Structured Exception Semantics for Concurrent Loops

Joel Winstead and David Evans
{jaw2u,devans}@cs.virginia.edu

University of Virginia, Department of Computer Science

Abstract. Concurrent languages have offered parallel loop constructs for some time to allow a parallel computation to be expressed in a simple and straightforward fashion. Modern programming languages include exceptions to allow for clean handling of errors or unexpected conditions, but few concurrent languages incorporate exception handling into their models for parallel loops. As a result, programmers that use parallel loops cannot use exceptions to simplify their programs. We present a semantics for handling exceptions in parallel loops that is predictable and that reduces to the familiar semantics for sequential loops. This semantics provides guarantees about the behavior of parallel loops even in the presence of exceptions, and facilitates the implementation of parallel algorithms. A Java library implementation of this semantics is presented, along with a description of a source-to-source translation.

1 Introduction

Exceptions generated in parallel loops create problems that can be difficult to resolve. Parallel loop semantics allow more than one iteration of a loop to be executed at a time. Because any statement can generate an exception, this makes it possible for more than one unhandled exception to be raised concurrently in the same loop, a situation that does not occur in sequential loops. However, the exception semantics of most languages do not allow more than one exception to be raised at a time. It is not clear how to deal with this situation in a consistent way. It may not be consistent with the language's exception model to propagate both exceptions; if only one exception is allowed, one must be chosen and the others ignored, and there may be consequences for ignoring an exception generated by the program. Although sequential loops simply stop executing once an exception occurs, there are several ways abnormal termination could be handled in the parallel case, and one must be chosen that is reasonable and understandable.

Exception handling should be well-integrated with other parts of the language and exceptions should be handled in a consistent way regardless of the context in which they are generated. One proposed way of dealing with the problem of exceptions in parallel loops is simply to forbid them, and treat any exception that reaches the top level of a parallel loop as a fatal error. This is an unsatisfactory solution because exceptions in the context of a parallel loop are not handled in a

manner that is consistent with the way exceptions in other contexts are handled, and it does not allow them to be caught by handlers higher in the call chain than the parallel loop, even if matching handlers exist. Ideally, an uncaught exception generated in a parallel loop should propagate out of the loop like any other exception, and be handled in a way that is consistent with the way exceptions are handled in other contexts.

Even in the event of exceptions, a loop may produce partial results that are still useful, so it is important to be able to make strong assertions about the state of the program after a parallel loop has terminated exceptionally. A good semantics for exception handling in parallel loops should provide a way to assert strong postconditions for both normal and exceptional loop terminations.

We propose a semantics for exceptions in parallel loops that solves these problems by always propagating the exception that occurs structurally first in the loop, not the exception that occurs first in time. This removes the nondeterminism that results when more than one iteration of the loop throws an exception, and allows exceptions to be handled in a manner that is consistent with the way exceptions are handled in sequential loops. It allows stronger postconditions to be asserted about the result of the loop even in the case when an exception is thrown, because it allows the exception handler to know that all structurally prior iterations of the loop have completed without exceptions.

The semantics presented here should be useful for scientific computing and for applications where parallelism in the program is used to improve performance, and where partial results are useful. It allows a large class of sequential loops to be parallelized with predictable and consistent semantics even in the presence of exceptions. It may be useful for debugging these kinds of applications, even where partial results are not useful, because it allows determinism in the case of multiple exceptions. It may also be useful in libraries where exceptions are caused not by programming errors in the library, but by bad data passed to the library. Because it integrates the exception semantics with the concurrency constructs, a programmer need not know whether a method in a library uses parallel loops in order to write correct code that uses exceptions.

Our semantics is most likely not useful for systems programming or real-time systems where concurrency is an inherent part of the problem to be solved, rather than a means to better performance. These kinds of computations do not benefit from parallel loops, and are better expressed as separate routines executing concurrently rather than one loop executing in parallel. The kind of concurrency normally provided by Java Threads [1] or Ada tasks [2] is probably more appropriate for this kind of system.

2 Parallel Loops

A parallel loop is a `for` loop in which the iterations of the loop execute concurrently rather than sequentially. When the program reaches a parallel loop it spawns multiple threads to execute the iterations of the loop, and requires that all threads complete before the program continues with the next statement. Each

thread within the loop receives its own copy of the iteration variable so that the threads can perform independent computations. For example, the following loop computes the vector sum of two arrays in parallel:

```
parfor (int i=0; i<N; i++)
    a[i] = b[i] + c[i];
```

Each iteration of the loop has its own value for i and operates on a different part of the data. Assuming that a does not share storage with b or c, all iterations of the loop can execute simultaneously. The program does not return to the surrounding block and continue with the next statement until all iterations have completed. This particular loop can be executed efficiently in parallel because it has no data dependencies between iterations.

Parallel loops may be either synchronous or asynchronous. Synchronous loops imply synchronization between statements in different iterations of the loop; the form of the implied synchronization varies from language to language, but all generally require each iteration to execute the same statement at the same time. A Fortran `forall` loop containing a single assignment statement as its body, for example, requires that the value of the right-hand sides of the assignments must be computed first for every iteration of the loop before any actual assignments are made, thus avoiding potentially harmful effects. This semantics is particularly well-suited to vector machines.

