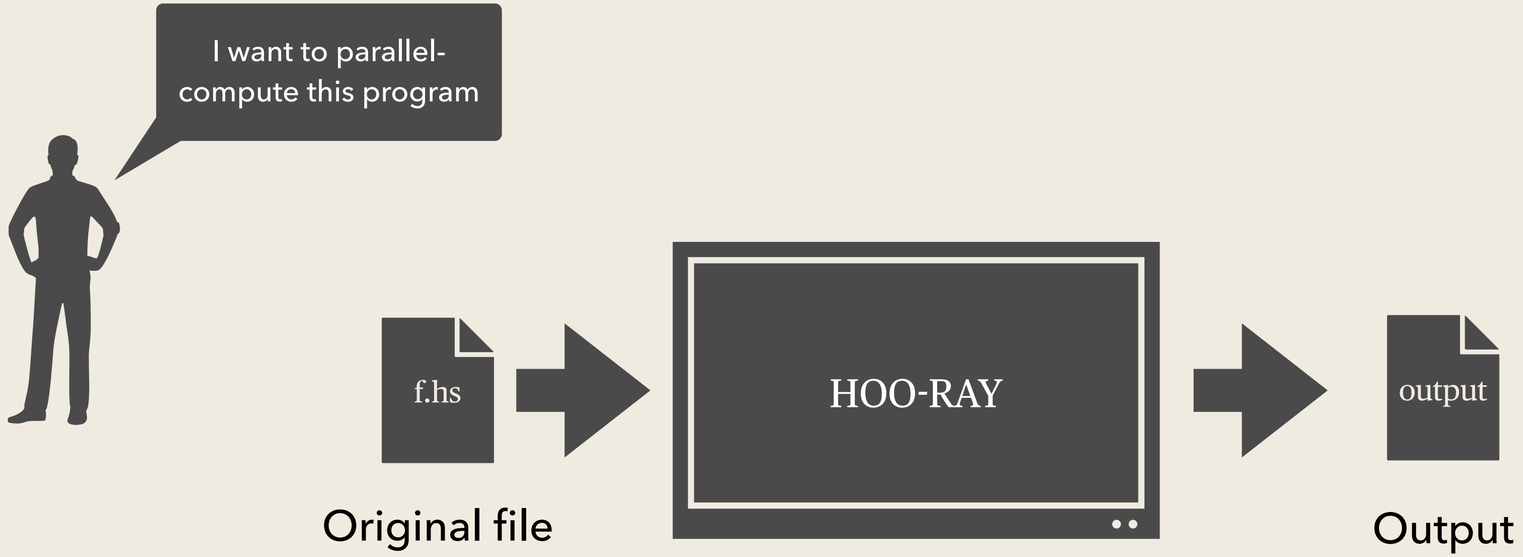
JERRY HOU, JADEN LONG, TONY WU, DENNIS XU

Hoo-Ray

A Distributed Execution Engine for Haskell, Written in Haskell

Workflow



A programmer with <u>little knowledge</u> about how the underlying function calls work can *parallelize*.

Haskell

Distributing Pure Functions

- Pure functions: Functions that
  1. Have consistent output given the same inputs
  2. Have no side effects
- Pure functions can be executed *in any order*.
  - The underlying idea behind MapReduce and Spark

Pure Functions

- Hard to identify in e.g. Java, Python
  - Especially since functions can be nested

```python
def pure1():
    return 3.14

def pure2(a, b):
    return a + b

def pure3(l: list):
    return sorted(l)
```

```python
def impure1():
    return time.time()

def impure2(a):
    print(a)

def impure3():
    with open('file.txt') as f:
        text = f.read()
    return text
```

Examples of pure and impure functions in Python

What if there is a language that enforces purity?

Pure Functions

- Every call to a pure function can be distributed
- We only have to enforce linearizability of impure operations
- Any program can be easily distributed!

