, the exception will be propagated to `catch` clauses for the outer try block. The exception handler is referred to using dynamic scope; thus, the nesting of `try` blocks is dynamic: for example, if an exception is not caught in a method body, the exception is propagated to the calling point of the method.

If the `try` statement has a `finally` clause, then the `finally` block is executed, no matter whether the `try` block completes normally or abruptly (with an exception), and no matter whether a `catch` clause is first given control. If an exception is thrown during the execution of the `finally` block, the old exception (if any) is discarded.

3.2 Synchronization using dynamic scope

For synchronization, join targets (i.e. synchronizers) may have dynamic scope. More precisely, we can define that a synchronizer established by `waitfor` in Section 2.3 is dynamically scoped. For example, the execution of the following statement can be illustrated in Fig. 1:

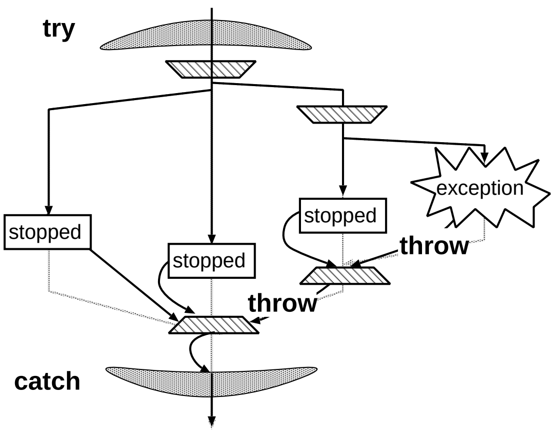

Figure 2: Handling an exception which is thrown during parallel processing

```
  waitfor {
    spawn f1();
    f();
  }
```

where `f1` and `f` are defined as follows:

```
  f1(){ ...
    waitfor {... spawn f1_1(); ...}
  }
  f(){ ... spawn f2(); ... }
```

The join block in the figure represents the interval during which the body of `waitfor` is executed, where `f1` has a nested `waitfor` statement. The join target of `spawn f2()` executed in `f` is referred to using dynamic scope. The join target of `spawn f2()` will not change even if we replace `f()` with `spawn f()`. The same `waitfor` construct is employed in COOL [4] which is a parallel dialect of C++, where thread creation is performed by calling parallel functions (functions defined with keyword `parallel`); however, COOL does not involve exception handling.

In terms of the concerns discussed in Section 2, the description by these constructs using dynamic scope has the following features. (1) It is concise. (2) It supports a variety of parallel processing including irregular computations to some degree with the restriction that that the programmer cannot change the synchronization point at runtime. The restriction that the programmer cannot make a thread to survive across function-call boundaries is removed by using dynamic scope, which means that one can define an independent function such as `f` to perform `spawn f2`. (3) An exception that is thrown during the execution of a `waitfor` statement (including parallel execution) can be properly handled as the exception of the `waitfor` statement itself just like a `forall` statement. We can define the language semantics so that, if the `waitfor` statement is wrapped by a `try-catch` statement, the exception thrown during the parallel execution can be caught by the exception handler of the `try-catch` statement and the whole parallel execution is stopped without describing individual `stop` or `abort` operations:

```
try{
  waitfor {
    spawn f1();
    f();
  }
}catch( ... ){
  ...
}
```

The execution of the above statement can be illustrated in Fig. 2. If an exception cannot be handled by a thread, the exception is propagated to the join target of the thread, which then stops the other threads sharing the same join target. Thus, the description is simple and the handling of the exception during the parallel execution is not subtle.

In languages where thread creation and thread synchronization are described with syntactic constructs, the user can easily picture a configuration of the current parallel-execution context. This is because the synchronization point is known when a thread is created; such a synchronization point is the point where the result of the thread execution is necessary and is regarded as a goal why the thread is being executed. In sequential languages, an execution context forms a data structure, namely *stack*. Here we imagine an ideal control stack which only saves control transfer information and does not rely on an automatically-incremented program counter. The stack top holds the information about the statement to be executed. If the execution of a function body (consisting of statements) or a block body (consisting of (sub)statements) is required to execute the current statement (which is popped from the stack), those (sub)statements are pushed onto the stack. As such a stack, the user can picture the configuration of the current execution context or can know the goal of the current execution. On the other hand, the stack for parallel execution in Fig. 1 would be a *cactus stack*, which changes as is shown in Fig. 3. Every created thread has a goal to continue the work after the join; it has its own (sub)stack on a *join frame* which is used as a join target (Fig. 3). The

