RICE

Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns

ECOOP 2014
August 1, 2014

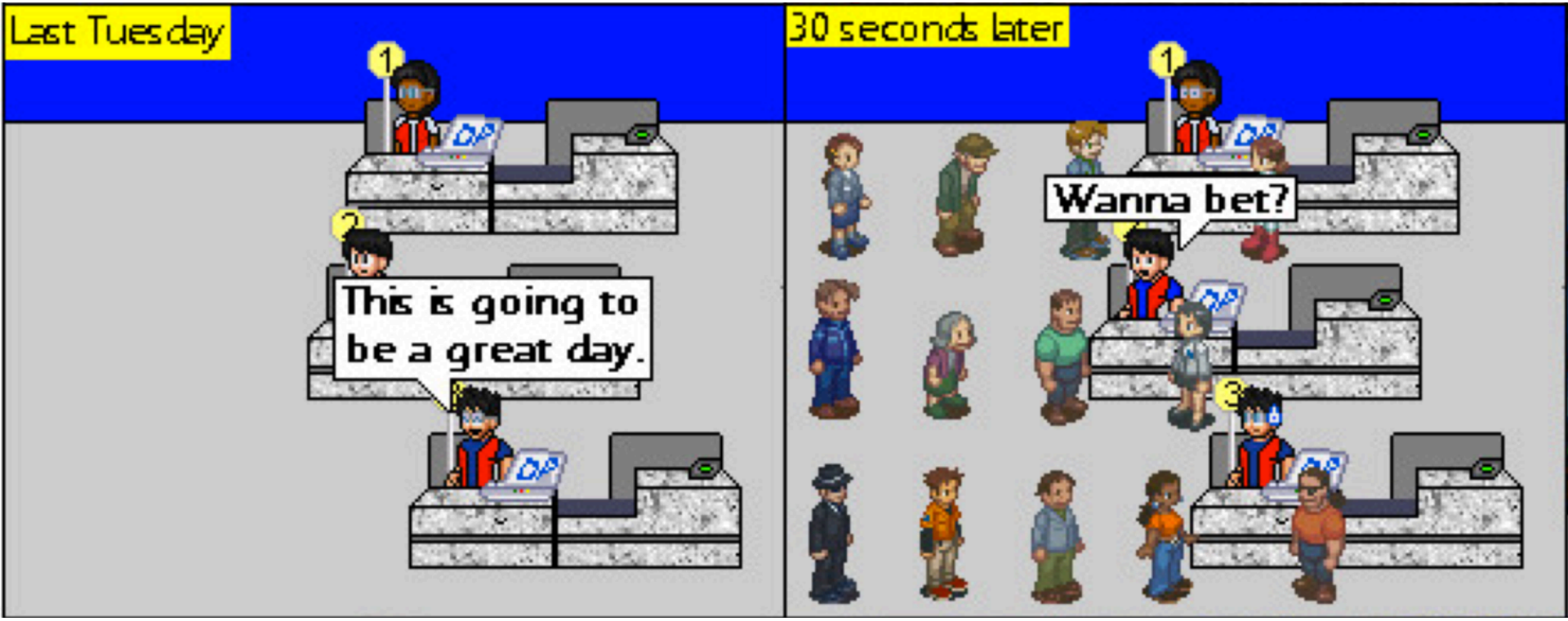
Shams Imam, Vivek Sarkar
Rice University

Task-Parallel Model

- Worker Threads
  - Typically one per core

source: http://www.deviantart.com/art/Randomness-20-178737664

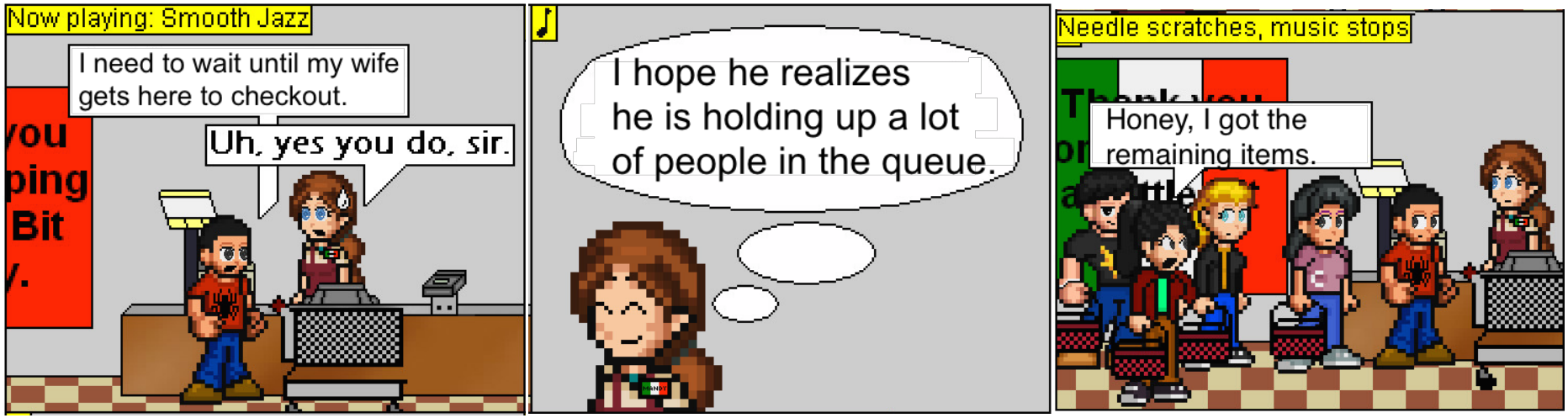
Task-Parallel Model



- Tasks, Work Queues, and Worker Threads
- Runtime manages load balancing and synchronization

source: http://www.deviantart.com/art/Randomness-20-178737664

Synchronization Constraints



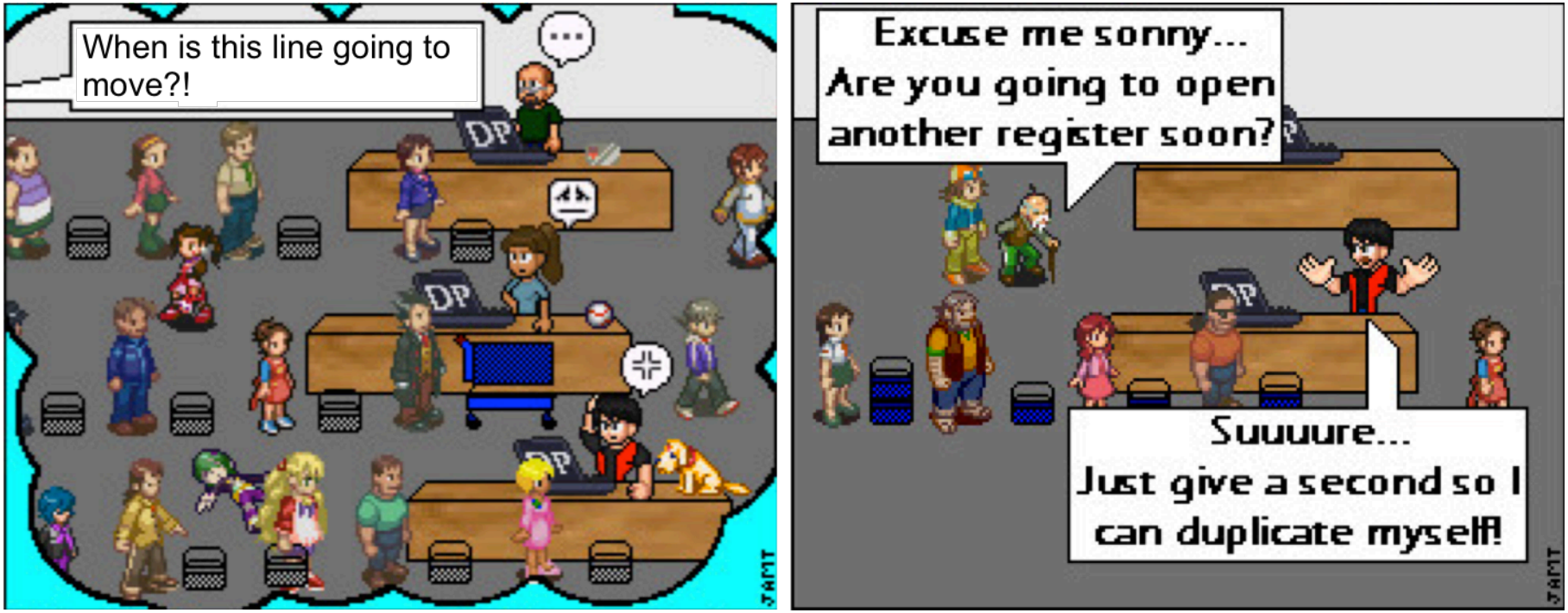
- Dependences between tasks
- Prevent an executing task from making further progress
  - Needs to synchronize with other executing task(s)

source: http://viper-x27.deviantart.com/art/Checkout-Lane-Guest-Comic-161795346

Common synchronization constructs

- Join operations
- Futures
- Barriers / Clocks / Phasers
- Atomic Blocks
- More in the future?

Common Solution to Synchronization: Block Worker Threads



Each thread needs its own system resources

Can lead to exhaustion of memory or other system resources

Thread blocking approaches do not scale!

source: http://www.deviantart.com/art/Randomness-5-90424754

Common Solution to Synchronization: Block Worker Threads

RICE

- Creating additional threads is contradictory to goal of TPM
- 1100 ns per thread context switch (without cache effects)

- In contrast
  - Object allocation takes around 30 ns
  - Method call takes around 5 ns
  - Setting fields takes around 1 ns

- Key assumption of this work
  - Creating threads and context switching is expensive
  - Task creation and synchronization constructs are cheaper

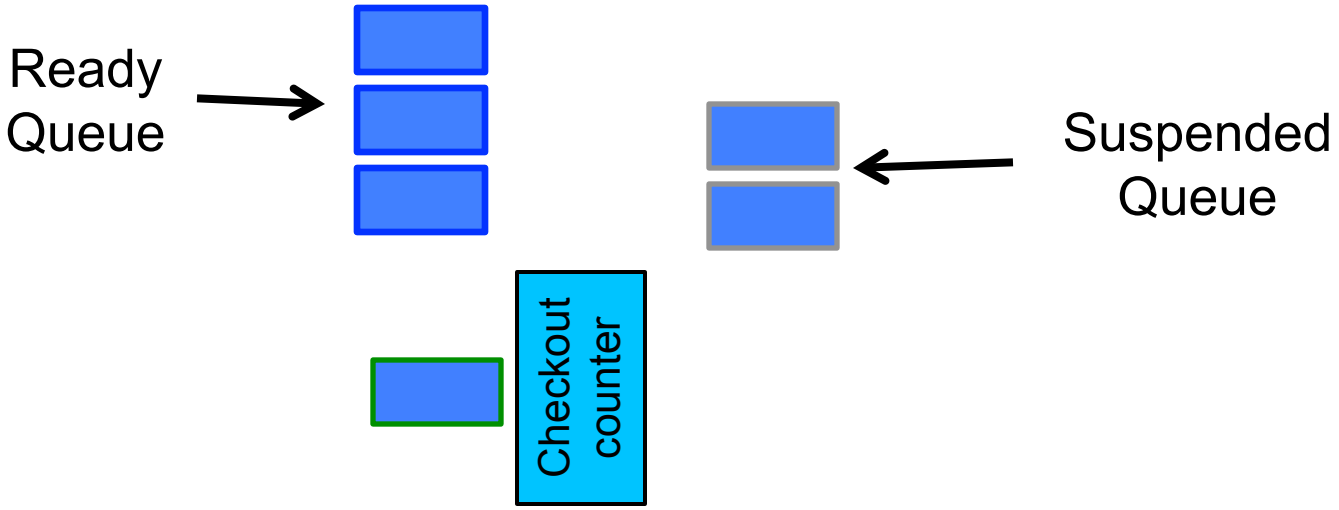
Proposed Solution

- A Cooperative Approach is more efficient

source: http://www.deviantart.com/art/3-000-Views-120361172

Cooperative Scheduling

- Task decides to actively suspend itself and **yield** control back to the runtime
- Task is added back into the ready queue when the task can make progress



