Noname manuscript No.
(will be inserted by the editor)

Supporting Enhanced Exception Handling with OpenMP in Object-Oriented Languages

Xing Fan, Mostafa Mehrabi, Oliver Sinnen, and Nasser Giacaman

the date of receipt and acceptance should be inserted later

Abstract The proliferation of parallel processing in shared-memory applications has encouraged developing assistant frameworks such as OpenMP. OpenMP has become increasingly prevalent due to the simplicity it offers to elegantly and incrementally introduce parallelism. However, it still lacks some high-level language features that are essential in object-oriented programming. One such mechanism is that of exception handling. In languages such as Java, the concept of exception handling has been an integral aspect to the language since the first release. For OpenMP to be truly embraced within this object-oriented community, essential object-oriented concepts such as exception handling need to be given some attention. The official OpenMP standard has little specification on error recovery, as the challenges of supporting exception-based error recovery in OpenMP extends to both the semantic specifications and related runtime support. This paper proposes a systematic mechanism for exception handling with the co-use of OpenMP directives, which is based on a Java implementation of OpenMP. The concept of exception handling with OpenMP directives has been formalized and categorized. Hand in hand with this exception handling proposal, a flexible approach to thread cancellation is also proposed (as an extension on OpenMP directives) that supports this exception handling within parallel execution. The runtime support and its implementation are discussed. The evaluation shows that while there is no prominent overhead introduced, the new approach provides a more elegant coding style which increases the parallel development efficiency and software robustness.

Keywords: OpenMP, parallel programming, exception handling, error recovery, software robustness

Department of Electrical and Computer Engineering
The University of Auckland

1 Introduction

Even though the evolution of OpenMP has made it increasingly comprehensive for shared-memory applications, the framework still has some way to go before it is widely used for general software development. In particular, the current OpenMP standard lacks support for essential programming features such as mechanisms for error recovery. As a matter of fact, OpenMP is mainly used for compute-intensive applications that are deterministic and less error-prone, such as batch-like, or numerical and scientific computations. For other kinds of parallel programs (such as server-side applications [10], games [7], desktop and mobile platform software [13]), which are typically interaction-based, handling unexpected situations is essential for robustness.

Exception handling is an error recovery mechanism which enables programs to anticipate and recover from abnormal situations and consequently avoid any abrupt termination of applications. Compared with other error handling approaches (e.g. error code based, callback function based [4]), exception-based recovery is more compliant with object-oriented principles, due to its support for user-defined exceptions. In object-oriented languages, useful information about an error is typically stored in an instance of an `Exception` class. Moreover, it is lexically clearer and more flexible to directly surround code that could potentially throw exceptions in *try-catch-finally* blocks. OpenMP does not provide rich support for object-oriented exception handling in parallel environments. If anything, considering that a parallelized application is likely to introduce more potential problems than that in a sequential application, this lack of support for exception handling makes it especially difficult to write robust object-oriented parallel code using the OpenMP approach. This is especially important to recognize in an object-oriented language such as Java, where exception handling is an integral part of the language. As Android and multi-core mobile devices continue their dominance, the relevance of parallel programming is evermore relevant and presents another opportunity for OpenMP to embrace this community of developers.

A direct combination of the conventional object-oriented exception handling model with OpenMP is not feasible. This is due to the conventional exception handling mechanism is only compatible with single-stream control flow. The *catch* block works as a backup execution stream and does not execute unless the specified exception occurs inside its paired *try* block. Moreover, it is guaranteed that at most one exception may occur within the contained *try* block, since the control flow is executed sequentially. Therefore, either an exception is handled adequately and the program continues running, or the exception is propagated upwards (and may potentially terminate the program if not handled at a higher level). However, this procedure becomes more complicated when developing code in OpenMP, because OpenMP directives change the context of the (otherwise lexically sequential) source code. That is, certain regions of the code might be executed in parallel (i.e. the parallel regions), and multiple exceptions may in those parallel regions may need to be handled differently. In other words, exception handling with multiple control

flows cannot simply adopt the conventional sequential try-catch-finally policy. Since the program deals with multiple threads, there are several factors that need to be considered, such as:

- Differentiating between handling single-thread exceptions and thread-group exceptions;
- Whether to stop the entire execution if one of the threads encounters a non-handled exception;
- Ensuring that stopping a thread does not interfere with the execution of other threads.

In this paper, an in-depth examination for exception handling in an OpenMP environment has been proposed. The contributions of this paper can be divided into three parts. First, the categorization and formalization of object-oriented exception handling in OpenMP parallel regions. Second, the concept of flexible thread cancellation is proposed, which provides a better approach for managing the control flow of a program, as well as facilitating exception handling on threads. Finally, the usability and performance are evaluated through an OpenMP implementation for Java [12].

This paper is organized as follows. Section 2 reviews latest researches related to OpenMP error recovery model, and we find that there is no dedicated error recovery model for object-oriented programming. Section 3 gives an overview of the current situation using error recovery in object-oriented languages, and lists the problems programmers are facing. In Section 4, we discuss an extended parallel cancellation solution, which will give programmers more flexibility to express parallel control flow and it is also useful with regard to exception handling. In Section 5 exception handling with OpenMP is comprehensively discussed. Some implementation issues are discussed in Section 6 and we give the evaluation of our new runtime support in Section 7.

2 Related work

Although the official OpenMP standard does not have a comprehensive error handling mechanism at the moment, several error handling models have been proposed for OpenMP. Gatlin [4] initially classifies error handling into three categories based on exception, callback function and error-code. Exception-based error handling is widely used in object-oriented languages such as C++ and Java, but combining this mechanism with parallelization approaches in OpenMP has not been studied in depth so far. On the contrary, error recovery models that are based on callback functions are widely used in different domains, but they seem to be too complicated to use. Low level languages such as C and Fortran mainly use this approach to handle errors, as these languages lack proper exception handling mechanisms. For this category, Duran et al. [3] introduces a model for error recovery in OpenMP that is based on callback functions. The model proposes a mechanism for registering callback functions using the `onerror` clause to specify a function that is called in

case of a specific error. Moreover, Wong et al. [14] discussed the necessity of error-handling models in OpenMP. However they argue that the model must support exception-unaware languages (e.g. C and Fortran), thus their model does not include the semantics of exception throwing and *try-catch* blocks.

3 Problem Overview

In this section, we itemize the obstacles towards efficient and robust exception handling programming with the help of some code snippet examples.