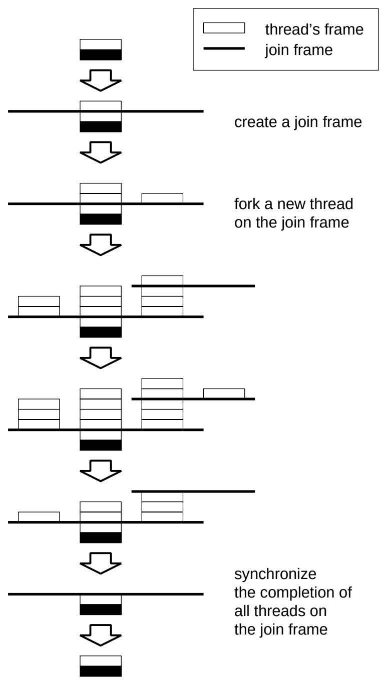

Figure 3: Transition of the cactus stack based on the structured synchronization

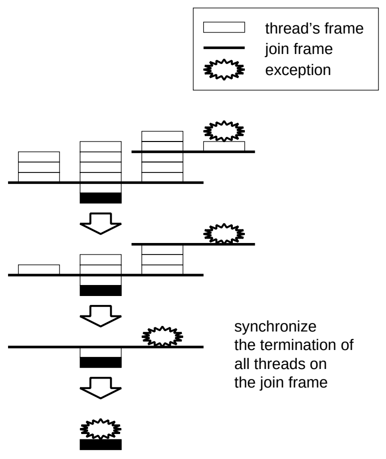

Figure 4: Transition of the cactus stack for an exception thrown during parallel processing

join frame returns its control only after all the stacks on the top of it become empty. The user can picture the configuration of the current parallel execution context as a cactus stack. Furthermore, an exception that is thrown during the parallel execution (Fig. 2) makes the cactus stack change as is shown in Fig. 4. This transitions are also intuitive to the users.

So far we have not discussed the subtle cases caused by multiple threads and finally clauses. Only one exception should survive if two or more exceptions reaches to the same join frame; thus, a problem is how to determine the survivor. There is a similar case in the sequential execution: if an exception is thrown during the execution of the finally block, the old exception (if any) is discarded in Java. However, there is no difference among parallel exceptions in terms of execution order. One solution to this problem would be to give a priority to each exception. The other problem is how to precisely define the behavior of the stopped thread. It is an elegant idea to define the behavior as to automatically throw a special (non-user) exception `stopped` except for two cases: when `stopped` is being thrown and when a finally block is being executed. Although the execution of a `finally` block is not stopped by the other threads, the `finally` block itself may throw an exception, possibly discarding the `stopped` exception. However, this is not a problem since re-throwing of `stopped` is automatic.

3.3 An object-oriented parallel language OPA

We are developing an object-oriented parallel language OPA, [5,6] where we employ the proposed synchronization and exception handling using dynamic scope. OPA (Object-oriented language for PArallel processing) is a parallel extension of Java language[1]; we remove specifications on Threads and Monitors from Java then add new constructs for structured synchronization and relaxed mutual exclusion. Its design is intended to realize both ease-of-use and high performance of parallel processing.

OPA supports irregular parallelism with dynamically forked threads. In OPA, patterns of parallel processing can be divided into three types according to the relation among threads, namely fork-join parallel, cooperative parallel, and exclusive parallel. In fork-join parallel, we divide a task into two or more subtasks; when a divided subtask can be executed in parallel, we can fork and join a new thread for the subtask in a structured manner. OPA also supports cooperative parallel processing in which the related threads synchronize/communicate with each other in the course of their execution. In addition, mutual exclusion (serialization) is necessary for concurrent accesses to an object to read/write the object's data consistently.

OPA employs structured syntactic constructs using dynamic scope for the fork-join parallel; thus, the user can easily and safely describe synchronization among the completions of multiple threads. Objects are employed for synchronization among cooperatively parallel threads which perform message passing to the shared objects. OPA provides useful classes for synchronization, such as an I-store class where threads executing `get` method are suspended until a thread performing `put` method sets a value. For mutual exclusion, OPA provides, in addition to `synchronized` methods, `instant` methods to support efficient concurrent access to an object by automatically dividing `instant` methods into read-only (RO) type and read-write (RW) type. Both RO and RW methods read the necessary variables of the object into implicit local variables atomically at the beginning of the method. An RW method writes the values of local variables into the object atomically at the point where the rest of the method execution no longer updates the local variables. The compiler automatically determines the update point with flow analysis.

The proposed structured constructs and their implementation do not depends on the OPA specific language features; however, OPA provides objects and mutual exclusion to enclose or split side effects with variable updates spatially or temporally as will be mentioned in Section 4.

OPA employs a `par` construct and a `join` construct instead of `spawn` and `waitfor` in Section 3.2. The rest of paper will use these `par` and `join` constructs. By attaching keyword `par` to a method call (or a statement), the execution of the method call (or the statement) is performed by a newly forked thread. By "`join` *statement*," *statement* is executed by the current thread and the completions of the new threads created during the execution of *statement* are joined with the completion of the `join` statement:

```
join{
   par obj1.m1();  // create a thread
   par obj2.m2();  // create a thread
}  // synchronize the completions of the created threads
```

When a value calculated by a created thread is used for the rest of computation, components of a compound statement can be separated with a `join` label to indicate that the part before the `join` label is a join block:

```
{
  int x = par f1(n);
  int y = par f2(m);
join:
  z = x + y;
}
```

where the scope of the bindings of the variables (such as `x`, `y`) initialized by created threads is below the `join` label. The presence of the `join` label and the appropriate use of variables can be checked at compile time.

