Programming with Intervals

Nicholas D. Matsakis and Thomas R. Gross

ETH Zurich

Abstract. Intervals are a new, higher-level primitive for parallel programming with which programmers directly construct the program schedule. Programs using intervals can be statically analyzed to ensure that they do not deadlock or contain data races. In this paper, we demonstrate the flexibility of intervals by showing how to use them to emulate common parallel control-flow constructs like barriers and signals, as well as higher-level patterns such as bounded-buffer producer-consumer. We have implemented intervals as a publicly available library for Java and Scala.

1 Introduction

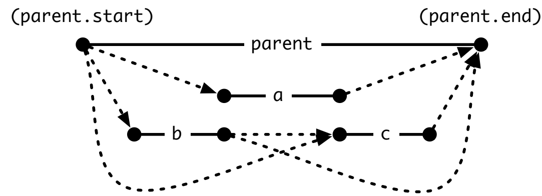
Existing primitives for synchronizing the control-flow of parallel threads, such as barriers and signals, are low-level and dangerous to use. Without careful attention, programs can easily deadlock due to misaligned barriers or missed signals.

Intervals are a higher-level alternative that make parallel programming safer, without sacrificing flexibility. Intervals allow programmers to directly encode the *happens before* relationships [14] between different parallel tasks. A modular static analysis then validates that these relationships cannot lead to a deadlock. Intervals have been implemented as a publicly available library for Java and Scala. The static analysis has been implemented for Java as a `javac` extension.

The aim of this paper is to demonstrate the flexibility of intervals by using them to model common patterns in parallel programs. The paper begins by introducing the basic interval model and API. We then show how to model a number of parallel patterns using intervals. Next, we discuss our implementation and the features it offers. Finally, we compare intervals against a number of other approaches to synchronizing parallel programs.

2 Interval Model and API

Intervals are first-class objects that represent the slice of time in which some parallel task will execute. Programmers create intervals by supplying a task object as well as any dependencies that the task may have on existing intervals. The scheduler is then responsible for executing the tasks in such a way that all dependencies are respected. By properly arranging the dependencies between intervals, traditional synchronization tools such as barriers and signals can be emulated, along with other patterns.




```
1 class Intro {
2     void setup(Interval parent) {
3         Interval a = parent.newInterval().schedule(emptyTask);
4         Interval b = parent.newInterval().schedule(emptyTask);
5         Interval c = parent.newInterval().startAfterEndOf(b).schedule(emptyTask);
6     }
7 }
```

Fig. 1. A sample interval graph and the code to create it

The interval model is easiest to understand when visualized as a graph. Figure 1 depicts a graph with four intervals, `parent`, `a`, `b`, and `c`. Each interval consists of two nodes, its start and end points, separated by a solid edge. We generally label the intervals but omit labels on the individual nodes; in this graph, however, we included labels on the start and end point of the `parent` interval for illustrative purposes. Dependencies between intervals in the graph are depicted with dotted edges; we will see that such edges are directly specified by the programmer. The transitive closure of all edges, dotted and solid, in the graph is the *happens before* relation. For example, in Figure 1, `parent.start` *happens before* `c.end`, as indicated by the path `parent.start` $\rightsquigarrow$ `c.end`.

Figure 1 also contains Java code which, when executed, creates the depicted intervals `a`, `b`, and `c`. The definition of `setup()` illustrates the salient features of the interval API.

Intervals are represented by the interface `Interval`. As intervals are first-class objects, they may be stored in fields or passed as parameters. In `setup()`, the interval `parent` is not created but rather provided by the caller. In this way, `setup()` creates the sub-tasks and their interdependencies, but the caller ultimately decides when they shall execute. This kind of parameterization aids in code reuse.

Creating a new interval begins by invoking `newInterval()` on an existing interval, as seen on line 3. `newInterval()` does not create the new interval immediately, but rather returns a builder object whose methods can be used to specify the dependencies and task of the new interval. The receiver of the call to `newInterval()` will become the parent of the new interval once it is created. The new interval is always contained within its parent, meaning that the parent's start point *happens before* the new start, and the new end *happens before* the parent's end.