Asynchronous loops do not have any implicit synchronization between statements in different iterations. This allows the iterations of the loop body to proceed independently of one another. Explicit synchronization statements can be used to provide a stronger ordering on loops that require it. Asynchronous loop semantics are well-suited to systems that implement parallelism as threads that are scheduled independently. Asynchronous loops have appeared in Compositional C++ [3], Modula-3* [4], and other parallel languages.

Some implementations, such as the `forall` loops in Fortran 95 [5], require that the number of iterations of the loop, and their index values, be known when the loop starts; this simplifies the implementation and allows all iterations to start immediately, at the cost of flexibility. Other parallel loop constructs, such as the `parfor` construct in Compositional C++, do not have this restriction, and have the control portion of the loop execute sequentially while spawning threads to execute the body of the loop [3]; this allows loops where the number of iterations is not known in advance, or that do not have integer indices.

3 Exception Semantics for Parallel Loops

Although parallel loops have been implemented in a number of languages, few have been implemented in languages that have exceptions, and even fewer have attempted to address the semantics of an uncaught exception that occurs within a parallel loop. Exceptions in parallel loops introduce several difficult situations that must be addressed, especially when exceptions occur within more than one iteration of the loop. When an exception occurs within a sequential loop, the

loop simply stops executing and the exception is propagated to the calling block; because of sequential loop semantics, the handler may assume that this was the only exception that occurred, and that all iterations up to the exception completed normally. When an exception occurs within a parallel loop, the iteration that generated the exception necessarily stops executing, but it is less clear what to do about the other iterations, and how or if the exception should be propagated. The question of what to do if more than one iteration of the loop throws an exception is a difficult issue as well, because the exception semantics of most languages do not allow two exceptions to be raised simultaneously.

3.1 Goals for Exception Semantics

A simple approach to the problem of exceptions in parallel loops is to ignore uncaught exceptions in concurrent code, or to forbid them altogether. The designers of Ada, when confronted by the issues exceptions raise in a concurrent context, chose to ignore uncaught exceptions that reach the top level of a task. If an uncaught exception occurs in an Ada task, the task terminates without handling the exception or passing it on to an outer block that can handle it; this policy was chosen to prevent the problems that would result if the exception was passed to a parent task asynchronously [6]. The designers of pSather chose to forbid exceptions in a concurrent context [7] [8] [9]. These solutions deny concurrent programmers the expressive and robustness benefits of exceptions.

Programmers should be able to use exceptions in concurrent constructs as they would normally in the programming language, and be able to reason about their behavior. A good semantics for uncaught exceptions in parallel loops should be consistent. If the exception which is propagated from the loop is chosen nondeterministically, this could make the program difficult to reason about. Some forms of nondeterminism are unavoidable and even desirable in an asynchronous loop, because the parallel loop semantics impose only a partial order on the statements in the loop; however, the ability to predict and reason about the exception, if any, produced by the loop would make programs easier to write.

In order to recover from an exception in a loop, it is important to know the state of the program after loop termination. A good exception semantics should allow a strong postcondition to be asserted about the state of the program after an exception has been thrown. This postcondition can then be assumed by the exception handler, which can use the information in its recovery.

An exception semantics should also enable a clear termination condition for the loop. If an iteration within a loop terminates with an exception, and other threads within the loop depend on the normal completion of that iteration, a deadlock could result. A good exception semantics should provide a way to prevent this sort of problem from occurring whenever possible.

3.2 Exception Propagation

Java, C++, and other object-oriented languages which have exceptions can only raise one exception at a time. In a parallel loop, however, it may be possible to

raise multiple exceptions within the same block of code. It is important to have a clear semantics for what should happen in this case.

One possibility is to handle each exception in the loop as it happens, while allowing other iterations in the loop to continue, possibly generating more exceptions. This solution would create consistency problems because it would allow code within the loop to run at the same time as code in outer blocks, and could even result in destroying stack frames that are part of the loop's context. This would create code safety problems, and is inconsistent with the concurrent loop semantics defined in the previous section, because the semantics state that all iterations of the loop must complete before any outside exception handler begins.

Another possibility is to merge all of the exceptions together into a single exception that contains an array storing the exceptions generated by each iteration of the loop. This allows all exception information to be kept. The cost of this approach is that each iteration of the loop must be allowed to run to completion in order to determine which exceptions will be thrown. This could result in deadlock if there are dependencies between events in different iterations of the loop. Code written to catch and handle exceptions would need to be more complicated, because either `catch` expressions would need to be able to match patterns in the array, or some mechanism would be required to apply the chain of exception handlers to each exception in the array. This would be particularly complex if some exceptions were handled higher in the call chain, while others were handled at lower levels.

A scheme for collapsing multiple exceptions into a single exception in a Modula-3* parallel loop is described by Heinz [10]. If a single exception is generated by the loop, or if all exceptions in the loop are identical, a single copy of the exception is propagated out of the loop. If the exceptions are not all identical, a special exception is generated to note the inconsistency. This approach allows multiple exception cases to be handled in languages that can only raise a single exception at a time. However, it requires all iterations of the loop to complete, and requires programmers to handle the case of inconsistent exceptions.

In many cases, if multiple exceptions occur in a parallel loop, the exceptions are all related to the same problem, and it is only necessary to propagate one of the exceptions to ensure a proper recovery; other exceptions can be discarded. Philippsen and Blount describe an implementation of asynchronous parallel loops in Java that uses this approach [11] [12]. The first exception that occurs in time is propagated out of the loop; other iterations are canceled, and any exceptions they generate are ignored. This approach loses some information and introduces nondeterminism in the exception that is returned, but keeps within the single-exception model of the language, and does not require every iteration of the loop to complete.