The syntax for exception handling in OPA is the same as in Java. In OPA, however, an exception thrown during parallel execution within a `try` block is also properly handled as was described in Section 3.2. In the following example, an exception thrown during the execution of `obj1.m1()` is examined for the exception handler of the `catch` clause:

```
try {
  join{
     par obj1.m1();  // create a thread
     par obj2.m2();  // create a thread
  }  // synchronize the completions
}catch(Exception1 ex1){  // exception handler
  ...
}
```

The thread which handles an exception has to execute necessary `finally` clauses before the control is transfered to a catch clause. Other threads that are stopped due to the exception have to execute necessary `finally` clauses before their termination. If a thread has acquired a lock for an object, the lock should be released as if the unlock instruction were written as a `finally` clause. With an `instant` method, the update of the object's data is performed atomically at a single update point; therefore, if an exception is thrown before the update point, the object's data remains unchanged. OPA introduces a `vflush` statement to enforce the update before throwing an exception. The consistency control over multiple objects is not performed automatically and must be specified explicitly.

3.4 Examples written in OPA

Some examples are presented to see we can describe a variety of parallel programs easily and safely with a small set of language constructs, such as `join` and `par` constructs for synchronization, and `try-catch-finally` and `throw` constructs for exception handling.

To perform a parallel method call for each element of an array `objs` in a data-parallel manner, we can write the program as follows:

```
join for(i=0; i<objs.length; i++)
         par objs[i].doit();
```

To calculate the sum of `items` held in leaf objects of a binary tree which is composed of instances of the class `BinTree` in parallel, we can write the program as follows:

```
  class BinTree {
    int item; boolean isLeaf;
    BinTree left, right;
    ...
    instant int getSum(){
      if( isLeaf ) return item;
      else{
        int x = par left.getSum();
        int y = par right.getSum();
       join:
        return x + y;
      }
    }
  }
```

When we can wrap the calculation with `try-catch`, we can naturally handle an exception which may be thrown during the parallel execution:

```
  try{
    sum = root.getSum();
  }catch(...){
    ...
  }
```

To find two answers which are searched in parallel and stop the whole execution with an exception (the fact that the answers were found), we can write the program as follows:

```
 // main class  (to catch the exception)
 public class SearchStart {
   public static
    void main(String argv[]) {
     Table table = new Table();
     try{
       join{
         Node node = new Node(0);
         node.search(table);        // start parallel search
       }
       System.out.println("NotFound");
     }catch(Found excp){            // if answers are found
       System.out.println("Found");
     }
   }
 }
```

```
// Node for search space
class Node {
  int p;
  Node(int p0){ p = p0; }
  void search(Table table) throws Found {
     ...
    if (an answer is found) table.add(the answer);
    else{
      Node node1 = new Node(2*p+1),
           node2 = new Node(2*p+2);
      par node1.search(table);
      par node2.search(table);
    }
  }
}

// Table for answers (throws an exception)
class Table {
  int[] answers = new int[2];
  int n = 0;
  instant void add(int ans) throws Found {
    answers[n++] = ans;
    if (n >= 2) {
      vflush;
      throw new Found(this); // non local exit with exception
    }
  }
}
```

4 Language semantics

The cactus stacks illustrated in Fig. 3 and Fig. 4 are considered to represent the rest of computation from some point during parallel execution, or a hierarchical parallel continuation. In this section, we formalize continuations for a simplified imperative sequential language, in which only boolean values are supported and the storage locations are directly expressed. Then we extend the language to the parallel one and try to formalize hierarchical parallel continuations for approaching to the concept of hierarchical parallel continuations. The similar notation to report[7] is used to describe the semantics below.

Abstract syntax of the simplified imperative language:

$\Gamma \in$ Com	commands
E $\in$ Exp	expressions
L $\in$ Loc	locations
K $\in$ Con	constants
T $\in$ Tag	catch-throw tags

$\Gamma ::=$ L `:=` E $|$ Γ `;` Γ
 $|$ `if` E `then` Γ `else` Γ $|$ `while` E `do` Γ
 $|$ `throw` T $|$ `catch` T `in` Γ $|$ `try` Γ `finally` Γ
E ::= K $|$ L

Domain equations:

$\epsilon \in \mathtt{E} = \{\mathit{false}, \mathit{true}\}$	values
$\sigma \in \mathtt{S} = \mathrm{Loc} \to \mathtt{E}$	stores
$\theta \in \mathtt{C} = \mathtt{S} \to \mathtt{A}$	continuations
$\kappa \in \mathtt{K} = \mathtt{E} \to \mathtt{C}$	expression continuations
$\rho \in \mathtt{U} = \mathrm{Tag} \to \mathtt{C}$	tag environments
$\mathtt{A}$	answers

Semantic functions:

$\mathcal{K} : \mathrm{Con} \to \mathtt{E}$
$\mathcal{E} : \mathrm{Exp} \to \mathtt{K} \to \mathtt{C}$
$\mathcal{C} : \mathrm{Com} \to \mathtt{U} \to \mathtt{C} \to \mathtt{C}$
$\mathcal{C}\,[\![\mathrm{L}\ \mathtt{:=}\ \mathrm{E}]\!]\,\rho\,\theta = \mathcal{E}\,[\![\mathrm{E}]\!]\,\lambda\epsilon.\lambda\sigma.\theta(\sigma\{\mathrm{L} \mapsto \epsilon\})$
$\mathcal{C}\,[\![\Gamma_1\ \mathtt{;}\ \Gamma_2]\!]\,\rho\,\theta = \mathcal{C}\,[\![\Gamma_1]\!]\,\rho\,(\mathcal{C}\,[\![\Gamma_2]\!]\,\rho\,\theta)$
$\mathcal{C}\,[\![\mathtt{if}\ \mathrm{E}\ \mathtt{then}\ \Gamma_1\ \mathtt{else}\ \Gamma_2]\!]\,\rho\,\theta = \mathcal{E}\,[\![\mathrm{E}]\!]\,(\lambda\epsilon.\epsilon \to \mathcal{C}\,[\![\Gamma_1]\!]\,\rho\,\theta,\ \mathcal{C}\,[\![\Gamma_2]\!]\,\rho\,\theta)$
$\mathcal{C}\,[\![\mathtt{while}\ \mathrm{E}\ \mathtt{do}\ \Gamma]\!]\,\rho\,\theta = \mathit{fix}\ \theta'.\ (\mathcal{E}\,[\![\mathrm{E}]\!]\,(\lambda\epsilon.\epsilon \to \mathcal{C}\,[\![\Gamma]\!]\,\rho\,\theta',\ \theta))$
$\mathcal{C}\,[\![\mathtt{throw}\ \mathrm{T}]\!]\,\rho\,\theta = \rho(\mathrm{T})$
$\mathcal{C}\,[\![\mathtt{catch}\ \mathrm{T}\ \mathtt{in}\ \Gamma]\!]\,\rho\,\theta = \mathcal{C}\,[\![\Gamma]\!]\,\rho\{\mathrm{T} \mapsto \theta\}\,\theta$
$\mathcal{C}\,[\![\mathtt{try}\ \Gamma_1\ \mathtt{finally}\ \Gamma_2]\!]\,\rho\,\theta$
$= \mathcal{C}\,[\![\Gamma_1]\!]\,\{\mathrm{T}_1 \mapsto \mathcal{C}\,[\![\Gamma_2]\!]\,\rho\,\theta_1, \ldots, \mathrm{T}_n \mapsto \mathcal{C}\,[\![\Gamma_2]\!]\,\rho\,\theta_n\}(\mathcal{C}\,[\![\Gamma_2]\!]\,\rho\,\theta)$
where $\rho\ = \{\mathrm{T}_1 \mapsto \theta_1, \ldots, \mathrm{T}_n \mapsto \theta_n\}$
$\mathcal{E}\,[\![\mathrm{K}]\!]\,\kappa = \kappa(\mathcal{K}\,[\![\mathrm{K}]\!]\,)$
$\mathcal{E}\,[\![\mathrm{L}]\!]\,\kappa = \lambda\sigma.\kappa(\sigma(\mathrm{L}))\sigma$

where $\sigma\{\mathrm{L}_1 \mapsto \epsilon_1\}$ denotes a function σ' such that $dom(\sigma') = dom(\sigma) \cup \{\mathrm{L}_1\}$ and $\sigma'(\mathrm{L}) = \sigma(\mathrm{L})$ for $\mathrm{L} \in dom(\sigma) - \{\mathrm{L}_1\}$ and $\sigma'(\mathrm{L}_1) = \epsilon_1$ (the substitution "σ with ϵ_1 for L_1"), $\epsilon \to \theta_1, \theta_2$ is McCarthy conditional "if ϵ than θ_1 else θ_2" and $\{\mathrm{T}_1 \mapsto \theta_1, \ldots, \mathrm{T}_n \mapsto \theta_n\}$ denotes a function ρ such that $dom(\rho) = \{\mathrm{T}_1, \ldots, \mathrm{T}_n\}$ and $\rho(\mathrm{T}_i) = \theta_i$ for $i \in \{1, \ldots, n\}$.

The imperative language with the above denotational semantics also has an operational semantics in terms of contexts; the operational semantics is defined by the following context rewriting rules where the store $s \in \text{Loc} \to \text{Con}$ is slightly different from $\sigma \in \text{Loc} \to \texttt{E}$.

