Exception Handling with OpenMP in Object-Oriented Languages

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

Department of Electrical and Computer Engineering
The University of Auckland, New Zealand

Abstract. 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.

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 [6], games [5], desktop and mobile platform software [8]), 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.

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 [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. [9] 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, even if it complies with OpenMP specifications. For example, Figure 1b shows a *try-catch* block embedded inside an OpenMP parallel region. Although this syntactically conforms to the OpenMP standard (and therefore deceivingly appears correct), it has a potential 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 `omp 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 main reason why OpenMP insists exceptions are managed and handled within the same thread. This is because a thread encountering an uncaught exception results in that thread escaping the parallel region (and out of OpenMP's control). Thus, the escaped thread would not have the opportunity to synchronize with other threads in the team, and the entire parallel execution could malfunction.

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, 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: (a) The semantics for checking whether catching an exception can cause other problems. (b) Convenient and flexible mechanisms for controlling or

```
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* mechanism that does not syntactically and semantically conform with the OpenMP specification; (b) Syntactically conforms with OpenMP specification, but semantically incorrect.

canceling execution within parallel environments. (c) A reliable runtime support for the default behavior of parallel execution 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 added some directives related to regional cancellation [1]. According to this omp cancel directive, programmers are allowed to cancel the innermost parallel/for/sections/taskgroup region where the omp cancel directive appears. This specification provides an approach to stop execution of a parallel region, with the combination of omp 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 `omp cancel` directive lacks the ability to stop a single thread locally without interfering 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 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 `omp 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. We however propose an additional optional clause, *`throw`*`s`, which works in combination with the exception throwing considerations explained in Section 5.3. Figure 2 demonstrates the extended syntax of the `omp cancel` directive.

#**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***)**
and *throw-clause* is:
throws(*exception-identifier***)**

Fig. 2: Extended cancellation directive

The extended `omp cancel` directive expands the control over a group of threads. That is, by combining different clauses, programmers can express customized behaviors of the parallel control flow. Figure 3 visualizes the omp 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 cancel directive is a local cancellation, it will only cancel a certain scope of the code and the thread resumes when all threads of a parallel execution reach the next statement following the cancellation 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

nearest cancellation checking points (indicated by white nodes). Afterwards, all threads resume from the next statement after the cancellation region.

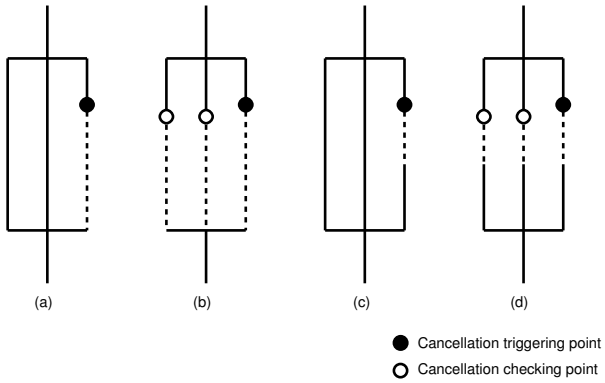


Fig. 3: **Different uses of `omp cancel` directive.** (a) `omp cancel parallel local` Only single thread quits the innermost parallel region; (b) `omp cancel parallel global` Entire thread group quits current parallel region; (c) `omp cancel for local` Single thread quits current worksharing for-loop,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

The omp cancel directive can be used for two purposes. First, this directive makes an OpenMP parallel source code more expressive about its control flow. Second, it works as an instruction for exception handling and jumping within control flow. The latter is explained in more detail in Section 5.3.

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 other thread's evolution. 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 with a stack-trace dump.

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 evolution.

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 with other threads in the corresponding parallel region. A typical control-flow synchronization point is the omp barrier directive. Other directives, may contain an implicit barrier if the nowait clause is not specified. Those directives include `omp for`, `omp section` and `omp 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. `omp master`) or non-specified thread (e.g. `omp single`); (c) Code regions to be executed by every thread, but the executions need serialization (e.g. `omp critical`). thread-context switching boundary

works as a dividing line to change this thread-context property. Notice sometimes a control-flow synchronization point is also a thread-context switching boundary, such as `omp 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.

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.

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.

Global Exception Catch Procedure In a sequential program, if an exception happens, it needs to be handled by the encountering thread; otherwise the program propagates up the call stack and then the entire program stops. However, in parallel execution, if an exception happens in a thread, it is not always necessary to 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 4 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 `omp cancel parallel global`, which triggers the cancellation of that parallel region. In another situation a program may encounter an `omp cancel` directive. As discussed in Section 4, `omp cancel` directives can also be used for deliberate control-flow stops. Therefore, we may not regard all `omp cancel` directives as exceptions.

Exception registration In some cases, programmers may need to throw an exception when encountering a cancellation directive. Since cancellation is always the last reachable statement in a code scope, directly appending a throw

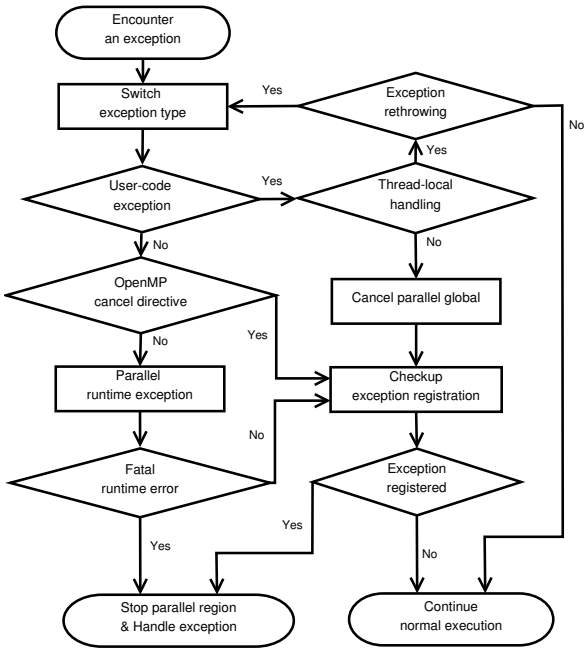


Fig. 4: Flowchart of exception handling within OpenMP parallel region

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 exits. In order to achieve this behavior, a new throws clause is added to the omp cancel directive, which indicates this cancellation is followed by an exception throwing from the innermost parallel region. The throws 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.

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 for such a manual approach is quickly tainted with multiple *try-catch* blocks, espe-

cially when programmers want to catch several potential exceptions in a parallel region.