Cooperative Scheduling: http://en.wikipedia.org/wiki/Computer_multitasking#Cooperative_multitasking

Cooperative Scheduling (contd)

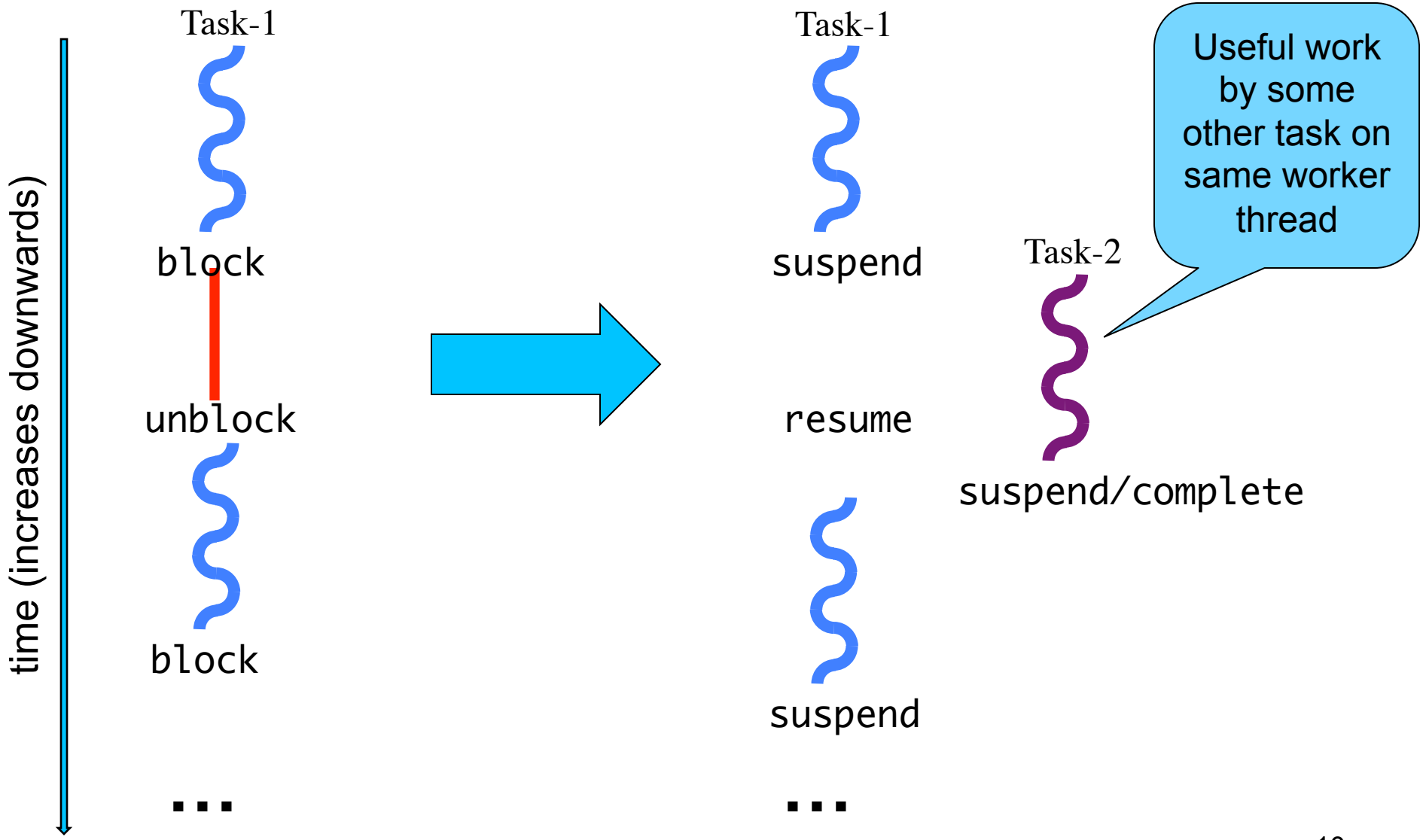


Figure represents work by a **single** worker thread

Technical Details

RICE

- Delimited Continuations
- Event-Driven Controls

One-shot Delimited Continuations

RICE

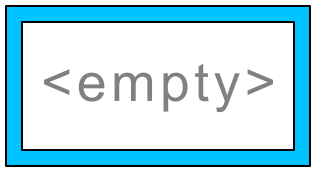
- Rest of the computation from a well-defined outer boundary
  - i.e. represents a sub-computation
- *Suspend* the state of a computation at any point
  - captures everything in the (Java) stack
    - current instruction pointer
    - return addresses
    - local variables
- *Resume* the computation, later, from that point
- One-shot: resumed at most once

Defined in “Continuations in the Java Virtual Machine”, Iulian Dragos et al.

Event-Driven Control (EDC)

RICE

- Binds a value and a list of runnable blocks
  - Runnable blocks are code that form the continuation
- Dynamic single-assignment of value (event)

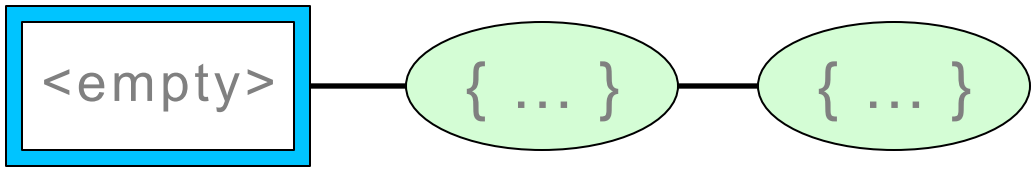


The EDC is initially empty

Event-Driven Control (EDC)

RICE

- Binds a value and a list of runnable blocks
- Dynamic single-assignment of value (event)

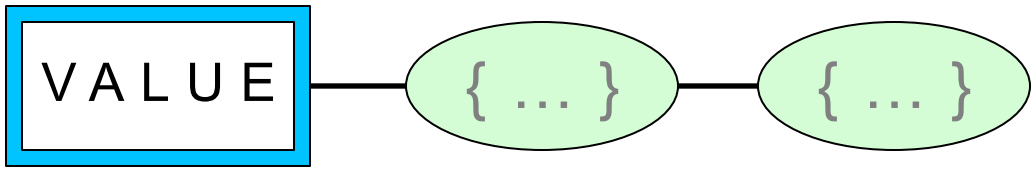


Continuations attach to the EDC and
are not triggered until value is available
(i.e. until event is satisfied)

Event-Driven Control (EDC)

RICE

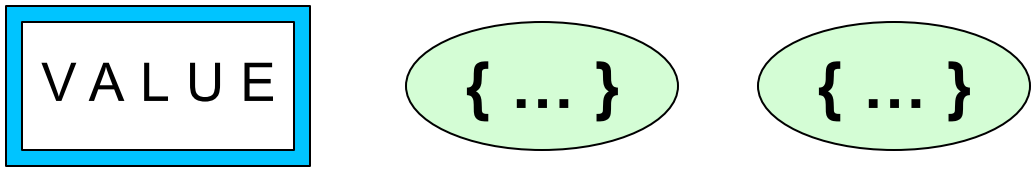
- Binds a value and a list of runnable blocks
- Dynamic single-assignment of value (event)



Eventually, a value becomes available in the EDC (follows from deadlock freedom property of finish, futures, clocks, atomic)

Event-Driven Control (EDC)

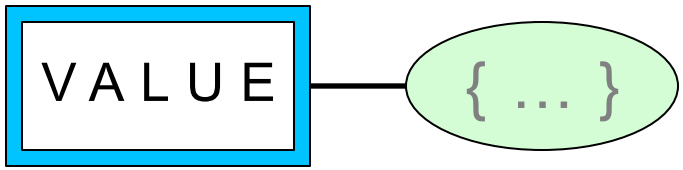
- Binds a value and a list of runnable blocks
- Dynamic single-assignment of value (event)



This enables execution of continuations attached to the EDC

Event-Driven Control (EDC)

- Binds a value and a list of runnable blocks
- Dynamic single-assignment of value (event)

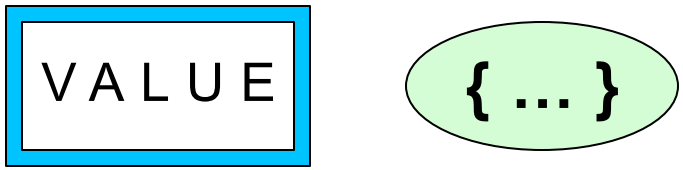


Subsequent continuation attachment requests…

Event-Driven Control (EDC)

RICE

- Binds a value and a list of runnable blocks
- Dynamic single-assignment of value (event)



Synchronously execute the continuation
(e.g. schedule a task into the work queue)

Event-Driven Control API

RICE

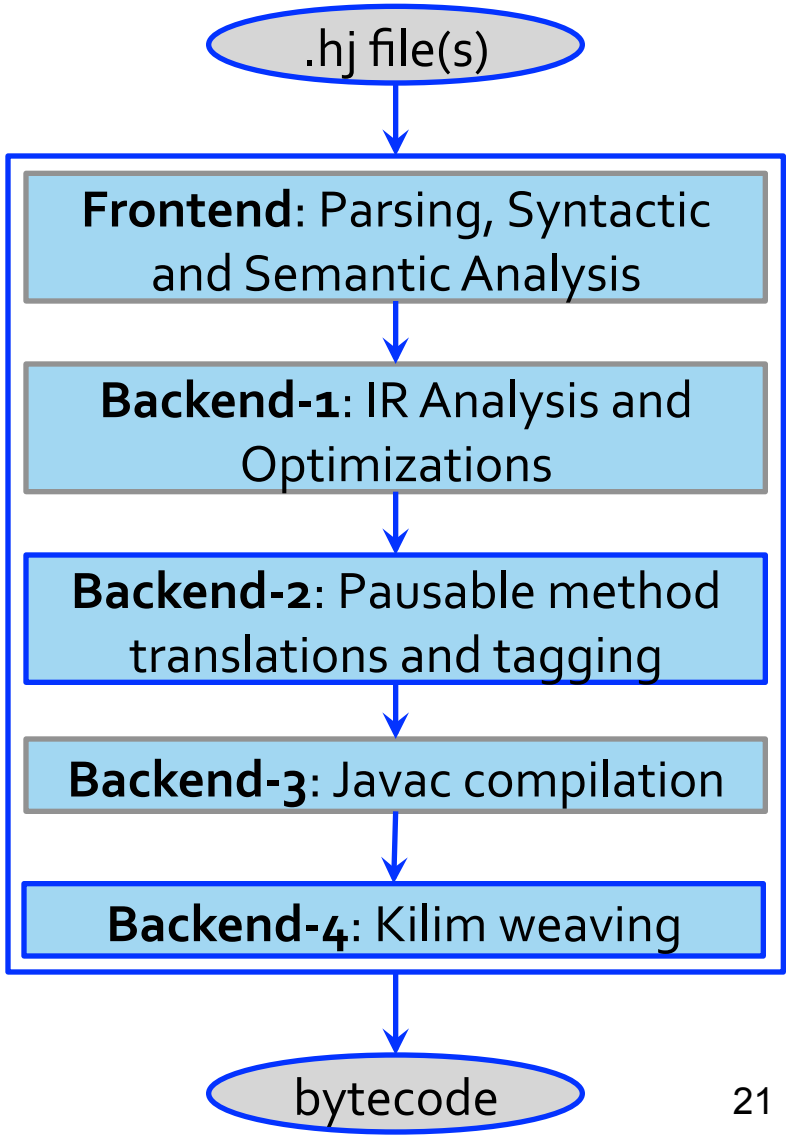
- currentTaskId():
  - returns a unique id of the currently executing task
- newEDC():
  - factory method to create EDC instance
- suspend(*anEdc*):
  - the current task is suspended if the EDC has not been resolved
  - Implementation attaches runnable block to resume task
- *anEdc*.getValue()
  - retrieves the value associated with the EDC
  - safe to call this method if execution proceeds past a call to suspend()
- *anEdc*.setValue(*aValue*)
  - resolves the EDC
  - triggers the execution of any EBs