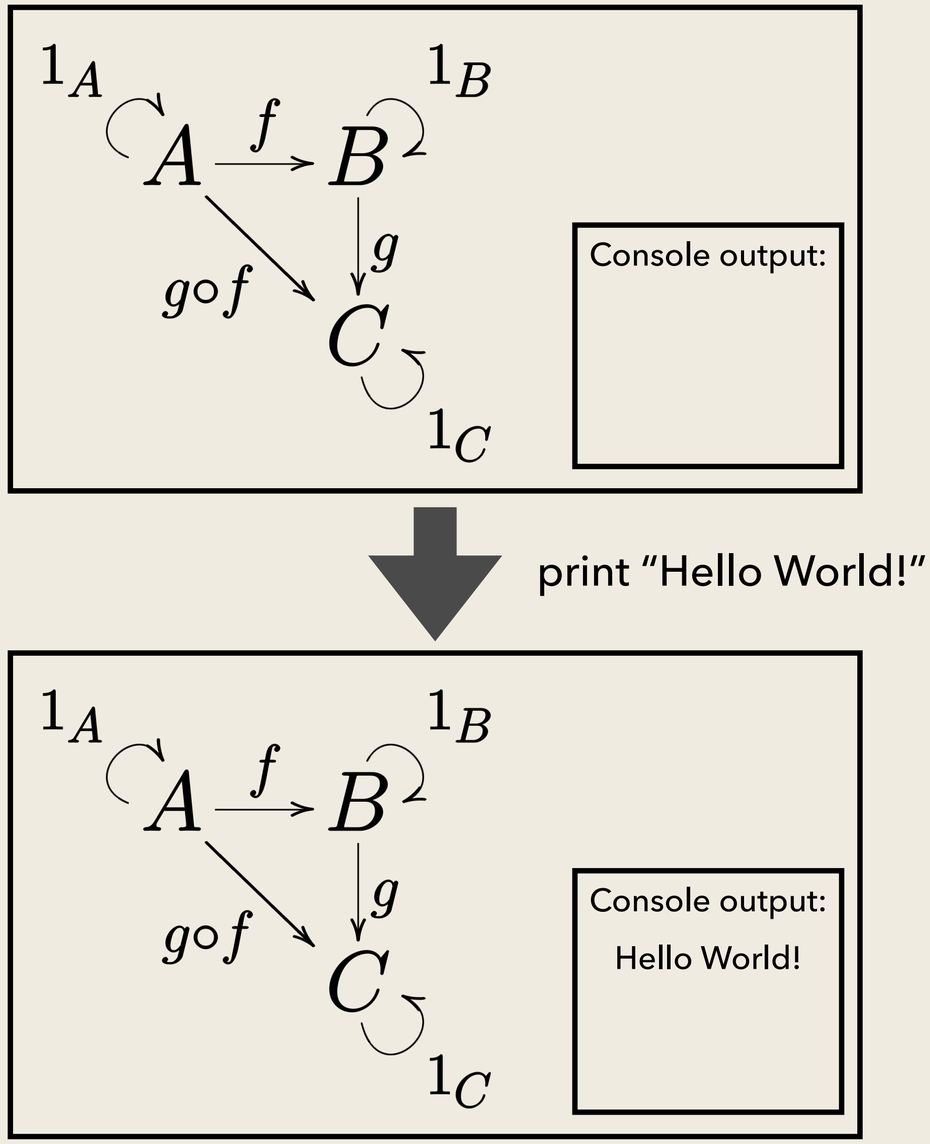
Functional Programming

- No classes or objects, just functions
- Variables are immutable
- Functions are “pure” by default

OO pattern/principle	FP pattern/principle
• Single Responsibility Principle	• Functions
• Open/Closed principle	• Functions
• Dependency Inversion Principle	• Functions, also
• Interface Segregation Principle	• Functions
• Factory pattern	• Yes, functions
• Strategy pattern	• Oh my, functions again!
• Decorator pattern	• Functions
• Visitor pattern	• Functions

(Im)Pure Functions in Haskell

- Technical note:
  - Functions are considered *morphisms* in the **Hask** category.
  - Impure functions are wrapped in *monads*, which are just *endofunctors* with two *natural transformations*.
  - When you say, write to a file or log to the console, you are actually *"teleported"* to a new world with these actions already done.