3.1 Current situation

Although it may be possible to handle exceptions thrown within OpenMP parallel regions, it is rather counter-intuitive, demanding and confusing to correctly implement since the semantics are evaded in the OpenMP standard. According to the specifications of OpenMP 4.0 [1], when an exception is thrown inside a parallel region, the only restriction is that the exception should be caught and handled within the same region and by the same thread. Therefore, a parallel region surrounded by a *try-catch* block does not comply with OpenMP specifications (see Figure 1a). Moreover, we also cannot guarantee that a *try-catch* block within a parallel region will function as it is expected, due to some semantic defects within OpenMP specifications [8]. For example, Figure 1b shows a *try-catch* block embedded inside an OpenMP parallel region. Although this syntax may get through an OpenMP compiler, it has a potential runtime bug. In this particular case, when an exception occurs before the barrier, the control flow of the encountering thread will jump to the `catch` block. This jump will skip the `barrier` directive, while the other threads that do not encounter an exception end up halting indefinitely at the barrier synchronization. This is similar to the reason why OpenMP standard strictly follows the Single Entry, Single Exit (SESE) principle as [8] indicated. Although there are already some static analysis techniques proposed such as [9] which is designed for checking the validation of barriers, it still lacks the consideration onto exception handling semantics.

3.2 Problem definition

The current situation of using *try-catch* blocks suggests that programmers encounter difficulties due to programming inconveniences and pitfalls of OpenMP error handling. Lacking a standard and consistent error handling mechanism in OpenMP makes programmers struggle in writing robust and efficient OpenMP code. The major consequence of the lack of exception handling mechanisms in OpenMP hinders the widespread use of OpenMP in object-oriented languages,

```
try{
  #pragma omp parallel for
  for (int i=0; i<4; i++)
  {
    cause_exception();
  }
}catch(Exception e){
  //handling exception
}
```

(a)

```
#pragma omp parallel{
  try{
    phase1_cause_exception();
    #pragma omp barrier
    phase2();
  }catch(Exception e){
    //handling exception
  }
}
```

(b)

Fig. 1: (a) Try-catch *m*echanism that does not syntactically and semantically conform with the OpenMP specification; (b) Syntactically conforms with OpenMP specification, but semantically it has a defect.

since there is no clear OpenMP conformity with contemporary software design paradigms. Generally, error handling in OpenMP needs to be improved in three major aspects:

- The semantics for checking whether catching an exception can cause other problems.
- Convenient and flexible mechanisms for controlling or canceling execution within parallel environments.
- A reliable runtime support for the default behavior of parallel executions when they encounter uncaught exceptions.

4 Cancellations

Before discussing exception handling within OpenMP parallel regions, it is helpful to discuss the significance of cancellation in a parallel context. In sequential programming, canceling execution at a certain part in the code is easily achieved by using the supported programming language keywords (e.g., `break` to cancel a loop, or `return` to cancel execution within a method). Because there is only one control flow, cancellation in sequential code simply means canceling the current scope of execution. In an OpenMP parallel region, such a cancellation keyword is lexically in a sequential program but semantically executing in parallel (the OpenMP philosophy that the original sequential code is intact when the OpenMP compiler directives are ignored). In this parallel context, does a cancellation indicate the termination for a single thread in the parallel region (i.e. the one encountering the cancellation), or it would it indicate termination of all threads participating within the current parallel environment? Therefore, when converting sequential code to parallel code, extra directives are needed to convey the programmer's intentions. This type of directive should be flexible and easy enough to express programming

logic, while still respecting the OpenMP approach of maintaining lexically sequential code.

OpenMP 4.0 standard has added some directives related to region cancellation [1]. According to the `cancel` directive, programmers are allowed to cancel the innermost parallel/for/sections/taskgroup region where the `cancel` directive appears. This specification provides an approach to stop execution of a parallel region, with the combination of `cancellation point` directive, which allows for user-defined cancellation points. The net effect of this directive is that it results in stopping the entire parallel execution. The `cancel` directive lacks the ability to stop a single thread locally without interfering with the execution of other threads. This would be useful when a thread encounters an exception and cannot recover from it, so it may be desirable to only stop execution of that current thread (since it no longer needs to continue its assigned workload), without canceling the entire parallel execution. Using a break-statement goes against OpenMP standards (since it is oblivious to OpenMP barriers). The status quo makes it difficult for programmers to specify the control flow of a parallel execution, and confines the use of exception handling when exceptions happen in a parallel execution.

Cancellation directive In order to support a more flexible thread canceling mechanism, and to better support the OpenMP exception handling model, the official `cancel` directive is extended. This extension is achieved by adding a *thread-affiliate-clause*, which can be `global`, indicating the cancellation of the entire thread group (the current OpenMP definition), or `local`, merely indicating the cancellation for the current thread encountering the directive. The optional `if` clause, signaling that the cancellation is active only when the condition inside the `if` statement holds true, remains unchanged. The optional *throw-clause*, indicating an extra exception throwing when the cancellation is applied. Figure 2 demonstrates the extended syntax of the `cancel` directive. We however propose an additional optional clause, `neglect_exception`, for constructs `parallel`, `for`, `sections` and `taskgroup`. This is under the consideration for the simplification of parallel exception handling, which will be explained in Section 5.3.3.

#**pragma omp cancel** *construct-type-clause thread-affiliate-clause [if-clause] [throw-clause]*

where *construct-type-clause* is one of the following:

parallel, sections, for, taskgroup

and *thread-affiliate-clause* is one of the following:

global, local

and *if-clause* is:

if(*scalar-expression***)**

Fig. 2: Extended cancellation directive.