Our approach is to return not the first exception that occurs in time, but the exception that occurs structurally first in the loop. This removes the nondeterminism in the exception that is propagated, and loops can be written in a way that guarantees that the most important exception, if any, will be the one that is propagated out of the loop. Our semantics requires structurally prior

iterations of the loop to complete so that any structurally earlier exceptions can be obtained. Structurally later iterations, however, can be canceled, thus eliminating deadlock in situations where no iteration of the loop waits for an event in a structurally later iteration.

3.3 Cancellation and Loop Termination

In order to determine what exception semantics is most appropriate for parallel loops, the termination condition imposed by each exception semantics must be considered. A straightforward solution is to require all iterations of the loop to terminate, even in the event of an exception. This approach is used by Heinz in Modula-3* loops [10]. In cases where some iterations of the loop wait or depend on events that occur in other iterations, the loop may deadlock. In particular, if an iteration of the loop terminates with an exception before creating an event that some other iteration is waiting for, the loop will deadlock because the second iteration is waiting for an event that will never occur. This situation can be created by the use of mutual exclusion for a shared resource, which is common in asynchronous parallel programs. Requiring all iterations of the loop to terminate even in the event of an exception is practical if it is known that there is no communication or explicit synchronization between iterations, but many loops do not satisfy this condition, so this solution cannot be used for a general parallel loop.

Another solution is to cancel some iterations of the loop in the event of an exception in order to guarantee that the loop terminates. One approach is to cancel all iterations in the loop other than the one that generated the exception. This approach guarantees that the loop will terminate in the event of an exception. Because it cancels some iterations, this cancellation strategy is not consistent with any approach to exception propagation that requires all iterations to complete. In addition, because this approach cancels both structurally prior and structurally later iterations, it cannot be used if the semantics requires the structurally first exception from the loop to be propagated. This approach is consistent with an exception propagation semantics that requires the first exception in time to be propagated.

Because canceling structurally prior iterations is inconsistent with a semantics that propagates the structurally first exception, we adopt a cancellation strategy that cancels structurally later iterations only. Because it requires some iterations of the loop to complete, it can result in deadlock if an iteration waits for an event that is generated by a structurally later iteration; however, if no iteration waits for an event generated by a later iteration, this cancellation strategy guarantees that the loop will terminate in the event of an exception, provided that the structurally prior iterations themselves terminate. This approach is consistent with a semantics that propagates the structurally first exception out of the loop.

3.4 Loop Postconditions

In order to write correct programs, it is important be able to make assertions about the state of the program after loop termination, even in the event of an exception. The exception semantics directly affects the kinds of postconditions that can be asserted about a loop that terminates with an exception.

Because different iterations of a parallel loop may share some context and data, and there is no explicitly defined order between events in one iteration of the loop and events in another, the postcondition that can be asserted of one iteration of the loop is not necessarily the same as the strongest postcondition that could be asserted of the same sequence of statements if executed in a sequential context. The postcondition for each iteration of the loop must be true for any possible interleaving of the different iterations of the loop and any potential interactions between them, and may not assume anything about the progress of other iterations. For example, consider the following loop:

```
parfor (int i=0; i<100; i++)
    A[i] = i*i;
```

If this loop were executed sequentially, the postcondition of iteration $i = 0$ would be $A[0] = 0$ with all other elements of A unchanged, or $A[0] = 0 \wedge \forall i : i > 0 : A[i] = A_{pre}[i]$. It it were executed concurrently, however, the strongest postcondition that can be asserted for iteration $i = 0$ is $A[0] = 0 \wedge \forall i : 0 < i < 100 : (A[i] = A_{pre}[i] \vee A[i] = i^2) \wedge \forall i : i >= 100 : A[i] = A_{pre}[i]$. This weaker postcondition takes into account the fact that the other iterations may not have completed by the time iteration $i = 0$ has. If different iterations of the loop attempted to write different values to the same variable without explicit synchronization, the postconditions for all iterations of the loop must account for each possible ordering of the writes.

The loop guarantee is the strongest condition that can be asserted of the loop at all times, and must allow for every possible interleaving of iterations, and every possible partial result. For this loop, this is $(\forall i : 0 \leq i < 100 : A[i] = A_{pre}[i] \vee A[i] = i^2) \wedge (\forall i : i \geq 100 : A[i] = A_{pre}[i])$, expressing the fact that any combination of the iterations may have completed.

The postcondition of the loop as a whole is the conjunction of the loop guarantee and the postconditions of all iterations of the loop that are known to have completed. For the above example, this is $(\forall i : 0 \leq i < 100 : A[i] = i^2) \wedge (\forall i : i \geq 100 : A[i] = A_{pre}[i])$, assuming that all iterations complete. The terms in the individual iteration postconditions that expressed uncertainty about the progress of the other iterations are absorbed when the fact that each iteration has completed is included in the postcondition.

If one or more iterations of the loop terminate with exceptions, the postconditions for these iterations may not be true, and they cannot be used in the loop postcondition. The cancellation strategy may prevent further iterations from completing, and the postconditions of these iterations may not be used either. In addition, although the exception propagation policy does not affect which iterations will complete, it may limit the knowledge the exception handler has

of the completion status of the loop. For any iteration that does not complete, the strongest postcondition that can be asserted is the uncertainty about how much progress the iteration made, which includes each potential partial result of the iteration; this condition of uncertainty is included in the loop guarantee.