Cooperative Runtime

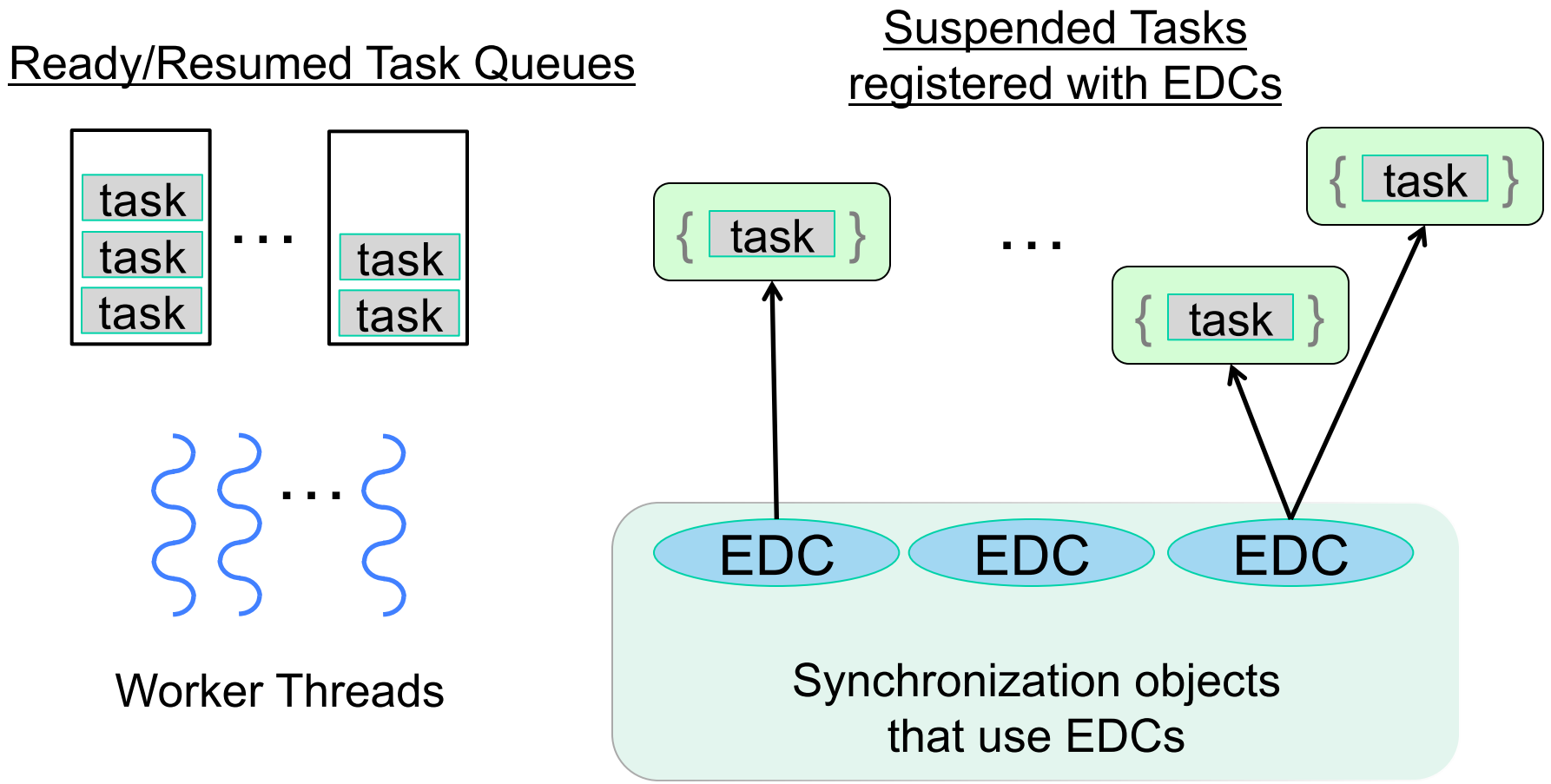
- We expose EDCs as an API in our runtime.
  - Read / Write / Query on value
  - Suspend till value becomes available
- Continuations not exposed to developer
  - Notorious for being hard to use and to understand
- Developers write thread-based code
  - Compiler handles CPS code transformations
  - One-shot delimited continuations implemented more efficiently than general continuations

Implementation: Compiler Infrastructure

- New Habanero-Java runtime
- Frontend, Backend-1, and Backend-3 reused from thread-blocking runtime compiler
- Backend-2 tags methods as 'suspendable'
  - Supports polymorphism
- Backend-4 CPS transforms code to support continuations



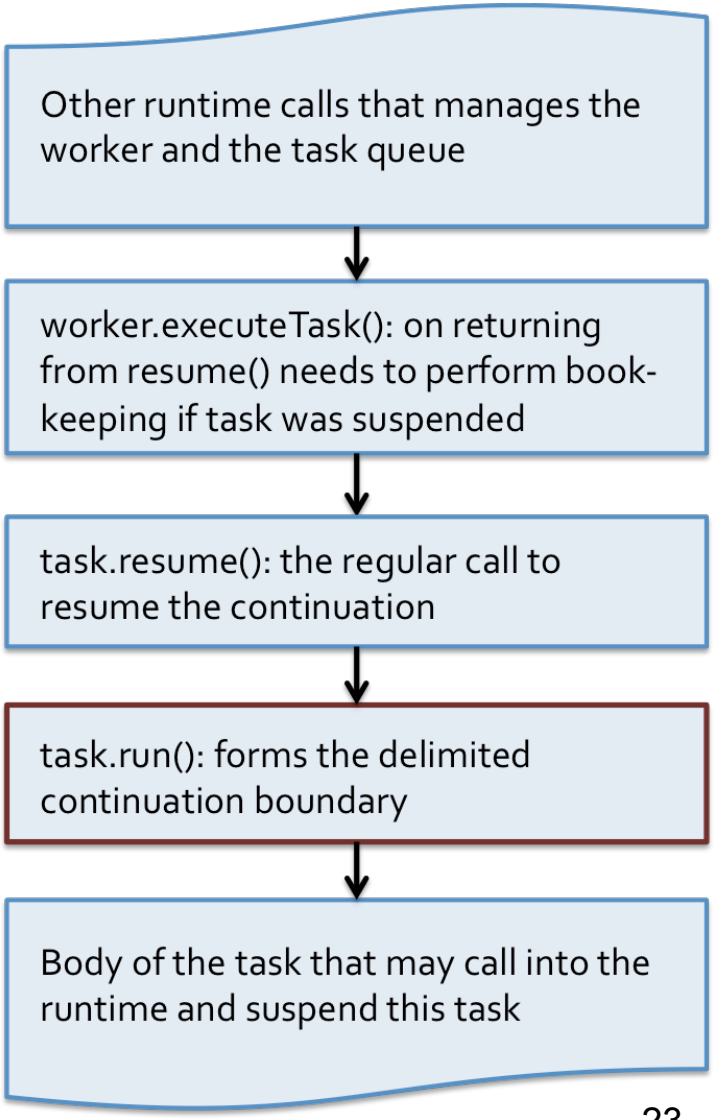
Cooperative Runtime



Cooperative Runtime – Call Stack

RICE

- **Help-first** policy
  - Task has a stack of its own
  - Task can be executed by any of the worker threads
- Task wrapped to form a Delimited Continuation
- Worker thread manages when tasks get
  - added to suspended queue
  - removed from ready queues for execution by worker threads



Benefits of Cooperative Runtime

RICE

- Bound the number of worker threads
- Threads never block
  - Additional threads do not need to be created
  - (Tasks may suspend)
- Do not need more than one worker thread
  - Computations can be made serializable
  - Can help in reproducibility and debugging

Synchronization Constructs

RICE

- Key idea is to:
  - Translate the coordination constraints into producer-consumer constraints on EDCs
  - Use Delimited Continuations to suspend consumers when waiting on item(s) from producer(s)
- Any task-parallel Synchronization Constraint can be supported.
  - Both deterministic and non-deterministic constructs
  - Including atomic/isolated and actors

Implementation Recipe

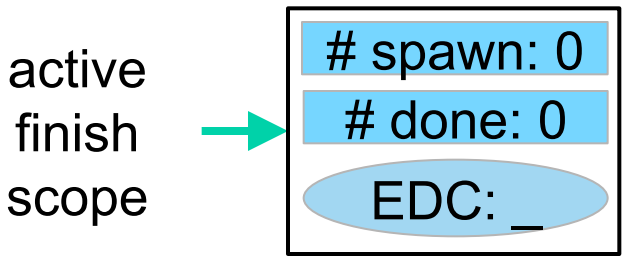
- **Async-Finish** (Join operations)
  - Counters to track in-flight spawned and completed tasks
  - Single EDC resolved when count reaches zero
  - Any async task maintains a Stack of nested finish scopes
  - Tasks suspends on EDC at the end-of-finish

Implementation Recipe

```
S0;
finish {
  S1;
  async {
    S2;
  }
  async {
    S3;
  }
}
S4;
S5;
```

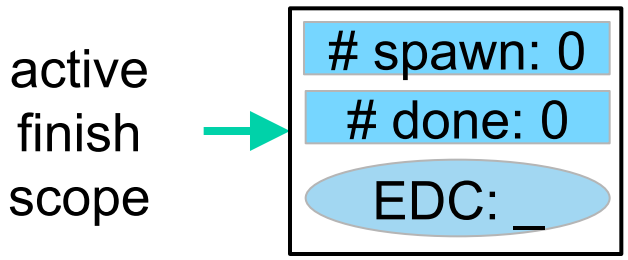
Implementation Recipe

```
S0;
finish {
  S1;
  async {
    S2;
  }
  async {
    S3;
  }
}
S4;
S5;
```



Implementation Recipe

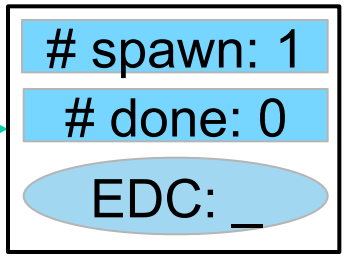
```
S0;
finish {
  S1;
  async {
    S2;
  }
  async {
    S3;
  }
}
S4;
S5;
```



