Handling Errors in Parallel Programs Based on *Happens Before* Relations

Nicholas D. Matsakis
Laboratory for Software Technology
ETH Zurich
Zurich, Switzerland
nmatsaki@inf.ethz.ch

Thomas R. Gross
Laboratory for Software Technology
ETH Zurich
Zurich, Switzerland
trg@inf.ethz.ch

***Abstract*—*Intervals* are a new model for parallel programming based on an explicit *happens before* relation. Intervals permit fine-grained but high-level control of the program scheduler, and they dynamically detect and prevent deadlocking schedules. In this paper, we discuss the design decisions that led to the intervals model, focusing on error detection and handling. Our error propagation scheme makes use of the *happens before* relation to detect and abort dependent tasks that occur between the point where a failure occurs and where the failure is handled.**

***Keywords*-exceptions; parallelism; intervals**

I. Introduction

Threads are a flexible and efficient model for parallel programming, but they are also very low-level. Like any low-level interface, they require careful attention to implementation details to achieve good performance, and they are prone to errors, particularly deadlocks and race conditions.

Typical thread primitives, such as signals and barriers, are operational in nature, meaning that the user does not describe the schedule they want to achieve, but rather the means to achieve it. As an example, consider barriers: the desired goal of a barrier is to separate one phase of computation from the next. The barrier primitive itself, however, knows nothing about which phase the program is in. It simply has the effect of suspending the invoking thread until all others have reached the barrier. This effect can be used to separate computation phases, but can also create a deadlock if one thread fails to reach the barrier the same number of times as the others. Even in a program which seems correct, it is often possible that one thread might encounter an unexpected exception and abort before it reaches the barrier, leaving the others permanently suspended.

A higher-level interface for phased computation would instead provide a means to designate which work belongs to the current phase of computation and which work belongs to the next. The runtime would then cause threads to block or assist each other as needed to ensure that the phases were completely as efficiently as possible but without overlapping.

The *intervals* model aims to provide such a high-level interface while retaining the flexiblity and efficiency of threads. In the Intervals model, users create lightweight tasks and order them using explicit *happens before* relations [1]. Users never explicitly block a thread or acquire a lock: rather, they specify when a task should execute relative to other tasks and what locks it should hold when it executes. The details of making this schedule come to pass are left to the runtime system.

Because the intervals API supports arbitrary *happens before* relations, the model is very flexible. Intervals can be used to emulate existing thread primitives [2], but they can also be used to easily create program schedules for which no standard thread primitives exist, such as peer-to-peer synchronization.

One of the primary goals in developing intervals is that program errors should not lead to deadlocks and should never be silently ignored. This includes not only errors that come about from misuse of the intervals API but also those from other sources, such as dereferencing a null pointer. Rather than causing deadlocks or (perhaps worse) having no effect at all, an error in one parallel task should prevent other, dependent tasks from executing until it is handled.

The focus of this paper is the detecting and handling of errors within the interval model, and in particular within the Java implementation. The paper begins by introducing the model and API in greater detail. We discuss the various illegal uses of the API and how each can be detected. Next, we show how to use the *happens before* relation to guide error propagation so that dependent tasks do not execute, and give a number of examples illustrating how the system works in various representative scenarios. Finally we compare our work to existing approaches for parallel error detection and propagation.

II. Intervals Model and API

Intervals are first-class objects representing the slice of program time used to execute some parallel task. Intervals are structured hierarchically, which allows an interval to make use of other parallel subintervals. The root of the interval tree, called `root`, represents the entire program execution. Program execution itself begins in a child of the `root` interval.

Each interval has two associated *points*, `start` and `end`, each representing a moment in time. The start point represents the moment when the interval begins execution.

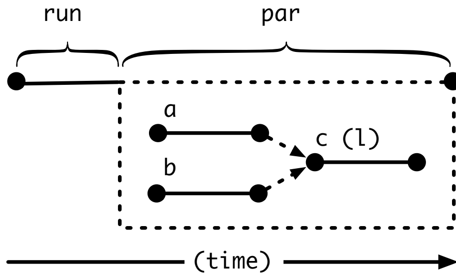

Figure 1. A sample interval diagram showing one interval and its subintervals a, b, and c. The two phases of an interval's execution are labeled.

