Saint Petersburg State University

Egor Porsev

Bachelor's Thesis

Development of a library for multithreaded and parallel computations based on the model of coroutines and channels

Education level: bachelor

Speciality *02.03.03 «Software and Administration of Information Systems»*

Programme *CB.5006.2019 «Software and Administration of Information Systems»*

Scientific supervisor:
C.Sc, S. I. Salishev

Reviewer:
Key projects engineer D. I. Solomennikov

Saint Petersburg
2023

Contents

Introduction

The performance of software products is one of the key attributes that determine its success. However, the speed at which processors execute instructions is limited by fundamental laws of physics [19]. *Multithreading*, a technology that enables the execution of multiple threads of instructions in *parallel* (simultaneously), is frequently employed to achieve performance improvements in software development.

Multithreading and parallelism are active areas of research and development. Recent works include lock-free data structures [11], performant weak memory models [38], dynamic load balancing algorithms [26], and transactional memory [44]. Programming languages are also evolving to emphasize multithreaded development, with newer languages such as Go [17], Kotlin [24], Swift [5], and Rust [14] placing a strong emphasis on this aspect. Mature languages like Python [7], Java [33], JavaScript [1], C++ [6], and C# [2] are improving their support for multithreading as well.

Despite all the advantages, multithreaded development is inherently complex, requiring developers to manage numerous interactions between simultaneously executing tasks. Classic low-level multithreaded development tools based on threads and locks do not offer reliable protection against programming errors. To alleviate this, modern programming languages offer approaches such as asynchronous functions, MapReduce, actors, coroutines, and channels to simplify multithreaded development. Similarly to how traffic laws provide safety by constraining our movement on the roads, these approaches narrow the class of possible programming errors by constraining the set of options available to the developers.

However, approaches to multithreaded development in modern programming languages do not address one of the principal complexities of multithreaded development: *non-determinism.* A program is said to be non-deterministic when its result is dependent not just on the input data, but also on other factors concerning the state of the execution. This unpredictability leads to challenges in identifying inputs (user actions) that cause errors, complicating testing, debugging, and bug-fixing procedures.

1 Problem Statement

This study aims to develop an approach to multithreaded programming that protects deterministic program logic from non-determinism by enforcing additional constraints and identifying violating code fragments.

The objectives of this study are as follows:

1. Review approaches to multithreaded programming in existing programming languages and compare their constraints.
2. Propose a modified approach to multithreaded programming.
3. Design and implement the new approach as a library for the Java language.
4. Test the implementation through representative test cases.

2 Related Work

To enable the development of multithreaded applications, programming languages provide specialized tools such as libraries and language constructs. Various multithreaded programming languages offer tools that implement similar features and ideas, including threads, coroutines, actors, async/await, MapReduce, and Synchronous Data Flow (SDF). These are examples of *approaches to multithreaded programming*, the next level of abstraction after concrete tools in programming languages. Informally, an approach is an interface with assigned execution semantics.

This chapter compares multithreaded programming approaches based on three criteria. Beginning with generic approaches, which are designed to address arbitrary multithreaded programming problems, it then discusses domain-specific approaches.

The criteria of comparison are as follows:

1. **Data-race-freedom (DRF):** This refers to approaches that protect user-defined state from data races.

2. **Progress guarantees (Progress):** This refers to approaches that guarantee that each task advances towards completing its job, including deadlock freedom.

3. **Determinism guarantees (Det):** This refers to approaches that protect the user program from non-determinism that arises from multithreading and message-passing.

2.1 Generic approaches

An approach is said to be *generic* if it is capable of addressing any multithreaded programming problem. More formally, to be considered generic an approach must correspond to a Turing-complete formal language.

The problem of deadlock detection is undecidable for Turing-complete languages that model concurrent programming [4]. Thus, generic approaches do not provide any progress guarantees.

Moreover, they introduce complexity to otherwise simple tasks. For instance, a program written using async/await (a domain-specific approach) never deadlocks. If we were to reimplement this program using one of the general approaches, say threads, we could by mistake introduce a deadlock. And since the latter approach does not provide progress guarantees we would have no way of deciding whether the code is correct.

2.1.1 Threads and virtual threads

Languages Java [42], C# [41], C++ [48], and Rust [14] provide low-level tools for managing kernel threads accompanied by atomics and locking mechanisms such as semaphores or monitors for memory access synchronization.

Virtual threads are the next level of abstraction. They share the same interface with threads but are scheduled by a runtime library or a virtual machine, instead of the kernel. Virtual threads are supported in Python [43], Java [33], OCaml [29], and Crystal [47] and are utilized with the same synchronization mechanisms, atomics and locks.

Data-race-freedom: Not provided. Users are responsible for the synchronization of shared memory access. Moreover, languages C/C++ explicitly state that the semantics of programs with data races is undefined [3].

Progress guarantees: Not provided. In thread-based approaches, the deadlock problem is well-known [10].

Determinism guarantees: Not provided. As illustrated in Figure 1, data races imply non-determinism. The program's result is undefined.

2.1.2 Coroutines and channels

Coroutines are supported in Kotlin [25], in Go [17], and in C++ [6]. They are lightweight suspendable tasks. Channels allow coroutines to communicate by message-passing.

This approach does not constrain the user. It allows the creation of arbitrary computational graphs where coroutines are the nodes and channels are the edges [4].

```
int counter = 0;

void increment() {
    for (int i = 0; i < 100000; ++i)
        ++counter;
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);

    t1.join(); t2.join();

    std::cout << ''Counter: '' << counter << std::endl;
}
```

Figure 1: Non-determinism of multithreading in C++

Data-race-freedom: Provided. As long as coroutines do not share any user-defined state, it is free of data races. For other scenarios, the said languages provide atomics and locking mechanisms.

Progress guarantees: Not provided. Figure 2a illustrates a program that deadlocks.

Determinism guarantees: Non provided. The output of the program shown in Figure 2b depends on the coroutines execution schedule.

```
func main() {
    ch := make(chan int)
    ch <- 100
    <-ch
}
```

(a) Deadlock

```
func foo(ch chan int, value int) {
    ch <- value
}
func main() {
    ch := make(chan int)
    for i := 1; i < 100; i++ {
        go foo(ch, i)
    }
    for i := 1; i < 100; i++ {
        println(<-ch)
    }
}
```

(b) Non-determinism

Figure 2: Coroutines in Go

2.1.3 Actors, processes, and workers

Swift [5], JavaScript [46], Erlang [13] and Elixir [12] provide approaches to multithreaded programming based on the Actor model [21]. In the Actor model, programs are represented as independent agents (actors), that communicate with each other by sending messages. In response to a message, an actor can update its state, send a finite number of messages to other actors, and create a finite set of actors.

However, in the said programming languages actors are given different names. In JavaScript they are called workers, in Erlang and Elixir they are processes and in Swift they are actors.