Implementation Recipe

```
S0;
finish {
  S1;
  async {
    S2;
  }
  async {
    S3;
  }
}
S4;
S5;
```

active finish scope

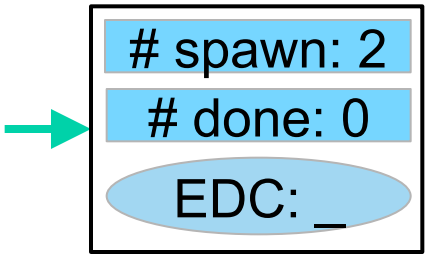


Implementation Recipe

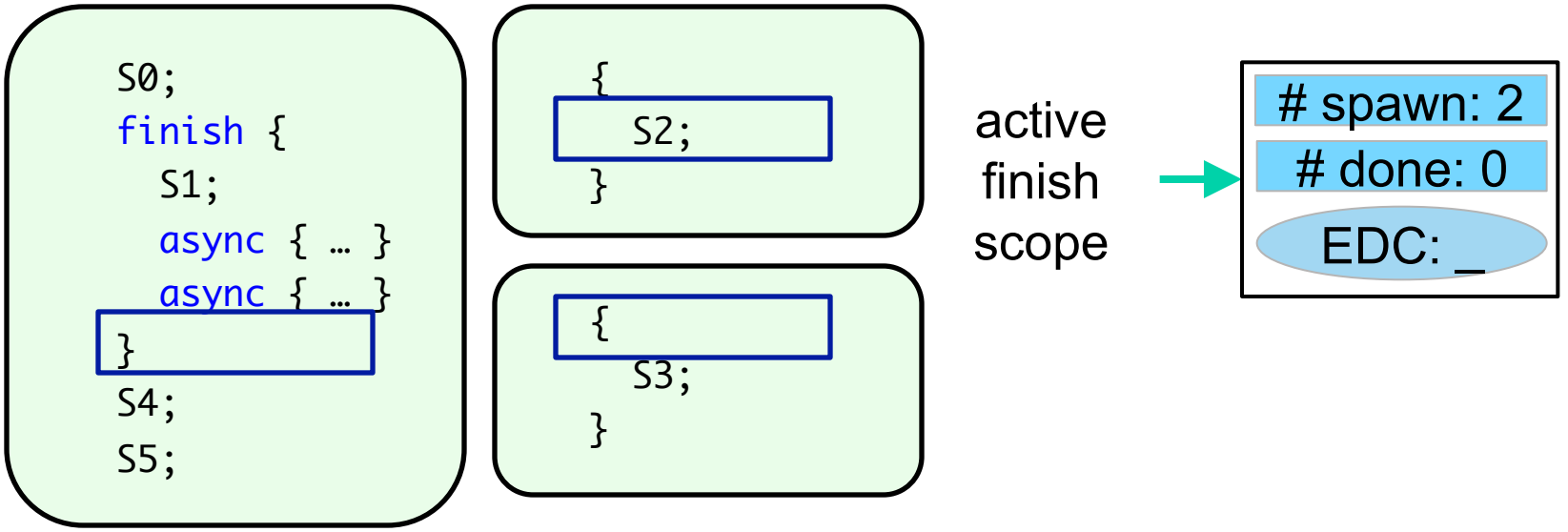
```
S0;
finish {
  S1;
  async { … }
  async {
    S3;
  }
}
S4;
S5;
```

```
{
  S2;
}
```