If the first exception thrown in time is propagated from the loop, and other exceptions are ignored, an exception handler may not assume anything about the termination of other iterations of the loop, regardless of the termination strategy used. Even if the cancellation strategy allows other iterations of the loop to complete, it is possible that some of them may terminate exceptionally, and it would not be safe for the exception handler to assume anything about them. In this case, no iteration of the loop can be assumed to have completed, and no postcondition can be asserted of the loop that is stronger than the loop guarantee.

If the structurally first exception is propagated from the loop, and other exceptions are ignored, then all iterations structurally prior to the one that terminated abnormally are known to have completed. However, nothing can be asserted about the completion status of any other iterations, because any of them may have terminated exceptionally. If this exception propagation policy is used, the strongest postcondition that can be asserted of the loop as the exception handler is entered is the conjunction of the loop guarantee with the postconditions of all iterations structurally prior to the one that terminated with the exception. In the above example, if an exception occurred in iteration $i = 50$, the exception handler could assume $(\forall i : 0 \leq i < 50 : A[i] = i^2) \wedge (\forall i : 50 \leq i < 100 : A[i] = A_{pre}[i] \vee A[i] = i^2)$.

Because propagating the structurally first exception in the loop yields the most useful loop postcondition, and because canceling structurally later threads in the loop increases the class of problems for which the loop terminates in the exception case, we propose a semantics for parallel loops that propagates the structurally first exception. Under this semantics, the effect and postcondition of a loop that terminates with an exception is determined by the structure of the loop, not by the accident of its execution.

4 Extending Java with Parallel Loops

We chose to implement this semantics for parallel loops using Java as the base language. Although Java is not as commonly used for scientific computing as C++, its language specification includes concurrency, and its class hierarchy in which all exceptions are derived from a single `Throwable` type makes it possible to implement the extensions using a source-to-source translation. Because the C++ specification does not offer these features, a C++ implementation could probably not be done as a source-to-source translator. While we chose Java for its simplicity, we believe these ideas could be implemented in C++ or other object-oriented languages.

Concurrency in Java is expressed using objects, rather than through a control flow construct. Each thread of control in the program is implemented as a sep-

arate object that either extends the `Thread` class or implements the `Runnable` interface and has a controlling `Thread` object. Every object in Java has its own associated monitor which is used for synchronization. There are no explicit parallel constructs similar to parallel loops. This model is well suited for writing programs with separate, unrelated tasks that execute concurrently, such as having one thread compute in the background while another thread handles the user interface. This technique is used mainly to separate the tasks in an application and to reduce latency in user interfaces. This thread model is not well suited for data parallelism, where many threads are to perform the same or similar operations on different pieces of data as part of the same task, because the constructs for declaring, starting, and joining tasks are not control flow constructs that can easily be used in an algorithm description.

In order to define a `parfor` loop for Java that uses the structured exception semantics, we must modify the Java language semantics for a `for` loop. The Java Language Specification [13] defines a sequential `for` loop in terms of a condition *Expression*, a *ForUpdate* statement, and a contained body *Statement*:

- If the *Expression* is present, it is evaluated, and if evaluation of the *Expression* completes abruptly, the `for` statement completes abruptly for the same reason. Otherwise, there is then a choice based on the presence or absence of the *Expression* and the resulting value if the *Expression* is present:
  - If the *Expression* is not present, or it is present and the value resulting from its evaluation is `true`, then the contained *Statement* is executed. Then there is a choice:
    - If execution of the *Statement* completes normally, then the following two steps are performed in sequence:
      - First, if the *ForUpdate* part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If evaluation of any expression completes abruptly for some reason, the `for` statement completes abruptly for the same reason; any *ForUpdate* statement expressions to the right of the one that completed abruptly are not evaluated. If the *ForUpdate* part is not present, no action is taken.
      - Second, another `for` iteration step is performed.
    - If execution of the *Statement* completes abruptly, see §14.13.3 below.
  - If the *Expression* is present and the value resulting from its evaluation is `false`, no further action is taken and the `for` statement completes normally.

In order to achieve parallel execution of the loop, we must modify this definition so that instead of executing the *Statement* directly, it starts a thread which executes the *Statement* concurrently. The *Statement* thread receives its own copy of the iteration variable so that it may execute independently of the loop control and the other iterations. In addition, we must add a rule that states that when the loop completes, either normally or abruptly, it must wait for all concurrent *Statement* threads to complete.

These changes allow the loop to execute concurrently, but further changes are needed to handle abrupt termination in the event of an exception. Section 14.13.3 of the specification describes abrupt termination of a `for` loop in the case of a `break` or `continue` statement, and states that "If execution of the *Statement* completes abruptly for any other reason, the `for` statement completes abruptly for the same reason." For simplicity, we do not allow `break`, `continue`, or `return` statements in `parfor` loops, although they could be introduced in ways consistent with our semantics. Hence, the only form of abrupt termination we consider here is through exceptions. In the event of an abrupt termination, the control portion of the loop should complete immediately, and no further *Statement* threads should be started. Any running *Statement* threads that were started after the *Statement* that generated the exception should be interrupted, which will allow them to terminate cleanly. Once all *Statement* threads have completed, the loop should terminate by propagating the structurally earliest exception.