Abstract syntax for contexts:

G ∈ CCtxt	command contexts
F ∈ ECtxt	expression contexts
C ∈ Ctxt	contexts

C ::= G | E :: F | `protected` :: C
F ::= `update`(L) :: G | `select`(Γ, Γ) :: G | `then`(Γ) :: G
G ::= Γ :: G | `mark`(T) :: G | `finally` Γ :: G | `pmark` :: G

Context rewriting rules $(s, \text{C}) \to (s', \text{C}')$:

1. (s, L `:=` E :: G) → (s, E :: `update`(L) :: G)
2. (s, Γ_1 `;` Γ_2 :: G) → (s, Γ_1 :: Γ_2 :: G)
3. (s, `if` E `then` Γ_1 `else` Γ_2 :: G) → (s, E :: `select`(Γ_1, Γ_2) :: G)
4. (s, `while` E `do` Γ :: G) → (s, E :: `then`(Γ `;` `while` E `do` Γ) :: G)
5. (s, `throw` T :: Γ :: G) → (s, `throw` T :: G)
6. (s, `throw` T :: `mark`(T') :: G) → (s, `throw` T :: G)
7. (s, `throw` T :: `mark`(T) :: G) → (s, G)
8. (s, `throw` T :: `finally` Γ :: G) → (s, `finally` Γ :: `throw` T :: G)
9. (s, `throw` T :: `pmark` :: G) → (s, `pmark` :: `throw` T :: G)
10. (s, `catch` T `in` Γ :: G) → (s, Γ :: `mark`(T) :: G)
11. (s, `mark`(T) :: G) → (s, G)
12. (s, `try` Γ_1 `finally` Γ_2 :: G) → (s, Γ_1 :: `finally` Γ_2 :: G)
13. (s, `finally` Γ :: G) → (s, `protected` :: Γ :: `pmark` :: G)
14. (s, `protected` :: `pmark` :: G) → (s, G)
15. (s, C) → (s', C′) infers
    (s, `protected` :: C) → (s', `protected` :: C′)
16. (s, `true` :: `select`(Γ_1, Γ_2) :: G) → (s, Γ_1 :: G)
17. (s, `false` :: `select`(Γ_1, Γ_2) :: G) → (s, Γ_2 :: G)
18. (s, `true` :: `then`(Γ) :: G) → (s, Γ :: G)
19. (s, `false` :: `then`(Γ) :: G) → (s, G)
20. (s, L :: F) → (s, s(L) :: F)
21. (s, K :: `update`(L) :: G) → ($s\{\text{L} \mapsto \text{K}\}$, G)

where "`throw` T" discards a command or mark at the top of the stack unless it matches "`mark`(T)" (rules 5–7), which may be set by "`catch` T `in` Γ" on the stack (rule 10). The execution of `finally` clause (rule 12) is not canceled by an exception (rule 8) and is protected by `protected` and `pmark` (rule 13).

This protection is necessary only for parallel execution and basically has no effect on the sequential semantics (rules 9,14,15).

A context can be mapped into a continuation; that is, the context (as a data structure) represents a continuation. The same context also represents every continuation which is used on the non-local exit with a catch-throw tag. The single context can represent these continuations simultaneously because the language is well structured with the proper constructs.

Now, the operational semantics of a parallel language can be described as an extension to that of the sequential language. We employ a parallel language where the following two commands, namely `par` command and `join` command, are appended to the sequential language.

Abstract syntax of the parallel language:

$\Gamma ::= \cdots \mid \texttt{par}\ \Gamma \mid \texttt{join}\ \Gamma$

We employ the following hierarchical parallel context H (instead of C) for the context of the parallel language.

Abstract syntax of contexts of the parallel language:

$\mathrm{H} \in \mathrm{PCtxt}$ hierarchical parallel contexts

$\mathrm{G} ::= \cdots \mid \texttt{term}$

$\mathrm{H} ::= \mathrm{G} \mid \mathrm{E} :: \mathrm{F} \mid \texttt{protected} :: \mathrm{H} \mid \langle\!\langle \mathrm{H} \| \cdots \| \mathrm{H} \rangle\!\rangle :: \mathrm{G} \mid \langle\!\langle \mathrm{H} \| \cdots \| \mathrm{H} \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}$

where, the order of H in $\langle\!\langle \mathrm{H} \| \cdots \| \mathrm{H} \rangle\!\rangle$ is not significant (exchangeable) and every H can be processed in parallel. $\langle\!\langle \cdots \rangle\!\rangle$ is corresponding to the join frame in Fig. 3 and $\langle\!\langle \cdots \rangle\!\rangle_{\mathrm{q}}$ is corresponding to the quitting join frame in Fig. 3.

Since the context rewriting rules $(s, \mathrm{C}) \to (s', \mathrm{C}')$ in the sequential language can be regarded as parallel context rewriting rules $(s, \mathrm{H}) \to (s', \mathrm{H}')$ in the parallel language, we show the only following appended part of parallel context rewriting rules.

context rewriting rules $(s, \mathrm{H}) \to (s', \mathrm{H}')$:

22. $(s,\ \texttt{join}\ \Gamma :: \mathrm{G}) \to (s,\ \langle\!\langle \Gamma :: \texttt{term} \rangle\!\rangle :: \mathrm{G})$
23. $(s,\ \langle\!\langle \texttt{par}\ \Gamma :: \mathrm{G} \| \cdots \rangle\!\rangle :: \mathrm{G}_0) \to (s,\ \langle\!\langle \Gamma :: \texttt{term} \| \mathrm{G} \| \cdots \rangle\!\rangle :: \mathrm{G}_0)$
24. $(s,\ \langle\!\langle \texttt{term} \| \cdots \rangle\!\rangle :: \mathrm{G}) \to (s,\ \langle\!\langle \cdots \rangle\!\rangle :: \mathrm{G})$
25. $(s,\ \langle\!\langle \rangle\!\rangle :: \mathrm{G}) \to (s,\ \mathrm{G})$
26. $(s,\ \langle\!\langle \texttt{throw}\ \mathrm{T} :: \texttt{term} \| \cdots \rangle\!\rangle :: \mathrm{G}_0) \to (s,\ \langle\!\langle \texttt{throw}\ \mathrm{T} :: \texttt{term} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0)$
27. $(s,\ \langle\!\langle \texttt{throw}\ \mathrm{T} :: \texttt{term} \| \texttt{throw}\ \mathrm{T}' :: \texttt{term} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0)$
    $\to (s,\ \langle\!\langle \texttt{throw}\ \mathrm{T} :: \texttt{term} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0)$
    where $\mathrm{T} \neq \texttt{stopped}$
28. $(s,\ \langle\!\langle \texttt{throw}\ \mathrm{T} :: \texttt{term} \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0) \to (s,\ \texttt{throw}\ \mathrm{T} :: \mathrm{G}_0)$
29. $(s,\ \langle\!\langle \texttt{throw stopped} :: \texttt{term} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}) \to (s,\ \langle\!\langle \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G})$
30. $(s,\ \langle\!\langle \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}) \to (s,\ \texttt{throw stopped} :: \mathrm{G})$
31. $(s,\ \langle\!\langle \langle\!\langle \cdots \rangle\!\rangle :: \mathrm{G} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0) \to (s,\ \langle\!\langle \langle\!\langle \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0)$

32. $(s,\ \langle\!\langle \Gamma :: \mathrm{G} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0) \rightarrow (s,\ \langle\!\langle \texttt{throw stopped} :: \Gamma :: \mathrm{G} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0)$
    where $\Gamma \neq \texttt{throw}\ \mathrm{T}$

rules 31,32 are used for stopping other threads to abort the whole parallel execution on the quitting join frame ($\langle\!\langle \cdots \rangle\!\rangle_{\mathrm{q}}$), whereas there is no rule for stopping `protected` contexts in order not to abort the execution of finally clauses.

The parallel context rewriting rules also include the following rule to realize the nondeterministic concurrent execution:

33. $(s,\ \mathrm{H}) \rightarrow (s',\ \mathrm{H}')$ infers
    $(s,\ \langle\!\langle \mathrm{H} \| \cdots \rangle\!\rangle :: \mathrm{G}) \rightarrow (s',\ \langle\!\langle \mathrm{H}' \| \cdots \rangle\!\rangle :: \mathrm{G})$

As is shown above, the operational semantics of the parallel language is defined by the rewriting rules with hierarchical parallel contexts. Since the single storage s is referenced and updated nondeterministically by multiple threads, the hierarchical parallel continuation represented by a hierarchical parallel context can be defined as a function from s into a set of answers (a subset of `A`), where every answer will be produced by a possible sequence of operations. However, such definition is not so helpful for understanding and hierarchical parallel contexts themselves are more useful; for example, the effect of `throw` can be well understood with a hierarchical parallel context.

5 Extending the synchronization construct

This section shows a way to extend the synchronization construct to the degree of the exception handling construct. In contrast to the exception handling construct where the handler for a `throw` statement is selected based on the tag (i.e. class) of the exception, the join target (synchronizer) for a `par` statement is fixed to the one corresponding to the nearest dynamically-enclosing `join` block specified by the synchronization construct.

A simple extension enables a join target to be selected based on a tag; that is, we can consider the dual case of `catch-throw` as follows:

```
do {
  f1();
}join(Total1 s1, Total2 s2, ...){
  ... // after the synchronization
}
```

In the above `do-join` construct, the `do-join` statement first executes the `do` block. If a thread is created during the execution of the `do` block and it computes an object of one of `Total1`, `Total2`, ... classes (or of one of their subclasses) as a result, the completion of the thread is synchronized at the `join` clause. All synchronized results which are matched with `Total1` are totaled (reduced) into an object `s1` of class `Total1`.

To compute a result with a new thread, the following `pthrow` construct can be employed:

```
  pthrow exp;
```

where a new thread is created and the thread executes (evaluates) *exp* to compute a result. The join target of the new thread is selected based on the tag that is the type of the expression *exp*. We cannot use the actual type (class) of the value of *exp* as a tag, since the join target has to be determined at the thread creation time. If we use the actual type of the value, unpleasant synchronization will be taken to wait for the value to be computed. An example of the `pthrow` construct is as follows:

```
 void f1(){ ... pthrow f2(); ... }
 Total1 f2(){ ... return new Total1(...); }
```

where the expression `f2()` of type `Total1` is evaluated by a new thread.