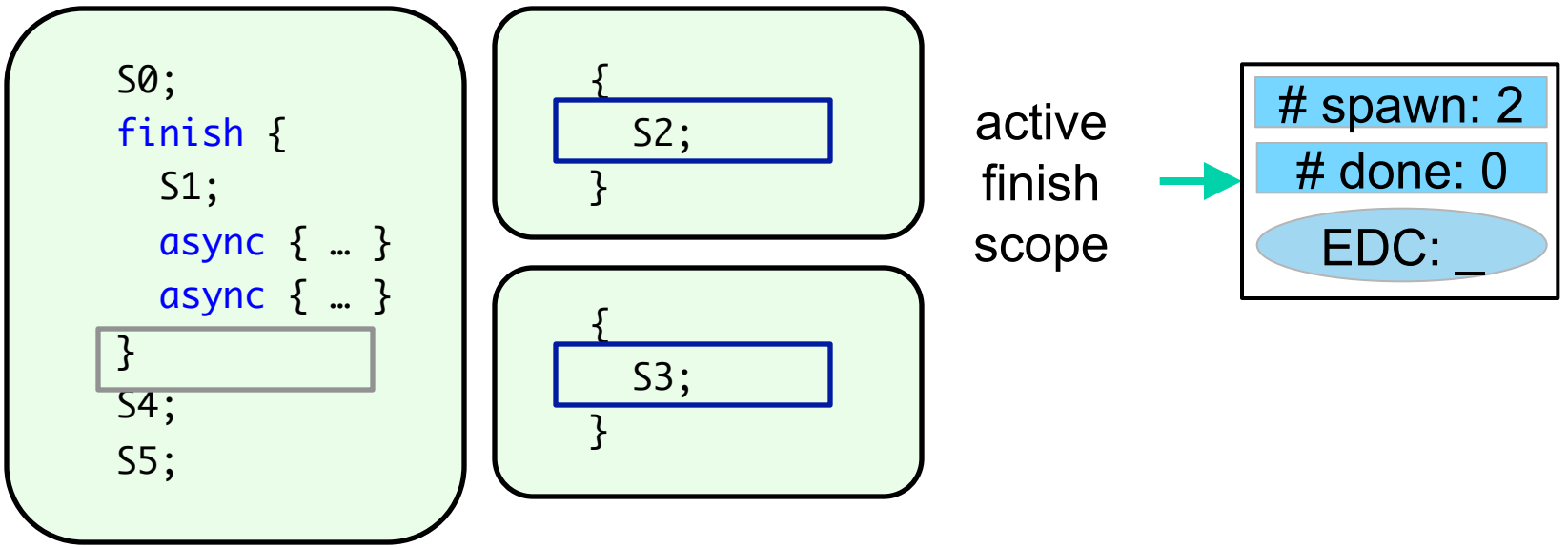
active finish scope



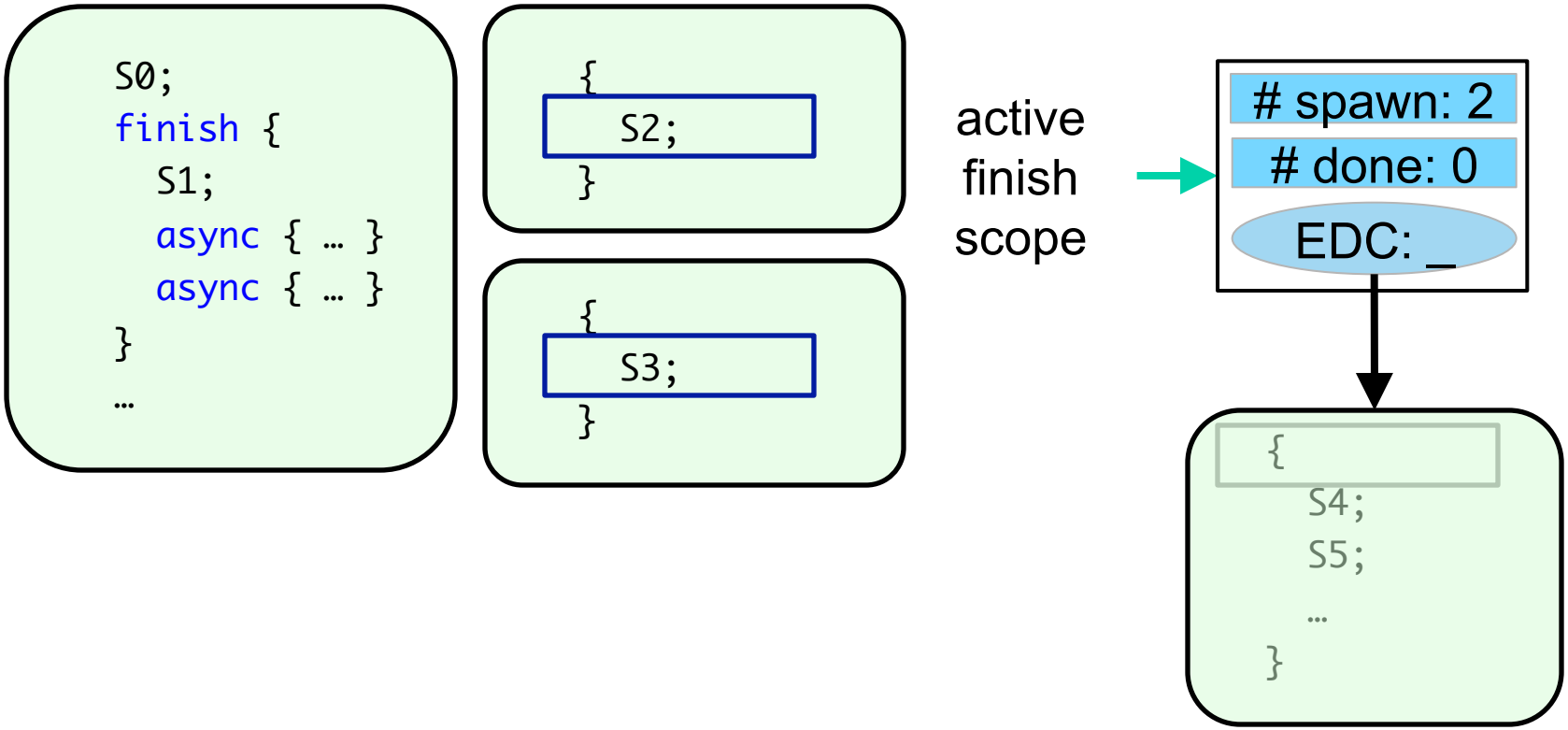
Implementation Recipe



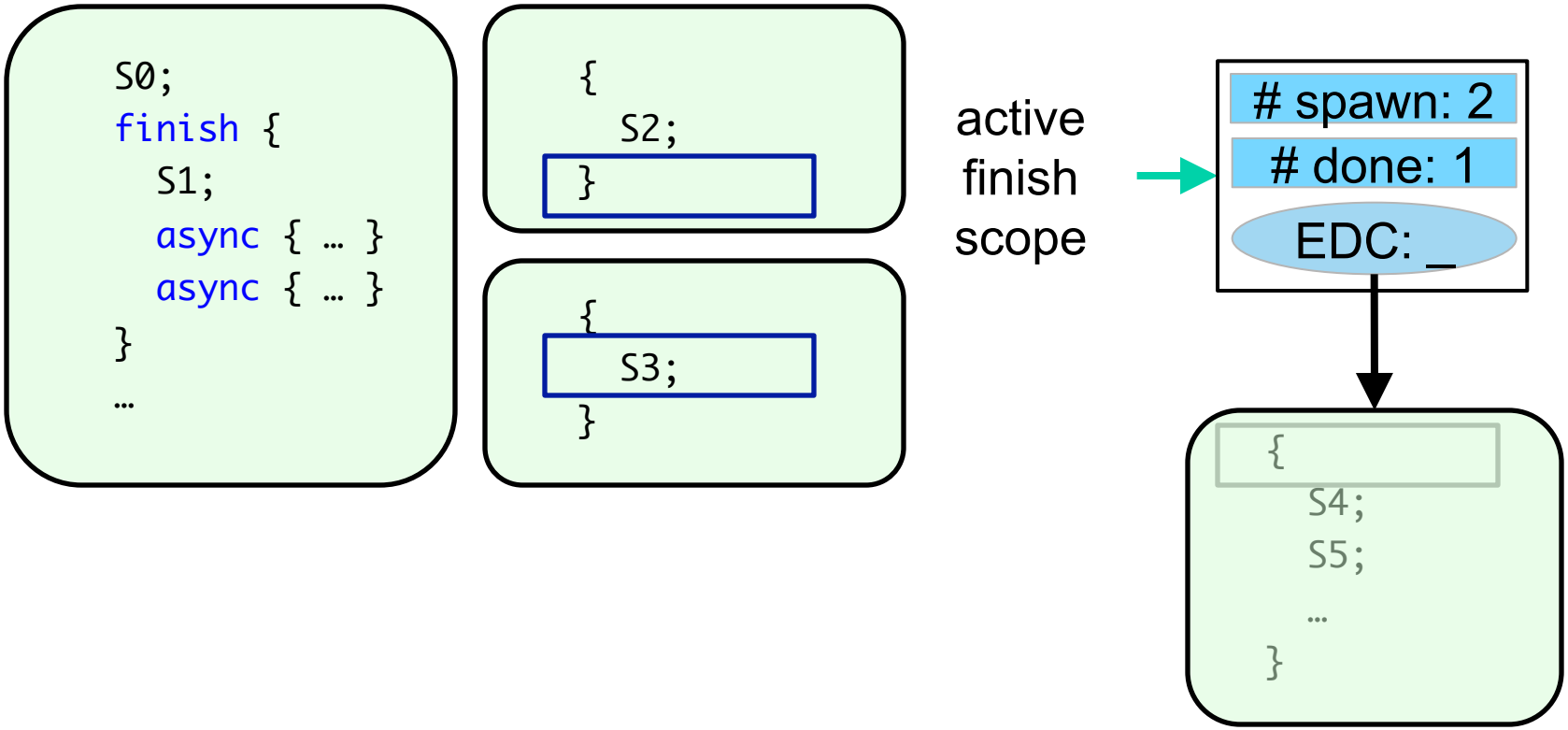
Implementation Recipe



Implementation Recipe



Implementation Recipe



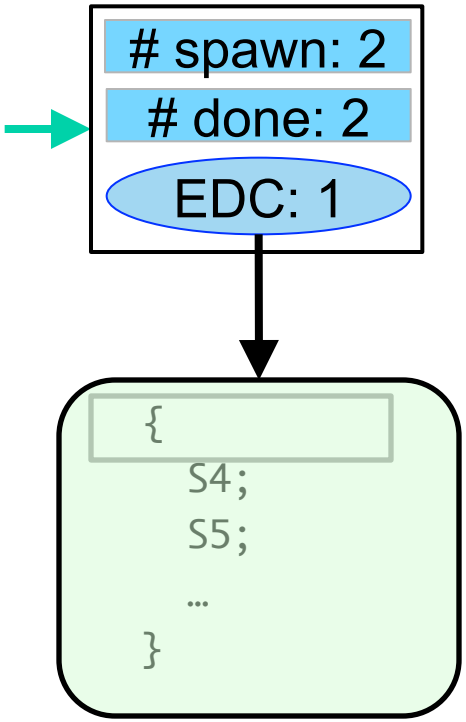
Implementation Recipe

```
S0;
finish {
  S1;
  async { … }
  async { … }
}
…
```

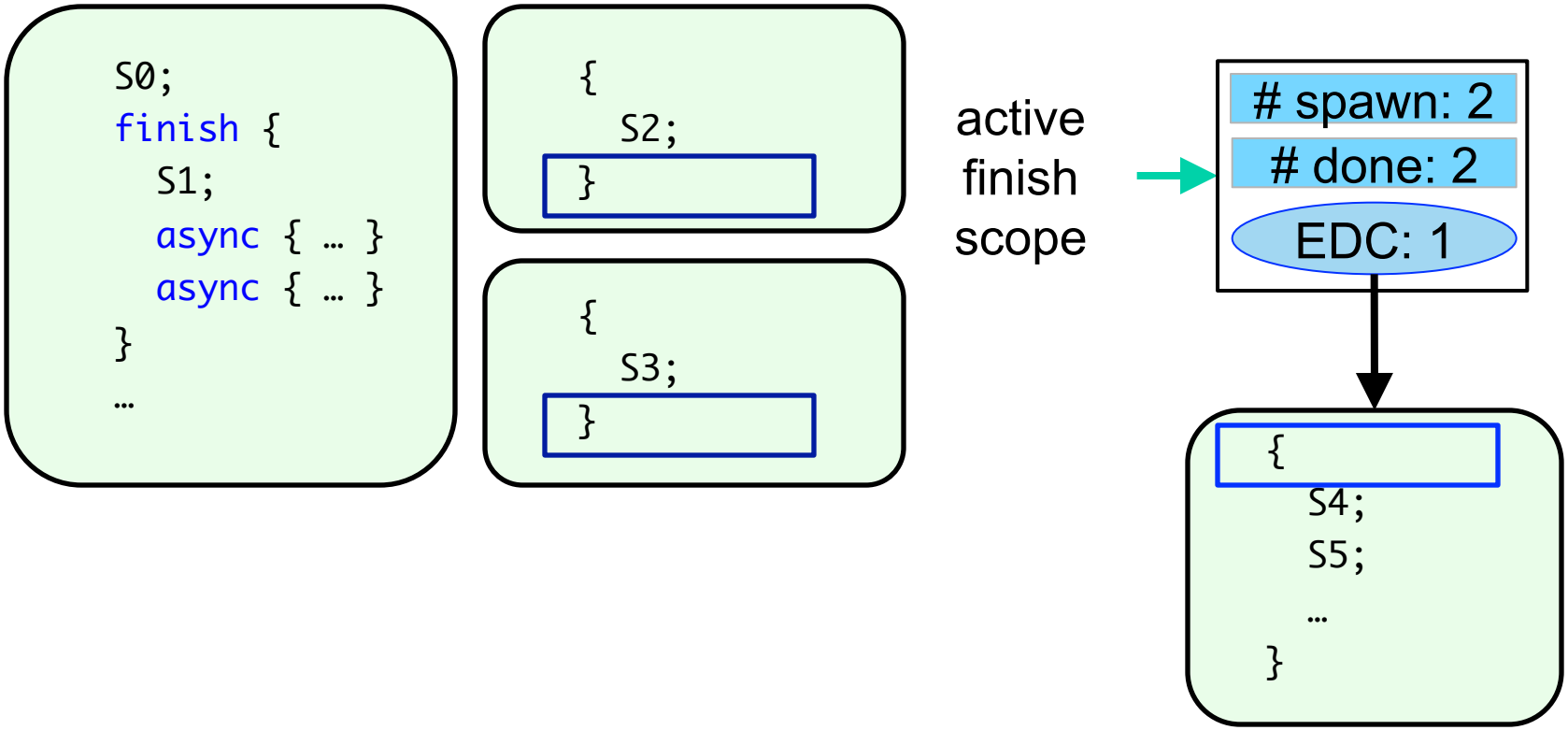
```
{
  S2;
}
```

```
{
  S3;
}
```

active finish scope

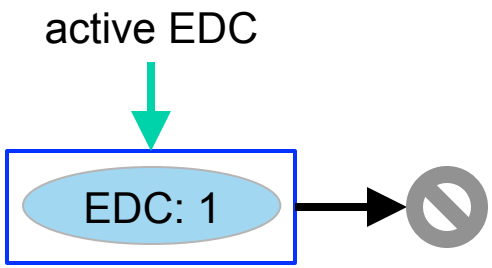


Implementation Recipe

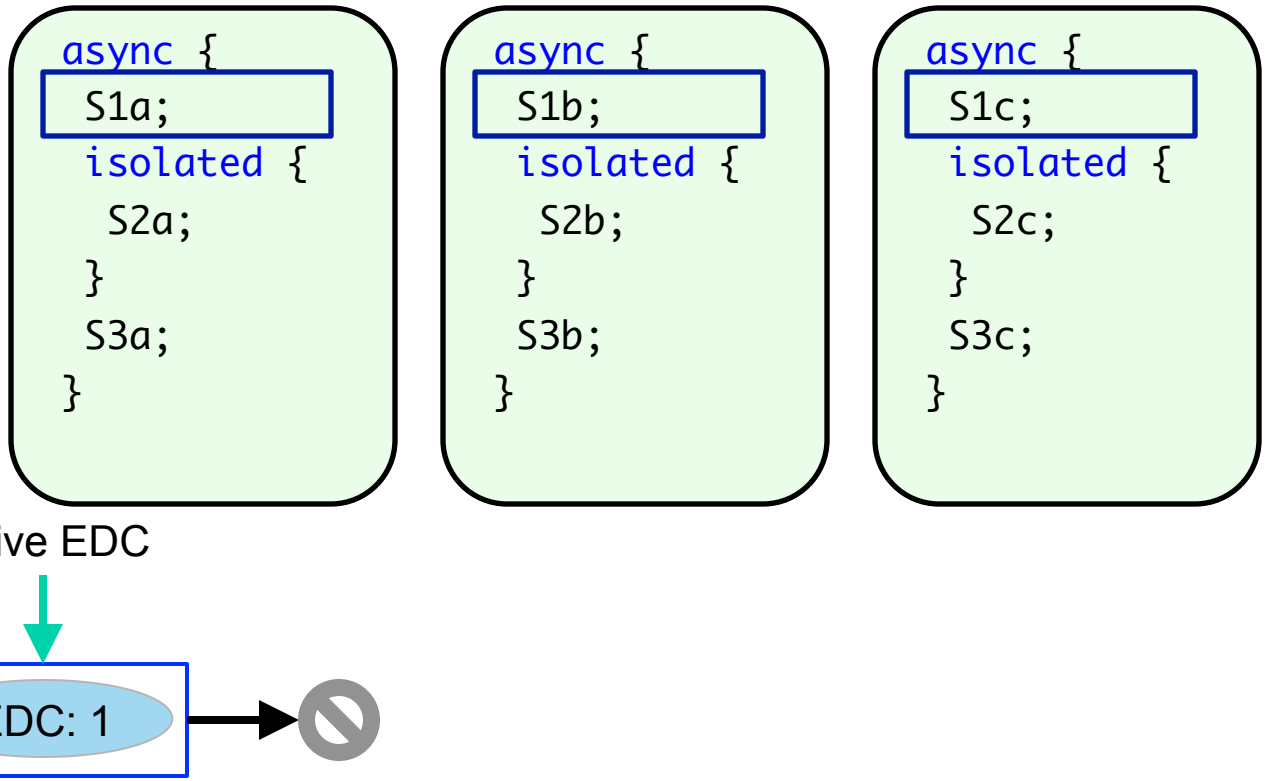


Implementation Recipe

- **Atomic/Isolated** blocks
  - Linked-list of EDCs to grant tasks permission to execute
  - First EDC in linked-list is resolved by default
  - ‘lock’ request causes task to suspend on next available EDC in the list
  - During ‘unlock’ resolve the value of the next EDC in the list