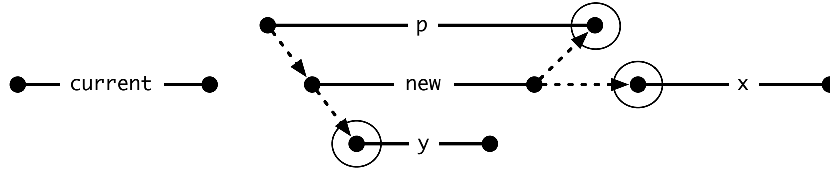


Fig. 2. An example of new interval creation. The interval `new` has just been created along with the various edges adjacent to it. For this creation to be legal, the end of the current interval `current` must *happen before* each of the circled points.

Additional scheduling constraints for the new interval are specified by invoking appropriate methods on the builder object returned by `newInterval()`. Each constraint method always returns the builder object, allowing multiple constraints to be specified in sequence. An example appears on line 5, where the call to `startAfterEndOf(b)` indicates that the start of the new interval (`c`) occurs after the end of `b`. In other words, `b.end` *happens before* `c.start`. There are similar methods for all combinations of start and end points and directionality, such as `endAfterStartOf()` or `startBeforeEndOf()`. Calls to dependency methods with `null` arguments (`startAfterEndOf(null)`, etc) have no effect.

The final step in creating an interval is to invoke `schedule(`t`)`. `schedule()` creates the new interval object, registers it with the scheduler, and returns it to the user. The parameter t is an instance of `Task` (similar to `Runnable`) which defines the behavior of the interval. The `Task` interface defines a single method, `void run(Interval current)`; once all dependencies of the interval's start point have been resolved, t`.run()` will be invoked with the newly created `Interval` object provided as its parameter. In this example, all three intervals simply use the dummy task `emptyTask`, which is provided by the API as a static field and has no effect.

2.1 Errors Constructing the Interval Graph

There are two kinds of errors that can result when creating new intervals. The first concerns attempts by the user to retroactively add incoming edges to a node in the interval graph that has already occurred. For example, the user might attempt to create a new interval whose parent has already closed. To accommodate this request, the scheduler would have to "step back in time" so as to execute the new interval before the parent had completed. Similar errors can occur with any outgoing edge from the new interval. We currently prevent these kind of errors with a dynamic check. If `current` is the interval invoking `newInterval()`, and T is the set of targets of any newly created edges, then we require that `current.end` *happens before* (or equals) each member of T. To help explain what points are in the set T, Figure 2 depicts the relevant intervals to a new interval with dependencies `endBeforeStartOf(x)` and `startBeforeStartOf(y)`.

and circles those points in T. Because the end of the current interval must happen before each of those points, there is no possible schedule in which they will have occurred. It would also be possible to perform these checks statically in many cases, but this is not currently implemented.

The second kind of error that could occur is a cyclic *happens before* relationship, which would be impossible to execute. To prevent this, we make use of a modular static analysis, the full details of which are out of scope for this paper. The analysis approximates the *happens before* relationships created by a method on its local variables and parameters. To keep the analysis modular, some annotations are required. For example, methods must be annotated to indicate the *happens before* relationships that they may create between their formal parameters. All the examples in this paper are checkable using our analysis if appropriately annotated.

2.2 Integrating Intervals and Locks

In addition to *happens before* relationships, invocations of `newInterval()` may specify a number of locks that should be held for the duration of the new interval. Locks are represented in our API as instances of the class `Lock`. Creating a new interval which executes under lock is done as follows:

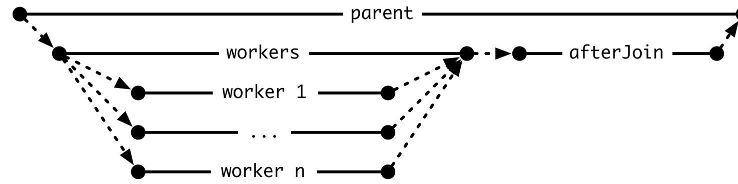
```
Lock lock;
Interval locked =
    parent.newInterval().exclusiveLock(lock).schedule(...);
```

Locks are acquired before the interval starts, but after any *happens before* relationships are resolved. In this case, the lock `lock` would be automatically acquired before the interval `locked` begins but after `parent` has begun (because `parent.start` *happens before* `locked.start`), and released after it finishes. We do not yet protect against deadlocks that result from acquiring locks.