The end represents the moment when the interval's task is completed. Programmers may introduce arbitrary ordering constraints by adding *happens before* edges between the start or end points of different intervals.

An interval can be associated with one or more locks. The intervals runtime will automatically acquire those locks before the interval's start point occurs and release them after its end point has occurred. Locks are useful when there are two intervals which cannot execute simultaneously, but for which a *happens before* relation would be too strong, as the two intervals could legally execute in any order.

We refer to the abstract schedule of an interval program as the *interval graph*. Interval graphs are depicted using a diagram like the one in Figure 1. Figure 1 contains a single interval with three subintervals, a, b, and c. The start and end points of each interval are represented as opaque circles. The subintervals of an interval are enclosed in a dashed box. This dashed box is omitted for leaf intervals.

The dashed edges connecting different points indicate user-specified additions to the *happens before* relation. For example, the ends of a and b both *happen before* the start of c.

Finally, the locks held by an interval are listed alongside its name in parentheses: the interval c, for example, acquires the lock l before executing.

A. Phases of Execution

When an interval executes, it begins by invoking a sequential method called run(). If the task of the interval is very short, then this method may simply perform the task directly. Otherwise, run() can create a number of subintervals to achieve the task in parallel. These subintervals begin to execute as a second phase once the run() method has completed. The two phases of an interval's lifetime are called run and par, and are depicted graphically in Figure 1. We sometimes omit the run phase from future interval diagrams.

The par phase lasts until all subintervals have completed; subintervals may themselves create new subintervals, so the par phase can be extended as needed.

```
1 class Interval {
2     final Point start;
3     final Point end;
4     final Interval parent;
5
6     abstract void run();
7 }
```

Figure 2. The class Interval, which serves as the base class for all intervals.

The separation of the run and par phases of an interval's execution gives the user an opportunity to construct any required dependencies between subintervals before they can begin execution. The precise rules used to ensure that modifications to the graph are safe are given in section Section II-D.

B. *The* Happens Before *Relation*

During execution, users can control the schedule by creating arbitrary *happens before* edges. An edge $p_1 \rightarrow p_2$ indicates that the point p_1 must occur before the point p_2; it also indicates that any memory writes which *happen before* p_1 must be visible to p_2.

Happens before edges are typically used to cause one interval to start after another has ended, as shown in Figure 1. However, it is also possible to create edges between two start points, or which delay an end point from occuring. One example where such unconventional edges might be useful is to implement hand-over-hand locking, as described in [2].

In addition to user-defined edges, each interval i_c is related to its parent i_p by two edges i_p.run $\rightarrow$ i_c.start and i_c.end $\rightarrow$ i_p.end. The point i_p.run refers to the moment when the interval's run method completes. Unlike the start and end points, there is no object representing this point at runtime, but it conceptually *happens after* the start of the interval and *happens before* the end.

C. Java API

In the Java API, intervals are represented as instances of the abstract class Interval, as shown in Figure 2. Interval provides immutable fields to access the interval's start point, end point, and parent, along with an abstract run() method which must be redefined in a concrete subtype.

Figure 3 provides an example of code to create the schedule from Figure 1. The code is presented in a Java-like language extended with a keyword interval that creates a new interval with the given run method.[1] The resulting object will be an instance of an anonymous subtype of the class Interval.

[1]In the library implementation, this keyword is replaced with a Java anonymous class.

```
void method(Interval parent, Lock l) {
    Interval inter = interval (parent) {
        Interval a = interval {
            // run method for 'a'
        };
        Interval b = interval {
            // run method for 'b'
        };
        Interval c = interval {
            // run method for 'c'
        };

        a.end.addHb(c.start);
        b.end.addHb(c.start);
        c.addLock(l);

        assert b.end.hb(c.start);
        assert c.locks(l);
    }
}
```

Figure 3. Code to produce the sample interval diagram shown in Figure 1.

The method shown in Figure 3 takes two parameters: an existing interval `parent` (not depicted in the diagram from Figure 1) and a lock `l`. It creates the interval `inter` on line 2 with `parent` as its superinterval. When `inter` is scheduled, its `run()` will then in turn create the intervals `a`, `b`, and `c` on lines 3–9 (because no super interval is specified, the default is the current interval, or `inter`).