A class for a synchronization tag (such as `Total1`) should have a `reduce` method to reduce multiple results to a single result. An example is the following `Sum` class:

```
 class Sum extends Total {
   internal int sum=0;
   instant int getSum(){ return sum; }
   instant void reduce(Sum a){
     sum += a.getSum();
   }
 }
```

By "`do{ ... }join(Sum s){ ... }`", multiple results $r_1, r_2, \ldots r_n$ of type `Sum` computed by forked threads are accumulated in a single `Sum s` with implicit "`s = new Sum()`" and "`s.reduce(`r_i`)`". By permitting hierarchical reduction, optimization is possible for an implementation: for example, each processor may accumulate local results in its own sub-result for the later global accumulation. For this purpose, each processor may prepare a unit element with "`new Sum()`".

Only expressions are allowed with the `pthrow` construct; let us enable the `par` construct to return a result:

```
 void f1(){
   Total1 par{ ... return new Total1(...);}
   Total2 par{ ... return new Total2(...);}
 }
```

where a new thread is created for the body of `par` statement and the thread `return`s a result of the type specified before the keyword `par`.

The use of tags for join targets has the following advantages: one is that we can fork a thread which survives after the synchronization for the nearest join block. We can create a thread which has indefinite extent as in Java's `Thread` class to reuse Java's class libraries in OPA. The other advantage is that we can exploit parallelism for a thread creation process itself.

6 Implementation issues

6.1 Implementation of synchronization

Synchronization of completions of related threads can be implemented with a *join frame* and *weight*. Weight is represented by an integer as is often employed in reference counting GC (garbage collection) schemes. At the beginning of a join block, the current thread creates a join frame and sets the maximum amount of weight to the join frame and holds the same amount of weight with the reference to the join frame. When the current thread forks a new thread, the current thread gives a part of weight (with the reference to the join frame) to the new thread and keeps the remaining weight. Both of the given weight and the remained weight must be positive. If the original amount of weight is just 1, this condition cannot be satisfied due to weight shortage. In such cases, a join frame with sufficient weight is created to cascade the original (parent) join frame as in indirect reference counting GC schemes. When the weight of a cascaded join frame becomes zero, the cascaded join frame returns weight 1 to the parent join frame. When a thread completes its whole execution or the execution of the current join block, it returns its weight to the corresponding join frame. A join frame subtracts its own weight by the returned weight; therefore, the detection of synchronization on a join frame is possible by checking if the weight of the join frame is zero.

The extension of synchronization described in Section 5 requires a more complex implementation. When the current thread forks a new thread, a naive implementation would be to search the join frame corresponding to the new thread backwards the cactus stack. An efficient implementation would be to divide (and postpone) the searching process and to make a piece of search every time the nearest join frame is deallocated (backwards the cactus stack); in order to divide the search, a cascaded join frame whose parent is the target of the search can be employed.

6.2 Implementation of exception handling

In the implementation of exception handling, an important point is how to deal with an exception thrown during the parallel execution as we can see the semantics in Section 4.

If an exception can be caught in the current method, the control is transfered to the catch clause (with the execution of necessary `finally` clauses). Otherwise, the exception is propagated to the calling point of the current method by *returning* the exception. If the exception cannot be caught within the current thread, the exception is propagated to the corresponding join frame and sets a quitting flag for stopping all threads on the join frame. The threads on the quitting join frame should be stopped as early as possible except for the execution of `finally` clauses.

In order to stop a thread, an implementation where the thread performs *polling* would be necessary. There is a trade-off between the overhead (frequency) of polling and the latency of thread stopping.[8] A possible optimization technique for a single polling operation is for a processor to poll a per-processor flag which is set for any quitting join frame and to investigate deeper part of cactus stack only when the flag is set.

The propagation of an exception to a deeper join frame might take a long time because of the execution of `finally` clauses or the synchronization on a shallower join frame. To make the propagation virtually faster, a provisional flag can be employed; a provisional flag of a (deeper) join frame should be set if an exception is likely to reach the join frame. Speedup would be possible by making the priority levels of the threads on the provisionally quitting join frame lower. The provisional flag will be cancelled if the exception is caught or discarded before reaching the join frame. This scheme is a sort of speculative computation that speculatively suppresses (probably) useless speculative computation.

7 Related work

7.1 Java

Parallel processing in Java[1] is described using the `Thread` class. A new thread is created by creating a new `Thread` object. The `join` operation is provided as a `join` method on the `Thread` object. As was described in Section 2, such explicit `join` operation makes the programming complicated. Exception handling in Java is designed to handle an exception within the current thread in which the exception is thrown and not to propagate outside the thread.

The design of OPA is intended to keep possible compatibility with Java; however, the synchronization and the exception handling are extended using syntactic constructs.

7.2 ABCL/1

Exception handling in a concurrent object-oriented language ABCL/1 [9] is described as methods for exception messages. In ABCL/1, every concurrent object has a thread of control and it can send a message to a concurrent object to invoke a method (script) on the target object concurrently. When an exception occurs during the execution of a method of an object which receives a message M, an exception message is generated and sent to the sender or the reply destination of M.