Together, these changes produce the following specification for a Java `parfor` loop:

- First, the `parfor` control loop is executed:
  If the *Expression* is present, it is evaluated, and if evaluation of the *Expression* completes abruptly, the `parfor` control completes abruptly for the same reason. Otherwise, there is then a choice based on the presence or absence of the *Expression* and the resulting value if the *Expression* is present:
  - If *Expression* is not present, or it is present and the value resulting from its evaluation is `true`, and no existing *Statement* thread has terminated exceptionally, then a thread is started to execute the contained *Statement* concurrently with the `parfor` control. A copy is made of the iteration variable, the new thread uses the copy.
    * Then the `parfor` control performs the following two steps in sequence:
      · First, if the *ForUpdate* part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If evaluation of any expression completes abruptly for some reason, the `parfor` control completes abruptly for the same reason; any *ForUpdate* statement expressions to the right of the one that completed abruptly are not evaluated.
      · Second, another `parfor` control step is performed.
  - If the *Expression* is present and the resulting value of its execution is `false`, no further action is taken and the `parfor` control completes normally.
- If a *Statement* thread terminates abruptly, all running *Statement* threads that were started after the one that terminated abruptly are interrupted. The `parfor` control then completes without starting any new *Statement* threads.
- When the `parfor` control completes, it must wait for all of its *Statement* threads to complete before terminating. If any errors or exceptions occurred

in the execution of either the `parfor` control or any *Statement* threads, the error or exception from the earliest iteration is propagated as the reason for the `parfor` statement's termination.

To illustrate the semantics, consider the array assignments example introduced in §3.4. If A is an array of size 50 instead of size 100, any attempt to access elements 50 through 99 will result in an `ArrayBoundsException`. Because the iterations of the loop execute in parallel, however, it is not possible to know in advance which iteration will cause an exception to be thrown first, nor is it possible, once that statement has been reached, to know which iterations of the loop have already completed or how much of the array has been filled.

With the proposed semantics for abnormal loop termination, although it cannot be known which iteration will generate the first exception, we do know that the structurally first exception will be the one that is propagated, and that all lower iterations will complete before the loop terminates. This means that even though we cannot know whether iteration $i = 50$ or iteration $i = 60$ will generate an exception first, we do know that the `ArrayBoundsException` generated by iteration $i = 50$ will be the one propagated by the loop, and the handler for that exception may safely assume that iterations $i = 0..49$ have completed and that the array up to the point the exception occurred has been filled, and this can be asserted as a postcondition of the loop.

The result of an exception with this semantics is consistent with the semantics of a normal sequential `for` loop. In a normal `for` loop, only one iteration of the loop executes at a time, and the iterations execute in order. If an exception is thrown in one iteration of the loop, the loop terminates immediately and propagates the exception to the calling block, and no further iterations of the loop are executed. The program may safely assume that all iterations up to the iteration that threw the exception have completed normally, and the handler for the exception can use this information when recovering. Even if there are additional problems that would have caused higher-numbered iterations of the loop to throw exceptions, only the first such problem is caught and reported. Our proposed exception semantics for parallel loops shares these properties: the lower-numbered iterations of the loop are guaranteed to have terminated normally, and the exception that is propagated is the one from the lowest-numbered iteration that threw an exception.

4.1 Handling Exceptions from Parallel Loops

In order for an exception handler to be able to make use of the loop postcondition described above, it must know how much of the loop completed and what iteration it was that generated the exception. To support this, we allow `catch` and `finally` clauses to be attached directly to `parfor` statements. The iteration variable for the loop remains in scope within these clauses. Because more than one copy of the iteration variable exists in the parallel loop semantics, we must specify what the iteration variable in the exception handlers refer to.

If an exception occurs within the body of the parallel loop, the value of the iteration variable in the handler should be the same as the value of the iteration variable in the instance of the body that raised the exception. If an exception is generated in the sequential control portion of the loop, then the handlers should see the current value of the iteration variable from the control portion when it stopped. This allows the handler to know which iteration caused the exception, so that it can use this information when recovering from the exception.

If an exception occurs in the initialization statement of the loop, which declares the iteration variable, the variable is undefined, and the `catch` and `finally` clauses of the loop cannot be allowed to execute because the result would be undefined and could introduce a code safety issue. For these purposes, the initialization statement can be considered to be outside of the loop, and outside of any attached `catch` and `finally` clauses. Exceptions generated by the initialization statement can still be handled, but they must be handled outside of the loop.

If no `catch` clause attached to the loop handles the exception, the exception will propagate up the call chain normally, and the information about which iteration caused the exception will be lost. An alternative might be to include the information about which loop and which iteration threw the exception in the exception object itself. However, this is undesirable. It would require a fundamental change to Java's exception model which would affect much more than just loops, and that would make the parallel exception semantics inconsistent with all normal Java programs. Furthermore, it is unlikely that the information about which iteration caused the exception would be useful outside of the context of the loop.

Although an exception handler inside the loop body, rather than outside the loop, would also have access to the value of iteration variable, such a handler would not have the knowledge that all iterations of the loop have completed. An exception handler outside the loop body, but without access to the iteration variable, would not be able to take advantage of the fact that all structurally prior iterations have completed normally. The possibility that an exception handler could use both the value of the iteration variable and the knowledge that all loop iterations have completed (and structurally prior iterations completed normally) justifies allowing a special handler to be attached to the loop which preserves the value of the iteration variable.