The methods `addHb()` and `addLock()`, shown on lines 13–15, are used to relate existing objects. $p1$`.addHb(`$p2$`)` creates a new *happens before* edge between $p1$ and $p2$, and i`.addLock(`l`)` ensures that interval i will acquire lock l.

Similarly, the methods `hb()` and `locks()`, shown on lines 17 and 18, can be used to query the schedule. $p1$`.hb(`$p2$`)` returns `true` if $p1$ *happens before* $p2$, and i`.locks(`l`)` returns `true` if the interval i will hold lock l when it executes.

D. Errors Creating the Interval Graph

Whenever the user creates a new *happens before* edge or a new interval, there are certain safety criteria which must be met to ensure that the request can be respected by the scheduler. In this section, we discuss those criteria and explain how to detect violations.

We have designed compiler extensions that can detect cyclic schedules and other errors ahead of time. Using these extensions should allow us to eliminate dynamic checks in most cases. However, the overhead from dynamic checks in practice is generally negligible.

1) The No-Rollback Requirement: Whenever an API command results in a new dependency $p_1 \rightarrow p_2$ that targets an existing point p_2, it is possible that the point p_2 might already have occurred. Because the scheduler cannot force p_2 to wait once it has already occurred, this *happens before* edge would not necessarily be satisfiable.

To avoid this scenario, we require that the point creating the dependency must *happen before* p_2. This way, the scheduler can be certain that p_2 has not occurred. This rule is known as the *no-rollback requirement*, because it prevents the need for rolling back points which have already occurred. The precise conditions required to enforce this rule depend on whether the dependency came about from a new interval creation or the addition of a *happens before* edge.

When a new interval is created with the parent i_p, the danger is that i_p may have already exited the `par` phase and thus may not accept new children. To prevent this, an interval i is only permitted to create a new child of an interval i_p if

1) the end of i *happens before* the start of i_p; or
2) $i = i_p$; or
3) i is a descendent of i_p.

In the first case, we know that i_p cannot have started yet, as the interval i has not yet ended. In the second case, we know that i_p is still in the `run` phase (since its `run()` method is the one creating the child). In the final case, we know that i_p is still in the `par` phase because it has an active descendant (namely i).

The other scenario where the no-rollback requirement could be violated is when the interval i adds a new *happens before* edge $p_1 \rightarrow p_2$ to the graph. In this case, we require that the target point p_2 either

1) be a point on some child of i; or,
2) be the end of i or some ancestor of i; or,
3) *happen after* the end of i.

Any one of these three conditions is sufficient to guarantee that the target cannot have occurred yet. In the first case, as i is still executing its `run()` method, any point on a child of i must not have occurred. In the second case, as i is still active, neither it nor its ancestors can have ended. In the final case, as i has not ended, none of the points which *happen after* its end can have occurred.

2) The Acyclic Requirement: The *happens before* relation must always be acyclic, as otherwise it could not be executed. *Happens before* relations which are derived purely from parent-child relationships are always acyclic. Whenever the user adds a custom edge into the graph, however, there is the potential for it to cause a cycle.

Protecting against a cycle must be done with care. The problem is that the check to see whether an edge $p_1 \rightarrow p_2$ creates a cycle and the actual addition of the edge must be performed atomically. Otherwise, it is possible for two parallel intervals to simultaneously add edges which are individually fine but together cause a cycle. If the timing is just right, both of these parallel intervals might pass the cycle check before either of them actually modifies the graph.

We elected to use an optimistic algorithm to resolve

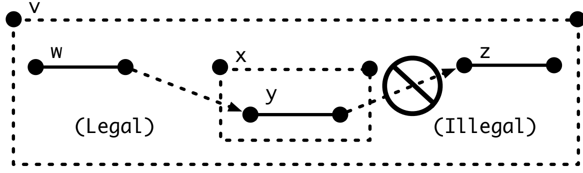

Figure 4. The edge from `y` to `z` violates the encapsulation requirement, because `z` is not a descendant of `y`'s parent, `x`.

this problem[2]. Prospective edges are added to the graph immediately, but marked as speculative. Then a cycle check is performed; if the cycle check fails, then the edge is removed and the graph state is recovered. If the cycle check succeeds, the edge is marked as confirmed. The speculative and confirmed markers on the edge are used to ensure that edges are not visible to the user until they are considered final. In the case of simultaneous but conflicting additions to the graph, this approach guarantees that at least one (if not both) will result in an exception.