(Im)Pure Functions in Haskell

- Purity can be directly inferred from function signature.
- Impure functions in Haskell have a monadic return type.
- e.g. IO operations are wrapped in the IO monad

```
def impure1():
    return time.time()

def impure2(a):
    print(a)

def impure3():
    with open('file.txt') as f:
        text = f.read()
    return f
```

(Im)Pure Functions in Haskell

- Purity can be directly inferred from function signature.
- Impure functions in Haskell have a monadic return type.
- e.g. IO operations are wrapped in the IO monad

```haskell
import Data.Time.Clock (getCurrentTime)

impure1 :: IO UTCTime
impure1 = getCurrentTime

impure2 :: Show a => a -> IO ()
impure2 x = print x

impure3 :: IO String
impure3 = do
    handle <- openFile "file.txt" ReadMode
    hGetContents handle
```

(Im)Pure Functions in Haskell

- When executing in main, pure and impure functions are called in different ways

```haskell
main :: IO ()
main = do
    result1 <- impure1
    let result2 = pure1 10 20
```

Existing Work

Previous Works (Haskell)

- **Par** monad:
  - A work-stealing scheduler for pure functions
  - Only for shared-memory machine
  - Same underlying idea as ours
- Cloud Haskell
  - Libraries and APIs for network transport, process spawning, etc.

Previous Works (Haskell)

- Parallel Haskell
  - A work-stealing scheduler for pure functions
  - Only for shared-memory machine
- Cloud Haskell
  - Low-level APIs for network transport, process spawning, etc.
- Par monad
  - A library providing APIs for parallel programming
- Same underlying idea as ours

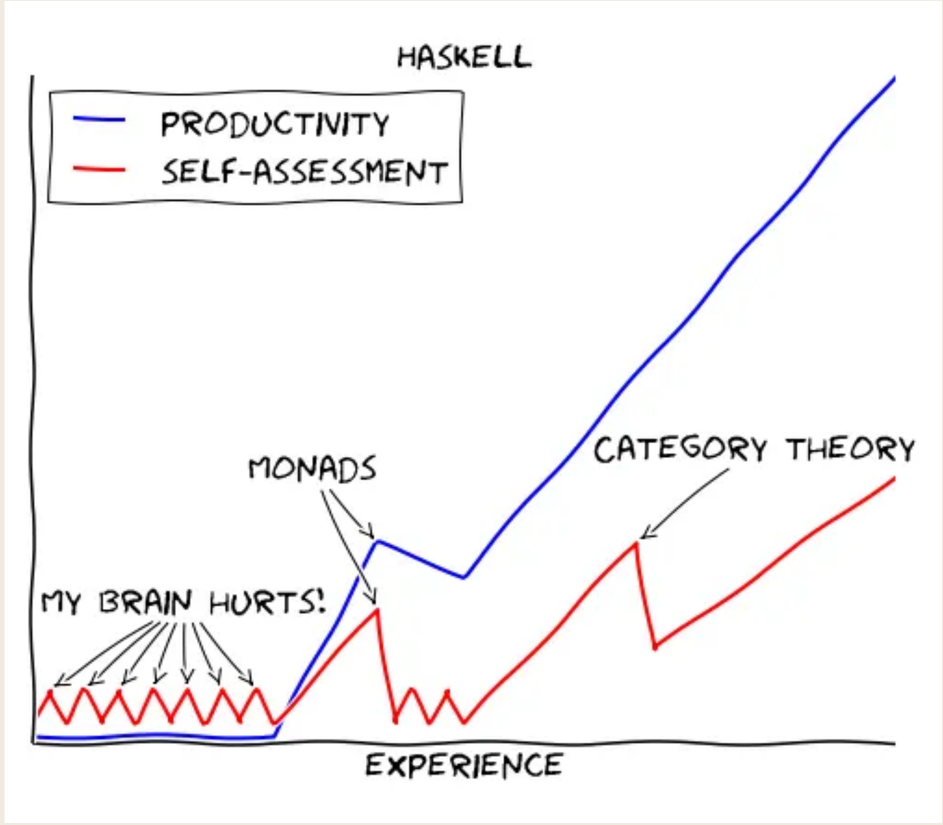
Hoo-Ray

A Distributed Execution Engine for Haskell,
Written in Haskell

Challenges

Functional Programming is an entirely new paradigm!