4.2 Safe Thread Cancellation

The semantics require that all iterations structurally prior to an iteration that causes an exception must be allowed to complete, because they may also throw exceptions, and we want to propagate the exception that is structurally first, not first in time. Any higher iterations that have not yet started will never be started, and higher iterations that are already running are interrupted. The purpose of this is to cancel those iterations that occur logically after the exception and are therefore invalid.

Cancellation of running threads is difficult to do safely in an object-oriented language [7]. Simply terminating a thread with no mechanism to allow cleanup is unsafe and deadlock-prone, because the thread may be holding locks or may have temporarily placed an object in an inconsistent state. For this reason, a mechanism to allow the thread to exit cleanly is needed.

One mechanism for terminating a thread provided by early versions of Java allowed an exception to be delivered to a thread asynchronously, though the `Thread.stop()` method. This is dangerous, however, because it can deliver an exception to code that was not designed to handle that particular exception. Although it is theoretically possible to write code that can handle asynchronous exceptions, it is very cumbersome to read and very difficult to get correct, particularly when the exception is delivered in a `finally` clause or `synchronized` block. In any case, the mechanism is not guaranteed to cause the thread to stop: the thread's code could be written in a way to trap such exceptions and ignore them. Because of these problems, Sun chose to deprecate this feature of Java [14].

Instead, Sun recommends using the `Thread.interrupt()` mechanism to cancel running threads, which is safe because it causes the thread to be interrupted only at well-defined points in the code. Certain library routines can generate an `InterruptedException`, which is a checked exception which must be caught or declared. Other code can explicitly check to see if it has been interrupted by using the `Thread.interrupted()` call, and it is the responsibility of the programmer to make sure that the thread exits cleanly. Although it is possible for code to ignore the interruption and continue running, this is not worse than the `Thread.stop()` method, which cannot provide this guarantee either, and is much harder to use safely [1].

For these reasons, we attempt to terminate running iterations of the loop using the interrupt mechanism. Because poorly written code could ignore the interrupt mechanism, the semantics cannot provide a guarantee that canceled iterations will terminate promptly, or even that they will terminate at all; however, correct Java code should terminate when interrupted. Because the loop semantics require all iterations of the loop to terminate before the loop itself terminates, the semantics cannot guarantee for arbitrary loops that the loop as a whole terminates either, even when a loop iteration raises an exception. Care must be taken to make sure that loops are written such that they will terminate cleanly, even in the presence of exceptions. This is already the case for sequential loops, and for all Java code, so this limitation of the parallel loop semantics is not worse than the limitations of the exception semantics for any other construct of Java.

4.3 Example

Suppose we wished to use a computer to render frames of an animation. Because each frame is independent of other frames in the video, they can be rendered in parallel, saving computation time. A procedure to do this is:

```
void render_frames(Frame[] frames,int start,int end) {
    parfor (int i=start; i<end; i++) {
        images[i] = frames[i].render();
    }
}
```

This loop would render the separate frames of the sequence in parallel. However, if an exception occurred while rendering the frames, the loop would stop. Because many frames have already been rendered at this point, and because rendering is a computationally expensive task, we would prefer to keep the partial results even in the event of an exception, and use these when recovering from the exception. For example, if the machine runs out of memory while rendering the frames, we would like to be able to recover from this situation by saving the completed frames to disk, and then continuing where the program left off. It is advantageous to save a contiguous block of frames together because this maximizes the effectiveness of a compression algorithm. Our proposed exception semantics allows this to be expressed easily:

```
void render_frames(Image images[],Frame[] frames,
                   int start,int end) {
    parfor (int i=start; i<end; i++) {
        images[i] = frames[i].render();
    } catch (OutOfMemoryException e) {
        // Delete frames that come after the exception
        for (int j=i; j<end; j++) {
            images[j] = null;
        }
        // Save earlier, completed images
        save_images(images,start,i-1);
        // Delete completed images to recover memory
        for (int j=0; j<i; j++) {
            images[j] = null;
            frames[j] = null;
        }
        // Render remaining images through recursive call
        render_frames(images,frames,i,end);
    }
}
```

This example recovers from an exception by deleting all rendered images that come logically after the frame that caused the exception, and saving all images that come logically before the exception. Images that come logically after the exception are not saved, because they may not be complete, and would not be contiguous with the earlier frames. Because the postcondition guarantees that the exception thrown is the structurally first exception and that all prior iterations have completed, these images can be saved to disk together and do not

need to be recomputed. Because these images are in a contiguous block without holes, it should be possible to compress them easily; this would not be possible if all completed images were saved, including any completed frames logically after the image that caused the exception.

5 Implementation

The proposed language extensions can be implemented using a library of new classes that encapsulate the loop and exception semantics, and a source-to-source translator that translates the extensions into standard Java code that uses the library. The resulting code can then be compiled by any Java compiler. This chapter describes the implementation of the library, and the design of a source-to-source translator. The source-to-source translator for our extensions has not yet been implemented, but similar extensions have been implemented as source-to-source translators before, and implementation of our extensions would be straightforward.

5.1 Strategy

An extension to Java could be implemented as a Java library, a source-to-source translator, a full compiler generating standard JVM bytecode, or a compiler generating special bytecode for a modified JVM.

A modified JVM would allow the most flexibility in what can be implemented, but would come at the cost of portability and interoperability with existing Java code. The JVM's built-in `Thread` class could be redefined so that uncaught exceptions generated by the thread would be passed to the parent thread upon termination; the parent thread could receive the exceptions when it attempts to `join` the child threads. This implementation technique would require a compiler to recognize the `parfor` loops and generate the code needed to implement them.