3 Examples

In this section we describe several parallel programming patterns and show how they can be modeled with intervals. The examples are given in Java. For brevity, we omit constructors, and assume that they simply assign their parameters to the fields of the class. We sometimes compress whitespace by collecting closing braces at the end of a line or together onto a single line. Finally, we make use of the notation introduced by the Java Closures [7] proposal to define `Task` objects inline. Accordingly, an expression like { `Interval i =>` *stmt* } indicates a `Task` instance whose `run()` method consists of *stmt* and takes a single parameter `i` of type `Interval`.

3.1 Fork-Join




```
1  class ForkJoin {
2      void method(Interval parent) {
3          Interval workers = parent.newInterval()
4              .schedule({ Interval workers =>
5                  for(int i = 0; i < N; i++)
6                      workers.newInterval().schedule(new WorkerTask(i));
7              });
8          Interval afterJoin = parent.newInterval()
9              .startAfterEndOf(workers)
10             .schedule(new AfterJoinTask());
11     }
12 }
```

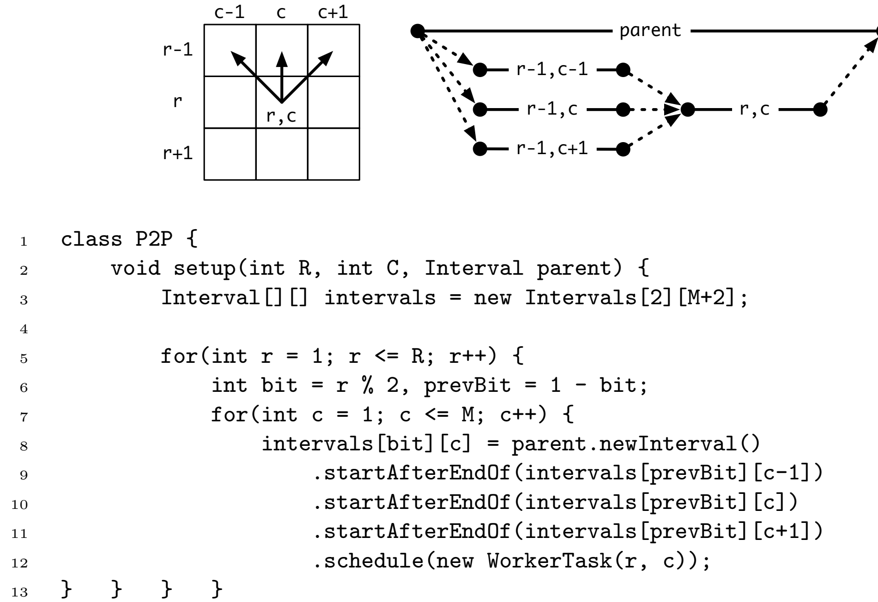
Fig. 3. Interval graph and code for the Fork-Join pattern. The notation `{ Interval workers => ... }` defines an anonymous task instance as explained in Section 3.

Fork-Join is one of the basic patterns for parallel programming. It consists of a master task which spawns a number of worker tasks. The master then waits for these workers to complete before proceeding. The desired interval pattern and Java code to create it are contained in Figure 3. The code creates `N` worker intervals, each with a common parent interval named `workers`. The interval `afterJoin` waits until `workers` has completed before starting, and therefore executes only after all workers have finished.

The `workers` interval's task creates its own children. Because the interval `workers` cannot close until its own task has completes, this structure ensures that `workers` remains open while its children are constructed, thus avoiding the dynamic error described in Section 2.1.

This example, like the others in this paper, assumes an event-based style of programming. This means that we assume the code has been refactored so that worker tasks can be neatly factored into `Task` instances. Event-based programs can appear quite different from their threaded equivalents. For example, a fork-join section in Cilk [20] is written:

```
/* stmts₁ */
sync;
/* stmts₂ */
```




```
class P2P {
    void setup(int R, int C, Interval parent) {
        Interval[][] intervals = new Intervals[2][M+2];

        for(int r = 1; r <= R; r++) {
            int bit = r % 2, prevBit = 1 - bit;
            for(int c = 1; c <= M; c++) {
                intervals[bit][c] = parent.newInterval()
                    .startAfterEndOf(intervals[prevBit][c-1])
                    .startAfterEndOf(intervals[prevBit][c])
                    .startAfterEndOf(intervals[prevBit][c+1])
                    .schedule(new WorkerTask(r, c));
}   }   }   }
```