- None of us had experience in Haskell before this project!
  - Hard to wrap head around monads
  - Lack of mutable states presents a challenge to write!
- Jaden assigned weekly tasks for onboard with Haskell.
- *Through hardships to the stars.*



Design Overview
- Assumptions, API, Architecture, Consistency Model

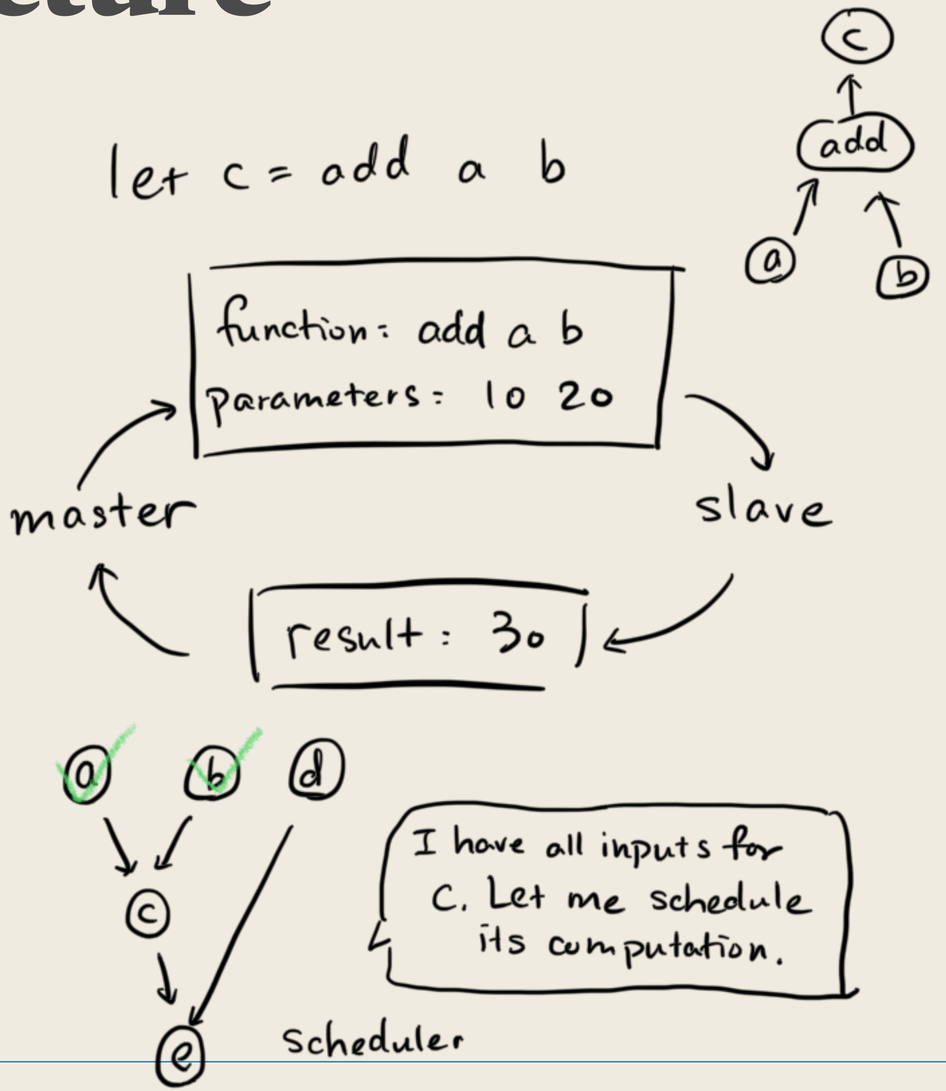
Architecture

HOO-RAY

Syntax tree parser

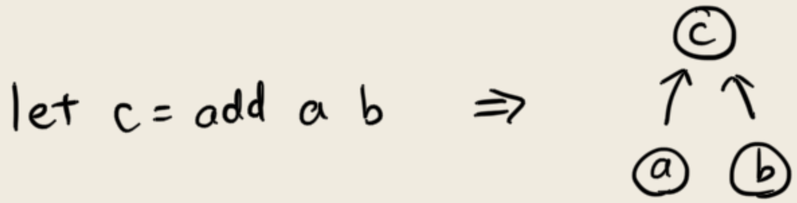
Function Serializer

Greedy Scheduler



Implementation

Syntax tree parser



- Parse the main function into an AST representation
- Traverse through the trees to collect function, variable, and argument bindings
- Ensure function names are unique and distinguish between functions and variables (ex. “f_add5”, “v_3”)