The Actor model is Turing-complete [20]. Similar to coroutines and channels, it is possible to define arbitrary computational graphs.

Data-race-freedom: Provided. Assuming actors communicate only via messages, no data races can occur. Memory isolation is enforced in Erlang and Elixir.

Progress guarantees: Not provided. Turing completeness implies the possibility of deadlocks.

Determinism guarantees: Not provided. For the reasons provided above, the Actor model allows non-deterministic computations.

2.2 Domain-specific approaches

An approach is said to be *domain-specific* if it is not generic. In other words, it constrains the user in their abilities and, therefore, cannot be employed to solve an arbitrary problem of multithreaded programming.

2.2.1 Async/await — asynchronous functions

Programming languages such as JavaScript [1], Python [7], Rust [23], Swift [5], C# [2], and Java [16] provide specialized language constructs for asynchronous programming. As illustrated in Figure 3a, users can create asynchronous functions with the `async` keyword and call them with the `await` statement.

The primary constraint of this approach is the structure of the data flow graphs. Data flow graphs where vertices correspond to asynchronous tasks are acyclic since directed edges point from tasks to their callers: async functions ‘send’ messages by returning values. In other words, two concurrent tasks cannot communicate with each other. It is important to note, that these statements hold only if all `Promises` [35], `Futures` [39], or `Tasks` [40] are created automatically by asynchronous calls, rather than manually.

Data-race-freedom: Provided. JavaScript, Python, and Rust execute asynchronous functions in a single thread using an event loop [31, 18]. Execution is not preempted and can be suspended only at `await` statements, eliminating data races. However, in Swift and C# asynchronous functions can be executed in parallel, creating a risk of data races when shared memory is used.

Progress guarantees: Provided. A valid execution schedule can be obtained by topologically sorting the data flow graph.

Determinism guarantees: Not provided. Functions `Promise.any()` [35] and `futures::select` [28] in JavaScript and Rust respectively allow executing multiple asynchronous tasks concurrently and waiting for the task that finishes first. The result of said operations depends on the task execution schedule.

```
async function fetch(url) {
    // fetches the resource
}
let resource = await fetch(url)
```

(a) Simple usage

```
await Promise.any([
    fetch(url1),
    fetch(url2)
])
```

(b) Non-deterministic `Promise.any()`

Figure 3: Async/await in JavaScript

2.2.2 MapReduce

MapReduce [9] is a computational model used for the parallel processing of data streams. The data processing is divided into two main phases: *map* and *reduce*. The former phase individually processes each element of the data stream, and the latter phase aggregates the result. The user is free to define the mapping and aggregation functions. Figures 4a and 4b illustrate the built-in support of MapReduce in Java and C++.

Data-race-freedom: Provided, under the assumption that mapping and aggregation functions are pure.

Progress guarantees: Provided. Since data flow graphs do not contain cycles.

Determinism guarantees: Provided. Even though the order in which elements are processed and aggregated is not defined, the result is deterministic, assuming that the reduction operation is associative. Additionally, some Java `Stream` implementations guarantee the fixed order of elements.

```java
var sum = numbers
    .parallelStream()
    .map(i -> i + 1)
    .reduce(Integer::sum);
```

(a) in Java

```cpp
auto sum = std::transform_reduce(
    std::execution::par,
    numbers.begin(), numbers.end(),
    0,
    std::plus<>(),
    [](int x) { return x + 1; }
);
```

(b) in C++

Figure 4: MapReduce

2.2.3 DAG data processing

In terms of our criteria, directed acyclic graphs (DAG) generalize the async/await approach and the MapReduce model. Approaches based on DAGs are used for batch and streaming data processing and are implemented in various frameworks, including Apache Spark, Hadoop, and Airflow, and in libraries, including TPL [8] and LTN12 [15]. This approach represents applications as DAGs which consist of computational nodes and connections between them.

Data-race-freedom: Provided, unless nodes share state.

Progress guarantees: Provided. Since data flow is acyclic.

Determinism guarantees: Provided, under the assumption that nodes joining multiple flows of data produce results that are independent of the execution schedule.

2.2.4 Synchronous data flow

Synchronous data flow (SDF) [27] is a model used in digital signal processing applications. It is implemented in Verilog [45] and SIGNAL [34] programming languages. SDF programs are directed graphs where each node represents a function and each directed edge represents a FIFO buffer. Nodes read data from input buffers and write results into output buffers. The data flow principle states that any node can fire whenever its input data is available. Nodes do not have any shared mutable state: they communicate only via message-passing.

The programs in SDF must satisfy strict constraints. The number of values that each node consumes from each of its input buffers as well as the number of values that it sends must be known ahead of time. These numbers are fixed for each node, i.e., do not change as execution progresses.

Data-race-freedom: Provided. Since nodes do not share any state.

Progress guarantees: Statically dedicable. Because of the strict constraints, an algorithm exists that statically constructs a schedule for a given SDF program. Thus, this algorithm determines whether a given SDF program achieves progress.

Determinism guarantees: Provided, under the assumption that functions computed by nodes are pure.

2.3 Summary

Table 5 summarizes the comparison presented in this chapter.

To conclude, the following contradiction holds for existing approaches to multithreaded programming:

- Complete approaches lack protection against many types of multithreaded programming errors, including non-determinism.

- Domain-specific approaches offer more guarantees but are associated with different interfaces. Therefore, in the event of changing software requirements, a switch from one approach to another would require the refactoring of the program.

	Approach	Constraints	DRF	Progress	Det
Complete	**Threads, virtual**	None	−	−	−
	Coroutines, shared state	None	−	−	−
	Coroutines, no shared state	No shared memory	+	−	−
	Actors, workers, processes	No shared memory	+	−	−
Domain-specific	**Async/await**[*]	Acyclic data flow	+	+	−
	MapReduce	2 types of operations, Acyclic data flow	+	+	+
	DAG	Acyclic data flow	+	+	+
	SDF	# of read and written messages is fixed	+	Statically decidable	+

Figure 5: Comparison of existing approaches to multithreaded programming

[*] In JavaScript, Python, and Rust asynchronous tasks are executed sequentially, in Swift, C#, and Java parallelism is supported

3 Proposal

Our approach is based on the model of coroutines and channels. In other words, the user creates coroutines and connects them with channels for communication. Figure 6a illustrates a coroutine that generates squares of integers from 0 to 5 and sends them to the output channel. This and all the following examples are oversimplified for clarity. Appendix A presents a more thorough overview of the library API.

```
class Squares extends Coroutine {
    SendChannel<Integer> channel;
    void run() {
        for (int i = 0; i < 5; i++)
            channel.send(i * i);
    }
}
```

(a) Coroutine definition

```
// Graph creation:
var graph = createDeterministic();
var channel = graph.channel();
graph.coroutine(new Squares(channel));
graph.build();
// Graph execution:
for (int i = 0; i < 5; i++)
    println(channel.receive());
// Prints: 0 1 4 9 16
```

(b) Building a graph and printing values

Figure 6: Generating squares of integers 0–5

Next, a coroutine graph must be created. Figure 6b illustrates a simple graph that consists of a single `Squares` coroutine and its output channel. Our approach explicitly distinguishes between graph creation and execution. A graph is created using the `coroutine()` and `channel()` methods which add coroutines and channels to the graph respectively. Graph creation must be finalized by the `build()` call. Then, the graph execution can be started by sending, or receiving from any of its channels. The graph in Figure 6b is executed by repeatedly receiving from its channel.