A full compiler has the ability to perform some transformations that generate valid JVM bytecode but cannot be expressed in Java in source form, but is also complex and can introduce interoperability problems. A source-to-source translator cannot perform as many transformations as a full compiler, but because its output is standard Java source code, it does not introduce any portability or interoperability problems. In addition, the code generated by a source-to-source translator is readable, which makes debugging and analysis easier. For these reasons, we chose to implement the extensions as a library along with a description of a source-to-source translation. A more detailed analysis of these options for implementing parallel loops in Java is provided by Philippsen [11].

The implementation described here uses an inner class to represent each parallel loop, and a set of methods that spawn threads, execute the loop, and wait for the threads to terminate. If an exception is generated by the loop, the loop class waits for structurally prior iterations to complete, and re-throws the structurally first exception to the calling code. This allows the parallel loop

constructs to be implemented without changing the virtual machine, and allows code with parallel loops to work transparently with standard Java code.

Although we have not implemented an automatic source-to-source translator for our proposed extensions, we describe here how the extended features could be translated mechanically into normal Java code. Other researchers have implemented automatic translators that generate standard Java code for parallel loops using similar techniques [12] [11] [15] [16], and it should be straightforward to modify one of these translators to use our proposed exception semantics.

5.2 Loop Translation

To implement `parfor` loops, the `ParforLoop` class manages a single parallel loop and is responsible for creating threads, waiting for them to complete, catching exceptions, interrupting running threads, and propagating exceptions back to the caller. `ParforLoop` has undefined abstract methods for the condition expression, update statement, and loop body; each method takes an `Object` parameter that holds the current value of the loop's iteration variable. The class is not instantiated directly: in order to use the class, a child class must be derived from it that defines these methods for the particular loop that is to be executed.

A `parfor` loop is translated as an inner class that extends the `ParforLoop` class and defines the abstract methods, and executed by calling the ParforLoop class's `loop` method. This transformation requires a few fairly simple substitutions.

The loop is described by overriding the abstract `condition`, `update`, and `body` functions to contain the loop's condition expression, update statement, and body implementation, respectively. These functions each take a reference to the iteration variable as a parameter. The loop is actually executed by a call to the `ParforLoop.loop` method, which is passed the initial value of the iteration variable. This method executes the control portion of the loop and starts threads to execute the body of the loop, and does not return until the loop has terminated.

Each iteration of the loop is executed by an instance of the `StructuredThread` class. This class calls the `ParforLoop`'s `body` method, and catches any exceptions that occur. If an exception occurs, it passes it to the `ParforLoop` object's `catchException` method, which actually implements the exception semantics by storing the exception along with its iteration number in the `ParforLoop` object, and interrupting any structurally later threads that are still running.

Once all iterations have stopped executing, `ParforLoop.loop` propagates any exception that occurred by re-throwing it to the calling block. If the loop has any extended `catch` or `finally` clauses attached, for which the iteration variable is to remain in scope, the handlers described there are made part of a `try..catch` block surrounding the call to `loop`. The value of the iteration variable is retrieved by calling the `getIteration()` method of the `ParforLoop` object, so that the handlers may use this information in their recovery.

5.3 Performance

The overhead associated with our exception semantics should not significantly affect the performance of concurrent Java programs. The cost of setting up the loop and starting threads is linear in the number of iterations of the loop. In the normal case in which there are no exceptions, the cost of joining the threads in the loop and cleaning up is also linear in the number of iterations of the loop, and introduces little overhead.

If the loop terminates exceptionally, the running time may be longer than other methods because our semantics requires all earlier iterations of the loop to complete when an exception is thrown, while other semantics allow all threads to be terminated immediately in this situation. However, the running time of this should not be any worse than that of normal execution of the loop. It is possible that a loop which uses concurrency control constructs incorrectly or which ignores the `InterruptedException` may deadlock or enter an infinite loop if an exception occurs; however, this is the result of an incorrect program and not the result of the translation.

The translation does increase code size. Any program using the translation must use the `ParforLoop` library. The translation itself creates a new inner class and several methods for each loop, and creates additional code to handle each type of exception that could be expected from the loop. The increase in code size is linear in the number of translated `parfor` loops. Relaxation of the Java requirement that all exceptions must be caught or declared (except for errors and run-time exceptions) would make the translation simpler and would reduce the size of the translated code.

The library described here uses the simple approach of spawning a separate Java thread to execute each iteration of the body of the loop. Other methods of implementing parallel loops in Java achieve higher performance by not using a one-to-one mapping of loop iterations to threads; see [15] for a comparison. However, such a strategy must deal with the possibility that there are dependencies between different iterations, and it can be difficult to determine when two iterations can be executed sequentially and when they must be executed concurrently. The mechanisms needed to deal with this situation are complex. Because our purpose here is to demonstrate our proposed exception semantics, rather than how to achieve optimum performance, we chose the simpler one-to-one mapping of iterations to threads. Our exception semantics could be incorporated into an implementation that uses a more efficient mapping of iterations to threads.

6 Conclusions

The goal of our semantics for exceptions in parallel loops is to facilitate the writing of parallel programs by making them more predictable and understandable, while allowing exceptions to be used in a consistent way with other language features. Under our semantics, exceptions can be used in any context, and have a well-defined meaning even in parallel loops. Exceptions in parallel loops behave

in a manner analogous to exceptions in sequential loops, because in both cases the structurally first exception thrown is the one that is propagated out of the loop. The attempt to cancel later iterations through interruption is consistent with the fact that later iterations in a sequential loop do not run. The proposed exception semantics are a generalization of the standard Java semantics for sequential loops, and remain consistent with them.