Fig. 4. Interval graph and code demonstrating Point-to-Point synchronization

The `sync` statement in this case causes all threads forked in $\texttt{stmts}_1$ to be joined. This same example transformed into an event-based style might look like:

```
/* stmts1 */
sync({ Interval i => /* stmts2 */ });
```

Now the statements which follow the synchronize have been moved into a `Task` object, which is given as a parameter to `sync`. Once the threads been synchronized, this `Task` object will be executed with a new interval as the parameter. As a rule of thumb, a new interval (and therefore a `Task` object) is required any time that a blocking operation is performed.

The need to translate code into an event-based style of programming is purely an artifact of implementing intervals using a Java library. If intervals were integrated into the runtime, or if Java were extended to support continuations, this transformation could be done implicitly during compilation. In the meantime, another option would be to use a bytecode rewriter such as Kilim [24]. In fact, our implementation includes a convenience method that permits threaded-style fork-joins. However, we chose to use an exclusively event-based presentation for consistency with the other examples.

3.2 Point-to-Point

Point-to-point synchronization patterns involve fine-grained dependencies between different threads. As an example, consider an algorithm that operates over

a matrix, where processing the cell at coordinates `(r,c)` requires the results from the neighboring cells in the previous row (if any).[1] The resulting dependency pattern is shown graphically in Figure 4, along with a corresponding interval graph. We only show the intervals for the cell `(r,c)` and its dependencies, but similar intervals would exist for each cell in the matrix.

The intervals for each matrix cell are created in the routine `setup()`. It uses an array, `intervals`, to store the interval objects for each cell in the matrix. The array has size `[2][M+2]`, where `M` is the number of columns in the matrix. Only two rows are needed because each cell only depends on the results of the previous row. The extra two columns (`0` and `M+1`) are included to handle the corner case for the left- and right-most cells in the matrix. Because they are never stored into, they are always `null`: for a cell in column 1, then, the interval at position `c-1` (i.e., `0`) is always `null`. As `startAfterEndOf(null)` has no effect, the correct dependencies are created.

Two nested `for` loops iterate over all the cells in the matrix and create appropriate intervals, storing them into the array. The iterations for even-numbered rows write to the `0`th row of the array and read from the `1`st. Odd-numbered rows do the opposite. In the case of the first row, the array contains only `null` pointers, and so no dependencies are created.

As an optimization, we are considering providing a special method for instantiating a regular matrix of dependencies. This method would replace the doubly nested loops in Figure 4. In addition to avoiding the need to instantiate an object for each matrix cell, this would permit us to use an optimized scheduler like the one found in Pluto [6].

3.3 Hand-Over-Hand Locking

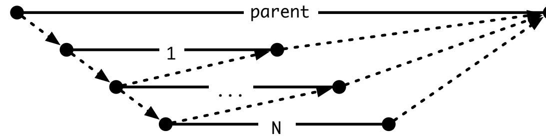
Hand-over-hand locking is a common pattern for parallelizing algorithms that need to walk down a linked list in parallel. The key idea is to acquire the lock on the next link of the list before releasing the lock on the current link. This ensures that no parallel thread can modify the `next` pointer until processing on the next link has already begun.

The interval graph in Figure 5 contains one interval per link in the linked list. Each interval i will execute while holding the lock for link i. There is an edge from the start of each interval to the end of its predecessor: this ensures that the lock for link i is acquired while holding the lock for link $i-1$.

The code in Figure 5 consists of two classes. `Link` simply defines the structure of the linked list. `Walk` defines the tasks that process each link. The static `Walk.begin()` method creates the first interval, given the start `l` of the linked list and a parent interval `p`. It simply creates a new `Walk` instance and ensures that it runs using `l`'s lock.

The processing of each link is defined by the method `Walk.run()`. It begins by doing some (omitted) processing (line 14). Next, it loads the `next` pointer (line

[1] This example is a simplification of the successive over-relaxation (SOR) algorithm (in which case the rows represent iterations).