Deterministic and non-deterministic graphs: Our approach distinguishes between deterministic and non-deterministic graphs. The model of non-deterministic graphs is unconstrained, allowing for the representation of arbitrary computational graphs. On the other hand, the deterministic model enforces additional constraints and does not permit operations that may introduce non-determinisms. These constraints are verified prior to the execution of coroutines.

For instance, let's say we need to process a list of numbers in parallel. Each number must be multiplied by two and the results aggregated into a list. Figure 7a illustrates one possible implementation. The coroutine definitions are omitted. For each element of the `data` list, we create a new `Twice` coroutine, which processes this element and sends the result into the `queue` channel. The `Aggregator` coroutine then reads all values from this channel and sends the resulting list into the `result` channel.

The result of the programs depends on the schedule according to which the `Twice` coroutines were executed, hence the non-determinism. To illustrate this further, assuming that the `data` list constaints only values $[1, 8]$ the result can be either $[2, 16]$ or $[16, 2]$.

To obtain a deterministic result, we may switch the type of the graph by using `createDeterministic`. As illustrated in Figure 7b, the program will report the error at the stage of graph creation before the coroutines are executed. The culprit is the `queue` channel, as it has multiple racing sender coroutines, thus, making the order of messages non-deterministic. The complete list of the deterministic model's constraints is as follows:

- Channels may have at most one sender and one receiver.

```
var data = List.of(1, 8);

void main() {
  var graph = createNonDeterministic()
  var queue = graph.channel()
  var result = graph.channel()
  for (var el : data) {
    graph.coroutine(
      new Twice(el, queue)
    )
  }
  graph.coroutine(
    new Aggregator(queue, result)
  )
  graph.build()



  println(result.receive())
  // Prints either [2, 16] or [16, 2]
}
```

(a) Different results in non-deterministic mode

```
var data = List.of(1, 8);

void main() {
  var graph = createDeterministic()
  // ...
  // ...
  // ...
  // ...
  // No changes
  // ...
  // ...
  // ...
  // ...
  // ...
  graph.build()
  // ^^^^^ Reports an error:
  //   Channel has multiple senders:
  //   Twice and Aggregator
  println(result.receive())
}
```

(b) The error that leads to non-determinism is reported

Figure 7: Parallel processing: multiplying each number of a list by two and aggregating the result into a list

- Merging multiple channels into a single channel is not permitted.
- Deterministic graphs must not include coroutines marked as non-deterministic.

Non-determinism can be eliminated by fixing the order of messages. Figure 8 shows how it can be achieved by creating a separate channel for each `Twice` coroutine, and then joining them. The `queue` channel returned by the deterministic `join()` operation contains fixed order pairs of values from both channels.

Sequential and parallel execution: For some programs single-threaded performance is satisfactory, and they do not benefit from multithreading as it introduces additional complexity. To address this issue, our approach supports both sequential and parallel modes. In the parallel mode, users must cautiously synchronize all access to user-defined shared coroutine state

```
var data = List.of(1, 6);

void main() {
    var graph = createDeterministic()
    var channels = empty list
    var result = graph.channel()
    for (var el : data) {
        var channel = graph.channel()
        channels.add(channel)
        graph.coroutine(new Twice(el, channel))
    }
    var queue = graph.join(channels)
    graph.coroutine(new Aggregator(queue, result))
    graph.build()
    println(result.receive())
    // Prints [2, 7]
}
```

Figure 8: The fixed deterministic program

with message ownership transfer or transactional data structures. On the other hand, in the sequential mode, coroutines are scheduled cooperatively on a single thread. As a result, synchronization in the library and user code is rendered unnecessary, improving the program's simplicity.

4 Design

The public API offers the ability to create coroutines, read and send values into channels and create coroutine graphs. The corresponding functionality is available via the coroutine classes, channel interfaces, and graph interfaces respectively. This is shown in Figure 9.

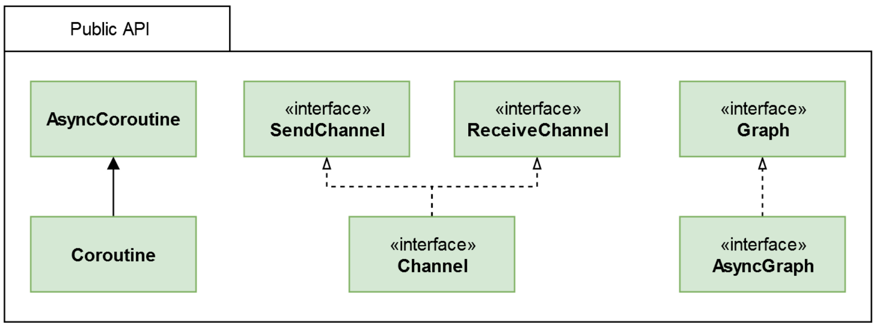


Figure 9: Classes and interfaces in the public API

Non-deterministic (*async*) coroutines are forbidden in deterministic graphs. However, deterministic coroutines can be used in deterministic and non-deterministic (async) graphs. These rules are enforced at compile-time by the Java typing system: `Coroutine` is a subtype of `AsyncCoroutine` and the inverse is true for `Graph` and `AsyncGraph`.

As illustrated in Figure 10, each public interface has multiple implementations which are decided based on one of four user-selected modes. The library provides optimized implementations that consider the constraints of each mode. For instance, non-deterministic channel implementations allow multiple senders and receivers (multichannels), whereas deterministic implementations do not. Also, channels used in sequential modes do not synchronize memory access, eliminating the performance overhead.

Stricter constraints allow more powerful optimizations. Apart from channels, implementations of algebraic operations (*join*, *select*) and schedulers are optimized for each specific mode.