Implementation

```
let res = multiply (add (x * y) (y + z)) (y - z)
```

```
let tmp0 = x * y
let tmp1 = y + z
let tmp2 = add tmp0 tmp1
let tmp3 = y - z
let res = multiply tmp2 tmp3
```

```haskell
extractParens :: Exp SrcSpanInfo -> State (Int, ExprInfo
SrcSpanInfo) (Exp SrcSpanInfo)
extractParens e = case e of
  Paren _ e1 -> do
    e1' <- extractParens e1
    var <- state (\(i, info) -> let var = "tmp" ++ show i in
    (var, (i + 1, (var, e1') : info)))
    return $ Var noSrcSpan (UnQual noSrcSpan (Ident noSrcSpan
    var))
  App _ e1 e2 -> do
    e1' <- extractParens e1
    e2' <- extractParens e2
    return $ App noSrcSpan e1' e2'
  _ -> return e
```

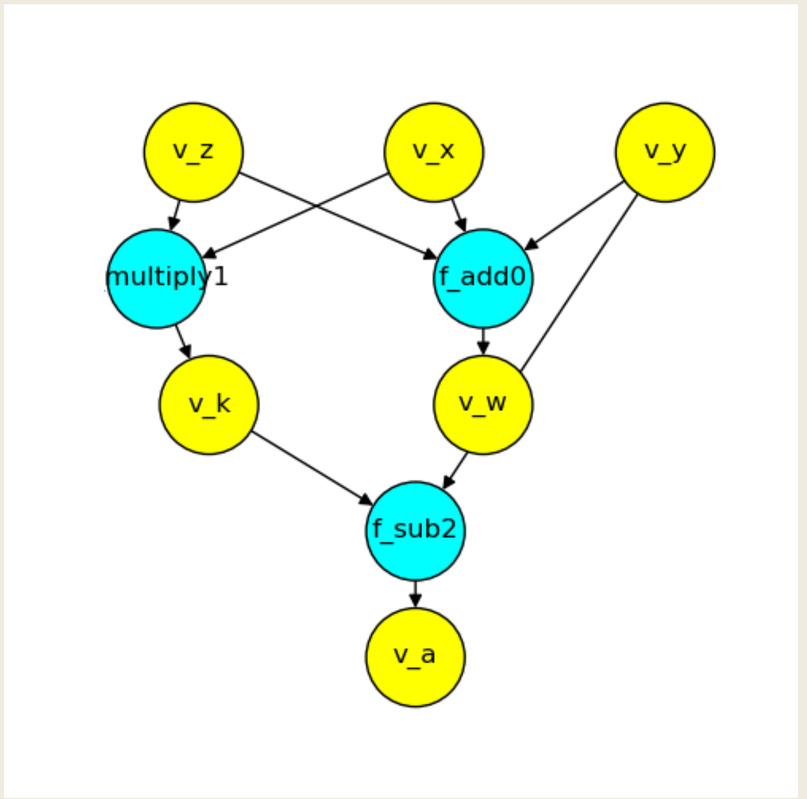
Consistency Model/Fault Tolerance

- Equivalent to a sequential execution of the same program
  - Random functions have to be explicitly seeded
  - Any state transitions are deterministic
- Assume that masters and workers do not crash
  - Any dispatched job will eventually finish with results returned

