Hoo-Ray: A Distributed Execution Engine for Haskell

Jerry Hou
Duke University

Jaden Long
Duke University

Tony Wu
Duke University

Dennis Xu
Duke University

Abstract

In this manuscript, we present *Hoo-Ray*, a prototype for a distributed execution engine for Haskell that allows distributed parallelization of Haskell programs with little user intervention. Given a Haskell program as input, Hoo-Ray builds its computational graph and automatically distributes workload across remote workers to execute in parallel. In our experiments, we find that Hoo-Ray is much faster than single-threaded execution on large matrix operations, and Hoo-Ray's speed scales with the number of remote workers. Current and future versions of our code can be found at https://github.com/denx20/Hoo-Ray.

1 Introduction

1.1 Related Works and Motivation

One of the main challenges in distributed computing is building interfaces and APIs that allow programmers with limited background in distributed systems to write scalable, performant, and fault-tolerant applications on large clusters. For example, MapReduce [3] provided the programming models of *Map* and *Reduce* that completely abstracts the distributed nature of the program away from the programmer, with the addition of clearly defined semantics. In this case, the programmer is able to first specify a *Map* function that groups the input data by some keys of interest. Then, these intermediate data are sent to the *Reduce* workers, who call the user defined *Reduce* function on the intermediate data to produce the final output. Although this model proved useful and efficient for many of Google's computational needs at the time, it also severely limited the scope of application to programs that can be expressed in this fashion. For instance, if we want to parallelize iterative, stateful computation using MapReduce, we would need several invocations of MapReduce, which would involve multiple reads and writes from persistent storage. Furthermore, it may take considerable effort for one to translate an existing task into the corresponding *Map* and *Reduce* functions, even without considering tasks that do not fit into the framework at all.

Another more recent work is Ray [5], which takes on a very different approach. The main problem that Ray set out to solve was distributed RL policy training. On one hand, this inherently involved stateful computations. While the environmental states need to be persisted throughout a given episode to track the agent's actions and rewards, the agent's state also needs to be kept in order for the agent to make incremental progress in future episodes. On the other hand, as the RL agent's actions are probabilistic in nature, the state graph cannot be determined in advance. Hence, a typical program in Ray involves both stateful and stateless computations, and Ray constructs a dynamic task graph computational model that differentiates between the two while respecting data dependencies between function calls.

While Ray's approach is effective for large-scale ML computations, we hypothesize that the task graph may be more easily constructed (perhaps even statically) if the program is deterministic and written instead in a purely functional language free of side-effects. Indeed, we found some prior work [7] that attempted to replicate a simplified version of the Ray model in PLT Scheme. In this case, the author took advantage of the powerful serialization framework built into PLT Scheme to serialize various data structures and functions into S-expressions, which can then be translated into jobs and scheduled across the workers.

Through our preliminary research, we found that many of Haskell's existing frameworks are more focused on shared memory parallelism (SMP) and have achieved impressive results [2] at the processor level. However, not much work has been done to exploit data and compute parallelism at the server level. Since the number of threads available on a given CPU is limited, we hypothesize that distributing tasks across a large number (potentially hundreds or even thousands) of servers will eventually pay off and even outperform optimized multi-threading executions.

1.2 Our Contribution

Our main contribution is Hoo-Ray, a powerful execution engine that automatically distributes and parallelizes computational workload of a Haskell program across different remote workers. Section 2 gives a more thorough illustration of the problem Hoo-Ray tackles and describes our high-level design principles. Section 3 describes the details of Hoo-Ray's architecture and implementation. Section 4 evaluates the performance of Hoo-Ray on bulky matrix operation tasks and compares it with single-machine benchmarks. Finally, Section 5 discusses directions for future work.

2 High Level Design

To adapt an existing program written in a common high-level object-oriented language (e.g. Java, Python) to a distributed system, one would require much understanding of the program to understand which functions do not depend on each other and to reason about which functions can be parallelized or executed out-of-order. We claim that a purely functional programming paradigm enforced by Haskell allows for easy reasoning about serializability. Since every access to the real world has to use the `IO` monad, one can precisely reason about when certain functions are ready to run.