Removing erroneous edges from the graph is not difficult. Due to the no-rollback requirement described above, we know that the target of the edge cannot have occurred yet, because it is dependent on the current interval. Therefore, we need only unlink the speculative edge from the graph.

3) The Encapsulation Requirement: An interval i must encapsulate its subintervals. This means that there can be no outgoing edge leading from one of i's subintervals to a point outside of i. This principle is illustrated in Figure 4, which shows five intervals named alphabetically `v` to `z`. The edge from `w` to `y` is legal, because `y` is a descendant of `w`'s parent `v`. The edge from `y` to `z`, however, is illegal, because `y`'s parent `x` is not an ancestor of `z`.

Checking this requirement is straightforward. Whenever the user adds a custom edge, we simply find the parent of the edge's source and ensure that it is also an ancestor of the target.

III. Propagating Exceptions

The interval model as described so far assumes that all intervals always terminate successfully. Real programs, of course, often encouter errors or unexpected conditions that cause them to fail. In most modern languages, such conditions lead to an exception being thrown, which skips all subsequent computations up until the point where the exception is caught.

In a multi-threaded world, exception semantics are somewhat murky. So long as an exception is always caught within the thread, the behavior is clear. Otherwise, however, there is no obvious candidate to receive the exception once the thread's own stack frames have all been popped. This is because the dependencies between different threads are never made explicit, so it is difficult to know what other threads may be affected by the error. In the intervals model, however, we can make use of the *happens before* relation to identify and skip dependent computation.

[2]Another alternative would be a fine-grained locking scheme.

```
1  class Interval {
2      final Point start;
3      final Point end;
4      final Interval parent;
5
6      abstract void run();
7
8      Set<Throwable> catchErrors(
9          Set<Throwable> errors)
10     {
11         return errors;
12     }
13 }
```

Figure 5. The class `Interval` extended with a method `catchErrors()` that allows it to intercept uncaught exceptions occurring in its `run()` method or in its subintervals.

The fundamental rule for error handling in intervals is that if a point p_t throws an exception which is caught at a point p_c, then the points p that *happen after* p_t but *before* p_c will be considered *skipped*. Any interval whose start point has been skipped does not execute its `run()` method and does not acquire its associated locks.

To define the point where an exception is caught, we extend the `Interval` class with a new method, `catchErrors()`, that allows an interval to handle errors originating in its `run()` method or children. The method's definition is shown in Figure 5.

The argument to the `catchErrors()` method is a set of unhandled exceptions which have occurred in this interval or its descendants; a set is required because it is possible for multiple descendants to have failed with different exceptions. The method is expected to handle some subset (or possibly all) of these errors. When it has finished, it should return those exceptions that it could not handle. The default implementation handles no errors, and so it simply returns the set of errors as is.

The method `catchErrors()` is invoked during a third phase of interval execution, called `catch`. The `catch` phase follows the existing `run` and `par` phases. Figure 6(a) depicts the ordering of the three phases. It shows the `catch` phase in a grey color, indicating that if no errors occur in the `run` or `par` phases, then the catch phase is simply skipped, as there are no errors to be caught. As is implied by the ordering of the phases, it is illegal for an interval to create new subintervals of itself while executing the `catchErrors()` routine.