Workflow, trace of a sample run

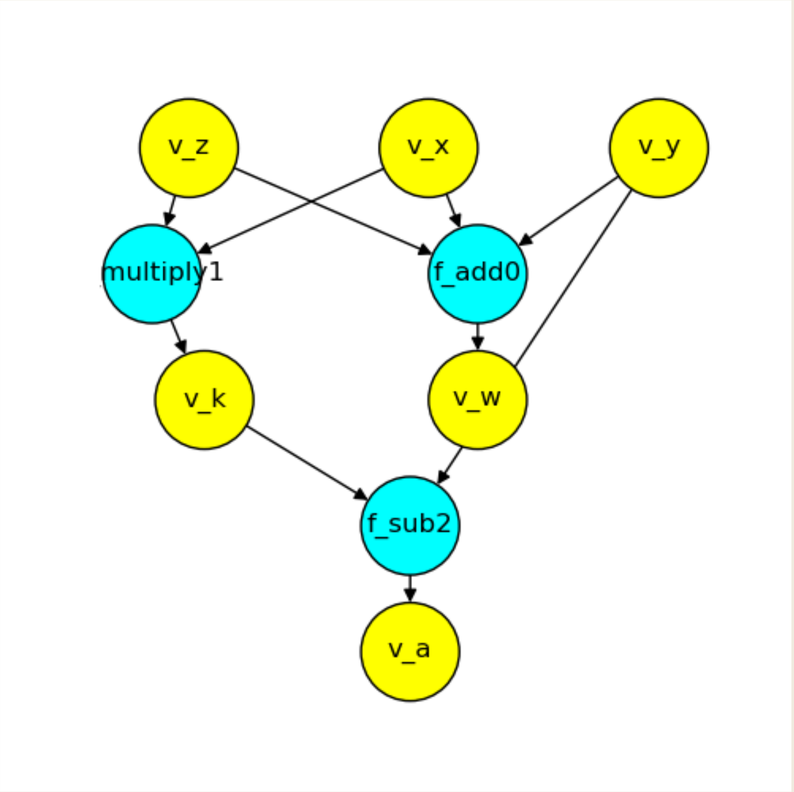
Dependency Graph from AST

```haskell
main :: IO ()
main = do
    let x = 10
    let y = 5
    let z = 3
    let w = add x y z
    let k = multiply x z
    let a = k `sub` y
    print
```



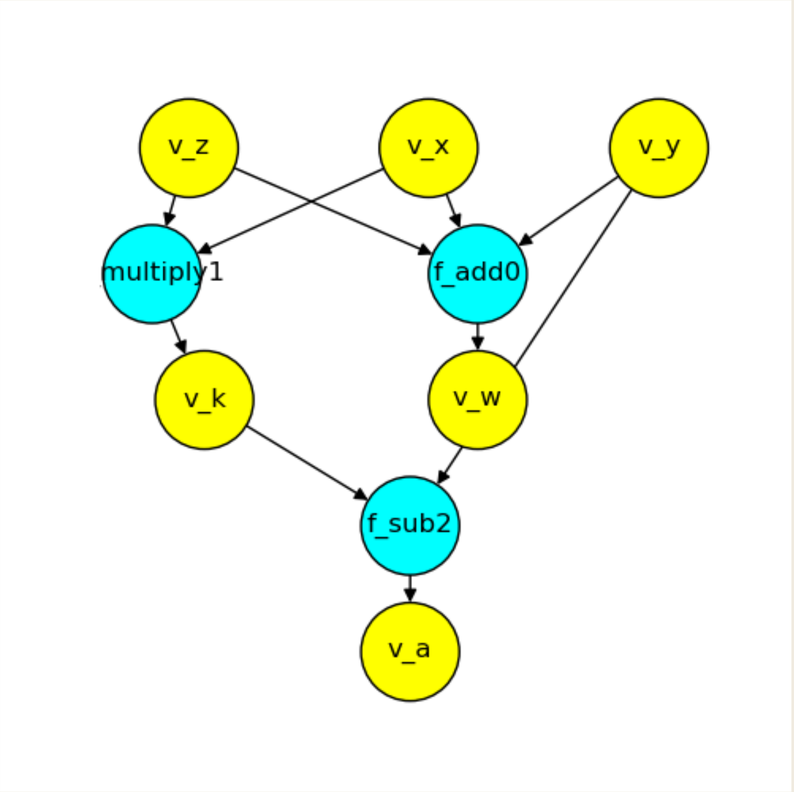
Scheduler

- Repeat:
  - Store nodes with in-degree zero in a FIFO queue
  - Send dependent task to a remote worker
  - When task finishes, store result in a function result hashmap
  - decrement in-degree of next node



