1st October 2024

MSc in Software Engineering

Thesis Defense

Improving Parallel and Concurrent Programming in FreeST

STUDENT

Guilherme João Correia Lopes

ADVISORS

Andreia Mordido

Vasco Vasconcelos

Improve FreeST's practical parallel and concurrent capabilities!

Analysis

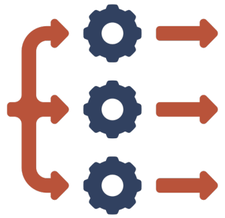
Identifying difficulties and challenges concerning usual parallel and concurrent programming practices.

Proposal

Elaboration of tools that address the identified challenges and improve the development experience.

Integration

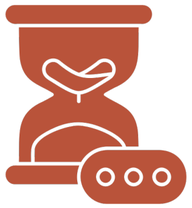
Development of tools into modules suited to the language characteristics.



Data parallelism

Addressing embarrassingly parallel problems

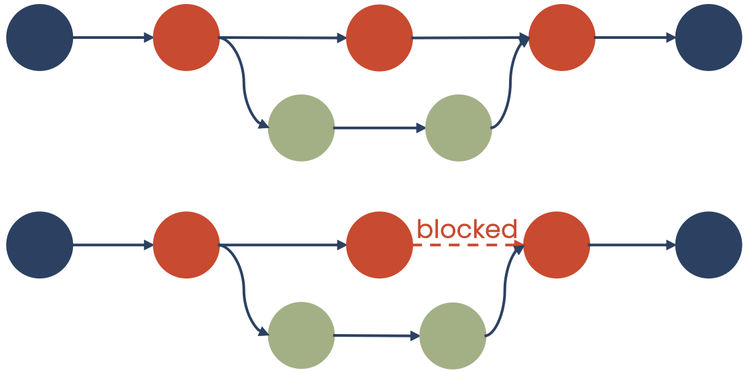
- Data is divided into subsets and processed simultaneously by multiple processors.
- An embarrassingly parallel problem where minimal effort is required to separate the problem into multiple parallel tasks.

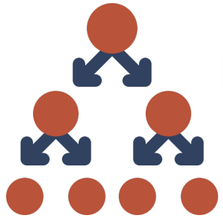


Futures

Asynchronous placeholders for pending results

- The representation of a promise to deliver the value of an expression at some later time.





Divide-and-conquer

Divide a problem, solve subproblems and merge solutions

1. **Divide:** Break down a problem into smaller subproblems.
2. **Conquer:** Solve the subproblems recursively.
3. **Merge:** Combine the results into the solution.

Streams

Efficient handling of continuous data processing

- Flows of data elements that can be continuously and sequentially read from or written to.
- **Filters:** Functions that process data by maintaining endpoints to send and receive data elements within streams.
- **Pipelines:** Multiple filters linked in a sequence.

MPI

- A message-passing specification.
- Facilitates data exchange between processes through cooperative operations.
- Harnesses data parallelism: optimal for embarrassingly parallel problems.

ForkJoin

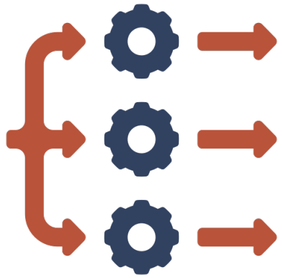
- A concurrent divide-and-conquer technique.
- Uses fork to divide and join to merge, which are analogous to futures.
- Implements a work-stealing scheduler.

StreamIt

- A programming language purposely designed for efficient stream programming.
- Uses standard stream mechanisms such as **filters** and **pipelines**.
- Additionally, it provides **splitters**, which distribute data from one filter to other two.

FreeST: What is it?

Functional



Concurrent

Message-passing

asynchronous communication

Session types

context-free

Session Types

- Focus on binary (two-party) sessions by well-defined protocols between endpoints.
- Guarantees that all agents involved in a communication strictly follow a protocol, ensuring reliable communication.
- Compile-time!

```
type ClientChannel = !Int;?Int
type ServerChannel = ?Int;!Int
```

```
type ClientChannel = +{IsOdd: !Int;?Boolean,
                       Succ : !Int;?Int}
type ServerChannel = &{IsOdd: ?Int;!Boolean,
                       Succ : ?Int;!Int}
```

Session Types

- Focus on binary (two-party) sessions by well-defined protocols between endpoints.
- Guarantees that all agents involved in a communication strictly follow a protocol, ensuring reliable communication.
- Compile-time!

```
type ClientChannel = !Int;?Int
type ServerChannel = dualof ClientChannel
```

```
type ClientChannel = +{IsOdd: !Int;?Boolean,
                       Succ : !Int;?Int}
type ServerChannel = dualof ClientChannel
```

new
create communication channels

fork
launch asynchronous computation

send & receive
… information through channels

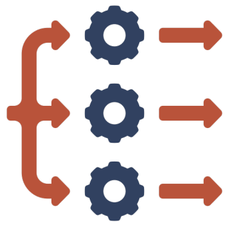
select & match
… a branch in a channel's protocol

FreeST is not simple!

- **Maturity:** FreeST is a new programming language in an early stage of development.
- **Accessibility:** Its features and type system might impose difficulties for newcomers (e.g., linearity).
- **Flexibility:** It has a limited set of tools, restricting its real-world utility in parallel and concurrent scenarios.

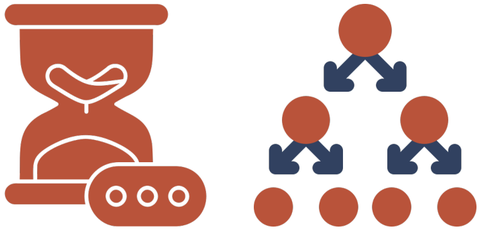
How can we mitigate these challenges?

- **Three new modules,** each providing user-friendly environments and abstractions for different parallel and concurrent programming concepts previously discussed.



Parallel module

A parallel and concurrent programming environment that addresses data parallelism.

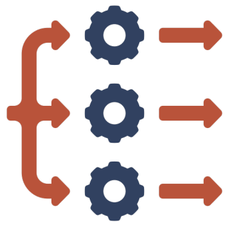


Futures module

Implements futures and allows divide-and-conquer algorithms.

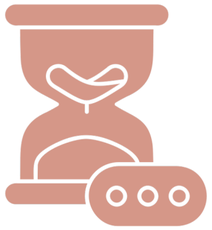
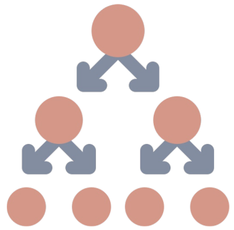
Streams module

Provides a set of abstractions suited for stream programming.



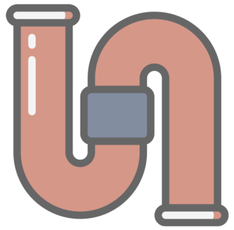
Parallel module

A parallel and concurrent programming environment that addresses data parallelism.