Figure 6(b) depicts the program flow if the `run()` method of an interval throws an uncaught exception. Because all

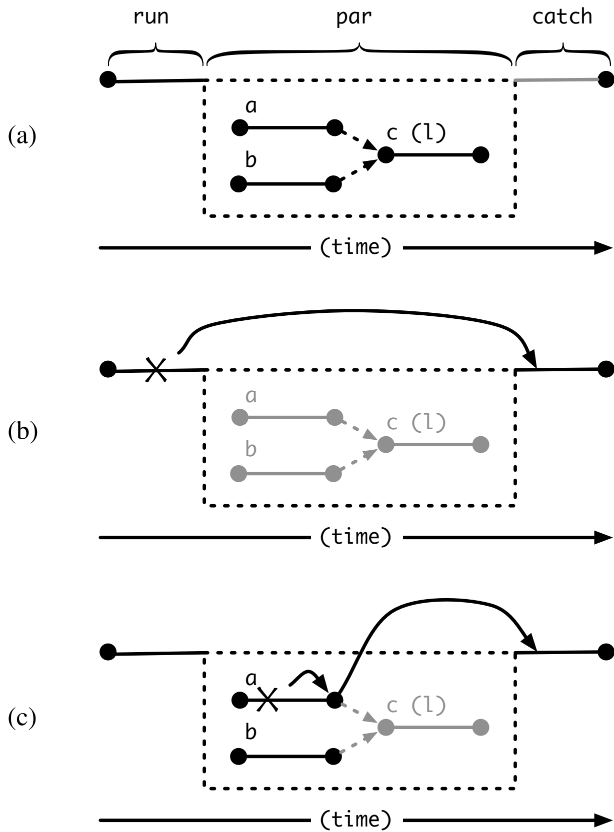

Figure 6. Extensions of the interval diagram from Figure 1 showing the behavior in the face of errors: (a) if no errors occur, the `catch` phase is skipped; (b) in the event of an error in the `run()` method, child computations are skipped; (c) in the event of an (unhandled) error in a child interval, dependent computations are skipped, but other children execute normally.

children of an interval always *happen after* its run method, they are all skipped, and flow proceeds directly to the `catch` phase. The `catchErrors()` method is invoked with a singleton set consisting of the exception thrown by the `run()` method. If it handles the error, then the interval is considered to have terminated normally. Otherwise, the interval has failed, and the error will propagate up to its parent.

Figure 6(c) depicts this final case, where some subinterval (in this case, interval `a`) has failed. The failure of an interval causes any other siblings which *happen after* the failed interval to be skipped (in this case, interval `c`). However, unrelated siblings may continue to execute normally (the interval `b`, for example, is unaffected). Once all siblings have either terminated or been skipped, the `catch` phase of the parent executes. It is provided all the errors which occurred amongst its children and given a chance to handle them.

Whenever the `catchErrors()` method returns unhandled errors, they are propagated upwards in the interval tree. This process terminates when an unhandled error reaches the root interval: in that case, the program is simply aborted (less drastic measures, such as printing the error to the terminal, would also be possible).

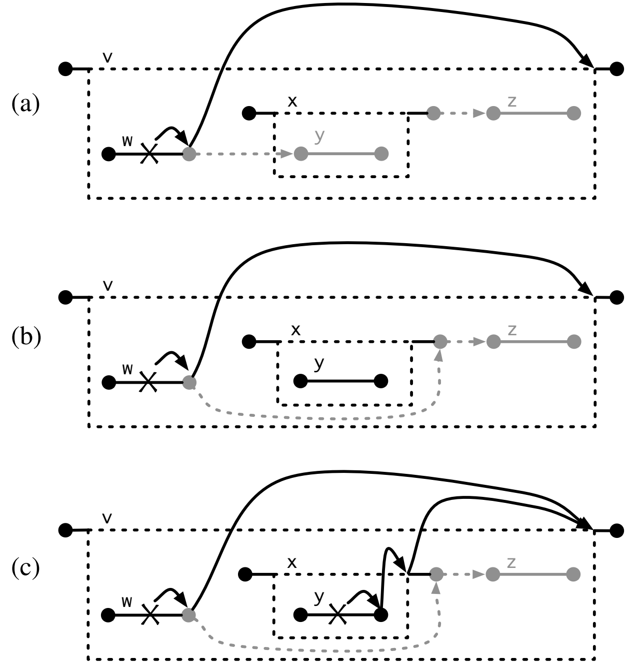

Figure 7. Exception propagation in scenarios involving more complex *happens before* edges that do not fit a rigid hierarchy. (a) An incoming edge to a subinterval can cause successors to the parent interval to fail, although the parent interval itself is not affected. (b) Incoming edges to the end point cause successor intervals to fail but do not abort internal computation. (c) When multiple descendants fail independently, many errors can collect at the mutual parent.