```
ExceptionA * ea = null;
ExceptionB * eb = null;
#pragma omp parallel shared (ea, eb) {
#pragma omp for
  for(int i=0; i<N; i++) {
    try {
      may_cause_ExceptionA();
    } catch (ExceptionA *e) {
      # pragma omp critical {
        ea = e;
        # pragma omp cancel parallel
      }
    }
  }
  try {
   may_cause_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
}
```

(a)

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

(b)

Fig. 5: Source code comparison between (a) without and (b) with exception runtime support

The source code can be easily simplified using new exception handling semantics with OpenMP directives. *Try-catch* block can directly surround a parallel region without code re-factoring (See Figure 5). 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 Evaluation

In this section, we evaluate the new exception handling mechanism in the new OpenMP version that we have implemented for Java.

6.1 Usability

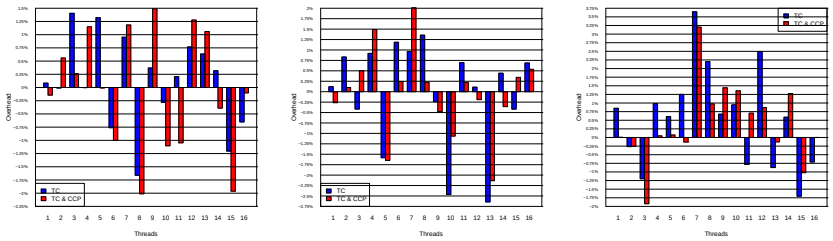
According to aforesaid concepts, the compiler does the semantic check to see whether programmers made a legal local thread exception handling in a parallel region. This could prevent the unexpected bugs such as the example shown in Figure 1b. Also, the runtime support ensures that even if an exception 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 5) and effort.

6.2 Performance

With regard to performance evaluation, 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.

The benchmark was run on a dedicated 16-core 2.4GHz SMP machine with 64 GB memory, and Java HotSpot 64-Bit Server VM is used. 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.

In the benchmark, 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 [2]. According to previous discussions, 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.



(a) omp parallel (b) omp for (c) omp barrier

Fig. 6: Overhead evaluations of different OpenMP directives

Figure 6 shows the benchmark result categorized by different types of directives. The overhead deviation of TC and TC&CCP is depicted. We notice that the overall average overhead is around 0.15% and the worst case happens with omp 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.

7 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. Bjoern Knafla and Claudia Leopold. Parallelizing a real-time steering simulation for computer games with OpenMP. *Parallel Computing: Architectures, Algorithms, and Applications*, 2008.
6. 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.
7. 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.
8. 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.
9. 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.