Consider this example program which first cleans the input files in-place with the `format()` function, then evaluates the result returned by that `format()` function with the `complex_evaluation()` function, then finally performs a semantic analysis on the cleaned-up files.

```
def clean_files():
    ... Cleans up all text files
        in current working directory.
        Overwrites the original
        copies, and returns a summary ...

def complex_evaluation(summary):
    ... A heavy calculation that only depends
        on `input`. Does not depend on
        these files ...

def semantic_analysis():
    ... depends on the formatted
        version of files in the folder ...

def main():
    folder = ...
    os.cwd(folder) # cd into folder
    summary = clean_files()
    eval_res = complex_evaluation(summary)
    sema_res = semantic_analysis()
    print(eval_res, sema_res)
```

To adapt such a program for distributed computing, an engineer can reason that the execution of `complex_evaluation()` and `semantic_analysis()` do not depend on each other, but only depend on completion of `clean_files()`. Thus the engineer deduces that these two functions can be executed in two separate threads or machines in parallel, without breaking correctness.

That is what we hope to achieve. However, in an object-oriented language, such reasoning about the linearizability of an out-of-order execution is usually not easy. In the above example, `complex_evaluation()` could import a few other functions in its function call, and without exact understanding of how `complex_evaluation()` works, our engineer cannot ensure that this function doesn't happen to depend on some files in the current directory.

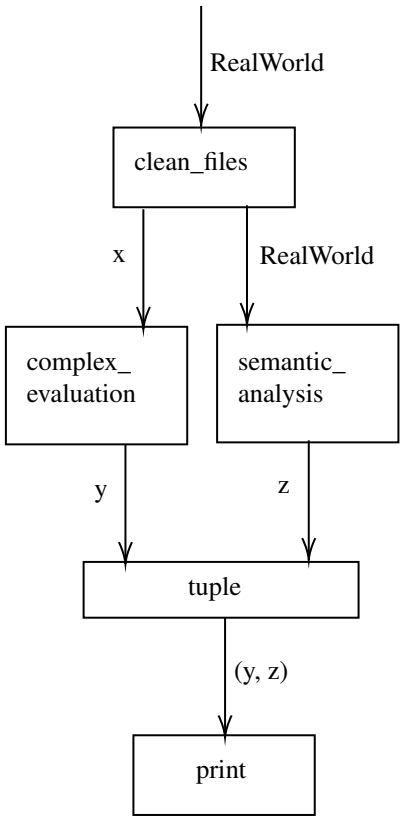
However, if the program were to be written in a purely functional programming language, this reasoning would be easy, since every interaction with the *RealWorld* has to be declared explicitly. The equivalent code in Haskell would be:

```
-- Define a custom data type called Summary
data Summary = ...

-- Define functions
clean_files :: IO Summary
clean_files = ...

complex_evaluation :: Summary -> Int
complex_evaluation x = ...

semantic_analysis :: IO Int
semantic_analysis = ...

-- Execute them
main :: IO()
main = do
    x <- clean_files
    let y = complex_evaluation x
    z <- semantic_analysis
    print (y, z)
```

From here, it should be rather easy to reason just from the function signatures that:

- The functions `clean_files` and `semantic_analysis` read from or write to the file system.
- The function `complex_evaluation` only depends on its input.

The execution of `main` can be more succinctly expressed as a graph:



At this point, it is very obvious that `complex_evaluation` and `semantic_analysis` do not depend on each other and thus can be executed in parallel, or even on different machines.

In general, such a dependency graph can be generated for any arbitrary Haskell program. Serializability can be ensured as long as one executes the `IO` operations in order, which is conveniently expressed as a variable *RealWorld* that is passed around in the graph. Hence, every function in this graph can be executed as long as all of its upstream neighbors are executed and serializability is ensured. In other words, such framework allows for parallel execution of a Haskell program while requiring essentially no knowledge from the software engineer about how the software is actually running.

3 Architecture and Implementation

In this section, we describe the architecture and implementation details of Hoo-Ray. Given a haskell program, Hoo-Ray performs three steps: building dependency graph of the program, serializing each task, and distributing workload to remote workers based on greedy scheduling.

3.1 The Dependency Graph Builder

The dependency graph builder is a critical component of our project. Its purpose is to extract the data dependencies from a given Haskell program and represent them as a directed acyclic graph (DAG). The dependency graph represents the relationships between variables and functions in the program, and the direction of the edges reflects the dependencies.