Futures module

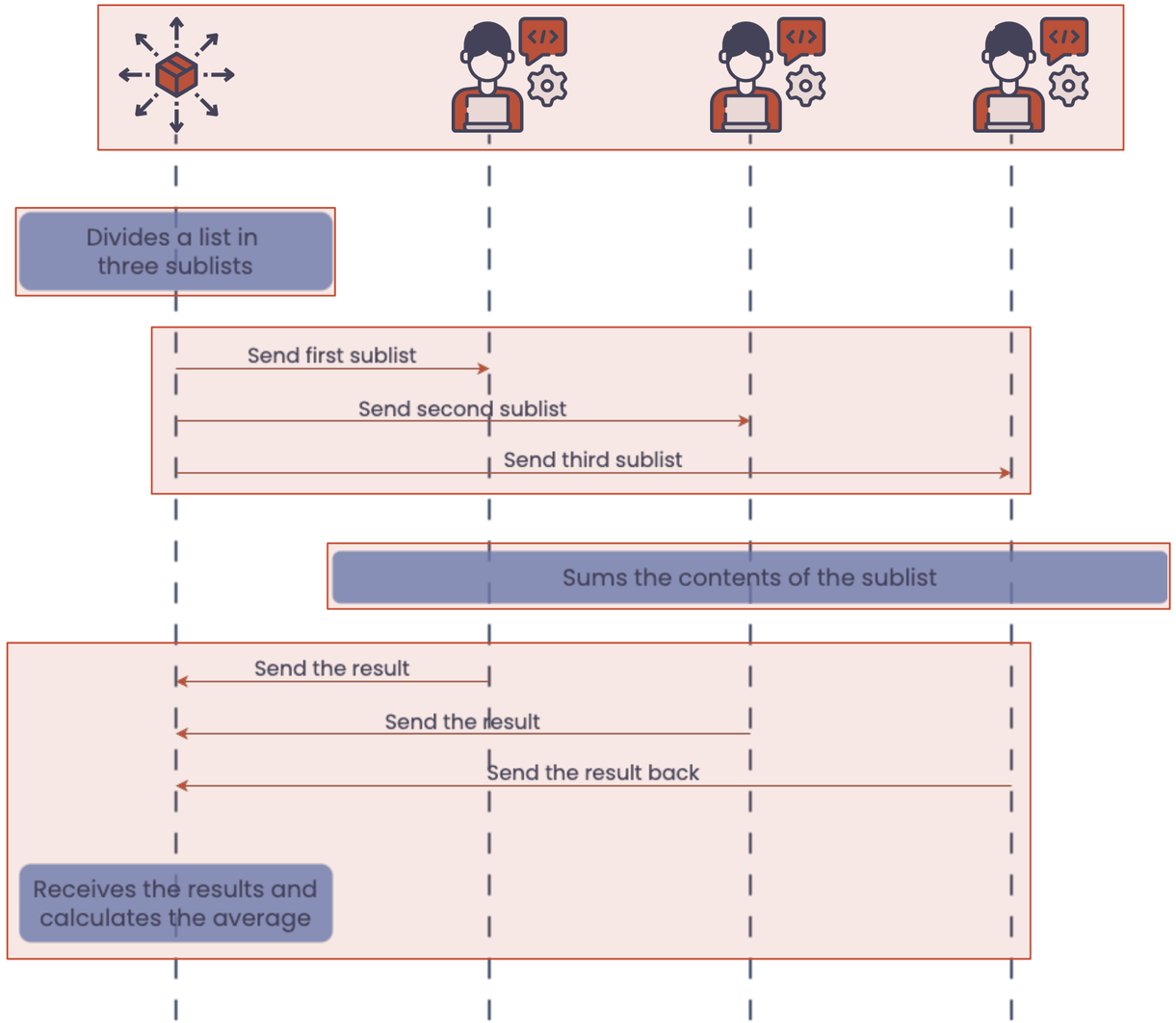
Implements futures and allows divide-and-conquer algorithms.



Streams module

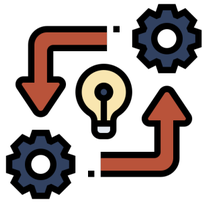
Provides a set of abstractions suited for stream programming.

Parallel module: Parallel average example




```
type ParallelStream = ![Int];?Int;Wait

process : dualof ParallelStream -> ()
process c = let (xs, c) = receive c in
            c |> send (sum xs) |> close

parallelAverage : [Int] -> ParallelStream -> ParallelStream
                        1-> ParallelStream 1-> Int
parallelAverage xs w1 w2 w3 =
        let (xs, ys) = splitAt 3 xs in
        let (ys, zs) = splitAt 3 ys in

        let w1 = send xs w1 in
        let w2 = send ys w2 in
        let w3 = send zs w3 in

        let (x1, w1) = receive w1 in
        let (x2, w2) = receive w2 in
        let (x3, w3) = receive w3 in
        wait w1; wait w2; wait w3;

        (x1 + x2 + x3) / 9

main : Int
main = let w1 = forkWith process in
       let w2 = forkWith process in
       let w3 = forkWith process in
       parallelAverage[1,2,4,8,16,32,64,128,256] w1 w2 w3
```