The extended `cancel` directive expands the control over a group of threads. That is, by combining different clauses, programmers can express customized

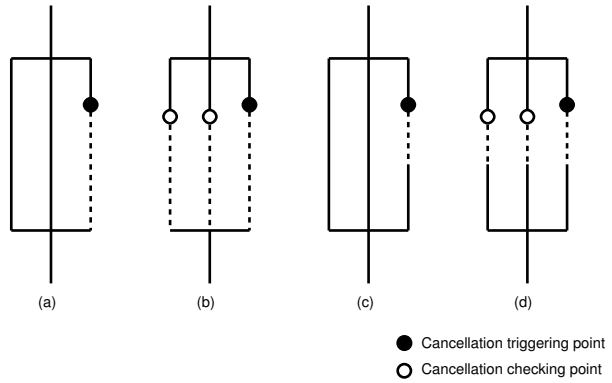

Fig. 3: **Different uses of `cancel` directive.** (a) `cancel parallel local` Only single thread quits the innermost parallel region; (b) `cancel parallel global` Entire thread group quits current parallel region; (c) `cancel for local` Single thread quits current worksharing for-loop, but continues when other threads finish this for-loop iteration; (d) `cancel for global` All threads quit current worksharing for-loop and continue with following statement.

behaviors of the parallel control flow. Figure 3 visualizes the cancel directive with the combinations of different clauses. Black nodes indicate the cancellation triggering points. A thread with black node is the cancellation triggering thread. If a thread encounters a cancellation directive with the local property, it will only stop executing the innermost OpenMP construct thread-locally. Afterwards, the thread resumes when all other threads within the parallel execution reach the next statement following the canceled region. On the other hand, if the cancellation is a global cancellation, the triggering thread will set a global cancellation flag. Other threads check this cancellation flag at next cancellation checking points (indicated by white nodes). Afterwards, all threads resume from the next statement after the cancellation region.

The `cancel` directive can be used for two purposes. First, programmers can explicitly use this directive to express parallel control flow. Second, it works as an implicit operation when an exception happens within a parallel execution. The latter is explained in more detail in Section 5.3.3.

5 Exception Handling

In this section we demonstrate the exception handling model. In order to ensure the robustness and flexibility, several limitations and extensions are discussed. The discussion is categorized into two parts: Local exception handling and global exception handling.

5.1 Overview of Categorization

In proposing a comprehensive model for parallel exception handling, we discuss different categories of exception handling in order to set up a standard for using exception handling with OpenMP directives to prevent unexpected execution behaviors. There are two kinds of exception handling scenarios that would be useful in an OpenMP environment. One involves handling exceptions within a single thread, while the other involves exception handling across a group of threads:

Local exception handling: This means an exception is handled by the same thread that threw the exception in the parallel region. A successful local handling must ensure that the procedure of error recovery does not influence with the execution of other threads. A local exception handling *try-catch* block does not surround the entire parallel region, but is rather handled internally within the parallel region.

Global exception handling: A global exception means an exception potentially influences the entire parallel region. If an exception in a parallel region is not caught by its throwing thread, or handling this exception causes another exception to be thrown, then the exception will affect the entire parallel execution. The OpenMP standard does not categorize this behavior, since it insists it should never occur. Lexically, the *try-catch* block for handling of these types of exceptions would surround the parallel region in which the exception might happen. An uncaught global exception will make the entire parallel execution stop. If this exception is still not caught afterwards, the entire program will stop as well.

5.2 Local Exception Handling

Local exception handling ensures that errors are recovered inside their local threads, and the local threads continue working/progressing. In order to avoid an unexpected execution behavior (examples in Section 3.1), this type of handling requires two conditions to be met: (a) Any potential exception inside a *try-catch* block does not interfere with other thread's execution; (b) Any operation inside a *catch/finally* block does not affect the entire parallel region's progress.

Technically, as a legal local exception handling, the entire exception handling region requires there is no *OpenMP synchronization point* present, in either of the *try-catch* or *finally* blocks. Furthermore, it should be ensured that (a) there is no exception re-throwing or (b) if exception re-throwing happens, the re-thrown exceptions need to be handled by another legal local handling.

With regards to parallel synchronization points in the parallel region, usually represented by various OpenMP directives, this can be categorized into two groups:

Control-flow synchronization point: A control-flow synchronization point is defined as a point where a thread cannot evolve until it is synchronized

control-flow synchronization point	context-property switching boundary
`omp barrier`	`omp parallel`
`omp for` (implicit barrier)	`omp section`
`omp sections` (implicit barrier)	`omp single`
`omp single` (implicit barrier)	`omp master`
	`omp critical`

Table 1: Two types of synchronization point in OpenMP.

with other threads in the corresponding parallel region. A typical control-flow synchronization point is the barrier directive. Other directives, may contain an implicit barrier if the `nowait` clause is not specified. Those directives include `for`, `sections` and `single`. If there is a control-flow synchronization point inside the *try* block, there is a risk of not being reached by one of the threads when this thread encounters an exception.

Thread-context switching boundary: The attribute of source code changes when encountering a thread-context switching boundary. In an OpenMP parallel region, there are mainly three types of source code regions: (a) Code regions to be executed by every thread at the same time; (b) Code regions to be executed only by one specified thread (e.g. `master`) or non-specified thread (e.g. `single`); (c) Code regions to be executed by every thread, but the executions need serialization (e.g. `critical`). Thread-context switching boundary works as a dividing boundary to change this thread-context property. Notice sometimes a control-flow synchronization point is also a thread-context switching boundary, such as `for`. If a *try* block contains several OpenMP code blocks which represent different thread-contexts, it is easy to cause an ambiguous exception handling semantic and unexpected runtime behavior. So avoiding thread-context switching boundaries inside a local exception handling *try-catch* block is a better programming practice.

Example In order to illustrate the concept of parallel synchronization point, Figure 4 shows a visualization of all synchronization points onto a piece of dummy OpenMP source code. All parallel synchronization points slice the code snippet into several pieces, indicated by the lines. A legal local handling, a.k.a *try-catch(finally)* block, should not traverse any line as indicated. This restriction confines programmers using *try-catch* blocks inside certain code span, to ensure the exception handling does not interfere any other thread's execution.

According to this limitation, a robust compiler should be able to throw a warning to inform programmers if the OpenMP source code does not conform with local exception handling rules. This warning reminds programmers to double check the code whether the exception handling could cause any side effect.

Implementing this feature can be safely achieved by adding an extra semantic checking pass at the front-end parsing stage of compilers. The general idea is, when encountering a *try* block, parser becomes sensitive to checking

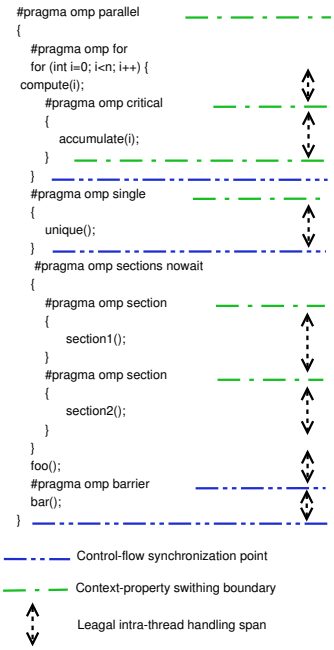

Fig. 4: A legal local exception handling *try* block cannot traverse any line as indicated.

certain types of OpenMP directives, when encountering a directive or an ending boarder of a directive which has the property as like a synchronization point, compiler throws a warning from it.