Schedulers are used by channels to run coroutines. The strictest constraints of the sequential deterministic mode (top left) allow the scheduler to use only *O(1)* memory. The sequential non-deterministic scheduler (top right) uses an event loop with a dynamic queue of size *O(n)*. It supports

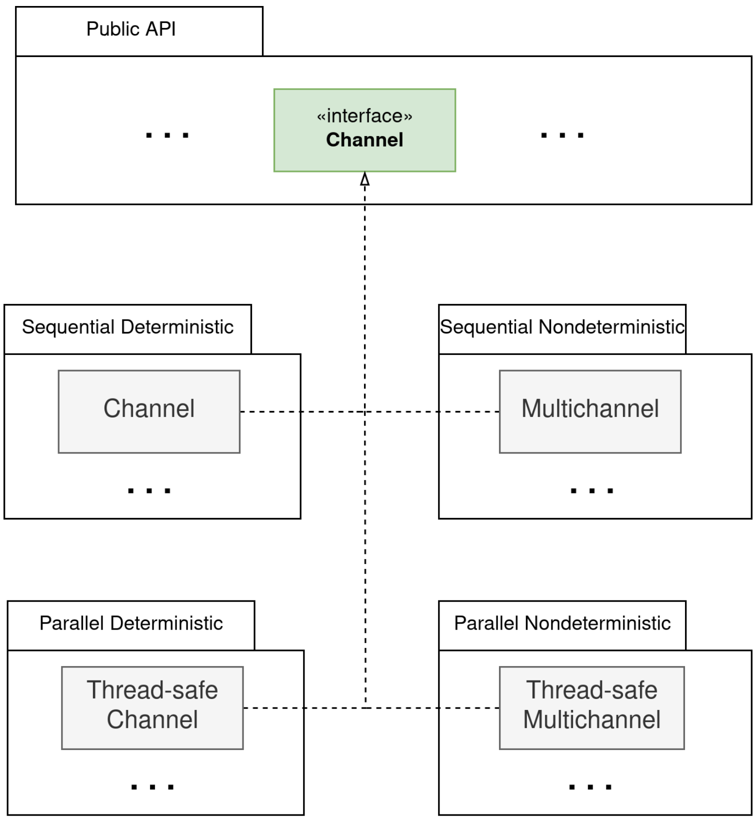


Figure 10: Four different implementations of `Channel`, optimized for each mode

delayed tasks and asynchronous I/O and has no synchronization overhead. The schedulers used in parallel modes (bottom left and bottom right) run coroutines on kernel threads and synchronize memory access.

The concrete implementations of the provided interfaces are not instantiated by the end user, instead, the abstract factory pattern is used. The `Graph` interfaces, shown in Figure 11, can be used to create channels, coroutines, and algebraic channel operations. There are four implementations of the `Graph` interface, one for each mode. When the mode is selected the appropriate `Graph` instance is returned.

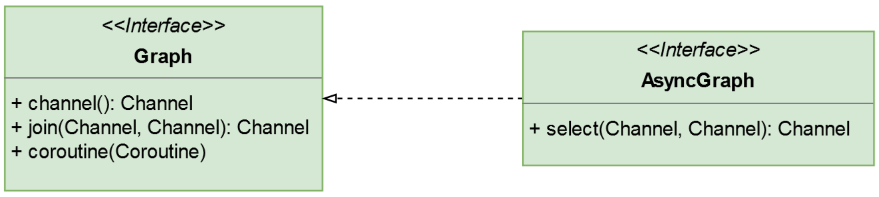


Figure 11: Interfaces for creating graphs, channels, and algebraic operations

5 Implementation

Two separate implementations of the proposed approach have been developed. The former implementation is open source[1], but the latter cannot be disclosed due to the NDA in place. The main difference between them lies in the method of saving the state of coroutines between suspension points. The former uses *stackful* coroutines and the latter uses *stackless* coroutines [30, 36].

This chapter describes the key aspects of the implementation. It begins with stackful and stackless coroutine implementations. The chapter then focuses on the primary implementation details of each mode, starting from the sequential deterministic (most constrained) and ending with the parallel non-deterministic (least constrained).

5.1 Stackful coroutines

Stackful coroutines are implemented with lightweight virtual threads [33] from Java Project Loom. Each coroutine is executed on a separate virtual thread. They are mapped onto a fixed number of kernel threads.

Coroutines, similarly to regular functions, store state in local variables on the stack. Internally, suspending operations use `LockSupport.park()`[2] to suspend coroutines' virtual threads. JVM is responsible for saving and recovering stacks of these threads.

5.2 Stackless coroutines

Stackless coroutines are implemented as objects that store their state in instance fields. They have a single method `run()` that is invoked each time the coroutine is executed, it returns when the coroutine suspends.

Generally, the `run()` method contains a single switch statement surrounded by the `while (true)` loop. When a coroutine performs a suspending channel operation it jumps from one switch branch to another.

[1] https://github.com/Furetur/Concurrency4D

[2] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/LockSupport.html

The latter switch branch starts with the corresponding suspending operation.

Each switch branch may be executed multiple times. The underlying JVM exhibits spurious wakeups. As a result, a coroutine that is waiting for a suspending operation to complete may be woken up when the operation is still pending. Additionally, some suspending operations require multiple attempts, e.g. sending into a full channel. Therefore, it is necessary to keep track of the state of each suspending operation.

For atomicity methods that perform suspending operations also check their state. For instance, the method `trySend(T, Status)` that attempts to send a message into the channel accepts the current `Status` of the operation and returns the updated status. The status object is saved by the coroutine which uses the `Status.isSuccessful()` method to determine whether the operation has been completed. Internally, `Status` is an algebraic data type (synonyms: variant type, tagged union) with values `IDLE`, `SENT`, `CLOSED`, or `PENDING(int)`. When a message is pending this object contains its unique identifier.

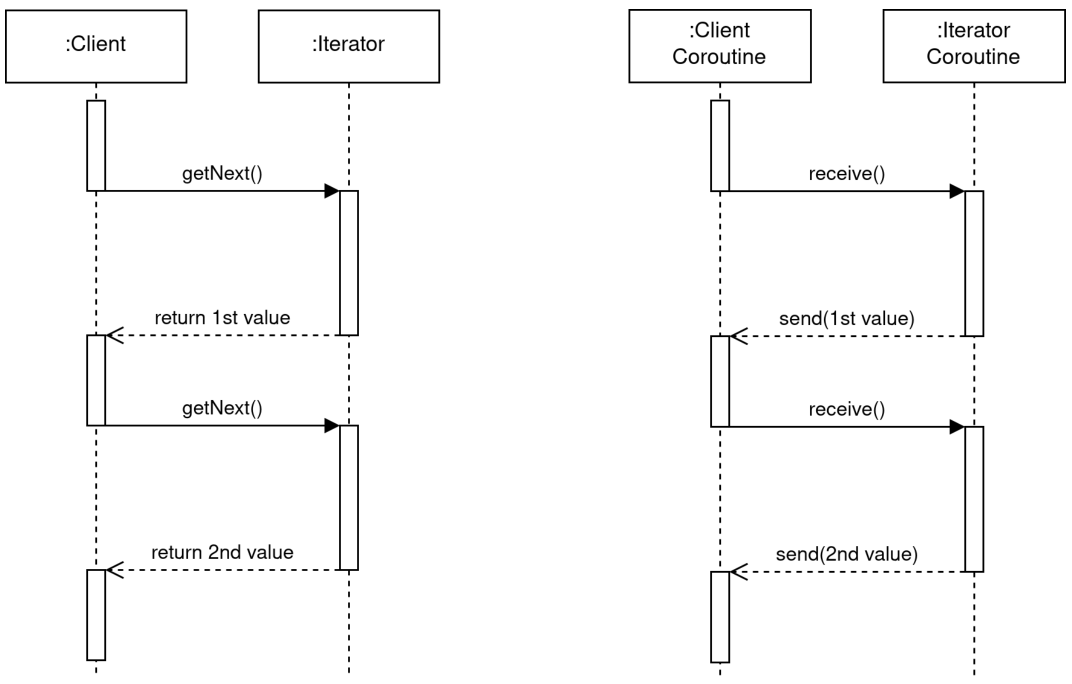
5.3 $\mathcal{O}(1)$ memory usage scheduler

The design of the sequential deterministic scheduler allows coroutines to compete in performance with other constructs like iterators, generators, and finite state machines. The performance evaluation is presented in Section 6.1.2.