Parallel module: Motivation



Repetitive

could benefit from
some abstraction

Unscalable

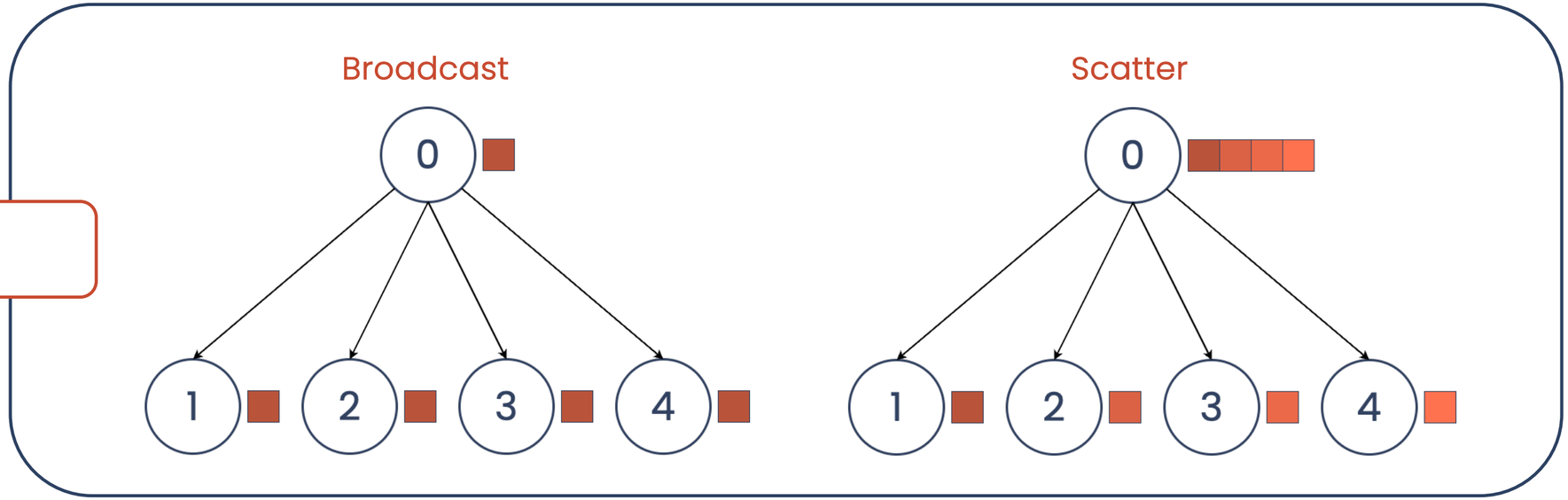
not ready to expand
the workforce

Complex

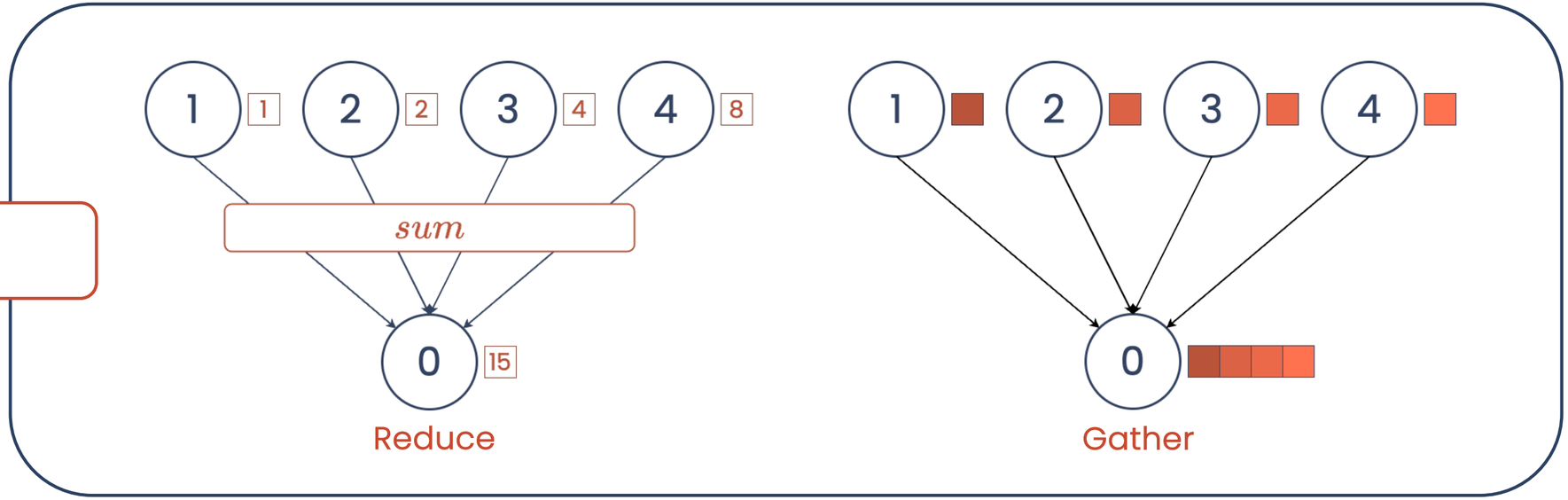
simple problem,
unintuitive implementation

```
type ParallelStream = ![Int];?Int;Wait

process : dualof ParallelStream -> ()
process c = let (xs, c) = receive c in
            c |> send (sum xs) |> close

parallelAverage : [Int] -> ParallelStream -> ParallelStream
                        1-> ParallelStream 1-> Int
parallelAverage xs w1 w2 w3 =
        let (xs, ys) = splitAt 3 xs in
        let (ys, zs) = splitAt 3 ys in

        let w1 = send xs w1 in
        let w2 = send ys w2 in
        let w3 = send zs w3 in

        let (x1, w1) = receive w1 in
        let (x2, w2) = receive w2 in
        let (x3, w3) = receive w3 in
        wait w1; wait w2; wait w3;

        (x1 + x2 + x3) / 9

main : Int
main = let w1 = forkWith process in
       let w2 = forkWith process in
       let w3 = forkWith process in
       parallelAverage[1,2,4,8,16,32,64,128,256] w1 w2 w3
```

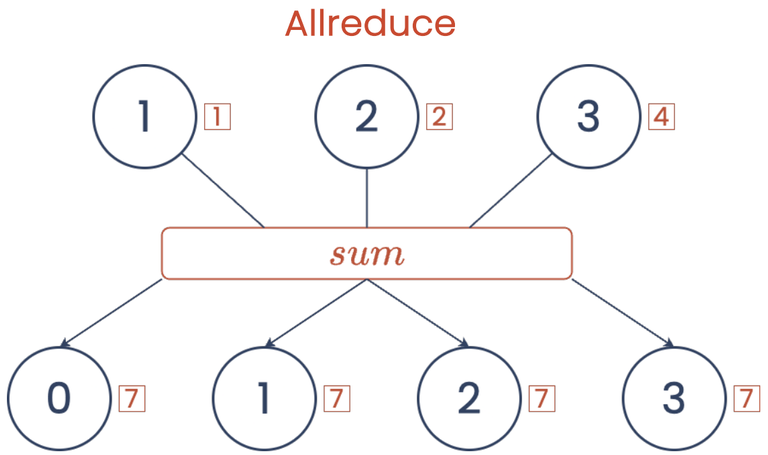
Distribution

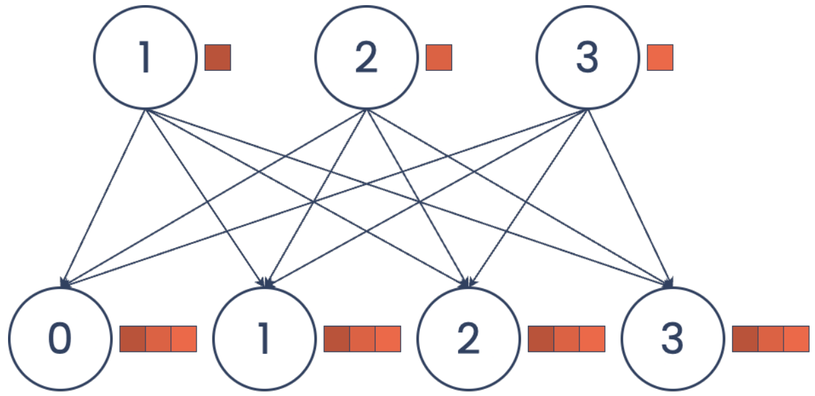


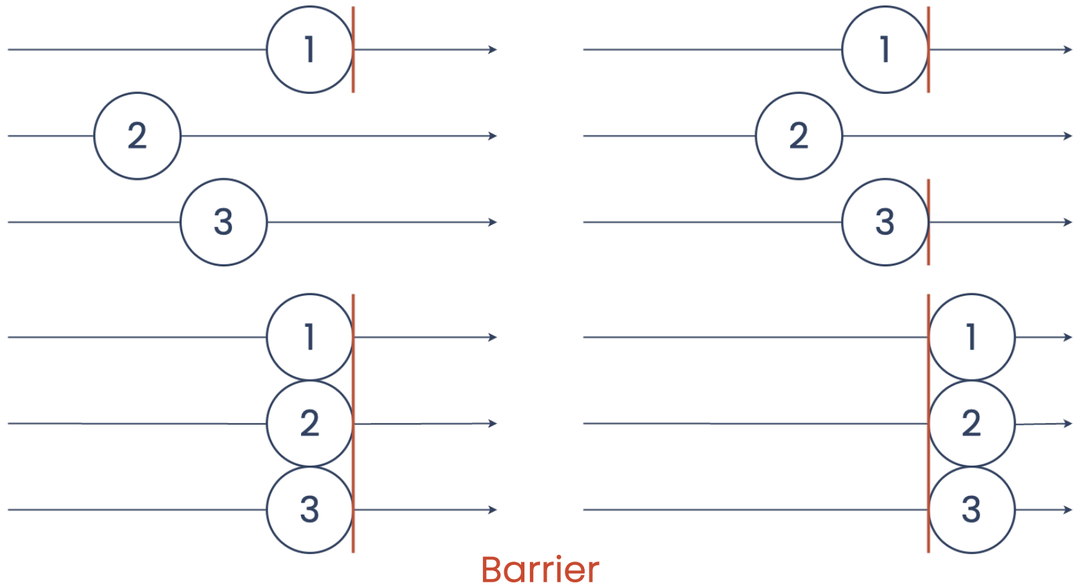
Gathering



Extra


Allreduce
1
1
2
2
3
4
sum
0
7
1
7
2
7
3
7


Allgather
1
2
3
0
1
2
3


1
2
3
1
2
3
1
2
3
1
2
3
Barrier

Parallel module: Implementation

Session types to outline the communication:

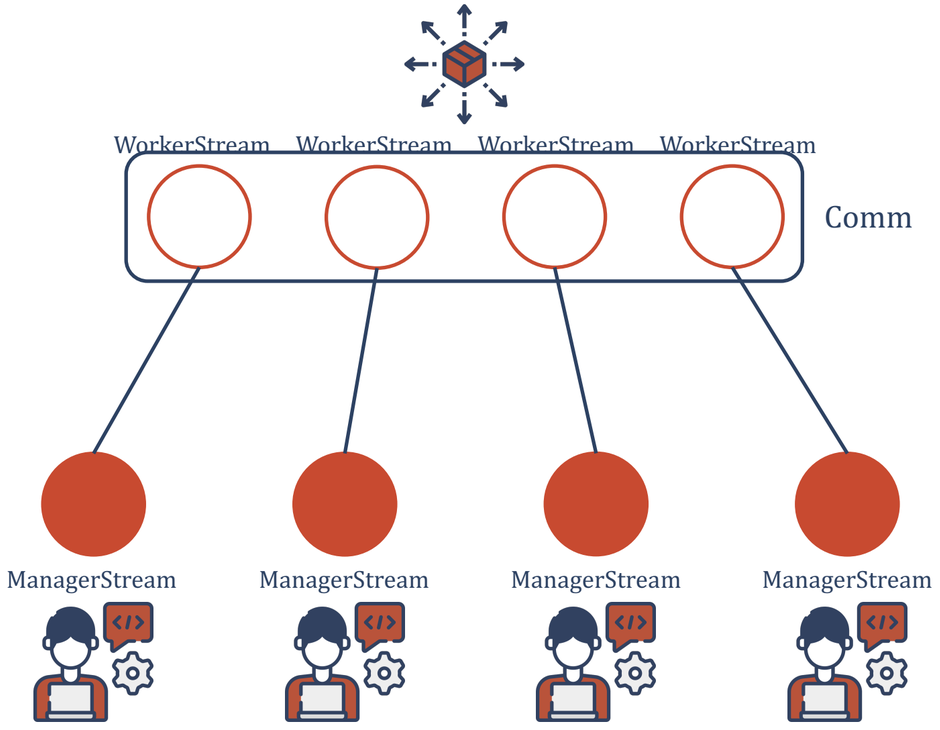
```
type ManagerStream = +{ Broadcast: ![Int]
                      , Scatter  : ![Int]
                      , Gather   : ?[Int]
                      , Reduce   : ?Int
                      , Done     : Wait} ; ManagerStream

type WorkerStream = dualof ManagerStream
```

List of endpoints to communicate with the workers:

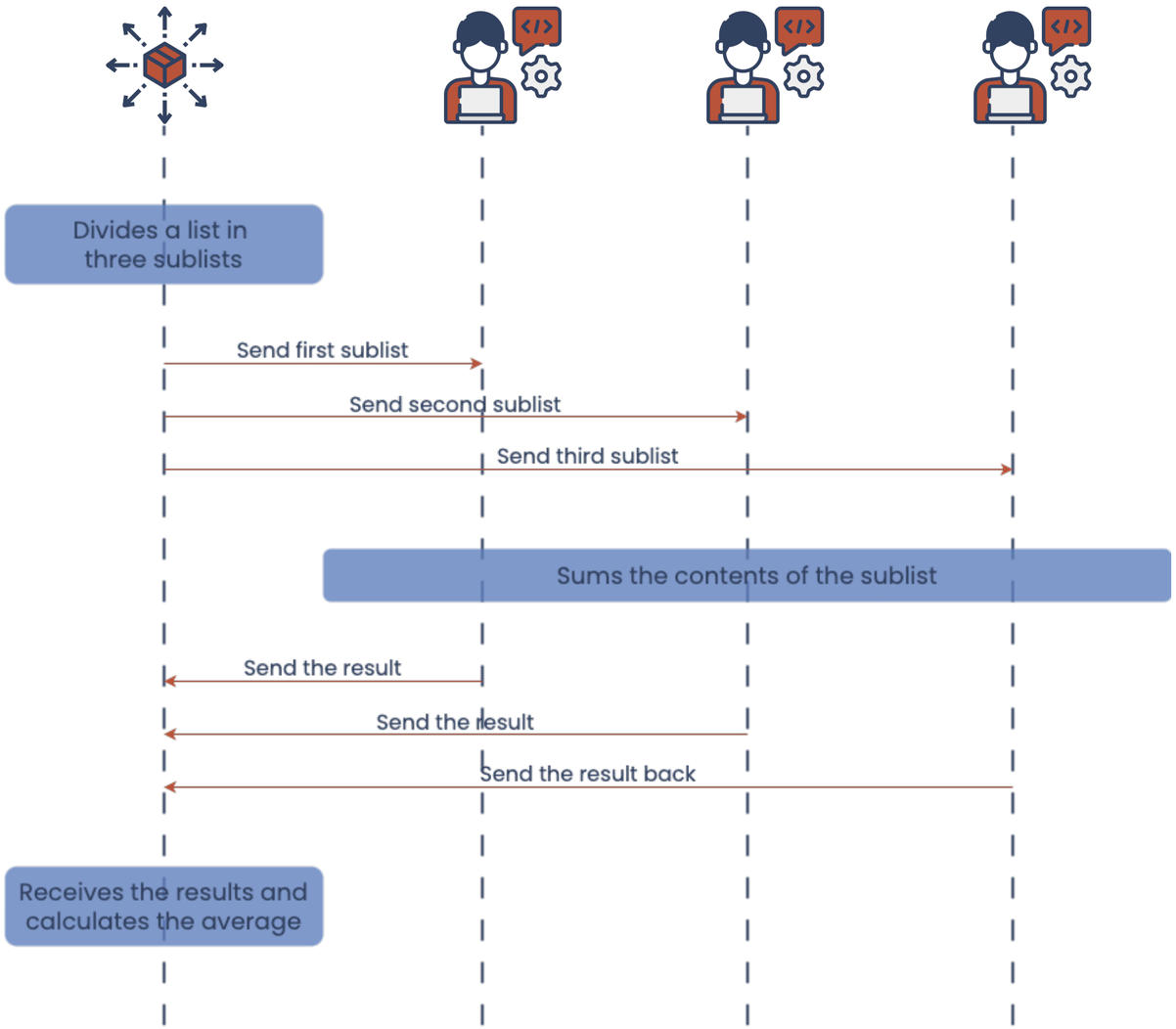
```
data Comm = WNil () | Worker ManagerStream Comm
```

Initialize the manager-workers communication framework:

```
initialize : (Comm -> a) -> (WorkerStream -> ())
                         -> Int -> a
```