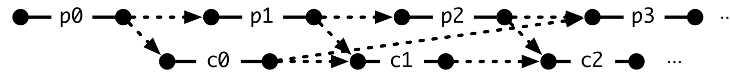

```
1  class Link { Lock lock; Data data; Link next; }
2
3  class Walk implements Task {
4      final Link link;
5      final Interval parent;
6
7      static void begin(Link l, Interval p) {
8          p.newInterval()
9              .exclusiveLock(l.lock)
10             .schedule(new Walk(l, p));
11     }
12
13     public void run(Interval current) {
14         // ... process link ...
15         Link next = link.next;
16         if(next != null)
17             parent.newInterval()
18                 .exclusiveLock(next.lock)
19                 .startBeforeEndOf(current)
20                 .schedule(new Walk(next, parent));
21 }   }
```

Fig. 5. Interval graph and code for hand-over-hand locking

15) and, unless it has reached the end of the list, creates the next interval (line 17). Due to the `exclusiveLock()` dependency on line 18, the next interval executes while holding the next link's lock `next.lock`. To ensure that `next.lock` is acquired before the current interval ends, and hence before the lock for the current link is released, a `startBeforeEndOf()` dependency is specified on line 19.

3.4 Bounded-Buffer Producer-Consumer

In general, a producer-consumer pattern describes any instance where data is produced and consumed in parallel. In our example we assume that consumers should consume the data in the same order that it is produced. A bounded-buffer producer-consumer refines the problem so that the production of the data is not permitted to exceed its consumption by more than a fixed amount. Put another way, when using a bounded-buffer, only a fixed number `N` of unconsumed data items are permitted.



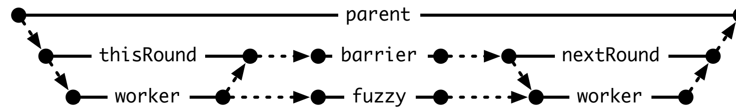

```
1  class BoundedBufferProducerConsumer {
2      public final static N = 3;
3      final Interval parent, consumers[] = new Interval[N];
4  
5      class ProducerTask implements Task {
6          final int i;
7  
8          void run(Interval current) {
9              int nextI = (i + 1) % N;
10             Data data = /* produce data */;
11             Interval prevConsumer = consumers[i % N];
12             Interval waitConsumer = consumers[nextI];
13             consumers[nextI] = parent.newInterval()
14                 .startAfterEndOf(prevConsumer)
15                 .schedule(new ConsumerTask(data));
16             Interval nextProducer = parent.newInterval()
17                 .startAfterEndOf(waitConsumer)
18                 .startAfterEndOf(current)
19                 .schedule(new ProducerTask(i + 1));
20 }   }   }
```

Fig. 6. Interval graph and code for bounded-buffer producer-consumer

Figure 6 gives the interval graph and code to implement a bounded-buffer producer-consumer with intervals. The producer intervals p1...pn follow each other in a linear fashion. Beginning with p3, however, each producer interval also has an incoming edge from the end of the consumer three items back, ensuring that it does not execute until the consumer has finished. Each consumers ci has an incoming edge from both the corresponding producer pi and the previous consumer c$(i-1)$.

The class ProducerTask in Figure 6 implements the producer for the ith data item, where i is a field. The various producers communicate using an array consumers; the size of this array bounds the number of outstanding consumers. Each producer performs three tasks: first, it creates the ith element of data. Second, it starts a consumer for this data. Finally, it starts the next producer, ensuring that it has a dependency on consumer i-N. The next producer also includes a dependency which guarantees that it starts after the the current interval; this is not strictly necessary, as this statement is the last one in the interval.

In Figure 6, the producer creates not only the next producer but also the consumer. This looks a little different from the traditional threaded approach, where the producer and consumer are independent threads. It is possible to




```
1  class FuzzyBarrier {
2      private Interval parent, thisRound, nextRound;
3      private boolean shouldContinue = true;
4
5      public void setup(Interval parent) {
6          this.parent = parent;
7          startRound(parent.newInterval().schedule(emptyTask));
8          for(int id = 0; id < N; id++)
9              thisRound.newInterval().schedule(new WorkerTask(...)); }
10     private void startRound(Interval inter) {
11         if(shouldContinue) {
12             shouldContinue = false;
13             thisRound = inter;
14             Interval barrier = parent.newInterval()
15                 .startAfterEndOf(thisRound)
16                 .schedule({ Interval _ => startRound(nextRound) });
17             nextRound = parent.newInterval()
18                 .startAfterEndOf(barrier)
19                 .schedule(emptyTask); } }
20     void barrier(Task fuzzyTask, Task afterBarrierTask) {
21         shouldContinue = true;
22         Interval fuzzy = parent.newInterval().schedule(fuzzyTask);
23         nextRound.newInterval()
24             .startAfterEndOf(fuzzy)
25             .schedule(afterBarrierTask); } }
```