5.3 Global Exception Handling

Global exception means an exception is emitted from a parallel region and it is not handled thread-locally. It indicates an unexpected behavior occurred and escaped from within the parallel execution. If this exception is not handled by its local thread, this exception will be forwarded to the parallel region. Because a thread-locally-uncaught exception could influence the correctness of parallel execution, this exception changes its property and becomes a global exception and handling this type of exception is defined as global exception handling.

5.3.1 Global Exception Catch Procedure

In a sequential program, if an exception happens, it needs to be handled by the encountering thread. If the thread cannot find a matching catch block, the program will stop with throwing an unhandled exception. However, in parallel execution, if an exception happens in a thread, it is not always necessary to

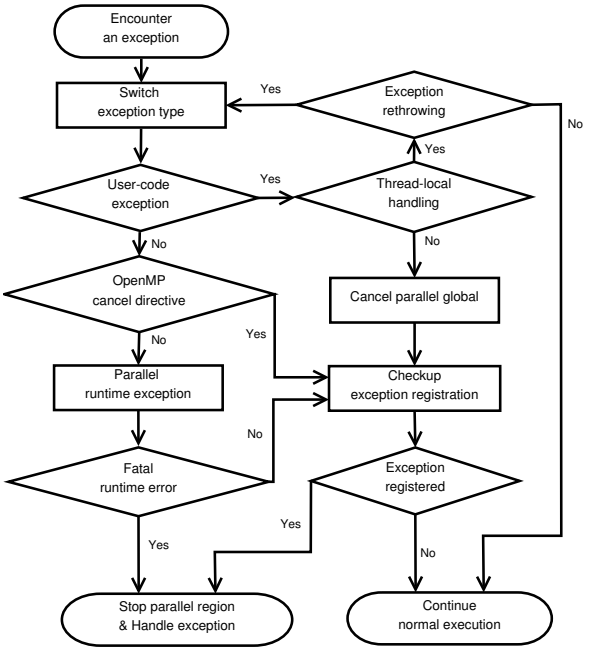

Fig. 5: Flowchart of exception handling within OpenMP parallel region.

stop the parallel execution. Programmers can specify the behavior when an exception happens. That is, to handle it by the encountering thread, to expose it to the parallel environment, or to stop the encountering thread only.

Figure 5 shows the flowchart for the case of an exception within an OpenMP parallel execution. When a parallel program is executing, if it encounters an exception, it first checks whether a local exception handler is defined. If yes, this exception will be handled using the thread-local approach, and then the encountering thread continues processing. Notice that it is possible to throw another exception from the handling code (i.e. *catch* or *finally* block), in which case the program continues looking for another local handler until the exception cannot be handled locally. If a thread encounters an exception and this exception is not handled locally, the default behavior will be `cancel parallel global`, which triggers the cancellation of that parallel region. In another situation a program may encounter a `cancel` directive. As discussed in Section 4, `cancel` directives can also be used for deliberate control-flow stops. Therefore, execution stops due to OpenMP cancellation are not always regarded as exceptions.

5.3.2 Exception Registration

In some cases, programmers may need to throw an exception when encountering a cancellation directive. Since cancellation is always the last reachable

```
Array<Object> arr = ...;
Value target = ...;
Object tarObj = null;
try {
  #pragma omp parallel for shared(arr)
  for (int i=0; i<arr.length; i++) {
    if (arr[i] == null) {
      #pragma omp cancel parallel global throw(NullElementException)
    }
    val = process(arr[i]);
    if (val == target) {
      #pragma atomic write
      tarObj = arr[i];
      #pragma omp cancel parallel global
    }
  } //end of parallel region
} catch (NullElementException *e) {
  //handle exception
}
```

Fig. 6: An example of using `throw` clause.

statement in a code scope, directly appending a throw statement does not work. In order to solve this problem, we enable cancellation directives to register exceptions. This mechanism allows programmers to define which kind of cancellation inside a parallel region requires error recovery. Furthermore, during runtime, parallel execution only regards the cancellation directives with exception registration as unexpected exit. In order to achieve this behavior, a new `throw` clause is added to the `omp cancel` directive, which indicates this cancellation is followed by an exception throwing from the innermost parallel region. The `throw` clause provides more flexibility for expressing parallel regions, and makes code more readable with cancellation directives that throw exceptions explicitly. The latter is important for the maintenance of parallel source code, from software engineering point of view.

An example of exception registration is showed in Figure 6. Inside the parallel region, there are two `omp cancel parallel global` directives. In order to distinguish the differences, the first cancellation directive is appended with an exception throwing, which means regarding the first parallel cancellation as an exception. Then if the first cancellation happens, an exception is thrown out from parallel region and it can be caught by the *catch* block outside parallel region. On the contrary, if the second cancellation directive is activated, though the parallel region will be canceled, there is no exception will be propagated.

5.3.3 Exception Neglecting

In some cases, it is not desirable to stop the entire parallel processing once an exception is exposed to the parallel environment. Programmers may want the remaining threads keep executing even if one or more threads fail within the

```
#pragma omp parallel
for (;;) {
  #pragma omp single
  requests = collect_requests();
  #pragma omp for neglect_exception(Exception)
  for(int i=0; i<requests.size(); i++) {
    try {
      data = process_request(requests[i]);
    } catch(DataNonconformityException *e) {
      data = response.NONCONF;
    }
    response[i] = data;
  }
  #pragma omp single
  send_responses(responses);
}
```

Fig. 7: A demo code using `neglect_exception` clause to simplify the recovery procedure when uncaught exception happens inside a parallel region.

thread group. This can be achieved by explicitly declaring a local cancellation at the end of local exception handling code to make the encountering thread stop locally. However, If there is no other recovery operations within the handling code, the semantic can be simplified by using `neglect_exception` clause after the corresponding OpenMP construct. When an exception happens and it is registered by the exception neglecting mechanism, it will not trigger the parallel or worksharing execution cancellation. Instead, only the encountering thread will stop. In the meanwhile, since it is possible that some works distributed by the stopping thread is not finished, for the compensation, a dynamic work redistribution is run when the thread stopping happens.