Of course, since the `catchErrors()` method contains user code, it may itself fail. If the `catchErrors()` method throws an exception, we add that exception to the set of uncaught errors which was its input and propagate the entire set to the parent as before.

A. More Complex Examples

In Figure 6, all incoming *happens before* edges targeted the start point of an interval with the same parent. However, it is also possible to have *happens before* edges which target the end point of an interval or which target a nephew node. Figure 7 illustrates three such scenarios.

In case (a), the interval `w` has an edge to its nephew `y`. When `w` fails with an exception, this causes the nephew `y` to be skipped in its entirety. The end point of `x` and the interval `z` are also considered skipped as they fall along the path from the exception being thrown to being caught. However, the *start point* of `x` is unaffected, and so the `run()` method of `x` still executes.

In case (b), the interval `w` which fails is not connected to a subinterval of `x` but rather directly to `x.end`. In this case,

the interval y executes as normal but z is still considered skipped.

Case (c) is the same as case (b) except that the interval y independently throws an exception. In this case, the `catchErrors()` methods of y and x both have an opportunity to catch the error but, assuming they do not, it will propagate to v along with the error from w.

B. Sequential vs Interval Exceptions

Error handling in the intervals library is analogous in many ways to the traditional sequential approach. The interval hierarchy acts similarly to the stack, allowing each parent interval to catch the errors from its descendants, just as a caller method can catch errors that occur in its callees. There are however some important differences.

First, sequential code only allows a single error at a time, whereas in a parallel setting it is possible for some subtasks to fail while others succeed. The `catchErrors()` method of an interval must determine whether the subtasks which failed were critical to the interval's task — in which case their errors should be propagated further — or optional — in which case the errors can be handled locally.

Second, in a traditional try/catch block, simply entering the `catch` region is considered to have handled the error. In our system the `catchErrors()` method must affirmatively return a modified set for the error to be considered handled. This is reflected in the behavior when an exception is thrown in the error handling code. In a try/catch region, the exception thrown within the catch block becomes the active error, and the original exception is forgotten. In our system, the new exception is added to the original set, and so both the original and new exception are passed upward.

The fact that `catchErrors()` can choose not to handle an error also allows it to play a similar role to a `finally` block. If an interval wishes to take some cleanup action when an error occurs, but not to handle the exception itself, it can simply override `catchErrors()` to perform the cleanup action but return the set of errors unchanged (unlike a `finally` block, however, `catchErrors()` does not execute if no error occurs).

C. Language Integration

The `interval` keyword discussed earlier created an anonymous interval class with a single run method. Figure 8 shows how we extend this syntax to support an interval with a `catchErrors()` method.

```
1 void method(Interval parent) {
2     interval (parent) {
3         // Run method
4     } catch (Set<Throwable> errors) {
5         // Optional catch block
6         return ...;
7     };
8 }
```

Figure 8. Extended syntax to support intervals which catch exceptions.

D. Implementation

The state machine shown in Figure 9 is used to achieve the exception semantics. Each interval begins in the WAIT state and eventually terminates in the END state. The current state of the interval also determines whether its start or end point are considered to have occurred. The vertical bars in the diagram indicate the transitions that cause the start and end points to occur.

We assume a background scheduler that tracks the points which have occurred and also tracks which points have been skipped due to uncaught exceptions. The scheduler informs the interval when its start or end point becomes *ready*, meaning that it could occur next without violating the *happens before* relation. A point is considered ready when all of its predecessors have occurred and the parent of its interval has entered the `par` state. The ready state of the start and end points is used as a predicate on transitions in the interval state machine. The interval in turn informs the scheduler when it taken a transition which causes a point to occur.

The WAIT state indicates that the interval is waiting for its predecessors and parent to complete. The wait state is only exited when the start point is ready. If the start point was skipped due to an error in some predecessor, it proceeds to the SKIP state. Otherwise, the interval enters the LOCK state.

The LOCK state is where an interval acquires its locks. This occurs after the start point is ready to occur, but before it has actually occurred. Once all locks have been acquired, execution proceeds to the RUN state. This transition triggers the start point to occur.