The scheduler maintains a single coroutine reference `next`. It is updated when coroutines receive or send messages into channels. In essence, the implementation of the scheduler is: `while (next != null) next.run()`.

The scheduling algorithm can be best understood when a comparison is drawn with Java iterators. When the client code calls the `getNext()` method it transfers control from itself to the iterator. The iterator computes the resulting value and returns it, transferring the control back to the client code. The flow of control is illustrated in Figure 12a.

Suppose the same scenario is implemented with two coroutines, namely



(a) Iterator (b) Coroutines

Figure 12: Control flow

client and iterator. The iterator coroutine sends values into the channel that connects both coroutines and the client receives them. As shown in Figure 12b, the flow of control is identical to the previous one. Initially, a reference to the client coroutine is stored in `next`. The client requests the next value by calling the channel's `receive()` method. The channel updates the `next` reference, transferring the control to the iterator. The coroutine computes the resulting value and sends it into the channel updating `next` and returning the control to the client.

5.4 Event loop

The sequential non-deterministic mode's scheduling is performed by the event loop. It supports delayed tasks, microtasks, and asynchronous I/O. The diagram in Figure 13 shows the phases of the implemented event loop. The phases are repeated while active tasks exist.

The event loop maintains four queues: a priority queue of delayed tasks, a task queue, a microtask queue, and a thread-safe I/O queue. Each I/O operation is run in a separate thread. The results of these operations are

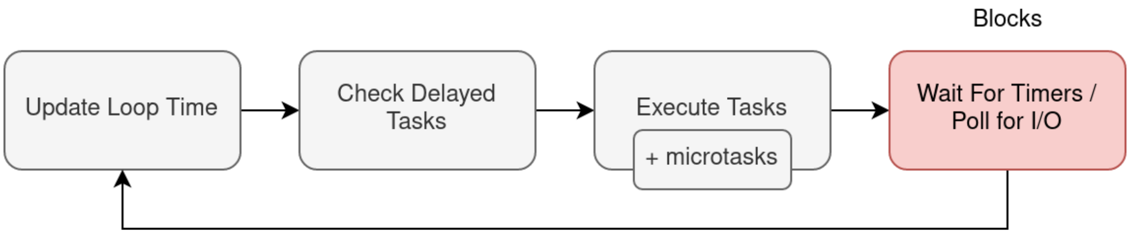


Figure 13: Event loop implementation

added to the I/O queue.

Phases overview:

- **Checks the delayed tasks:** iterates through the delayed tasks queue and adds ready tasks to the task queue.
- **Execute tasks:** runs tasks from the task queue, ignoring newly added tasks.
- **Microtasks:** after each task all microtasks from the microtask queue are executed until the queue is empty.
- **Wait for delay, poll for I/O:** if the task queue is empty the loop waits for new tasks, moving all complete I/O tasks from the I/O queue to the task queue.

5.5 Scheduling coroutines with `park`/`unpark`

In parallel modes, each coroutine is executed on a dedicated virtual thread. Coroutines are suspended and resumed with `LockSupport.park()` and `LockSupport.unpark()` methods which suspend and resume virtual threads.

It is crucial, that `unpark` permits are not lost, otherwise, coroutines may remain waiting forever. Nonetheless, logging and lock acquisition and release may consume `unpark` permits. As a result, it occurred to us as if the permits were lost, even though they were taken by the underlying libraries.

To prevent this scenario, the library implements a thread wrapper illustrated in Figure 14. Apart from a thread reference, it stores a boolean value that represents a permit. This wrapper offers custom `park()` and `unpark()` methods that consume the aforementioned permit. Internally, these methods still use `LockSupport.park()` and `LockSupport.unpark()`.

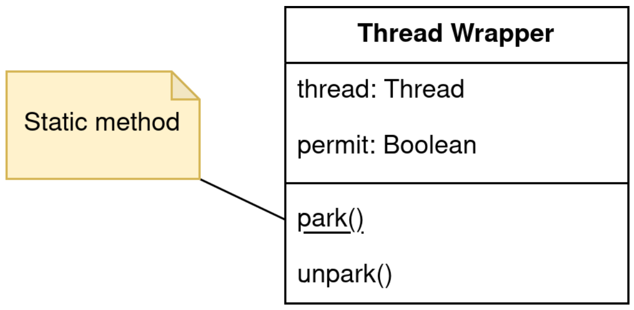


Figure 14: Thread wrapper

5.6 Transactional *Join* implementations

In deterministic modes, channels are permitted to have at most one receiver and one sender, avoiding data races. The sequential *join* implementation performs all suspending `receive()` calls one by one. The parallel implementation schedules all senders before executing `receive()` calls in a similar manner. To prevent deadlocks, channels are queried in a consistent order. This order is based on unique numbers that are assigned to channels upon creation.

The sequential non-deterministic *join* is not thread-safe. However, this implementation must account for multi-receiver scenarios. Firstly, it schedules all senders and uses the `peek()` method (equivalent to `Queue.peek()`[3]) to look into the channels' memory. Then, if all channels contain messages they are received and returned.

In contrast, the parallel non-deterministic *join* is thread-safe. Therefore, it does not use `peek()`, since the composite action of peeking and then receiving is not atomic. It repeatedly tries to complete a transaction of receiving from both channels, using non-suspending `tryReceive()` methods. The transactions are executed optimistically, with no locking beforehand. If at least one channel does not contain a message, the transaction is rolled back by returning all previously read messages to the channels.

[3]https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html#peek--

6 Testing

6.1 Correctness

To ensure that the implementation conforms to the semantics of our approach, extensive end-to-end testing was conducted using representative applications. The entire suite of tests, including modular and integrational tests, covers 97% of instructions and 95% of branches in the library. This chapter highlights four representative applications and describes the method used for calculating the test coverage.

6.1.1 Test cases

The table in Figure 15 illustrates four highlighted test cases and the modes they validate. Two test cases (Two servers and Dining philosophers) are non-deterministic, thus, they are executed only in non-deterministic modes. The remaining two deterministic test cases are run in all modes.

