POSTER: Lock-free Channels for Programming via Communicating Sequential Processes

Nikita Koval
IST Austria, JetBrains
ndkoval@ya.ru

Dan Alistarh
IST Austria
dan.alistarh@ist.ac.at

Roman Elizarov
JetBrains
elizarov@gmail.com

Abstract

Traditional concurrent programming involves manipulating shared mutable state. Alternatives to this programming style are communicating sequential processes (CSP) [1] and actor [2] models, which share data via explicit communication. *Rendezvous channel* is the common abstraction for communication between several processes, where senders and receivers perform a rendezvous handshake as a part of their protocol (senders wait for receivers and vice versa). Additionally to this, channels support the `select` expression.

In this work, we present the first efficient lock-free channel algorithm, and compare it against Go [3] and Kotlin [4] baseline implementations.

1 Channel Algorithm

Channel Structure. The overall channel structure is represented as Michael-Scott lock-free queue [5] of waiting processes. However, instead of dynamically creating a new `Node` for each operation to be suspended, our `Node` has a fixed-size `waiters` array of `NODE_SIZE` structures of `Waiter` type, which stores both the process and the element to be sent (a special marker element `RECEIVE_EL` is used for `receive` operations). To manipulate with these slots we maintain global 64-bit `enqIdx` and `deqIdx` indices, which indicate the positions to enqueue a new waiter and to dequeue the oldest one correspondingly.

While updating these indices, we maintain the invariant that `deqIdx` $\leq$ `enqIdx`, and these indices are equal if the channel is empty. An additional invariant is that the first waiter slot in `waiters` array of a node is always occupied when this node is added to the queue, what guarantees lock-freedom in a similar way to LCRQ [6].

The `send` and `receive` Operations. Both `send` and `receive` look for a potential rendezvous with a waiter of the opposite type (`send` rendezvous-es with `receive`, and vice versa) or add themselves as new waiters. This complex operation has to be performed atomically with respect to other ones, maintaining the invariant that the queue contains waiters of one type only (either senders or receivers), or is empty. We consider the algorithm for `send` operation only since `receive` works similar.

The high-level pseudo-code for `send` operation is presented in the Listing 1. Without interference from other threads, it firstly reads both `enqIdx` and `deqIdx` and checks if the queue is empty, adding itself to the queue in this case. If the queue contains waiters, it reads the first element and checks its type. If a rendezvous is possible, it removes the first waiter from the queue and resumes the corresponding process with the element to be sent, completes the operation after that. Otherwise, the queue contains senders, and the current process is added as a new waiter. The whole `send` operation is enclosed in an infinite loop to retry when interference from other threads is detected.

```
1  fun send(el: Any) = while true {
2    (enqIdx, deqIdx) := (this.enqIdx, this.deqIdx)
3    if enqIdx < deqIdx: continue // Inconsistent
4    if deqIdx == enqIdx: // Is the queue empty?
5      if addToQueue(enqIdx, element):
6        suspend(); return // Wait for a receiver
7    else:
8      head := this.head // Read head
9      if deqIdx / NODE_SIZE < head.id:
10       continue  // State is not consistent
11     if (deqIdx / NODE_SIZE > head.id):
12       CAS(&this.head, head, head.next)
13       continue // Head was outdated
14     // Read the first element
15     firstEl := readEl(head, deqIdx % NODE_SIZE)
16     if firstEl == BROKEN: // Is the slot broken?
17       CAS(&this.deqIdx, deqIdx, deqIdx + 1)
18       continue // Skip the broken slot
19     if firstEl == RECEIVE_EL:
20       if resumeWaiter(head, deqIdx, elem):
21         return // Made a rendezvous
22     else:
23       if addToQueue(enqIdx, element):
24         suspend(); return // Wait for a receiver
25 }
```

Listing 1. High-level algorithm for `send`.

At the point of adding the current process to the queue, the `enqIdx` has been already read in the beginning of the operation, and references the slot into which the waiter information is going to be written (we write

PPoPP '19, February 16–20, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3297000

the current process pointer at first, then the element to be sent). The algorithm either writes the information to the corresponding slot in the tail segment, or creates a new segment using the same logic as the Michael-Scott queue. Since we also maintain the global `enqIdx`, we update it after the tail pointer, and other operations can help with this update as well.