The RUN state executes the `run()` method. Assuming that the `run()` method returns normally, execution proceeds to the PAR state. If the `run()` method throws an exception, however, then the interval moves to the CATCH state and the start points of any subintervals are considered skipped.

The PAR state is when child intervals execute. This state terminates once all children and other predecessors of the end point have completed. If all children completed normally, then the end point occurs and the interval moves to the END state. Otherwise, the interval moves to the CATCH state.

The CATCH state invokes the `catchErrors()` method with the set of errors collected from the interval's children. This state is only entered when the end point is ready to occur. If the return value from `catchErrors()` is not the empty set, then the unhandled errors are propogated to the parent and the end node is considered skipped (along with

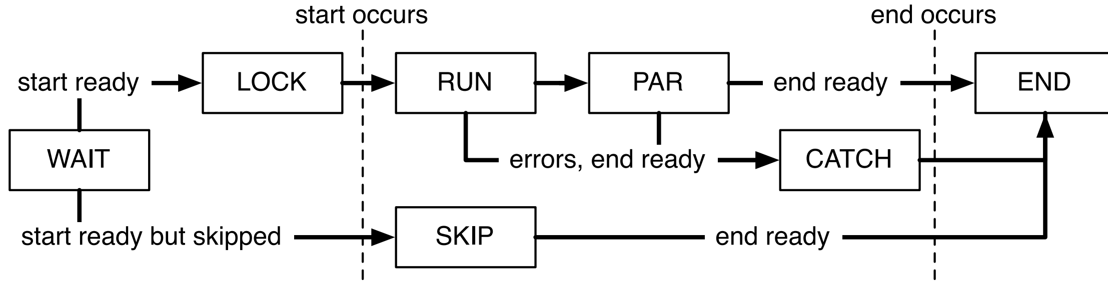

Figure 9. The state machine for the execution of a single interval.

its successors). Exiting this state causes the end point to occur and leads to the END state.

The SKIP state is entered if the start point of the interval was skipped. In that case, the `run()` method and child intervals are not executed, and the interval merely waits for the end point to become ready before proceeding to the END state and causing the end point to occur.

IV. AVAILABILITY

Libraries for Java and C are freely available from our website [3]. Both libraries are still in their early stages and under active development. Also available online are a number of examples such as a parallel version of the nqueens problem or the Game of Life, as well as ported versions of Java Grande Forum benchmarks. Although we have not heavily optimized the intervals library, we have found that threaded programs converted to use intervals perform competitively with their original versions, if not faster. In addition, the overhead of the various safety checks we perform is very small, generally on the order of a few percent.

V. RELATED WORK

A. Parallel Programming Models

Erlang [4] embodies a strict share-nothing philosophy, in which actors with disjoint heaps communicate with messages. The simplicity of this approach is appealing, but we believe there are many scenarios where shared memory is an easier and better choice, given the right tools.

Cilk [5] and OpenMP 3.0 [6] both offer lightweight task frameworks where tasks are executed in a tree structure. Tasks in these languages are not first-class objects, however, and they do not support arbitrary dependency graphs. Java's Fork-Join Framework [7] and Intel's Threading Building Blocks (TBB) both offer a more flexible alternative, but lack a higher-level interface to task dependencies. The fork-join framework permits lightweight tasks to be joined, and TBB allows tasks to delay starting until an associated counter is decremented to zero.

The parallel extensions for .NET [8] offer a task library with a similar feeling to intervals. In addition to the usual join-based task routines, they also permit tasks to have continuations, which are dependent tasks that execute and are given the result of the previous task to begin (the equivalent of edges from the end of one interval to the start of another). This approach is powerful but does not permit the full range of *happens before* edges supported by intervals.

X10 [9] offers a revised threading model which includes a number of innovative synchronization constructs. Among them are phasers [10], a combination of barriers and signals which can guarantee data-race freedom. Intervals can be used to construct the same patterns as phasers with similar guarantees, but also go further by replacing thread joins and other constructs in the X10 toolset.

Intervals in Java align nicely with the Java Memory Model [11]: The *happens before* relation defined by intervals can be seen as a deterministic subset of the full *happens before* relation defined by the memory model, which includes edges due to constructs like `volatile` fields or `synchronized` sections.