	Deterministic	**Nondeterministic**
Sequential	*k-means*, FSM	*k-means*, FSM Two servers, Dining philosophers
Parallel	*k-means*, FSM	*k-means*, FSM Two servers, Dining philosophers

Figure 15: Highlighted representative test cases

***k-means*:** This is a parallel clustering algorithm. It has been adopted from the Renaissance benchmarking suite [37] for performance evaluation. The results are presented in Section 6.1.2.

FSM: A coroutine implements a finite state machine by reading inputs from an input channel, performing state transitions, and sending outputs into an output channel. In this manner, two state machines are implemented. The output of the first is piped as input into the second. The test generates inputs for the first state machine and validates the outputs of the second.

Two servers: Two coroutines concurrently send requests to two separate servers. Each coroutine forwards the response to the main coroutine.

The main coroutine waits for the response that arrives first and then cancels the second coroutine.

Dining philosophers: This classical problem involves five philosophers who are sitting around a circular table. Each philosopher has a plate of spaghetti and two forks. However, only five forks in total are available. This test implements philosophers as coroutines and forks as messages in channels. The *join* operation is used to atomically acquire both forks avoiding deadlocks. As a result, the code of philosopher coroutines is declarative. As illustrated in Figure 16, each philosopher picks up both forks, eats, releases the forks, and then goes to sleep.

```
while (isHungry()) {
    var bothForks = forks.receive();
    eat();
    forks.send(bothForks);
    sleep();
}
```

Figure 16: Implementation of dining philosophers

6.1.2 Test coverage

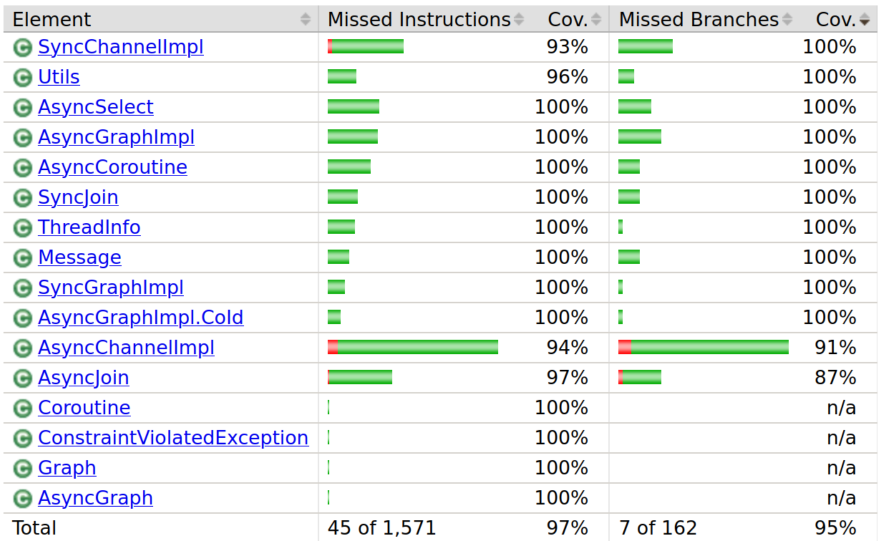
Test coverage was calculated using the JaCoCo[4] library. Figure 17 shows the results of 97% instruction and 95% branch coverage.

3% of instructions and 5% of branches were not covered by tests. Unreachable conditional branches and never-failing runtime assertions (each assertion adds two paths) are also included in the JaCoCo report, negatively affecting the overall metric. Collectively, these two categories account for the largest share of the code not covered by tests.

6.2 Performance

The approach developed in this study offers additional protection against non-determinism. This chapter explores the impact of these guarantees on performance.

[4]https://www.eclemma.org/jacoco/



Element	Missed Instructions	Cov.	Missed Branches	Cov.
SyncChannelImpl		93%		100%
Utils		96%		100%
AsyncSelect		100%		100%
AsyncGraphImpl		100%		100%
AsyncCoroutine		100%		100%
SyncJoin		100%		100%
ThreadInfo		100%		100%
Message		100%		100%
SyncGraphImpl		100%		100%
AsyncGraphImpl.Cold		100%		100%
AsyncChannelImpl		94%		91%
AsyncJoin		97%		87%
Coroutine		100%		n/a
ConstraintViolatedException		100%		n/a
Graph		100%		n/a
AsyncGraph		100%		n/a
Total	45 of 1,571	97%	7 of 162	95%

Figure 17: JaCoCo test coverage reports

The user can select between two modes of coroutine execution, sequential and parallel. To evaluate the performance of both modes, we conducted two benchmark tests. Each benchmark compares the performance of two semantically equivalent programs. One program was implemented using only built-in Java features, while the other program depends on our library. The output of both programs is validated for equivalence.

The first test involves a level-order (BFS) binary-tree iterator implementation and an equivalent coroutine, which traverses the tree and sends each value into its output channel. The second test adopts the *k-means* benchmark from the Renaissance benchmarking suite [37] by reimplementing it using coroutines. The results are illustrated in Figure 18.

Test environment: The JMH [22] benchmarking framework was used with the following JVM arguments: `-Xms28g -Xmx28g -XX:+UseSerialGC -XX:+AlwaysPreTouch` on a 3.20GHz Intel i7-8700 processor running Ubuntu 20.04. Garbage collection was forced between each run and the performance was not measured until the methods were C2-optimized [32]. The JIT compiler's assembly was assessed for constant folding and dead code elimination.

Summary:

- The coroutines implementation did not show any performance overhead in the single-threaded benchmark.

Benchmark	**Input size**	**Impl**	**Score (ms)**	**Sample size**
Sequential BFS Iterator	2^{14}	Java	$15.52 \pm 0.9\%$	7498
		Coroutines	$15.51 \pm 0.8\%$ (**–0.05%**)	7502
	2^{16}	Java	$280.03 \pm 0.3\%$	7411
		Coroutines	$272.35 \pm 0.6\%$ (**–0.7%**)	7430
	2^{18}	Java	$5708.24 \pm 0.1\%$	370
		Coroutines	$5723.48 \pm 0.1\%$ (**+0.2%**)	367
Parallel k-means [37]	25000	Java	$0.040 \pm 2.5\%$	2239
		Coroutines	$0.045 \pm 2\%$ (**+12.5%**)	2517
	50000	Java	$0.090 \pm 1\%$	1263
		Coroutines	$0.101 \pm 1\%$ (**+12.2%**)	1118
	75000	Java	$0.134 \pm 0.7\%$	750
		Coroutines	$0.146 \pm 0.6\%$ (**+8.9%**)	689

Figure 18: Execution time (in ms) of sequential and parallel benchmarks

- The multithreaded benchmark indicated a 12.5% performance overhead of the coroutines implementation, but as the input size increases this relative overhead falls.