In case of rendezvous, the first element is already read, and we only need to increment the `deqIdx` to remove it from the queue, what should be performed by CAS.

Similarly to LCRQ, we guarantee progress of concurrent manipulations with a slot by marking it as "broken" in case a read happens before the write. This way, when a new waiter is adding, the slot should be updated from `null` to the current process pointer by `CAS`. If a concurrent operation comes before this update, it marks the slot as "broken" by `CAS` from `null` to `BROKEN`.

The `select` Expression. In addition to the standard `send` and `receive` operations, the CSP model supports selection among several alternates. This `select` expression makes possible to await multiple `send` and `receive` invocations on different channels, and guarantees that only the first one which becomes available is selected.

Each invocation is represented by a small descriptor with this `select` invocation state, which is either `PENDING` or `SELECTED`. The operation of the `select` algorithm proceeds in several phases. In the first *registration* phase the descriptor is added to all the corresponding channels as a waiter. During this phase, it can make a rendezvous with an opposite operation, become `SELECTED` and jump to the *removing* phase. If the registration completes without rendezvous, then this `select` operation is in the *waiting* phase until another opposite operation performs a rendezvous with it. In this case, the state is changed to `SELECTED` and the *removing* phase starts, when all the registered waiters for this `select` invocation are removed from other channels in order to avoid memory leaks.

A rendezvous with the `select` operation is complicated by the fact that it is (being) registered into multiple channels and can concurrently rendezvous with multiple operations of the opposite type, which can be `select` operations as well. So, rendezvous becomes a complex operation that must update multiple memory locations atomically. For this, we use descriptors similarly to the Harris multi-word CAS [7].

2 Evaluation

We implemented the proposed channel algorithm in Kotlin and Go, and compared it with the provided by these languages baseline implementations.

In our experiments we use multiple-producer single-consumer and multiple-producer multiple-consumer

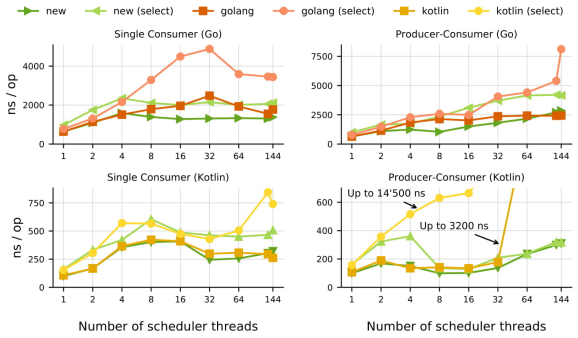

Figure 1. Performance of the proposed channel algorithm compared against Go (top) and Kotlin (bottom).

benchmarks. Since both Go and Kotlin use coroutines, we increase parallelism by increasing the maximum number of threads for the coroutines scheduler, and use the same number of coroutines (the results for bigger number of coroutines are similar, and therefore omitted). To benchmark the `select` expression, we use it to send and receive elements, receiving from a coroutine-local channel at the same time, what simulates receiving a cancellation token (this is a standard pattern in Go).

We simulate some amount of work between operations, consuming 100 CPU cycles in a non-contended local loop. We also have chosen a `NODE_SIZE` size of 32.

We evaluated the performance on a server with 4 sockets, Intel Xeon Gold 6150 (Skylake) with enabled hyper-threading in each, 144 hardware threads in total.

According to Figure 1, our `send`/`receive` algorithm shows a comparable performance on the small number of threads and beats both Go and Kotlin in case of high parallelism level, demonstrating much better scalability. The `select` expression performance has the same trends and also outperforms both implementations in almost all scenarios. However, it is a bit slower than Go in case of low concurrency due to overhead in using descriptors.

References

[1] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[2] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[3] The Go Programming Language. https://golang.org/, 2018.

[4] Kotlin Coroutines. https://github.com/Kotlin/kotlin-coroutines, 2018.

[5] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

[6] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Notices*, volume 48, pages 103–112. ACM, 2013.

[7] Timothy L Harris, Keir Fraser, and Ian A Pratt. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing*, pages 265–279. Springer, 2002.