Implementation Recipe

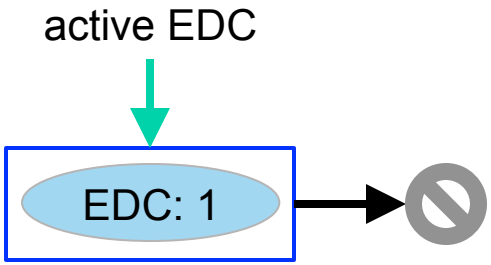


Implementation Recipe

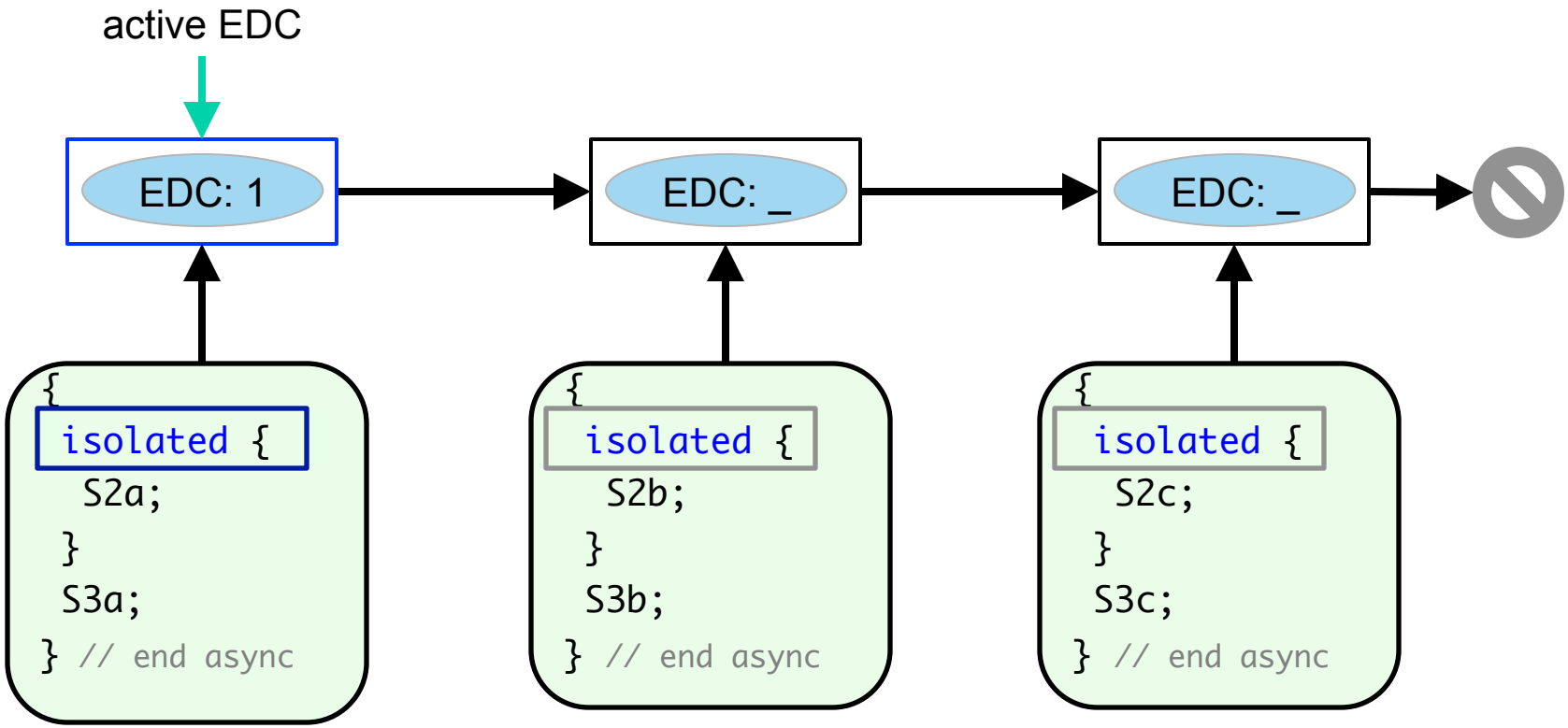
```
async {
 S1a;
 isolated {
  S2a;
 }
 S3a;
}
```

```
async {
 S1b;
 isolated {
  S2b;
 }
 S3b;
}
```

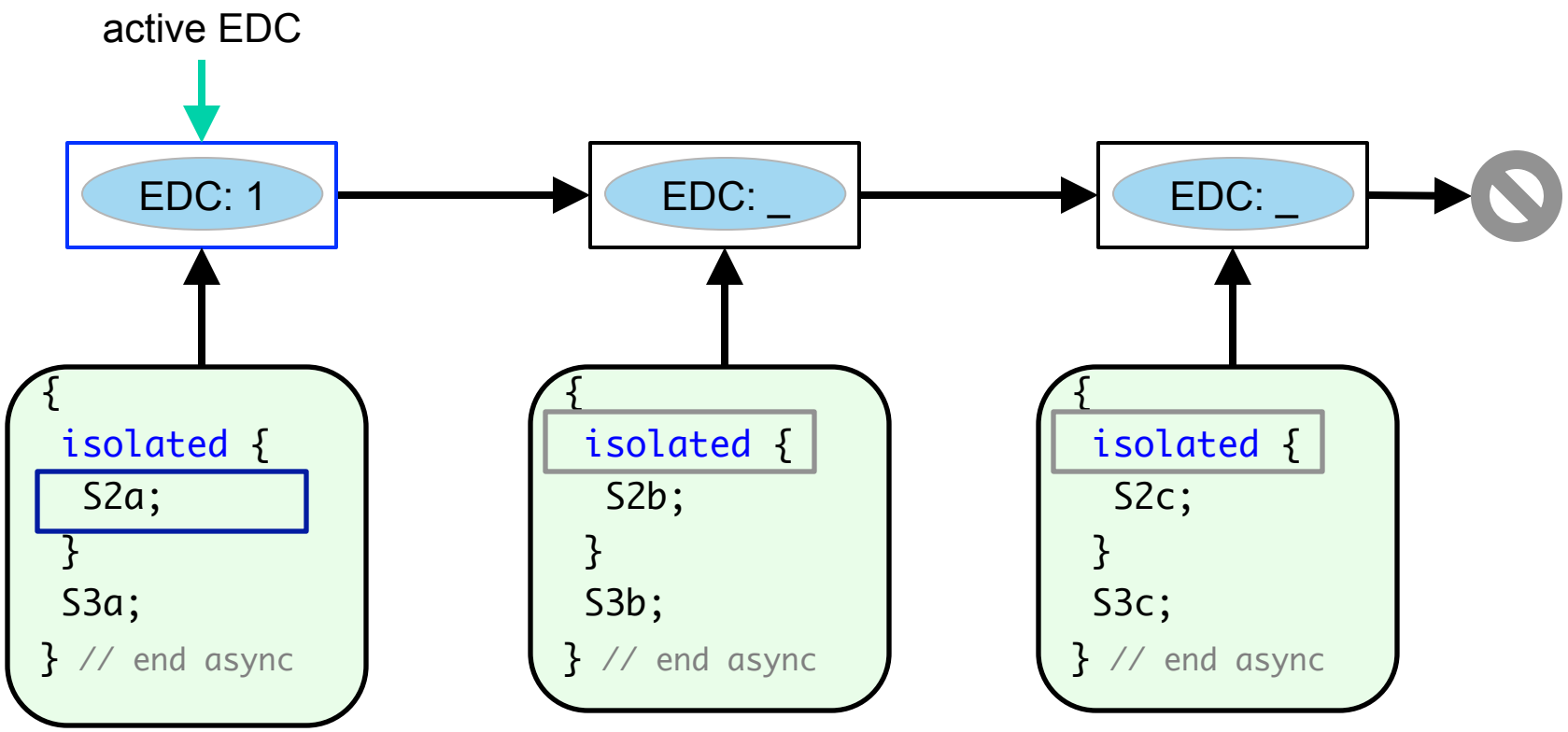
```
async {
 S1c;
 isolated {
  S2c;
 }
 S3c;
}
```