Conclusion

The results of this work are as follows:

- A review of 5 existing approaches to multithreaded programming in 9 programming languages including Go, Kotlin, Rust, and Swift, has been conducted.
- A modified approach to multithreaded programming has been proposed.
- To enable developers to utilize the new approach, a Java library has been developed using the Java Project Loom technology.
- Representative test cases were implemented, achieving the test coverage of 97% of instructions and 95% of branches.
- These results were presented at the "Modern Technologies in Theory and Practice of Programming" SPbPU conference.

Two separate implementations of the proposed approach have been developed. The former implementation is open source[5], but the latter cannot be disclosed due to the NDA in place.

[5]https://github.com/Furetur/Concurrency4D

References

[1] Async Functions. — 2015. — URL: https://tc39.es/proposal-async-await/ (visited on: 2023-05-09).

[2] Asynchronous programming. — 2012. — URL: https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-scenarios (visited on: 2023-05-09).

[3] Boehm Hans-J., Adve Sarita V. Foundations of the C++ Concurrency Memory Model // Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI '08. — New York, NY, USA : Association for Computing Machinery, 2008. — P. 68–78. — URL: https://doi.org/10.1145/1375581.1375591.

[4] Cieslak R.A., Variaya P.P. Undecidability results for deterministic communicating sequential processes. — 1990.

[5] Concurrency. — 2021. — URL: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/concurrency/ (visited on: 2023-05-09).

[6] Coroutines (C++20) — cppreference.com. — 2020. — URL: https://en.cppreference.com/w/cpp/language/coroutines (visited on: 2023-05-09).

[7] Coroutines and Tasks. — 2015. — URL: https://docs.python.org/3/library/asyncio-task.html (visited on: 2023-05-09).

[8] Dataflow (Task Parallel Library). — 2023. — URL: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/dataflow-task-parallel-library (visited on: 2023-05-09).

[9] Dean Jeffrey, Ghemawat Sanjay. MapReduce: Simplified Data Processing on Large Clusters // OSDI'04: Sixth Symposium on Operat-

ing System Design and Implementation. — San Francisco, CA, 2004. — P. 137–150.

[10] Demartini Claudio, Iosif Radu, Sisto Riccardo. A deadlock detection tool for concurrent Java programs // Software: Practice and Experience. — 1999. — Vol. 29, no. 7. — P. 577–603.

[11] Efficient Lock-Free Durable Sets / Yoav Zuriel, Michal Friedman, Gali Sheffi et al. // Proc. ACM Program. Lang. — 2019. — oct. — Vol. 3, no. OOPSLA. — 26 p. — URL: https://doi.org/10.1145/3360554.

[12] Elixir Processes. — 2012. — URL: https://elixir-lang.org/getting-started/processes.html (visited on: 2023-05-09).

[13] Erlang Processes. — 2003. — URL: https://www.erlang.org/doc/reference_manual/processes.html (visited on: 2023-05-09).

[14] Fearless Concurrency — The Rust Programming Language. — 2015. — URL: https://doc.rust-lang.org/book/ch16-00-concurrency.html (visited on: 2023-05-09).

[15] Filters Sources And Sinks. — 2023. — URL: http://lua-users.org/wiki/FiltersSourcesAndSinks (visited on: 2023-05-09).

[16] Future (Java Platform SE 8). — 2014. — URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html (visited on: 2023-05-09).

[17] The Go Programming Language Specification. — 2022. — URL: https://go.dev/ref/spec (visited on: 2023-05-09).

[18] HTML Living Standard. — 2022. — URL: https://html.spec.whatwg.org/multipage/webappapis.html (visited on: 2023-05-09).

[19] Hennessy John L., Patterson David A., Asanovic Krste. 1.5 Trends in Power and Energy in Integrated Circuits // Computer Architecture:

A quantitative approach. — 5th edition. — Morgan Kaufmann, 2019. — P. 21–26.

[20] Hewitt Carl. What is Computation? Actor Model versus Turing's Model // A Computable Universe. — P. 159–185.

[21] Hewitt Carl, Bishop Peter, Steiger Richard. A Universal Modular ACTOR Formalism for Artificial Intelligence // Proceedings of the 3rd International Joint Conference on Artificial Intelligence. — IJCAI'73. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1973. — P. 235–245.

[22] Java Microbenchmark Harness (JMH). — URL: https://github.com/openjdk/jmh (visited on: 2023-05-09).

[23] Keyboard async. — 2018. — URL: https://doc.rust-lang.org/std/keyword.async.html (visited on: 2023-05-09).

[24] Kotlin 1.3 Released with Coroutines, Kotlin/Native Beta, and more. — 2018. — URL: https://blog.jetbrains.com/kotlin/2018/10/kotlin-1-3/ (visited on: 2023-05-09).

[25] Kotlin Coroutines: Design and Implementation / Roman Elizarov, Mikhail Belyaev, Marat Akhin, Ilmir Usmanov // Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. — Onward! 2021. — New York, NY, USA : Association for Computing Machinery, 2021. — P. 68–84. — URL: https://doi.org/10.1145/3486607.3486751.

[26] LB4OMP: A Dynamic Load Balancing Library for Multithreaded Applications / Jonas H. Müller Korndörfer, Ahmed Eleliemy, Ali Mohammed, Florina M. Ciorba // IEEE Transactions on Parallel and Distributed Systems. — 2022. — Vol. 33, no. 4. — P. 830–841.

[27] Lee E.A., Messerschmitt D.G. Synchronous data flow // Proceedings of the IEEE. — 1987. — Vol. 75, no. 9. — P. 1235–1245.

[28] Macro futures::select. — 2018. — URL: https://docs.rs/futures/latest/futures/macro.select.html (visited on: 2023-05-09).

[29] Module Thread. — 2022. — URL: https://v2.ocaml.org/api/Thread.html (visited on: 2023-05-09).

[30] Moura Ana Lúcia De, Ierusalimschy Roberto. Revisiting Coroutines // ACM Trans. Program. Lang. Syst. — 2009. — feb. — Vol. 31, no. 2. — 31 p. — URL: https://doi.org/10.1145/1462166.1462167.

[31] The Node.js Event Loop, Timers, and process.nextTick(). — 2022. — URL: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/ (visited on: 2023-05-09).

[32] Oaks Scott. Java performance: In-depth advice for tuning and programming Java 8, 11, and beyond. — O'Reilly Media Inc., 2020.

[33] Pressler Ron, Bateman Alan. JEP 444: Virtual Threads. — 2023. — Mar. — URL: https://openjdk.org/jeps/444 (visited on: 2023-05-09).

[34] Programming real-time applications with SIGNAL / P. LeGuernic, T. Gautier, M. Le Borgne, C. Le Maire // Proceedings of the IEEE. — 1991. — Vol. 79, no. 9. — P. 1321–1336.

[35] Promise. — 2015. — URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise (visited on: 2023-05-09).