Fig. 7. Interval graph and code to implement fuzzy barriers

create a more "agent-like" version using intervals. In that case, the next consumer would be created by the previous consumer, rather than the producer. However, we prefer the current technique, particularly in a bounded-buffer context, where the producer and consumer are dependent on one another in any case. In general, using intervals leads away from thinking about agents and instead encourages the programmer to focus on what code executes and when.

3.5 Barriers and Fuzzy Barriers

Barriers are a generalization of the Fork-Join pattern that allows threads to synchronize during their execution and not only in the end. When threads synchronize on a barrier, they wait until all threads in the group have arrived before continuing. A fuzzy barrier [11] is an extension of barriers that permits

threads to specify some final amount of work that they can do while waiting for others to synchronize. Fuzzy barriers can be more efficient because they are able to make use of time that would otherwise have been spent waiting.

Figure 7 contains the code to implement fuzzy barriers as well as an interval graph demonstrating the various relationships. The class `WorkerTask` which implements the workers is not included: the only constraint is that when it needs to synchronize, it should invoke the method `barrier()` with two tasks as arguments. The first, `fuzzyTask`, is permitted to start executing immediately. This represents remaining work that is safe to perform before the barrier is reached. The second, `afterBarrierTask`, is only executed once both the fuzzy task and the next round have begun. This represents code that must be executed after the barrier. We assume workers do not perform additional work after invoking `barrier()` and so the edge from end of the worker interval to the `fuzzyTask` does not need to be added explicitly.

The number of workers participating in each round of the barrier is not predefined or fixed. In the `setup` routine, `N` initial workers are instantiated. Thereafter, invocations of `barrier()` cause work to be scheduled in the next round: a worker could invoke the method multiple times to create multiple workers in the following round, or simply not invoke it at all if there is no more work to be done. Assuming processing is not meant to continue indefinitely, the final round will complete without any invocations of `barrier()`. At this point, no next round is created, and so processing stops. The flag `shouldContinue` is used to indicate when at least one worker has invoked `barrier()`. If it is false, then the barrier task will simply not schedule a next round.

The key method for performing the barrier is `startRound()`, which gets invoked once all workers in a given round have completed and also before the first round has started. Assuming the `shouldContinue` flag is set, `startRound(`i`)` sets i as the current round and creates both a barrier interval and a `nextRound` interval, where the former precedes the latter. The barrier does not execute until the current round has finished.

Because there are no locks used to access `shouldContinue`, one might expect that it should be declared `volatile`. No `volatile` flag is necessary here, however, because the writes to `shouldContinue` occur in the worker intervals, which *happen before* the barrier interval where the read takes place.

4 Implementation and Experience

A prototype implementation of intervals for Java and Scala is available from our website [1]. Both make use of a work-stealing scheduler similar to those found in Cilk [20] or Java 7 [15] but extended to support locks and *happens before* edges.

Intervals were originally developed to facilitate static analysis. Our website also offers a `javac` extension that can be used to statically check the safety of interval programs. It protects against cyclic *happens before* relationships as well as data races. Currently, the safety check supports a large subset of Java, but excludes certain advanced features such as nested classes.

To validate the design of intervals, we are in the process of porting a number of threaded benchmarks to use intervals, beginning with the Java Grande Forum's multithreaded benchmarks. The benchmarks we have ported so far make use of the many of the patterns presented in this paper, including point-to-point synchronization (SOR), barriers (MolDyn), and fork-join sections (Crypt, LUFact, Series, MonteCarlo, and RayTracer). We have found that using intervals results in fewer lines of code when compared against the original threaded versions. Performance generally improves as well due to the improved load balancing offered by a work-stealing execution engine.

4.1 Features and Extensions

The implementation includes a number of extensions to the basic intervals model presented in Section 2 which help to make using the library more convenient.