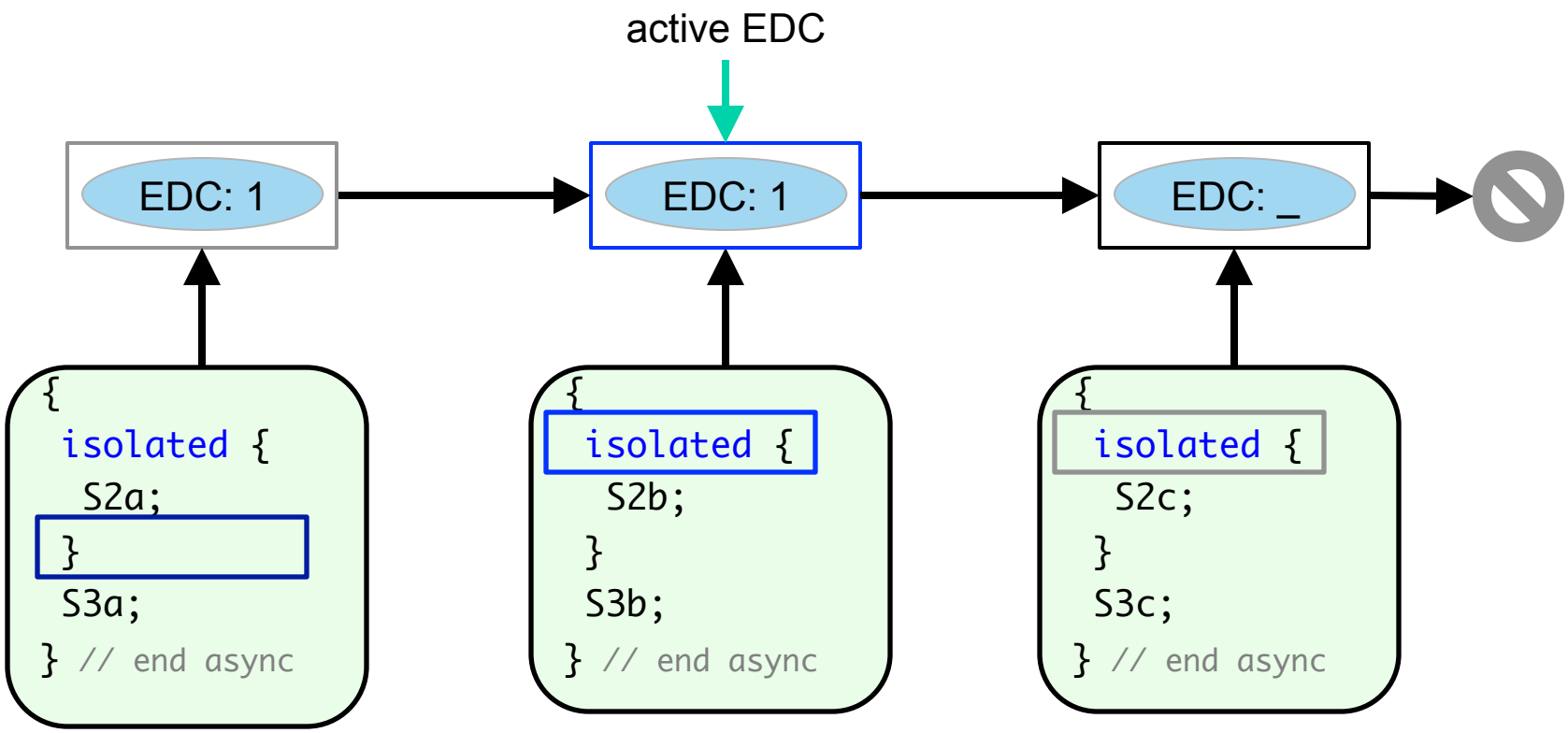
Implementation Recipe



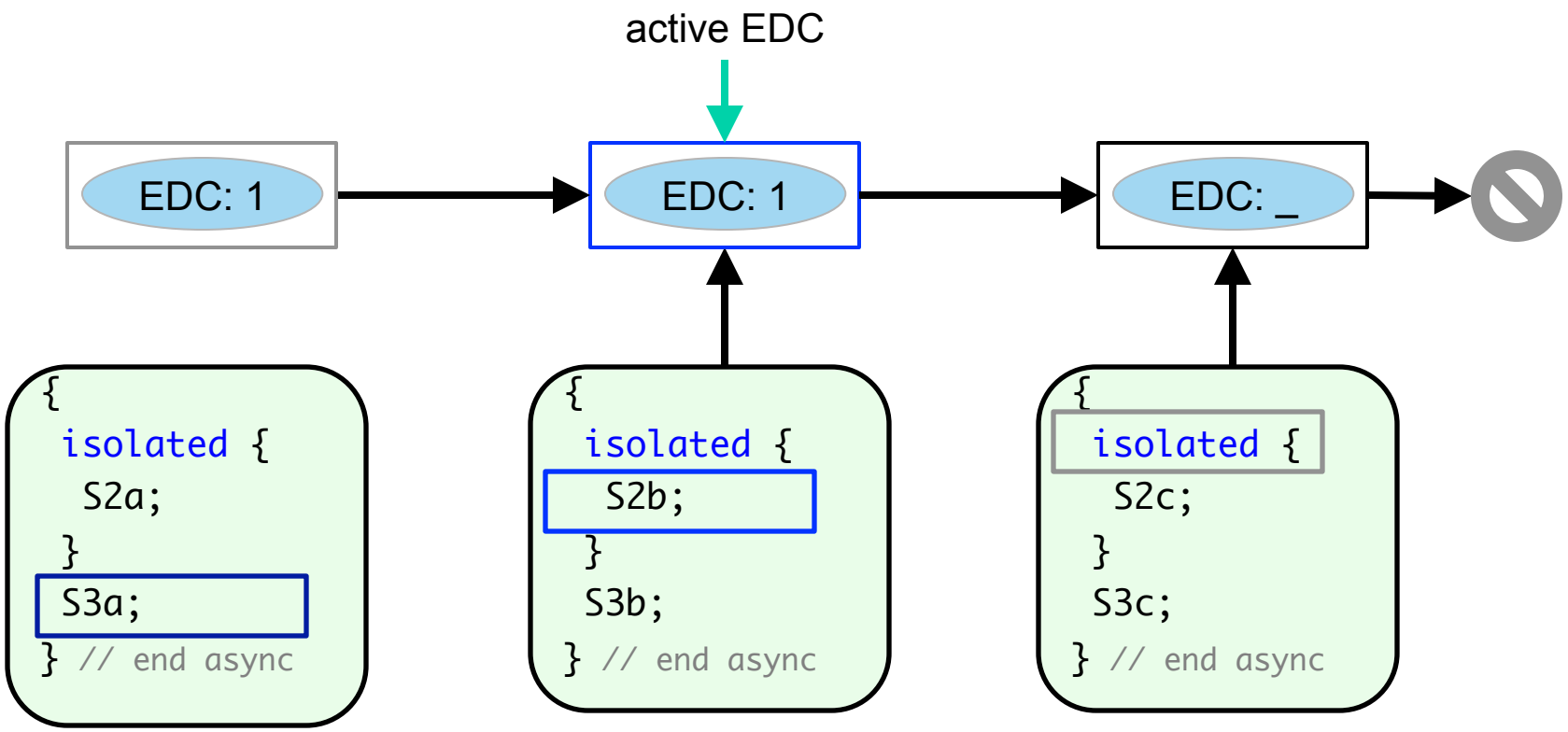
Implementation Recipe



Implementation Recipe

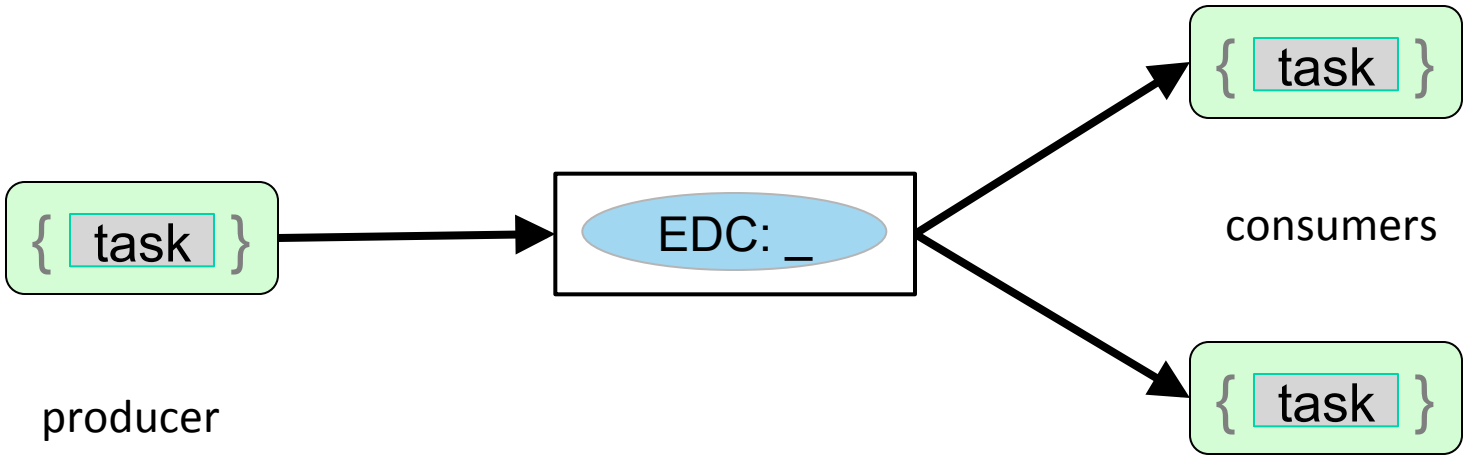


Implementation Recipe



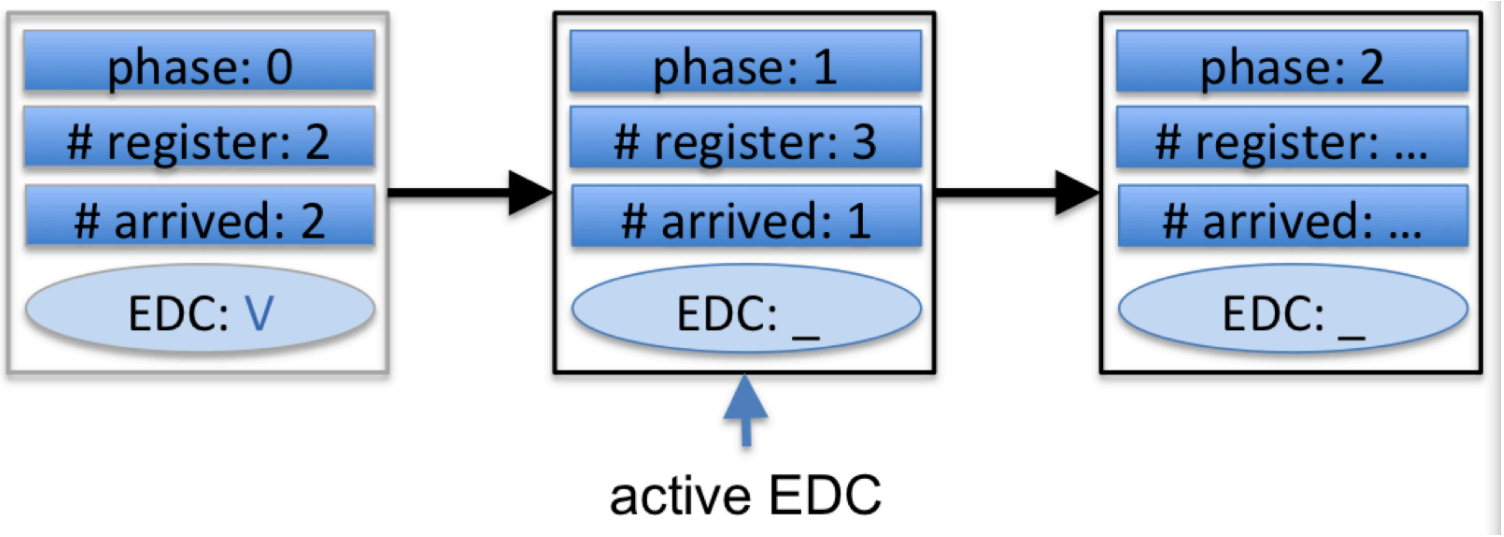
Implementation Recipe

- **Futures**
  - Single EDC to store future value
  - EDC resolved when future task is executed
  - Consumers suspend until EDC is resolved



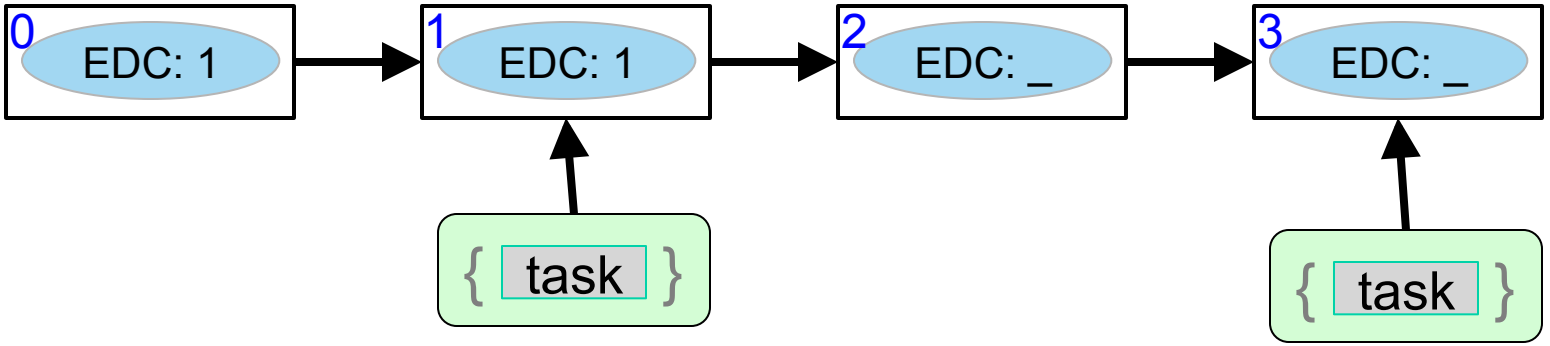
Implementation Recipe

- **Clocks / Phasers**
  - One EDC per phase
  - Tasks can leave and enter phases
  - Track number of registered and arrived tasks for each phase
  - Resolve EDC when counts become equal



Implementation Recipe

- **EventCount**
  - Counts number of events that have occurred so far
  - One EDC per count in a list
  - Resolve EDC when counts reaches particular value
  - Tasks await on specific element of list



Experimental Setup

- Four 8-core 3.8 GHz IBM POWER7
  - 256 GB of RAM
  - 32 KB L1 Cache
  - Threads bound to cores (using `taskset` command)
- IBM Java SDK Version 7 Release 1
  - Classes from `java.util.concurrent` package
- Habanero-Java language v1.3.1
  - Default scheduler = work-sharing
  - Cooperative scheduler enabled via option
- Benchmarks run with single place
  - 32 worker threads per place
  - 64 GB memory allocated to JVM
  - Mean of best 30 out of 100 execution times reported