Scheduler

- We have v_z and v_x. We can run multiply1
- We have v_z, v_x, and v_y. We can run f_add0
- Running remotely in parallel…
- Store result[multiply1] = 30, result[f_add0] = 18.
- Decrement in-degree of v_k, v_w
- We have v_k, v_y. We can run f_sub2



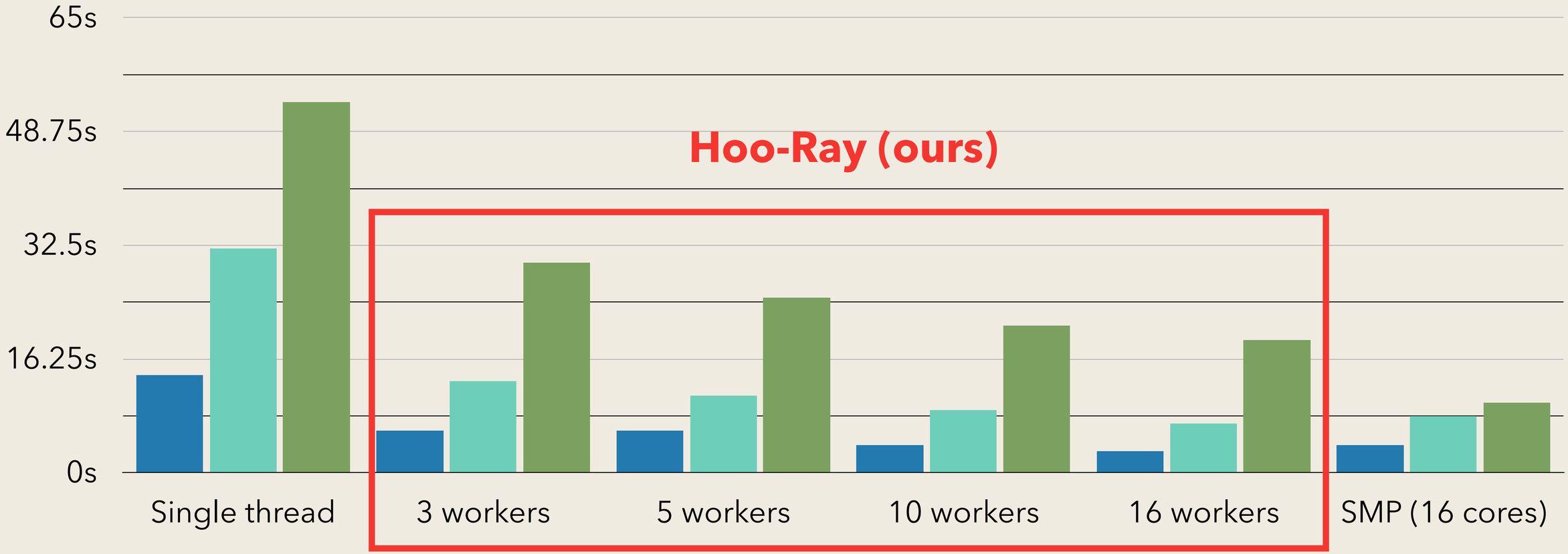
Benchmarks

Matrix Multiplication Benchmark

- The test program consists of a series of “operation sets”
- Each operation set generates two large random matrices, multiplies them, and computes the sum of all elements in the resulting matrix

```
let a1 = generateRandomMatrix 100 1000 1.0 486490
let b1 = generateRandomMatrix 1000 100 1.0 584180
let c1 = mmult a1 b1
let tmp1 = sumMatrix c1
```

Benchmark Results



Computation: matmul m=100, n=1000, p=100, range=(-1, 1)

Bars: 50 lines, 100 lines, 200 lines

Master and workers communicated over LAN in Hoo-Ray

Summary

Summary

- Hoo-Ray is a distributed execution engine for optimizing Haskell
- Generates Dependency Graph
- Serializes Functions
- Greedily Schedules Jobs on Workers
  - Schedule when we have all dependencies

Future Work

Future Work

- Fault tolerance
  - Re-run failed jobs
  - Zookeeper/Raft
- Handle network unreliabilities
  - adding retry-logic, redundancy, and timeouts
- Straggler mitigation
  - If many workers, can redundantly execute
- Locality-aware scheduling
  - Taking advantage of data locality and minimizing network data transfer

Q & A