B. Handling Uncaught Errors

Erlang's error handling [4] permits multiple processes to be associated with one another so that if one fails, all will fail. This served as an inspiration for the intervals technique of propagating errors forward.

Failboxes [12] are an abstraction that let users group dependent operations so that if one operation fails anything grouped with it will also fail. Failboxes are not a multi-threading mechanism in themselves, but they do help to make error handling with threads more robust by detecting threads which are blocked waiting for a signal from a failed thread. This aspect is very similar to the propagation of errors along *happens before* edges in our design.

Java threads have traditionally ignored uncaught exceptions unless users installed custom error handling routines. Java's `Future` class along with the upcoming Fork-Join Framework [7] take the approach of storing uncaught ex-

ceptions and rethrowning them when a task is joined. The effect is similar to our error propagation, but achieved in a lower-level fashion. Also, if a task is never joined, an exception may go unobserved.

The DBLFuture framework [13] delivers exceptions at the same point where they would have been thrown in a serial implementation. They do not, however, unroll side-effecting computations that occurred in the main thread while the future was being evaluated in parallel.

The parallel extensions for .NET 2.0 [8] introduces the notion that all errors must be observed in some way. Uncaught errors are generally observed when tasks are joined, similar to Java, but there are other options. The user may poll the task or create follow-on tasks which execute on error. If a task is collected by the garbage collector without its exception having been observed, then the process is killed. As in our approach, uncaught exceptions are aggregated.

JCilk, the Java port of Cilk, addresses exception handling [14] based on Cilk's notion of a tree-based task graph. They take the unusual approach that when a JCilk computation aborts, it also aborts its siblings asynchronously, even if they are not dependent on it. These semantics fit well with branch-and-bound problems but may not be suitable in other circumstances. Asynchronous aborts can also lead to very tricky reasoning when shared memory is involved. For these reasons, we chose not to implement similar semantics with intervals. Users can always emulate such a behavior by monitoring a shared flag which is triggered when an error occurs or the goal is found.

VI. Conclusion

We have presented the key points of the intervals model and API. Intervals present a flexible alternative to threads that can be used to construct virtually any parallel schedule, without the risk of deadlock even in the face of unanticipated errors. The use of explicit *happens before* relations allows users to express the dependencies between tasks in a high-level and concise fashion, which also permits for intelligent error propagation so that dependent tasks do not execute when their predecessors have failed. As parallel programming becomes increasingly popular, languages will need to evolve a higher-level means for describing possible parallel schedules than what is currently available. Intervals are a step in this direction.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, 1978.

[2] N. D. Matsakis and T. R. Gross, "Programming with Intervals," in *LCPC*, 2009.

[3] http://intervals.inf.ethz.ch.

[4] http://ftp.sunet.se/pub/lang/erlang/.

[5] K. H. Randall, "Cilk: Efficient multithreaded computing," Ph.D. dissertation, Dept. of EECS, MIT, May 1998.

[6] "OpenMP Specification: Version 3.0," http://openmp.org/, May 2008.

[7] D. Lea, J. Bowbeer, D. Holmes, and S. AG, "JSR 166: Concurrency Utilities," http://jcp.org/en/jsr/detail?id=166, September, 2004.

[8] "Parallel Extensions to the .NET Framework," http://msdn.microsoft.com/en-us/concurrency/default.aspx.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA*. ACM, 2005.

[10] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *ICS*. ACM, 2008.

[11] Manson, Jeremy and Pugh, William and Adve, Sarita V., "The Java memory model," *SIGPLAN Not.*, vol. 40, no. 1, 2005.

[12] B. Jacobs and F. Piessens, "Failboxes: Provably safe exception handling," in *ECOOP*. Springer-Verlag, 2009, pp. 470–494.

[13] L. Zhang, C. Krintz, and P. Nagpurkar, "Supporting exception handling for futures in Java," in *PPPJ '07*. ACM, 2007, pp. 175–184.

[14] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson, "Programming with Exceptions in JCilk," *Science of Computer Programming (SCP)*, vol. 63, no. 2, pp. 147–171, Dec. 2006.