Using exception neglecting mechanism also enables programmers to easily sustain the continuation of parallel processing when certain thread inside the thread group fails. Because remedy operations are automatically done by the underlying runtime support, programmers liberate from arduous works of converting sequential code to robust parallel code.

Figure 7 shows a code example which uses the `neglect_exception` clause to simplify the programming logic. Inside the parallel region, an infinite loop is executed. For each loop iteration, firstly one of the thread inside thread group collects requests from the network sockets. Afterwards, a series of requested are processed by the thread group. During the worksharing process, exceptions could happen. But only some types of the exception are handled thread-locally. Other unexpected exceptions (e.g. `OutOfMemoryException`) could still escape from the local thread. Under this circumstance, using `neglect_exception` clause followed by a more general exception type (`Exception`) enables the parallel execution ignore the exceptions which are exposed to the parallel en-

```
ExceptionA* ea = null;
ExceptionB* eb = null;
#pragma omp parallel shared(ea, eb)
{
  #pragma omp for
  for (int i=0; i<N; i++) {
    try {
      may_causes_ExceptionA();
    } catch (ExceptionA *e) {
      #pragma omp critical {
      ea = e;
      #pragma omp cancel parallel
    }
  }
  try {
    may_causes_ExceptionB();
  } catch (ExceptionB *e) {
    #pragma omp atomic write
    eb = e;
    #pragma omp cancel parallel
  }
  foo();
} // end of parallel region
if (ea) {
  //handle exceptionA if happens
}
if (eb) {
  //handle exceptionB if happens
}
```

Fig. 8: Conventional approach of handling global exception.

vironment. After an automatic work redistribution (if necessary), the parallel execution keeps processing.

5.3.4 Source Code Simplification

Due to the lack of specifications for parallel exception handling, a conventional traverse-parallel-region exception handling solution would have to use predefined references (or pointers) to store the exceptions that happen in a parallel region. That is, programmers have to manually store exceptions that could possibly occur in a parallel region, and then invoke a global cancellation directive to stop the parallel execution when handling that exception. Thus, a parallel region must be followed by a series of inspections to test whether any of the specified exceptions have happened. The source code (Figure 8) for such a manual approach is quickly tainted with multiple *try-catch* blocks, especially when programmers want to catch several potential exceptions in a parallel region.

The source code can be easily simplified using new proposed exception handling semantics with OpenMP directives. *Try-catch* block can directly sur-

```
try {
    #pragma omp parallel
    {
        #pragma omp for
        for (int i=0; i<N; i++) {
            may_causes_ExceptionA();
        }
        may_causes_ExceptionB();
        foo();
    } // end of parallel region
} catch (ExceptionA *e) {
    //handle exceptionA if happens
} catch (ExceptionB *e) {
    //handle exceptionB if happens
}
```

Fig. 9: Pyjama's new approach of handling global exception.

round a parallel region without code re-factoring (See Figure 9). The compiler source-to-source generation and runtime support will do all the routines in the background. This improvement makes the source code more elegant and more compliant with object-oriented design patterns.

6 Implementation

This section discusses about the implementation of enhanced exception handling support. The aforementioned concepts and proposals are implemented through a source-to-source compiler and its runtime support. This section mainly explains some noticeable issues with regard to the runtime implementation.

6.1 Adaptable synchronization barrier

The extended OpenMP cancellation directive allows the cancellation of single thread without stopping the entire parallel execution. Since a stopped thread could influence the following synchronization procedure of other remaining threads, it requires an on-the-fly thread consensus number adjustment when a local thread cancellation happens.

The requirement is achieved by implementing an adaptable synchronization barrier. Different from traditional cyclic barrier, adaptable barrier has the extra interfaces `decreaseConsensus()` and `increaseConsensus()` which enables the barrier to readjust consensus number when a thread quits or joins the thread group. The detailed implementation of the adaptable barrier is listed in Algorithm 1. Every thread local cancellation invokes `decreaseConsensus()` before real thread stopping, and the synchronization consensus number decreases from n to $n - 1$. The same, if a canceled thread rejoin the thread

group, the interface `increaseConsensus()` is invoked and the synchronization consensus number related to this thread group increased from n to $n+1$.

Algorithm 1 Adaptable synchronization barrier.

```
 1: class AdaptableBarrier {
 2:     int consensus;
 3:     int waitingCount;
 4:     Lock lock = new Lock();
 5:     Condition round = lock.condition();
 6: }
 7: void dowait() {
 8:     lock.lock();
 9:     if (- -waitingCount) {
10:         nextRound();
11:         lock.unlock();
12:         return;
13:     }
14:     while(true) {
15:         round.await();
16:     }
17:     lock.unlock();
18: }
19: void decreaseConsensus() {
20:     lock.lock();
21:     consensus -= 1;
22:     if (- -waitingCount) {
23:         nextRound();
24:     }
25:     lock.unlock();
26: }
27: void nextRound() {
28:     round.signalAll();
29:     waitingCount = consensus;
30: }
```

6.2 Dynamic work redistribution

As mentioned before, in order to ensure all remaining worksharing chunks are processed if a thread cancels its works in an OpenMP worksharing group, work redistribution is required. We adopt the similar way as Parallel Iterator [5] does. More specifically, if a thread quits from a worksharing execution, all its remaining allocated iterations are released. If there are still other threads working, then they share these remaining iterations (after those threads complete their normal iterations) using a dynamic schedule with chunk size 1.

If multiple threads attempt to exit from a local cancellation, then all of them will succeed except the last thread. Because if there is only one thread, the worksharing construct is at risk of half finish. Under this circumstance, if the last thread cancels from the parallel execution, an extra exception is thrown out to indicate the total fail of parallel execution.

6.3 Exception from synchronization regions

There is the possibility that an exception thrown from a critical region. A legal local-exception handling could be available to catch it, as long as it does not break the rule as Section 5.2 discussed. Otherwise, this exception exposes to the parallel region. In order to avoid a deadlock, from the implementation level, it should release the lock resource when the exception escapes from the critical region. In the Java implementation, a `finally` block is sufficient to ensure this, which always makes the lock to be released when quitting the critical region. Whereas considering the implementation for C++, which does not support `finally` keyword, RAII [11] is the suitable technique to ensure the life cycle of lock resources to be confined inside a certain lexical scope.