Our semantics allow strong postconditions to be asserted in the presence of exceptions, because the exception that is propagated is chosen deterministically, based on the structure of the program rather than on the accident of its execution. This also provides repeatability for some kinds of errors in concurrent programs, which facilitates debugging. Other semantics for exceptions in parallel loops, such as propagating the first exception that occurs in time, do not have this property. Of course, there are other sources of nondeterminism in concurrent programs, and this semantics does not eliminate all of them; indeed, that would not be possible without giving up the benefits of parallelism. But it does eliminate some nondeterminism in a way that should make program behavior more predictable and easier to understand and reason about.

The semantics also allows stronger assertions about what partial results have been computed in the exception case. Keeping partial results in the case of an exception can save time when recomputing or recovering from the exception. Partial results may be useful in some cases even when the program does not attempt to repair and recompute after the exception has been handled. Other choices for an exception semantics do not allow strong postconditions to be asserted.

For some loops, it does not matter which exception is structurally first, or what partial results have been computed, and for these loops, propagating the first exception in time and canceling or interrupting all remaining threads in the loop may make more sense. One possibility is to use one keyword to indicate parallel loops for which partial results are useful and deterministic exception semantics is desired, in which our exception semantics would be used, and another keyword to indicate parallel loops which express inherently concurrent algorithms for which it is not important which exception is structurally first.

Our semantics is presented in the context of Java, but it could be applied to other languages. The partial results feature may be particularly helpful in implementing the `retry` keyword in languages that have one, such as Eiffel [17]. This would allow the loop to restart at the point the logically first exception occurred, while keeping previously computed partial results.

The ability to express concurrency in a clean and natural way is an important feature to have in modern programming languages. Parts of programs that express concurrency or parallelism should be well integrated with the rest of the language, and it is important for language designers to consider how concurrency interacts with exception handling in particular. We have presented a concurrency construct for Java that provides exception semantics that are consistent and integrated with the language, and that can be implemented with modest overhead.

Acknowledgements

This work was supported by NSF CCR-0092945 and NASA NRC 99-LaRC-4. The authors thank Paul Reynolds and John Thornley for helpful comments and suggestions.

References

1. Sun Microsystems: Java 2 Platform, Standard Edition, v 1.2.2 API Specification. (1999) http://java.sun.com/products/jdk/1.2/docs/api/index.html.
2. Barnes, J.G.P.: An overview of Ada. Software - Practice and Experience **10** (1980) 851–887
3. Carlin, P., Chandy, K.M., Kesselman, C.: The Compositional C++ language definition. Technical Report 1993.cs-tr-92-02, Department of Computer Science, California Institute of Technology (1993)
4. Heinz, E.: Modula-3*: An efficiently compilable extension of modula-3 for explicitly parallel problem-oriented programming. In: Joint Symposium on Parallel Processing, Tokyo, Waseda University (1993) 269–276
5. Adams, J.C.: Fortran 95 Handbook. MIT Press, Cambridge, MA (1997)
6. Ichbiah, J.D., Heliard, J.C., Roubine, O., Barnes, J.G.P., Krieg-Brueckner, B., Wichmann, B.A.: Rationale for the design of the ADA programming language. ACM SIGPLAN Notices **14** (1979) 1–247
7. Fleiner, C., Feldman, J., Stoutamire, D.: Killing threads considered dangerous (1996)
8. Murer, S., Feldman, J.A., Lim, C., Seidel, M.: pSather: Layered extensions to an object-oriented language for efficient parallel computation. Technical Report TR-93-028, International Computer Science Institute, Berkeley, CA (1993)
9. Stoutamire, D., Omohundro, S.: The pSather 1.1 manual and specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, CA (1996)
10. Heinz, E.A.: Sequential and parallel exception handling in Modula-3*. In Schulthess, P., ed.: Advances in Modular Languages: Proceedings of the Joint Modular Languages Conference, Ulm, Germany (1994) 31–49
11. Philippsen, M.: Data parallelism in Java. In Schaefer, J., ed.: High Performance Computing Systems and Applications. Kluwer Academic Publishers, Boston, Dordrecht, London (1998) 85–99
12. Blount, B., Chatterjee, S., Philippsen, M.: Irregular parallel algorithms in JAVA. In: Parallel and Distributed Processing, 6th International Workshop on Solving Irregularly Structured Problems in Parallel. Number 1586 in Lecture Notes in Computer Science, Puerto Rico, Springer Verlag (1999) 1026–1035
13. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Second edn. Addison-Wesley (1997)
14. Sun Microsystems: Java 2 Platform, Standard Edition, v 1.2.2 API Specification. (1999) http://java.sun.com/products/jdk/1.2/docs/guide/misc/thread-PrimitiveDeprecation.html.
15. Oliver, J., Ayguade, E., Navarro, N.: Towards an efficient exploitation of loop-level parallelism in Java. In: Java Grande. (2000) 9–15
16. van Reeuwijk, C., van Gemund, A., Sips, H.: Spar: A programming language for semi-automatic compilation of parallel programs. Concurrency: Practice and Experience **9** (1997) 1193–1205

17. Meyer, B.: Eiffel: The Language. Prentice Hall International, Hemel Hempstead, UK (1992)