[36] Racordon Dimitri. Coroutines with Higher Order Functions. — 2018. — 1812.08278.

[37] Renaissance: benchmarking suite for parallel applications on the JVM / Aleksandar Prokopec, Andrea Rosà, David Leopoldseder et al. // Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. — 2019. — P. 31–47.

[38] Repairing and Mechanising the JavaScript Relaxed Memory Model / Conrad Watt, Christopher Pulte, Anton Podkopaev et al. // Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI 2020. — New York, NY, USA : Association for Computing Machinery, 2020. — P. 346–361. — URL: https://doi.org/10.1145/3385412.3385973.

[39] Rust Future. — 2018. — URL: https://doc.rust-lang.org/std/future/trait.Future.html (visited on: 2023-05-09).

[40] Task. — 2021. — URL: https://developer.apple.com/documentation/swift/task (visited on: 2023-05-09).

[41] Thread Class. — 2000. — URL: https://learn.microsoft.com/en-us/dotnet/api/system.threading.thread (visited on: 2023-05-09).

[42] Thread (Java Platform SE 8). — 2014. — URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html (visited on: 2023-05-09).

[43] Thread-based parallelism. — 2015. — URL: https://docs.python.org/3/library/threading.html (visited on: 2023-05-09).

[44] Transactional Memory: An Overview / Tim Harris, Adrián Cristal, Osman S. Unsal et al. // IEEE Micro. — 2007. — Vol. 27, no. 3. — P. 8–29.

[45] Verilog HDL Reference Manual. — URL: http://www.csit-sun.pub.ro/~cpop/VerilogHDL_Tools/synver.pdf (visited on: 1999-05).

[46] Worker. — 2022. — URL: https://developer.mozilla.org/en-US/docs/Web/API/Worker (visited on: 2023-05-09).

[47] class Fiber. — 2022. — URL: https://crystal-lang.org/api/1.8.2/Fiber.html (visited on: 2023-05-09).

[48] std::thread. — 2011. — URL: https://en.cppreference.com/w/cpp/thread/thread (visited on: 2023-05-09).

A Library API

This chapter presents a brief overview of the library API. It begins with deterministic graphs of coroutines that can be used to represent core program logic. Then, it describes how non-deterministic coroutines can communicate with deterministic subgraphs.

A.1 Deterministic graphs

The basics of this library can be described using a coroutine that calculates a sequence of squares. The code is shown in Listing 19.

```
class Squares extends Coroutine {
    SendChannel<Integer> channel;

    Squares(SendChannel<Integer> channel) {
        super(List.of(), List.of(channel));
        this.channel = channel;
    }

    @Override
    protected void run() {
        for (int i = 0; i < 5; i++) {
            channel.send(i * i);
        }
    }
}
```

Figure 19: A coroutine that calculates squares of integers from 0 to 5

Each coroutine must extend from the `Coroutine` class. A coroutine's constructor must register all its input and output channels by calling `super()` and passing the lists of the channels respectively. The `send()` channel operation suspends the calling coroutine and reuses the current thread for another one.

`SendChannel` is the channel interface that allows sending. The interface for receiving is `ReceiveChannel`.

To use this coroutine, a graph must be created. This is shown in Figure 20.

```
var graph = Graph.create();

var channel = graph.<Integer>channel();
graph.coroutine(new Squares(channel));

graph.build();

for (int i = 0; i < 5; i++) {
    System.out.println(channel.receive());
}

// Prints:
// 0 1 4 9 16
```

Figure 20: Running the `Squares` coroutine and receiving values

The `Graph.create()` method creates a graph builder. In this example, the constructed graph consists of an integer channel that is created by the `channel()` call and an instance of the `Squares` coroutine. The graph description must be finalized by the `build()` call.

The `receive()` call blocks the current thread and schedules the `Squares` coroutine. The coroutines are scheduled lazily by receiving and sending values. A coroutine is initially run only when it is expected to consume or produce values.

A.2 Receive from two channels with *Join*

If two threads try to acquire the same two locks, but in different order, the potential for deadlock arises. The same applies to two coroutines that are receiving values from the same two channels.

This issue is solved by the `Graph.join()` method, which is similar to the `zip`[6] function commonly used with lists. It can be used to receive pairs of values from two channels without the potential for deadlock.

The code in Figure 21 creates a graph of two coroutines and joins their output channels. The resulting channel contains pairs of squares and cubes of integers from 0 to 5.

[6] https://docs.python.org/3/library/functions.html#zip

```java
var graph = Graph.create();

var squares = graph.<Integer>channel();
graph.coroutine(new Squares(squares));

var cubes = graph.<Integer>channel();
graph.coroutine(new Cubes(cubes));

var result = graph.join(squares, cubes);

graph.build();

for (int i = 0; i < 5; i++) {
    System.out.println(result.receive());
}
// Prints
// (0, 0) (1, 1) (4, 8) (9, 27) (16, 64)
```

Figure 21: *Joining* two channels together

A.3 Adding non-determinism

The code in Listing 22 implements a non-deterministic coroutine that reads the file and sends its contents line-by-line into a channel. Non-deterministic coroutines extend from `AsyncCoroutine`. This is the only difference.

```java
class FileReader extends AsyncCoroutine {
    // ...
    @Override
    protected void run() {
        // read lines from file
        var lines = ...;
        for (String line : lines) {
            output.send(line);
        }
    }
}
```

Figure 22: A non-deterministic coroutine

This library allows separating deterministic core logic from non-deterministic code into separate subgraphs. The code in Figure 23 demonstrates how the previously defined deterministic graph can be reused together with the newly created non-deterministic `FileReader` coroutine.

```
// returns the previously created `squares` channel
var squares = getSquaresChannel();

var graph = AsyncGraph.create();
var lines = graph.<String>channel();
graph.coroutine(new FileReader(''file.txt'', lines));

var result = graph.join(squares, lines);

graph.build();

for (int i = 0; i < 5; i++) {
    System.out.println(result.receive());
}
// Prints
// (0, line0) (1, line1) (4, line2) (9, line3) (16, line4)
```

Figure 23: Separating deterministic core logic from non-deterministic code

A.4 Merging channels with *Select*

Suppose a resource is stored at two different locations. The objective is to fetch the resource from both servers simultaneously and return the response that is received first.

Assuming the coroutine `Fetch` fetches the resource from the given server this can be achieved with the code in Figure 24. The *select* operation is similar to the `select`[7] statement in *Go*.

```
var channel1 = graph.<Response>channel();
graph.coroutine(new Fetch(url1, channel1));

var channel2 = graph.<Response>channel();
graph.coroutine(new Fetch(url2, channel2));

var result = graph.select(channel1, channel2);
// Receives the response that arrives first
System.out.println(result.receive());
```

Figure 24: Using *select* to concurrently fetch from two servers

[7]https://go.dev/ref/spec#Select_statements