6.4 Global exception throwing

Different from exception handling in sequential execution, in a parallel environment, two or more exceptions may happen at the same time. If those exceptions are not caught thread-locally, then multiple exceptions are exposed to the parallel region. Considering one global exceptions is thrown from one thread, but before other threads reach the nearest cancellation points, another global exception happens from another thread. If there is no consensus about which global exception should be handled, the entire parallel environment is in the risk of inconsistency and unexpected behavior may happen.

So in order to ensure the exception handling consistency, it is important to guarantee that when multiple global exceptions happen, all the exception exposures to the parallel region should be linear [6] and immediately visible to any other threads within the thread group. This is implemented by endowing each parallel region an exception slot, on which the data can be modified using the compare-and-set (CAS) operation. If more than one threads throw exceptions at the same time, the CAS operation ensures that only one exception is set to the exception slot. All other threads which failed putting the exception will put their exception to a logger `ExceptionLogger`. The thread which succeeds on CAS operation will trigger the cancellation flag and all other threads which fail to register their exceptions will only end with thread cancellations. Algorithm 2 shows how a cancellation signal is processed in the runtime.

6.5 Parallel runtime exception

Parallel runtime exception indicates the potential runtime unexpected behaviors on parallel processing. Since this kind of exception is provided by parallel runtime library, the exception throwing cannot be reflected from user source code. Global exception handling enable programmers handle this type of exception by surrounding parallel region using try block. Because all these exceptions only happens at runtime, they are all runtime exceptions (inherent class `RuntimeException` in Java). Sometimes, this type of exception may

Algorithm 2 The cancellation signal processing in runtime.

```
1: switch (cancellationSingal) {
2: case GlobalException:
3:     if (exception.isNeglected) {
4:         return;
5:     } else {
6:         if (!ExceptionSlot.compareAndSet(null, exception)) {
7:             ExceptionLogger.add(exception);
8:         }
9:     }
10: case GlobalCancellation:
11:     cancellationFlag.set(true);
12: case ThreadLocalCancellation:
13:     barrier.decreaseConsensus();
14:     cancelCurrentThread();
15:     break;
16: }
```

Exception Name	Description	Severity
OmpParallelStartFailException	Failed to start a parallel execution	Fatal
OmpNotEnoughMemoryException	Does not have enough memory when spawning threads	Severe
OmpBrokenBarrierException	OpenMP barrier broken due to hardware interruption	Severe
OmpNotEnoughThreadsException	Cannot generate enough threads as expected	Medium
OmpSynchronizationTimeoutException	OpenMP synchronization time window runs out	Medium
OmpUnsupportedFeatureException	Encounter an unsupported feature in current OpenMP version	Medium

Table 2: OpenMP runtime exceptions.

be fatal for the parallel execution (e.g. `OmpParallelStartFailException`). In some situations, the parallel runtime exception may influence the correctness of parallel execution (e.g. `OmpNotEnoughThreadsException`). Programmers can optionally handle these exceptions according to their code purposes. Table 2 lists several selected OpenMP runtime exceptions.

7 Evaluation

In this section, we evaluate the new exception handling mechanism in the new OpenMP version that we have implemented for Java. The evaluation mainly contains two parts: The first part describes programmability, which shows how programming productivity and software robustness are improved by introducing the boosted model. Second, the performance is evaluated by running a series of benchmarks, showing that there is no salient performance degradation with new runtime support, compared with original unmodified one.

7.1 Programmability

According to aforesaid concepts, the compiler does the semantic check to see whether programmers made a legal local thread exception handling in a parallel region. This could prevent the unexpected bugs such as the example shown in Figure 1b. Also, the runtime support ensures that even if an exception is not handled inside the parallel region, the execution will stop the entire parallel execution instead of causing a deadlock (example showed in Figure 1a).

Generally, since the compiler and its runtime help to do most of the correctness checking and underground operations, programmers are able to write robust parallel code with less coding (Figure 9) and effort.

7.2 Performance

The overhead has always been the main concern in concurrent and parallel execution. Even though the new design decreases the developing time, having too much overhead introduced during runtime is undesirable, and it defeats the potential benefits of the design. Therefore, we measured performance using our new design to answers two questions:

- Does the parallel *try-catch* block exert a performance penalty when no exceptions happen during runtime?
- Does the new implementation introduce prominent overhead with respect to the conventional manual exception-checking in OpenMP?

The experiments were mainly performed on three systems. The first two are portable multi-core systems, with different infrastructure specifications: The first one has an i5-3570 quad-core Intel processor, with 8M cache, up to 3.90 GHz clock rate, and it would use the HotSpot 64-Bit Server JVM. The second portable system is equipped with a 1.90GHz quad-core Intel CPU i7-3517U and was running the OpenJDK 64-Bit Server JVM. For simplification, the two portable systems are named as System A and System B respectively. The last system is a more scalable system, which is a dedicated 16-core 2.4GHz SMP machine with 64 GB memory, and Java HotSpot 64-Bit Server VM is used. This system is named as System C.

7.2.1 The overhead of try-catch guarding onto parallel region

In this section, we mainly focus on whether the exception handling support degrades the performance even though no exception happens during the parallel execution. The possible overhead can arise from two aspects: (a) either the *try-catch* guarding on the parallel region, or (b) the explicit cancellation checking points the programmer added into the parallel region.

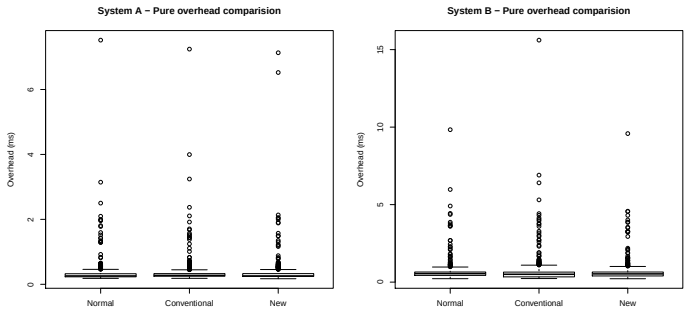

Fig. 10: pure overhead comparison in two systems.

(ms)	System A				System B			
Style	Min	Median	Mean	Max	Min	Median	Mean	Max
Normal	0.1826	0.2576	0.3228	7.5210	0.2222	0.5488	0.6034	9.8340
Conventional	0.1836	0.2737	0.3284	7.2440	0.2307	0.5068	0.6112	15.6100
New	0.1715	0.2568	0.3244	7.1320	0.2189	0.5310	0.5968	9.5830

Table 3: Key values of overhead measurement in two systems.