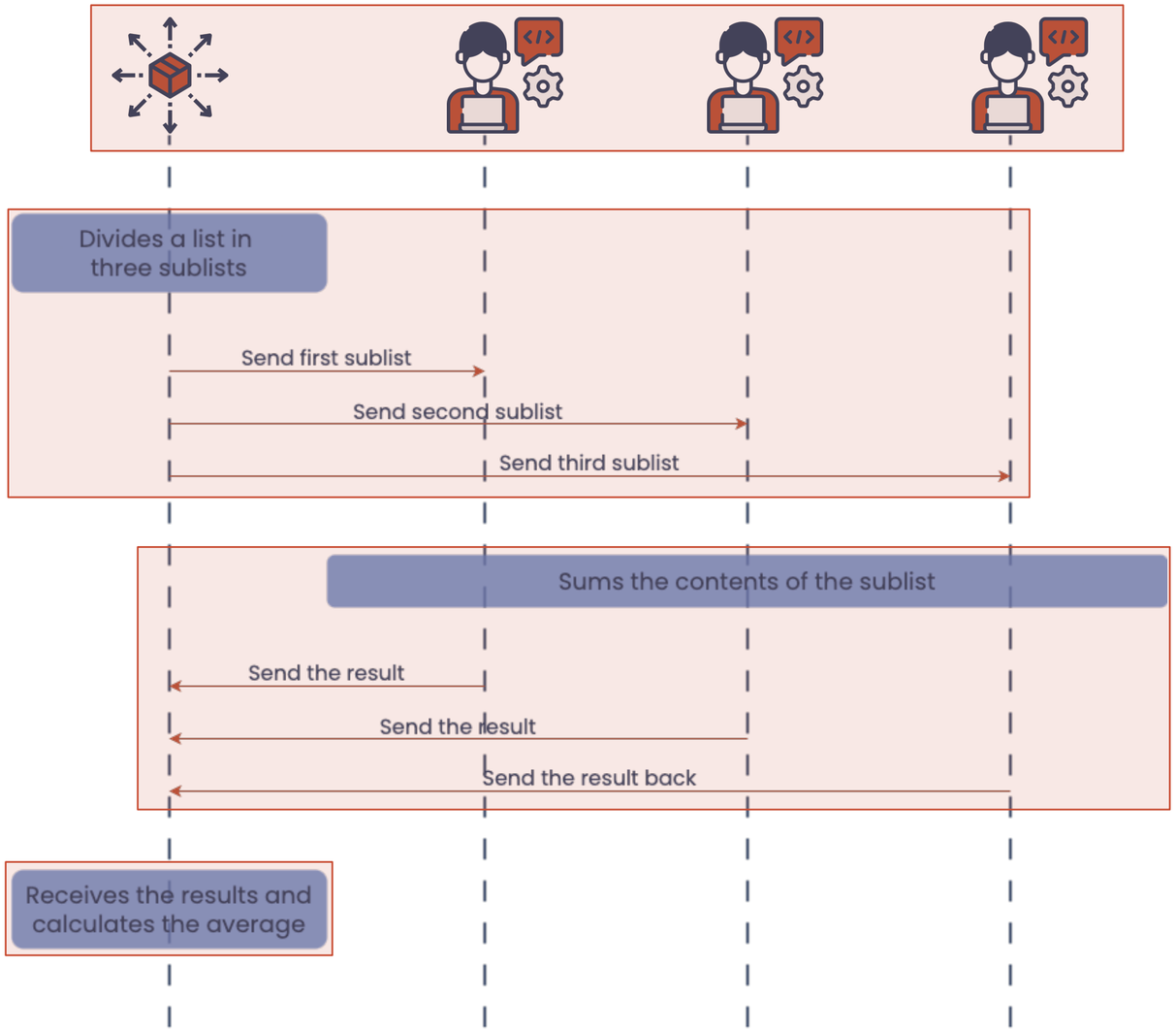


Parallel module: Revisiting the parallel average




```
type ParallelStream = ![Int];?Int;Wait

process : dualof ParallelStream -> ()
process c = let (xs, c) = receive c in
            c |> send (sum xs) |> close

parallelAverage : [Int] -> ParallelStream -> ParallelStream
                        1-> ParallelStream 1-> Int
parallelAverage xs w1 w2 w3 =
        let (xs, ys) = splitAt 3 xs in
        let (ys, zs) = splitAt 3 ys in

        let w1 = send xs w1 in
        let w2 = send ys w2 in
        let w3 = send zs w3 in

        let (x1, w1) = receive w1 in
        let (x2, w2) = receive w2 in
        let (x3, w3) = receive w3 in
        wait w1; wait w2; wait w3;

        (x1 + x2 + x3) / 9

main : Int
main = let w1 = forkWith process in
       let w2 = forkWith process in
       let w3 = forkWith process in
       parallelAverage[1,2,4,8,16,32,64,128,256] w1 w2 w3
```

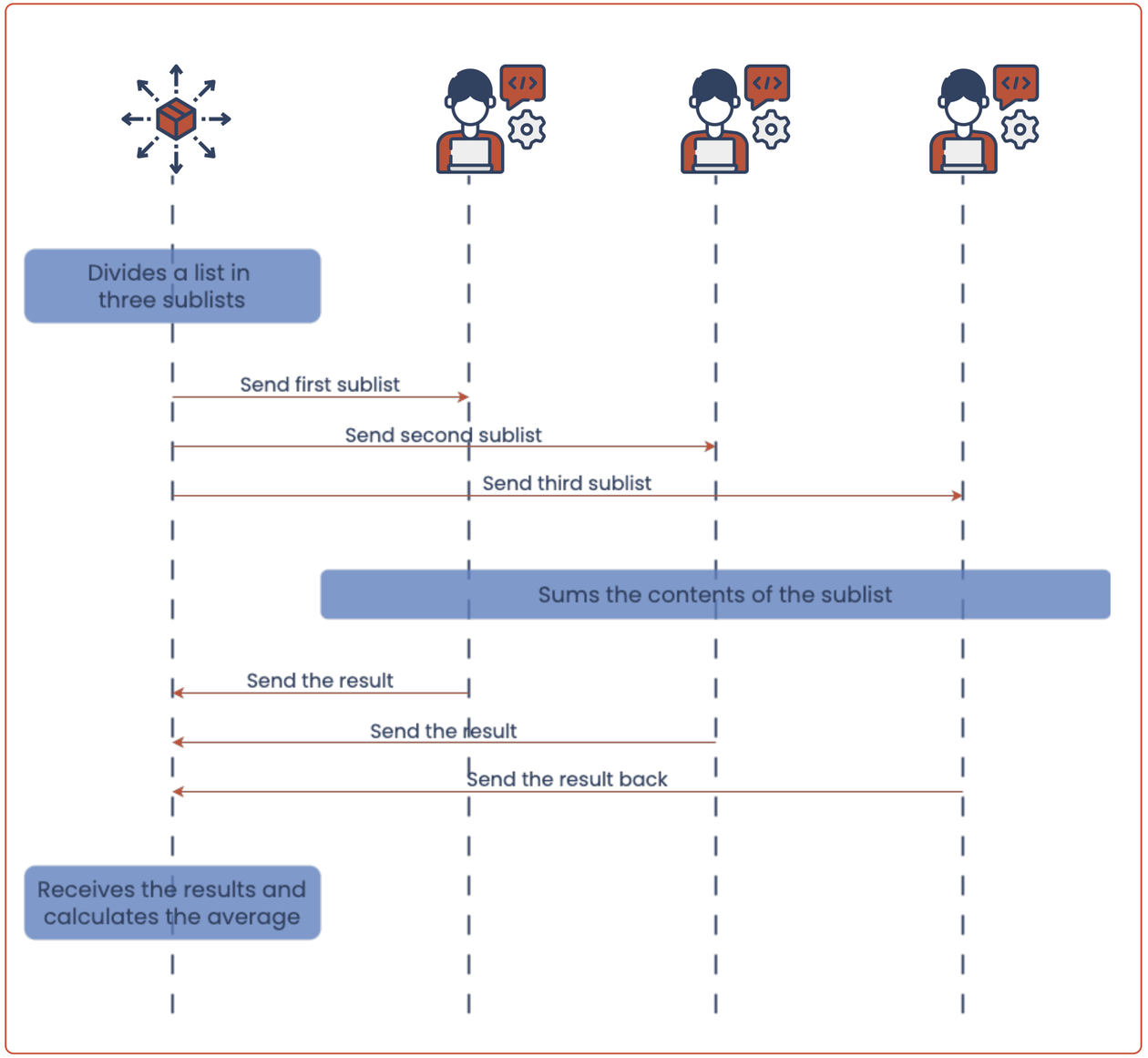
Parallel module: Revisiting the parallel average




```
import Parallel

worker : WorkerStream -> ()
worker c = let (xs, c) = wscatter c in
           c |> wreduce (sum xs) |> wdone

manager : [Int] -> Comm -> Int
manager xs comm = let xsl = length xs in
                  let comm = mscatter xs comm in
                  let (xs, comm) = mreduce (+) 0 comm in
                   mdone comm; xs / xsl

main : Int
main = let xs = [1, 2, 4, 8, 16, 32, 64, 128, 256] in
       initialize (manager xs) worker 3
```

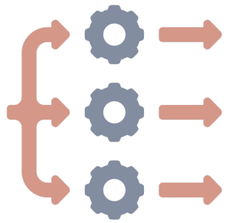
Note: The manager selects operations via m prefix; the worker matches with w prefix.