We extend the locking model to include recursive and read-write locks. Recursive locking is based on the the parent-child relationship: an interval may recursively acquire the same lock as its parent (or any ancestor). Locks may still only be acquired by one task at a time, so the child interval cannot start until its parent's task (but not the parent interval) has finished. When creating a new interval, the programmer can specify whether each of the interval's locks should be held in *shared* or *exclusive* mode. The lock may either be held by one interval with an exclusive lock or multiple intervals with shared locks.

To avoid the need to rewrite all programs in an event-based style, we support synchronous intervals. Creating a synchronous interval causes the current interval to block until the synchronous interval has completed. This makes it possible to write lexically scoped fork-join sections like those in X10 [8] or Cilk [20].

Intervals are also extended to carry a result value, similar to a future [12]. This result is defined to be the return value from the interval's task (for intervals that do not compute a value, the result is simply `null`). The result of an interval `i` may be accessed by invoking `i.result()`. To prevent race conditions, the result of an interval `i` is only accessible by an interval `j` if `i.end` *happens before* `j.start`. If this is not the case, a dynamic error results when `i.result()` is invoked.

4.2 Exceptions

One of the open design questions in the implementation is the proper treatment for exceptions. Currently, if the task of an interval `i` results in an exception, this exception is stored in place of the interval result. This exception is re-thrown when `i.result()` is called or when some ancestor blocks on `i`. This system has the downside that exceptions may never be rethrown.

In the future, we intend to modify the system so that exceptions propagate from a child interval to its parent if they are not handled. This raises several issues, however. For example, what happens when multiple child intervals raise exceptions? Or, what are the precise conditions when an exception should be considered handled? These issues are not unique to intervals: similar problems

arise whenever attempting to integrate error handling and asynchronous execution [10, 13, 25].

5 Related Work

Coroutines [9] and continuations [21] are two building blocks for manipulating control-flow in a sequential context. Either would make a useful primitive on which to build the intervals library and would provide an alternative to rewriting programs in an event-oriented style.

Futures are annotations for parallel execution which act similarly to a lazy or deferred execution. Expressions annotated as being safe for parallel execution are executed in parallel; when the program reaches a point where the result of the expression is needed, the main thread blocks until the evaluation of the expression has completed. Futures were first implemented for MULTILISP [12] but have since been ported to a number of other languages, including Java [19]. Futures can be seen as a subset of intervals that lack the extended *happens before* relationships. Furthermore, most implementations of futures make no guarantees with respect to deadlock-freedom or other safety properties.

Cilk [20] and JCilk [17] are supersets of C and Java respectively which add support for parallelism, primarily in the form of fork-join or barrier style computations. Cilk pioneered many of the dynamic scheduling and work stealing techniques used in the intervals implementation itself.

Jade [22] uses programmer-provided specifications to dynamically parallelize a program. Shared objects were specially integrated into the type system. Tasks declare those objects that they affect and how; adherence to these declarations is checked dynamically. The ability for intervals to be associated with locks works in a similar fashion, but Jade did not attempt to model *happens before* relationships in its task specifications.

X10 [8] introduced a number of innovative synchronization constructs. The most recent, phasers [23], are a combination of barriers and signals. Threads wishing to synchronize with one another make use of a shared phaser object. Threads indicate how they will use a phaser by placing it into different modes, such as signal-wait-next or wait-only, that grant different capabilities. A combination of static and dynamic safety checks ensures that programs cannot be deadlocked through using a phaser. When synchronizing on a barrier, a special "single" mode allows a small section of code to be executed by a single thread before the waiting threads resume. Intervals can be used to perform the same kinds of synchronizations as phasers and with similar safety guarantees. However, intervals are a standalone mechanism that also replaces threads, thread joins, and integrates locks, all of which are beyond the scope of phasers. On the other hand, phasers are closer to existing threading primitives and therefore can be adopted more easily.

OpenMP [4] and the Message Passing Interface (MPI) [3] are two higher-level alternatives to threads for writing parallel programs. Unlike intervals, they are focussed on SIMD programming, although both can be used more generally.

JSR166 [16] introduced a number of concurrency-related utility classes to Java, including futures, thread pools, read-write locks, and concurrent containers such as maps and queues. Java 7 will likely contain additional classes [15], among them the fork-join framework that intervals itself is built upon. For C#, the Parallel Extensions [2] library promises a similar lightweight task framework. Neither of these frameworks includes any mechanism for declarative or explicit *happens before* relationships; instead, users use traditional joins to wait for tasks to complete.