Experiment 1- Absolute overhead This experiment mainly evaluates the absolute overhead of launching a parallel execution with default thread number, with different exception guarding onto the parallel region. Three different coding style of Normal, Conventional, and New are studied in the experiment. The Normal style involves running a parallel region without any try-catch blocks. The Conventional style includes the traditional approach to handle exceptions, by registering them inside parallel regions and handling them outside parallel regions. The third style (New), is our proposed approach in which a try-catch block is able to directly surround a parallel region and catch any exceptions inside that region. In order to get reliable results, each style was run 100 times at one single test, and there were 10 groups of tests conducted in total. Between every two tests, there were random delay periods, in order to simulate a real-time scheduling of an operating system. Therefore there are 1000 (100x10) samples collected for each style in the each system.

The box plots in Figure 10 demonstrate the graphical dispersion of the samples. Despite the 20-30 outliers in each category, most of the runtime samples (970-980) are confined to a narrow interval, and the box regions are very similar to each other. More detailed key values are listed in Table 3. The results demonstrated by the data confirm that there is no obvious overhead deviation between three different approaches.

Experiment 2 - How try-catch guarding overhead influences scaling The second experiment was performed on a more scalable system (System C). We mainly

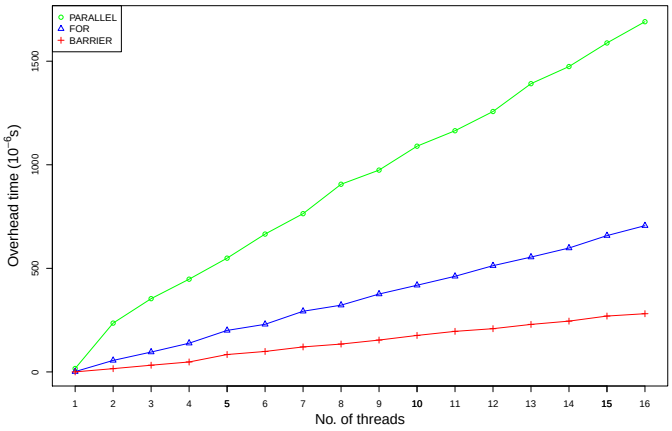

Fig. 11: Absolute time of synchronization overhead in a Java version of OpenMP (Pyjama).

use this system to evaluate how try-catch guarding influences the performance when increasing the number of parallel processing units.

Because there is no pre-existing OpenMP overhead benchmark suite for Java, according to the EPCC benchmarks [2], we develop the similar benchmarks to measure the OpenMP synchronization overheads of Java version. In the benchmarks, the parallelization overhead is defined as $T_p - T_s/p$, in which T_p is the parallel execution time on p processors and T_s indicated the sequential execution time of the same program with the same working load. In order to achieve a consistent and more accurate evaluation on the JVM, each benchmark case was run n (varies between different cases) times and before each benchmark case a $n/10$ times warmup is executed. Figure 11 illustrates the absolute time of synchronization overhead of `parallel`, `for`, and `barrier` respectively, before introducing the support of exception handling.

After the implementation of aforementioned exception handling support, two types of execution time were measured. The first is the parallel execution guarded with a *try-catch* block (TC). The second, in addition to the try-catch guarding, an extra cancellation checking point (CCP) is added. As a reference, the execution time without any exception handling is regarded as the baseline and the overhead differences are computed against it. Figure 12 depicts the absolute execution times of the EPCC-like Java benchmark cases onto `parallel`, `for` and `barrier` are measured, adopting different coding styles (no *try-catch*, using TC, and using TC&CCP respectively). It can be seen from the diagrams that the execution times do not fluctuate too much compared with the no try-catch guarded one.

In Figure 13, compared with the overhead of no try-catch guarding one, the overheads deviation of TC and TC&CCP are depicted. It can be noticed that the overall average overhead is around 0.15% and the worst case happens with

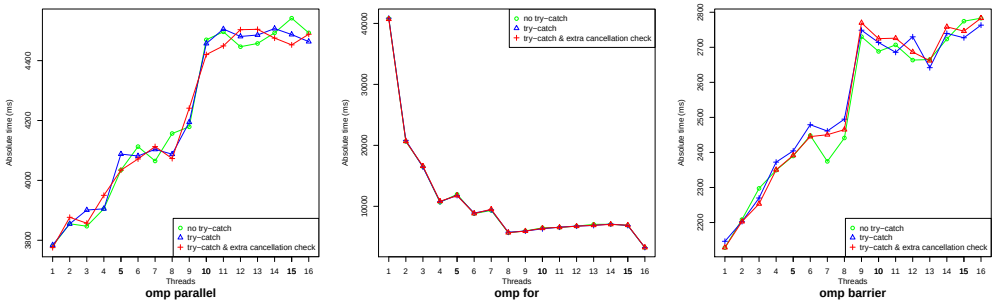

Fig. 12: Absolute execution time of EPCC.

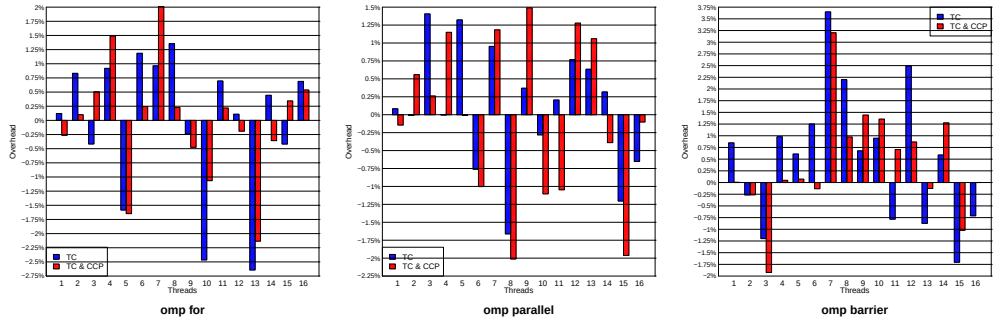

Fig. 13: Overhead of TC/TC&CCP evaluations onto different OpenMP directives.

`barrier` directive on TC and the overhead is 3.65% higher than non-exception-handling one. However, in many of the cases, the overhead is negative which means the execution time of TC or TC&CCP is faster. This phenomenon may be attributed to the operating system scheduling which has a much greater impact on the execution time, so the overhead of exception-handling support does not introduce a noticeable impact on execution time.