Parallel module: Revisiting the parallel average



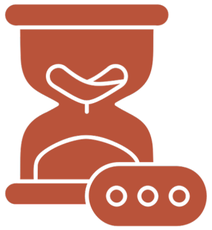

```
.
.
.

nfoldl : Int -> (Int -> Int -> Int) -> Int -> [Int] -> Int
nfilter : Int -> (Int -> Bool) -> [Int] -> [Int]
nzipWith : Int -> (Int -> Int -> Int) -> [Int] -> [Int] -> [Int]
```

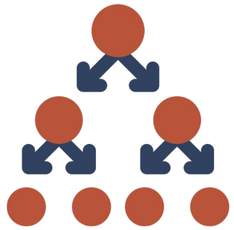
```
import Parallel

main : Int
main = let xs = [1, 2, 4, 8, 16, 32, 64, 128, 256] in
       nfoldl 3 (+) 0 xs / length xs
```



Parallel module

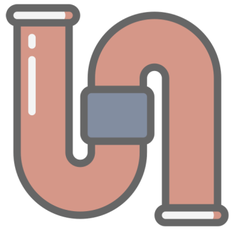
A parallel and concurrent programming environment that addresses data parallelism.





Futures module

Implements futures and allows divide-and-conquer algorithms.



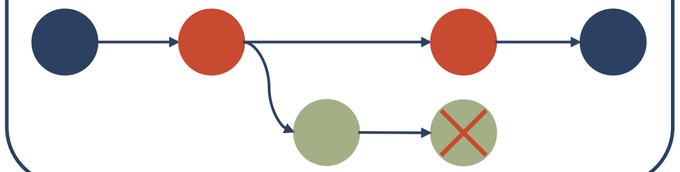
Streams module

Provides a set of abstractions suited for stream programming.

How can we retrieve the result of an asynchronous computation?

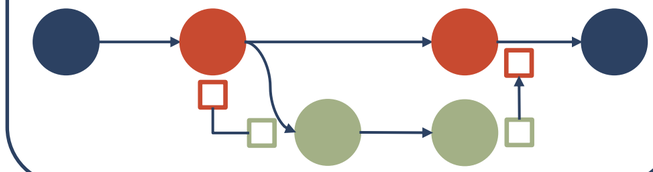
Problem

The fork primitive discards the result.



Work-around

Create an extra channel to exchange the result.



Could we avoid this hassle? YES!

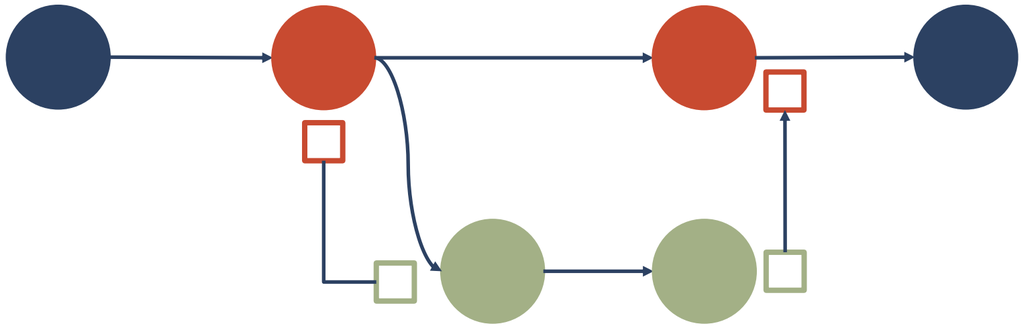
Futures

A mechanism that wraps the fork primitive by abstracting the creation of an extra channel, simplifying retrieving the result of an asynchronous computation.

Futures: Implementation

Launch an asynchronous computation:

```
future : (() -> a) -> ?a;Wait
```

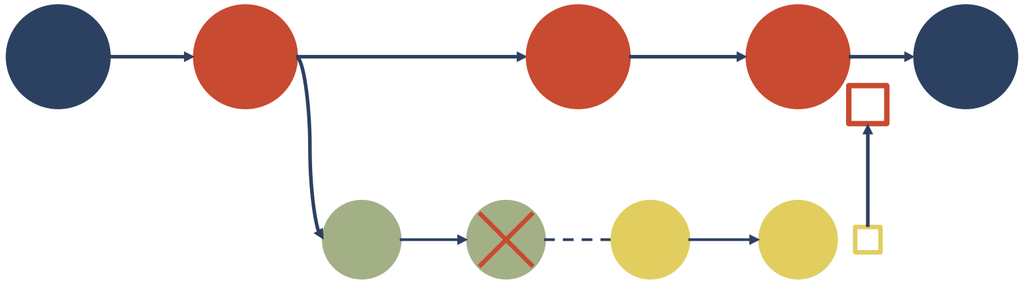


Retrieve the result of the computation:

```
block : ?a;Wait -> a
```

Asynchronously delay a computation:

```
delay : (() -> a) -> (() -> b)
                  -> ?b;Wait
```