Since ABCL/1 is based on concurrent objects, the exception handling is also dealt with by specifying the behaviors of objects to send/receive an exception message. On the other hand, in OPA, exception handlers can be specified independently of objects; furthermore, related threads can be stopped automatically.

7.3 KL1 and Shoen

For a concurrent logic language KL1, "shoen" [10] is used to manage a group of goals. A group consists of all goals which are derived from a single initial goal. Shoens can be nested. When a goal raise an exception, the exception is handled by the shoen. Each shoen has a report stream and a control stream for the communication and it can propagate an exception to the outside of the shoen.

The approaches to the exception handing in KL1 and in OPA are similar. However, a shoen in KL1 is a process with its own I/O and it deals with the internal exception. On the other hand, in OPA, the exception handler deals with the exception for a task possibly with parallel execution.

7.4 Qlisp

The earlier Qlisp [11] is intended to describe a variety of parallel processing easily and safely with a small set of language constructs. The approach of the earlier Qlisp is very similar to our scheme; in particular, the design for stopping the related threads was described in the report. [11] In the later Qlisp, [12] lots of constructs are added; in particular, the `qwait` construct serves as the `join` construct in OPA. In OPA, we consider the extension of synchronization as was described in Section 5.

7.5 Approaches based on first class continuations

In some sequential languages, the rest of computation (continuation) can be reified as a first class continuation. The first class continuations are useful to describe non local exit, exception handling and coroutines.

In parallel languages, without first class continuations, coroutines can be realized by simply using multiple threads of control. Non local exit and exception handling can also be realized by using the `catch`-`throw` constructs of this paper. Thus, we think the necessity of first class continuations is small in parallel languages. We think, however, the notion of continuation is important to describe the semantics of parallel languages.

The study by Katz and Weise[13] and the study by Hieb and Dybvig[14,15] are Scheme-based studies on first class continuations for parallel processing. The study by Katz and Weise proposes a scheme to navigate parallel execution and to obtain the same result in parallel execution with `future`[2] and first class continuations as in sequential execution removing `future`s. The study by Hieb and Dybvig[14,15] proposes constructs to extract (i.e. capture and remove) a part (subtree) of cactus stack (as in Fig. 3) and reify it as a first class datum. The reified subtree can be called at any point. However, their construct does not support `finally` clauses.

8 Conclusion

This paper showed the effectiveness of the parallel languages with hierarchically structured synchronization and exception handling using dynamic scope to describe a variety of parallel processing easily and safely with a small set of language constructs; In particular, we showed that handling an exception which is thrown during the parallel execution can be naturally expressed and performed.

Future work includes implementing the proposed functionality into language systems and evaluating on real parallel computers.

Acknowledgments

I would like to express my gratitude to Dr. Luc Moreau for his helpful suggestions and comments in improving this paper.

References

1. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley Publishing Company, 1996.
2. R. H. Halstead. New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools. In *Parallel Lisp: Languages and Systems*, volume 441 of *LNCS*, pages 2–57, Springer, June, 1990.

3. Supercomputing Technologies Group. *Cilk-5.1(Beta1) Reference Manual.* November 1997. http://theory.lcs.mit.edu/˜cilk.
4. Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, chapter 6. The MIT Press, 1996.
5. Masahiro Yasugi and Kazuo Taki. OPA: An Object-oriented Language for Parallel Processing — its Design and Implementation —. *IPSJ SIG Notes 96-PRO-8*, 96(82):157–162, August 1996. (in Japanese).
6. M. Yasugi, S. Eguchi, and K. Taki. Eliminating Bottlenecks on Parallel Systems using Adaptive Objects. In *Proc. of Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 80–87, October 1998.
7. W. Clinger and J. Rees, editors. *Revised^4 Report on the Algorithmic Language Scheme.* MIT AI Memo 848b. MIT, 1991.
8. Mark Feeley. Polling Efficiently on Stock Hardware. In *Proc. of FPCA'93*, pages 179–190, June 1993.
9. Y . Ichisugi and A . Yonezawa. Exception Handling and Real Time Features in an Object-Oriented Concurrent Language. In *Concurrency: Theory, Languages and Architecture*, volume 491 of *LNCS*, pages 92–109. Springer, 1990.
10. Takashi Chikayama, Hiroyuki Sato, and Toshihiko Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. of FGCS'88*, pages 230–251, 1988.
11. Richard P. Gabriel and John McCarthy. Queue-based multi-processing Lisp. Technical Report STAN-CS-84-1007, Department of Computer Science, Stanford University, 1984.
12. Ron Goldman and Richard P. Gabriel. Qlisp: Parallel processing in Lisp. *IEEE Software*, pages 51–59, July 1989.
13. Morry Katz and Daniel Weise. Continuing into the future: On the interaction of futures and first-class continuations. In *ACM Conference on Lisp and Functional Programming*, pages 176–184, June 1990.
14. Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. In *Conf. on the Principles and Practice of Parallel Programming (PPoPP)*, pages 128–136, March 1990.
15. Robert Hieb, R. Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110, 1994.