Apple's Cocoa framework includes a class `NSOperation` [5] that is similar to intervals. Like an `Interval`, each `NSOperation` embodies a particular task, and a user may declaratively specify that one operation cannot execute until another has finished (the equivalent of a `startAfterEndOf()` dependency). Unlike intervals, however, `NSOperation`s do not permit other kinds of dependencies nor are they integrated with locks. This means that they cannot easily be used to describe the patterns in this paper, with the exception of point to point synchronization.

The Java Memory Model [18] defines how parallel threads in Java interact with shared memory. In addition to defining formally what it means for a Java program to be correctly synchronized, it describes the legal behaviors of incorrectly synchronized programs which include data races. In the Java Memory Model, *happens before* relationships potentially result from imperative actions such as acquiring locks, accessing `volatile` fields, or joining a thread. This model can be easily adapted to the explicit *happens before* relationships used by intervals. When using our data race analysis, however, there is no need to define the semantics of incorrectly synchronized programs, because they cannot occur.

6 Conclusion

As concurrent programming becomes more common, the problem of managing parallel control flow will become increasingly acute. Structured programming constructs like `while` and `for` loops have been universally adopted for describing *sequential* control flow, but the primitives available to parallel programmers are generally much lower level.

Intervals aim to close that gap by providing a more declarative means of specifying the *happens before* relationships in a program. Explicit *happens before* relationships help to keep the code for synchronization concise and isolated. They also aid the compiler in checking for safety properties such as freedom from deadlocks and data-races. Moreover, as we have shown in this paper, intervals are expressive and flexible enough to implement the common parallel patterns in use today.

References

1. http://intervals.inf.ethz.ch.

2. Parallel Extensions to the .NET Framework. http://msdn.microsoft.com/en-us/concurrency/default.aspx.
3. *MPI Specification 2.1.* September 2008.
4. OpenMP Specification: Version 3.0. http://openmp.org/, May 2008.
5. Apple Developer Connection. Managing Concurrency with NSOperation. http://developer.apple.com/Cocoa/managingconcurrency.html.
6. M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *PPoPP*. ACM, 2009.
7. G. Bracha, N. Fater, J. Gosling, and P. von der Ahè. Closures for the Java Programming Language (v0.5). http://www.javac.info/closures-v05.html.
8. 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.
9. M. E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, 1963.
10. J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson. Programming with Exceptions in JCilk. *Science of Computer Programming (SCP)*, 63(2):147–171, Dec. 2006.
11. R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. *SIGARCH Comput. Archit. News*, 17(2), 1989.
12. R. H. Halstead, Jr. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4), 1985.
13. Joe Duffy. Concurrency and Exceptions. http://www.bluebytesoftware.com/blog/2009/06/24/ConcurrencyAndExceptions.aspx.
14. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.
15. D. Lea. Concurrency JSR 166 Interest Site. http://gee.cs.oswego.edu/dl/concurrency-interest/, June, 2009.
16. D. Lea, J. Bowbeer, D. Holmes, and S. AG. JSR 166: Concurrency Utilities. http://jcp.org/en/jsr/detail?id=166, September, 2004.
17. I.-T. A. Lee. The JCilk multithreaded language. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Aug. 2005.
18. Manson, Jeremy and Pugh, William and Adve, Sarita V. The Java memory model. *SIGPLAN Not.*, 40(1), 2005.
19. A. Navabi, X. Zhang, and S. Jagannathan. Quasi-static scheduling for safe futures. In *PPoPP*. ACM, 2008.
20. K. H. Randall. *Cilk: Efficient Multithreaded Computing.* PhD thesis, Dept. of EECS, MIT, May 1998.
21. J. C. Reynolds. The discoveries of continuations. *Lisp Symb. Comput.*, 6(3-4):233–248, 1993.
22. M. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language.* PhD thesis, Dept. of CS, Stanford University, 1994.
23. 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.
24. S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP*, 2008.
25. L. Zhang, C. Krintz, and P. Nagpurkar. Supporting exception handling for futures in Java. In *PPPJ '07*, pages 175–184. ACM, 2007.