The two experiments confirm that the implementation of new runtime does not introduce noticeable overhead in order to support a safer and cleaner semantic.

7.2.2 Overhead of Global Exception Handling

In this section we measure the running time of a successful global exception handling. When an exception happens inside a parallel region and it is not handled locally, the thread that encounters the exception should stop, and the cancellation signal for the parallel region is triggered. Moreover, other threads stop when receiving this signal as well. Then, the control flow jumps

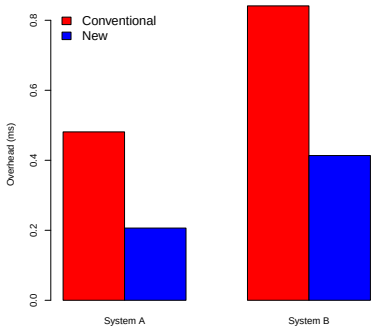

Fig. 14: Comparison of global exception handling overhead in two systems.

to the handling code (if any) which is outside the parallel region. The overhead means the time span from the throwing of the global exception to the stopping of the entire parallel execution. We measured the runtimes in this scenario for the Conventional and New approaches. Similarly, the tests were executed 100x10 times for each approach in System A and System B. The average overhead times are demonstrated in Figure 14. The differences suggest that the new proposed runtime support has better performance than the conventional manual approach.

The reason why conventional approach is slower is that in order to prevent data race, the exception throwing should be synchronized in parallel region. Conventional approach uses a critical region which is lock-guarded. Instead, the new approach does a code refactoring and the throwing of the exception is a CAS operation. Therefore, obtaining and releasing lock spend much more time than non-lock-guarded operations. The contention could be very high when many threads throw exceptions to the parallel region at the same time. In contrast, the new implementation uses source to source code conversion and all exceptions are assigned atomically, which is faster than lock guarded operations.

Figure 15 depicts the situation with System C. When a global exception happens inside a normal parallel region (`parallel` construct), the new approach of quitting of the parallel execution is always better than conventional approach. However, when it comes to a worksharing region (`parallel for` construct), it can be noticed that the new approach is slower than the conventional approach for most of cases. The explanation is that the CAS operation is quicker that lock-based operation, therefore other non-exception-encountering threads have little chance to get parallel cancellation signal at the nearest cancellation checking point and all these thread still processing until next cancellation checking point. Since the parallel execution cannot stop unless every thread stops, this situation slows down the entire stop of the parallel execu-

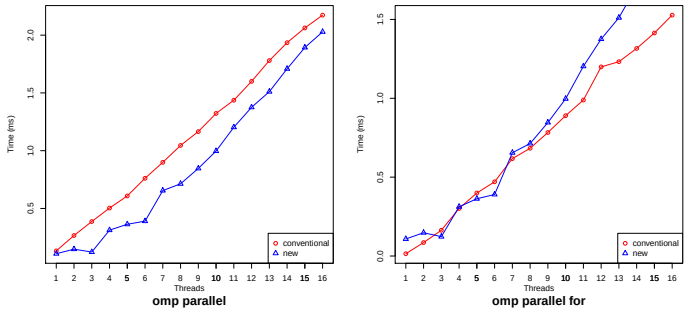

Fig. 15: Absolute quitting time when an exception happens in parallel execution.

tion. This reveals the fact that the position of cancellation checking points could influence the stopping time of a parallel region.

8 Conclusion

The ability to use exception handling mechanisms in OpenMP would be a powerful feature from a software engineering point of view. The OpenMP specification lacks the integration of exception handling in object-oriented languages. In this paper, a combination of exception handling and parallel programming (based on OpenMP directives) is discussed. A proposal on the semantics, and the runtime to support this semantics, is discussed. Programmers will gain a better programming experience when writing robust high-level parallel code with OpenMP. Evaluations suggest that the the new approach provides an elegant exception handling mechanism in OpenMP, without causing any performance degradation.

References

1. OpenMP Architecture Review Board. OpenMP application program interface 4.0, July 2013.
2. Mark Bull. Measuring synchronisation and scheduling overheads in OpenMP. In *Proceedings of First European Workshop on OpenMP*, volume 8, page 49, 1999.
3. Alejandro Duran, Roger Ferrer, JuanJosé Costa, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. A proposal for error handling in OpenMP. *International Journal of Parallel Programming*, 35(4):393–416, 2007.
4. Kang Su Gatlin. OpenMP 3.0 feature: Error detection capability. Available at: `http://www.nic.uoregon.edu/iwomp2005/Talks/gatlin-panel.pdf`, May 2005.
5. Nasser Giacaman, Oliver Sinnen, and Lama Akeila. Object-oriented parallelisation: Improved and extended parallel iterator. In *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on*, pages 113–120. IEEE, 2008.
6. Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

7. Bjoern Knafla and Claudia Leopold. Parallelizing a real-time steering simulation for computer games with OpenMP. *Parallel Computing: Architectures, Algorithms, and Applications*, 2008.
8. Jan Felix Münchhalfen, Tobias Hilbrich, Joachim Protze, Christian Terboven, and Matthias S Müller. Classification of common errors in OpenMP applications. In *Using and Improving OpenMP for Devices, Tasks, and More*, pages 58–72. Springer, 2014.
9. Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. Static validation of barriers and worksharing constructs in OpenMP applications. In *Using and Improving OpenMP for Devices, Tasks, and More*, pages 73–86. Springer, 2014.
10. Sébastien Salva, Clément Delamare, and Cédric Bastoul. Web service call parallelization using OpenMP. In *A Practical Programming Model for the Multi-Core Era*, volume 4935 of *Lecture Notes in Computer Science*, pages 185–194. Springer Berlin Heidelberg, 2008.
11. Bjarne Stroustrup. *The design and evolution of C++*. Pearson Education India, 1994.
12. Vikas, Nasser Giacaman, and Oliver Sinnen. Pyjama: OpenMP-like implementation for java, with gui extensions. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 43–52, New York, NY, USA, 2013. ACM.
13. Vikas, Travis Scott, Nasser Giacaman, and Oliver Sinnen. Using OpenMP under Android. In *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 15–29. Springer Berlin Heidelberg, 2013.
14. Michael Wong, Michael Klemm, Alejandro Duran, Tim Mattson, Grant Haab, BronisR. de Supinski, and Andrey Churbanov. Towards an error model for OpenMP. In *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, volume 6132 of *Lecture Notes in Computer Science*, pages 70–82. Springer Berlin Heidelberg, 2010.