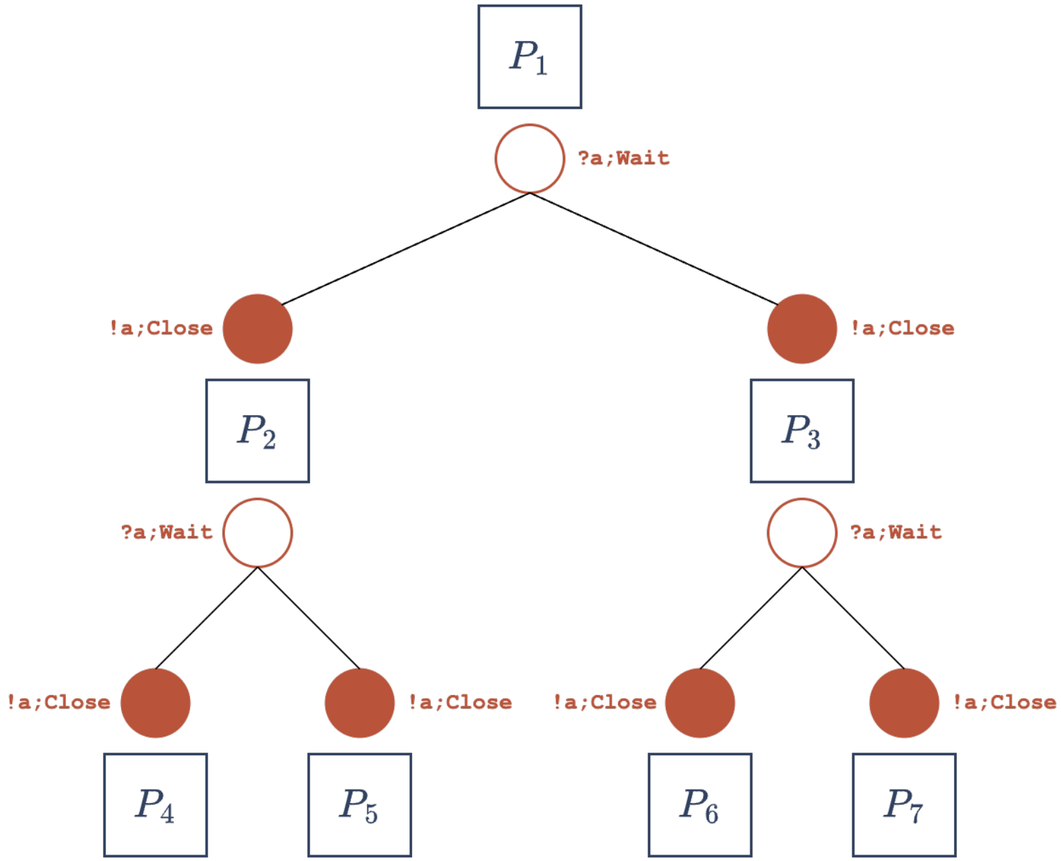
Futures: Divide-and-conquer

Launch an asynchronous computation:

```
future : (() -> a) -> ?a;Wait
```

Retrieve the result of the computation:

```
block : ?a;Wait -> a
```



Futures: Fibonacci sequence

Example

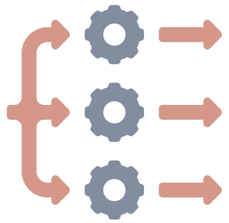
The Fibonacci sequence defines each number as the sum of the two preceding ones.

$$F_0 = 0, \qquad F_1 = 1,$$

$$F_n = F_{n-1} + F_{n-2}$$

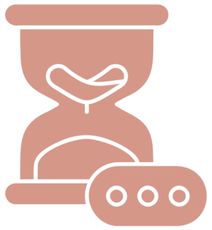
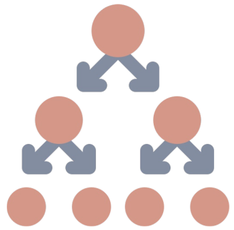
Implementation with futures

```
pFib : Int -> Int
pFib n | n == 0    = 0
       | n == 1    = 1
       | otherwise = let f1 = future (\_:() -> pFib (n - 1)) in
                     let f2 = future (\_:() -> pFib (n - 2)) in
                     block f1 + block f2
```



Parallel module

A parallel and concurrent programming environment that addresses data parallelism.

Futures module

Implements futures and allows divide-and-conquer algorithms.

Streams module

Provides a set of abstractions suited for stream programming.

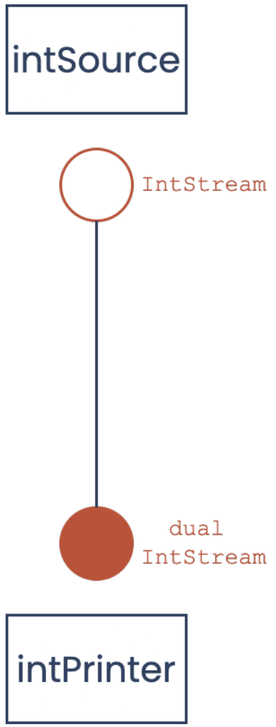
Session types and channels allow for a natural way of writing and handling streams in FreeST!

Example of a stream of integers

```
type IntStream = +{More: !Int;IntStream, Done: Wait}
```

Example of a stream program

```
main : ()
main = let (w, r) = new @IntStream () in
       fork (\_:() -> intSource w);
       intPrinter r
```



Streams: Implementation

Session types for streams:

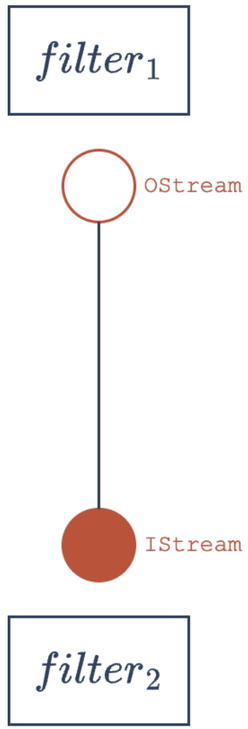
```
type OStream = +{More: !Int;OStream, Done: Wait}
type IStream = dualof OStream
```

Basic operations on streams:

```
sendS : Int -> OStream -> OStream
waitS : OStream -> ()
forward : IStream -> OStream 1-> OStream
```

List related operations on streams:

```
fromList : [Int] -> OStream -> ()
toList : IStream -> [Int]
```



Definition

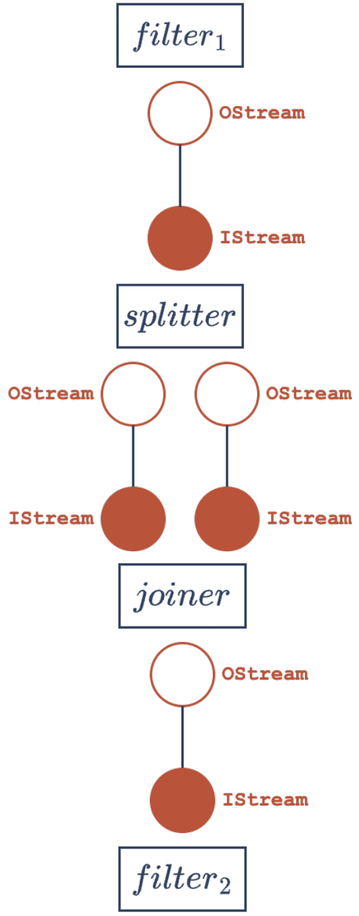
Splitters distribute data from a stream between two streams.

Divide a stream into two:

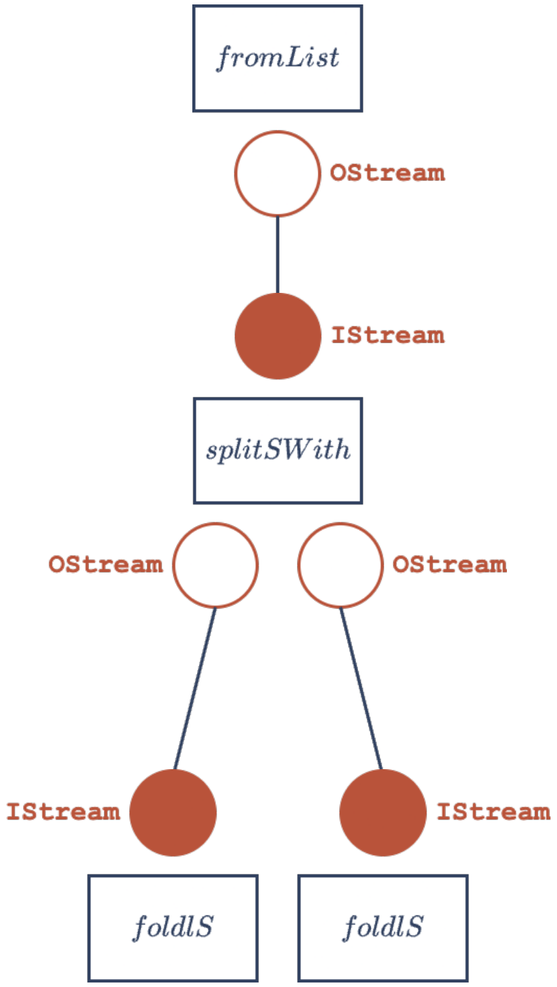
```
splitSDup : IStream -> OStream 1-> OStream 1-> ()

splitSAlt : IStream -> OStream 1-> OStream 1-> ()

splitSWith : (Int -> Bool) -> IStream -> OStream
                                1-> OStream 1-> ()
```