Async-Finish Benchmarks

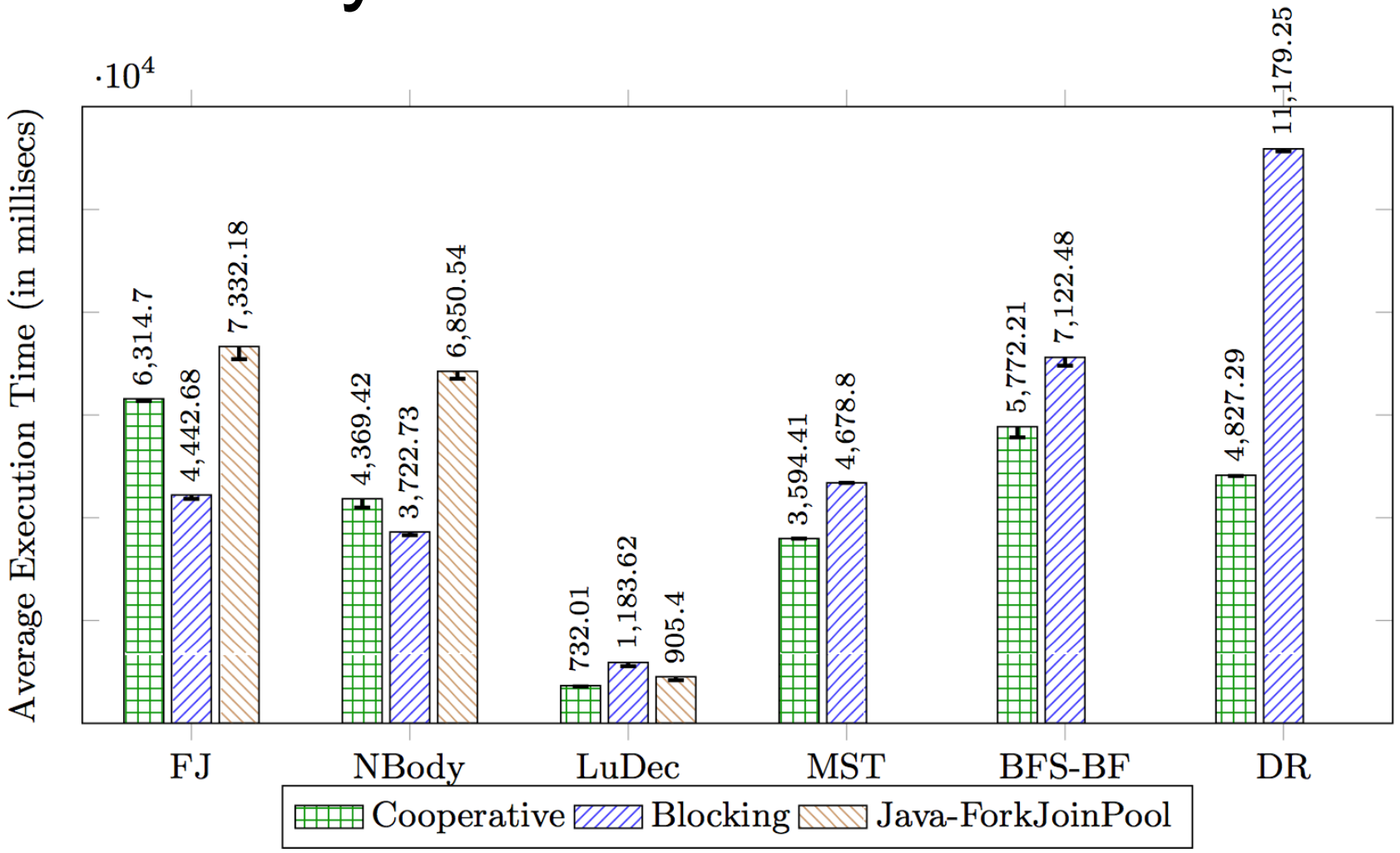


Fig. 11: Results for **async-finish** benchmarks. JGF Fork Join (FJ) with 4 million tasks. NBody with 300K steps. LU-Decomposition (LuDec) with an array size of 2K and block size of 128. MST, BFS-BF and DR with an input graph of size 512 nodes and artificial load values of 500K, 20M, and 8M respectively.

Future Benchmarks

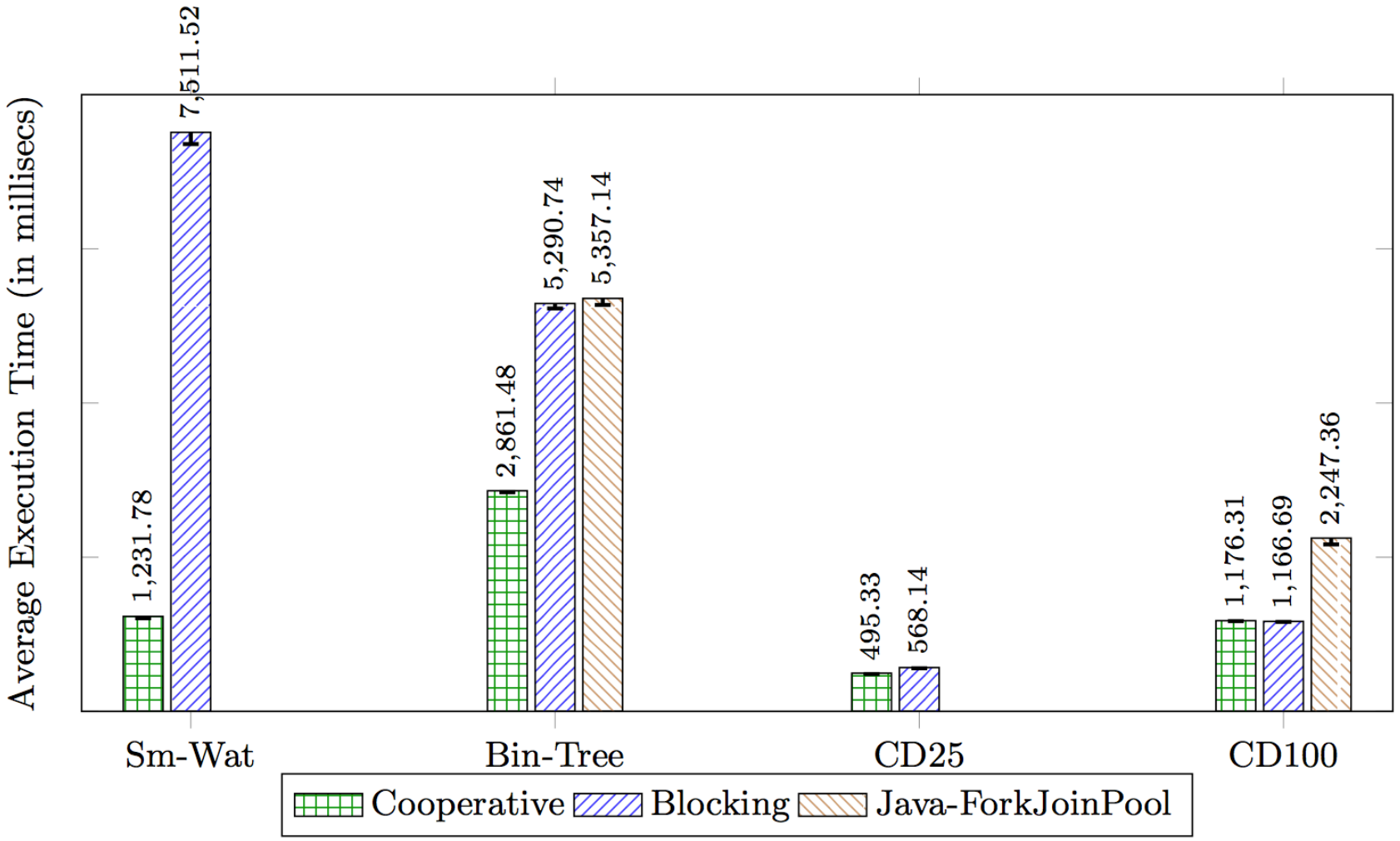


Fig. 12: Smith Waterman on strings of length 960 and 928. Binary Tree operating on a tree with depth of 14. Cholesky Decomposition on an input matrix of size 2000×2000 with tile sizes of 25 and 100.

Phaser Benchmarks

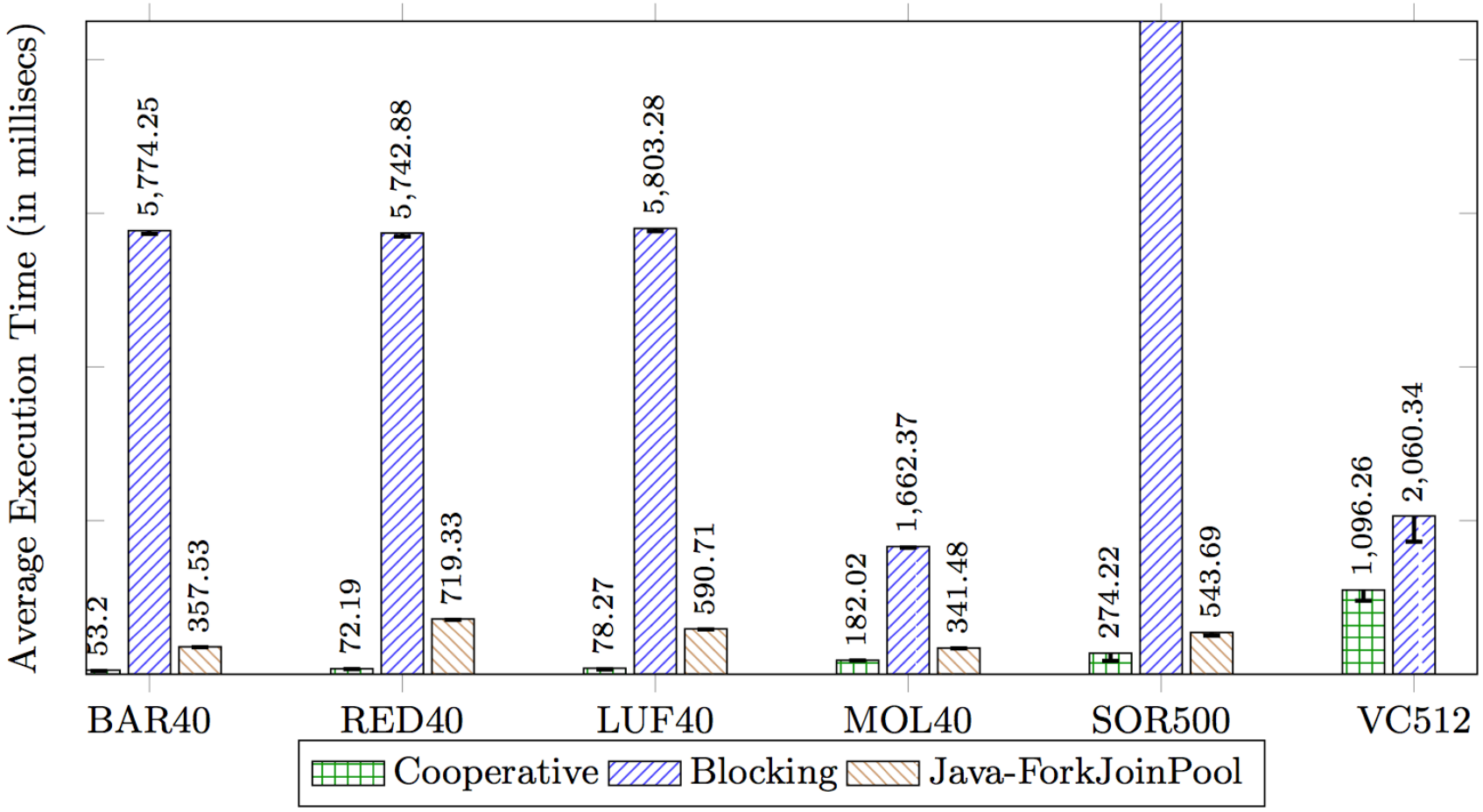


Fig. 13: Phaser benchmark results. BAR, RED, LUF, and MOL with 40 tasks registered on the phaser. SOR benchmark with an input array size of 500. VC coloring with an input graph of 512 nodes and artificial load of 10M.

Future work

- Cooperative scheduling for library implementation of Habanero-Java (HJlib)
- Pre-emptive Scheduling
  - Suspend long running tasks for fairness
  - Support priorities
- Eureka Computations
  - Support for Cilk-like abort statement with sound semantics
  - E.g. branch-and-bound computations

source: http://www.cartoonstock.com/cartoonview.asp?catref=bso0035

Related work

- Tasks / Kilim
  - Write event-driven programs in thread-based style
  - Compiler does CPS transforms of code
  - Nothing to do with task parallelism
- Qthreads
  - Continuations with lightweight call stack stitching
  - Stack size limited to 4kB

Related work

- Glasgow Haskell Compiler
  - Provides continuation support directly
  - Uses polling to resume continuations
- C++ implementation of X10
  - Work-first policy
  - Dedicated implementation for each construct
  - Supports async-finish and futures
  - Implementation did not support clocks (phasers)

RICE

Summary

- Cooperative runtime for scheduling tasks
- Using
  - One-shot Delimited Continuations
  - Event-Driven Controls
- Can support any task-parallel synchronization
- Performs better than runtimes that use blocking

Questions

- Cooperative runtime for scheduling tasks
- Using
  - One-shot Delimited Continuations
  - Event-Driven Controls
- Can support any task-parallel synchronization
- Performs better than runtimes that use blocking

```
import ecoop.audience.Questions;
```