Merge two streams into one:

```
joiner : IStream -> IStream 1-> OStream 1-> ()
```



Streams: Example




```
                                .
                                .
                                .
mapS : (Int -> Int) -> IStream -> OStream 1-> ()
filterS : (Int -> Bool) -> IStream -> OStream 1-> ()
foldlS : (Int -> Int -> Int) -> Int -> IStream -> Int
```

```
import Streams

foo : [Int] -> (Int, Int)
foo xs = let i1 = forkWith (fromList xs) in
         let (i2, i3) = forkWith2 (splitSWith even i1) in
         (foldlS (+) 0 i2, foldlS (+) 0 i3)

main : (Int, Int)
main = foo [1, 6, 23, 82, 34, 2, 5, 44, 302, 48, 17, 3]
```

Evaluation: Surveys

Sample

6 participants

Duration

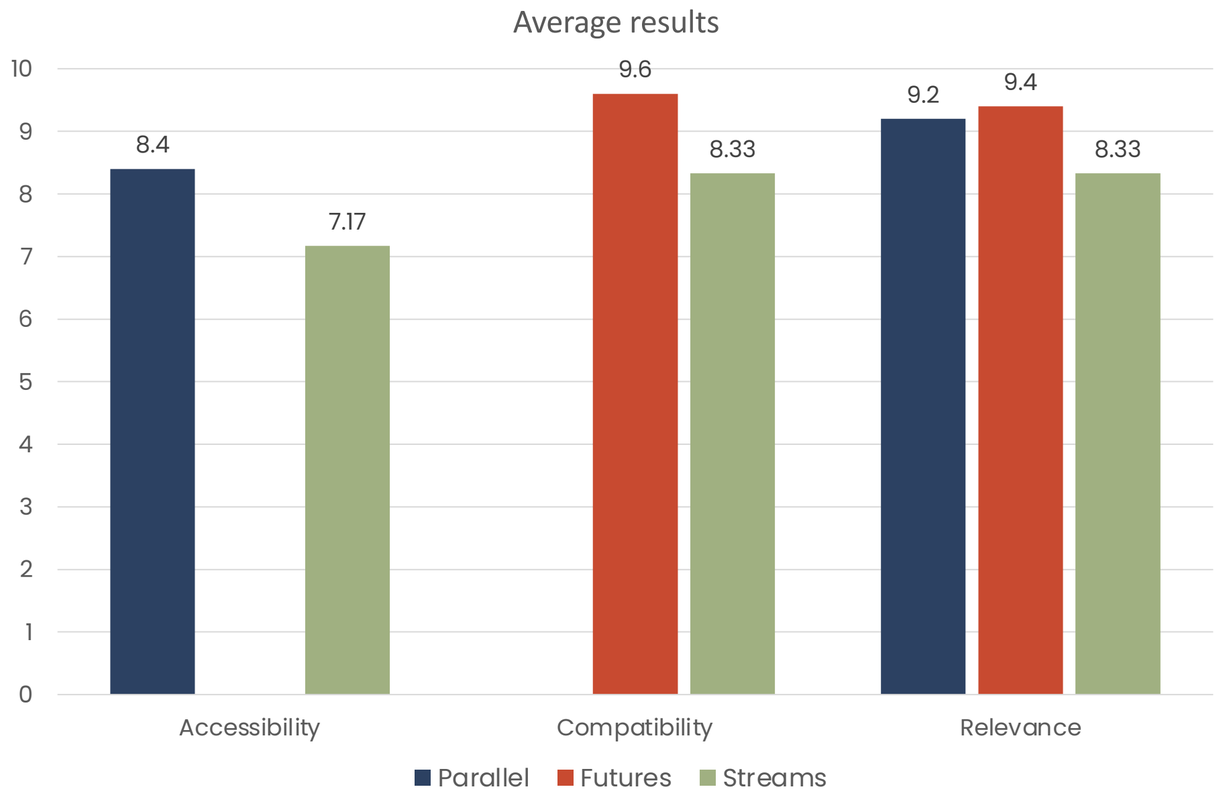
2024 March–June

Structure

1. Overview
2. Examples
3. Exercise
4. Feedback

Feedback

- 1 to 10 evaluation concerning several parameters.
- Open-ended question.



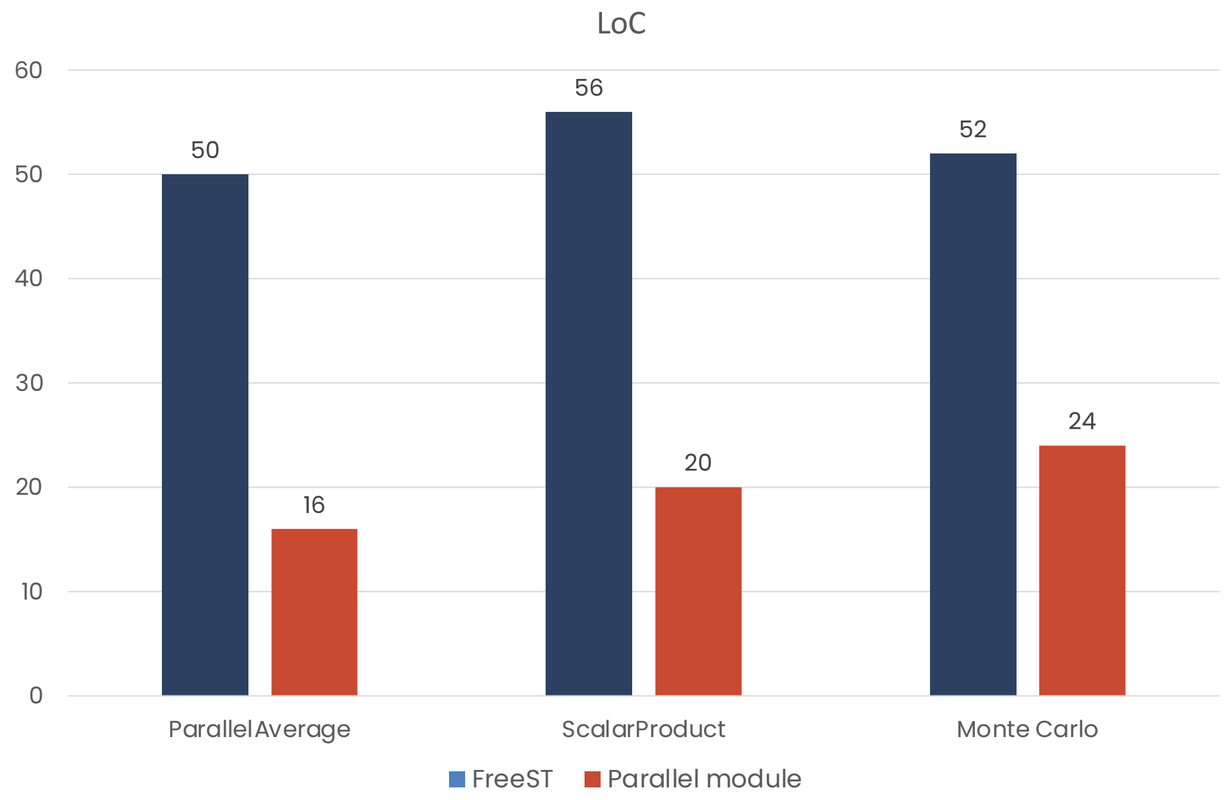
Evaluation: Lines of Code

Target

We only submit the Parallel module to this analysis.

Goal

LoC comparison between examples implemented with FreeST and with the Parallel module.



Thank you!

Questions

Answers

R-Q&A