Our implementation of the dependency graph builder starts by parsing the source code of a given Haskell program into an abstract syntax tree (AST). The AST represents the structure of the program in a way that is easily traversable and pattern-matchable. We then use pattern matching from Haskell's `PatBind` function to extract data dependencies from the AST by identifying variable bindings and function calls. The resulting data dependencies are returned as a list of tuples, where each tuple contains a variable name, and a list of the names of functions on which it. Function dependencies are also returned as a list of tuples, where each tuple contains the name of a function and a list of its arguments. The ordering of tuples follow the topological order of the dependency graph, meaning that the dependencies are listed in the order in which they must be evaluated.

In particular, there are two types of edges in the our dependency graph: variable-to-function edge, which denotes assigning function output to variables, and function-to-argument edge, which denotes the input arguments to a function.

To concretely illustrate what our dependency graph constructs, consider the following pure function haskell program with inputs x, y, z, intermediate variables w, k, and output a. We will refer to this program as `pure.hs` hereinafter.

```
add :: Int -> Int -> Int -> Int
add x y z = x + y + z

sub :: Int -> Int -> Int
sub x y = x - y

multiply :: Int -> Int -> Int
multiply x y = x * y

divide :: Int -> Int -> Int
divide x y = x `div` y

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 a
```

Our dependency graph builder will generate the following output:

```
("w", ["add"])
("add", ["x","y","z"])
```

```
("k", ["multiply"])
("multiply", ["x","z"])
("a", ["sub"])
("sub", ["k","y"])
```

The dependency `("w", ["add"])` is a variable-to-function type edge that indicates `w` is assigned to the return value of the `add` function, and the the dependency `("add", ["x","y","z"])` is a function-to-argument type edge that indicates `x`, `y`, `z` are arguments of the function `add`. Our implementation can be found at `DependencyGraph/graph.hs`.

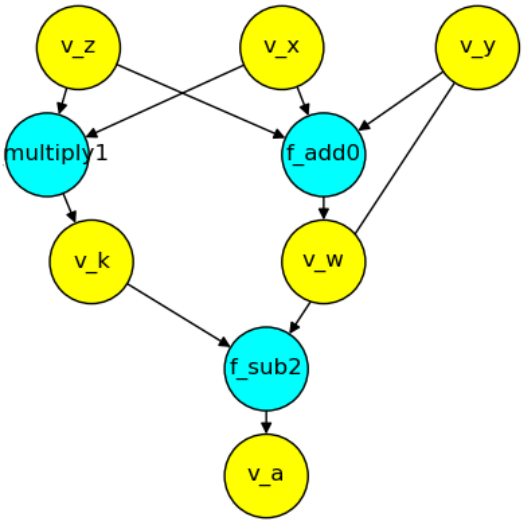


Figure 1: Dependency Graph constructed from example Haskell program `pure.hs` in Section 3.1. Yellow vertices represent variables and blue vertices represent functions. An edge from a yellow vertex to a blue vertex represents a function call argument relation, and an edge from a blue vertex to yellow vertex represents an variable assignment relation

3.2 Coarse-grained vs. Fine-grained Load Balancing

Coarse-grained load balancing and fine-grained load balancing are two core strategics that our system uses to handle parenthetical expressions.

Coarse-grained load balancing is a technique that treats nested parenthetical expressions as a single expression, which is then dispatched to workers. By bundling related expressions together, this approach reduces the preprocessing time needed to execute the code, and enables the workers to directly handle more complex, interdependent functions while still ensuring the correctness of the output. By default, pure.hs uses coarse-grained optimization.

Fine-grained load balancing is an alternative technique to coarse-grained optimization for improving the performance of software systems. It involves breaking down complex nested functions into smaller, more manageable expressions, which can be executed by multiple workers.

One practical example of how fine-grained load balancing can be applied is through the use of expressionparser.hs. Expressionparser.hs converts parenthetical expressions in the main function of a target file into serial expressions. For instance, when the input expression is

```
multiply (add (x * y) (y + z)) (y - z)
```

The tool can convert it into the following serial expressions:

```
Output:
    tmp0 = x * y
    tmp1 = y + z
    tmp2 = add tmp0 tmp1
    tmp3 = y - z
    multiply tmp2 tmp3
```

This conversion enables the smaller, individual expressions to be executed by multiple workers, rather than being processed by one worker that might be slowed down by a complex, nested expression. While the preprocessing time required to convert the expressions may add some overhead, the overall performance gain from distributing the expressions among multiple workers can be significant. Fine-grained load balancing is particularly useful in situations where complex and interdependent expressions can be broken down into smaller parts and distributed among multiple workers.

3.3 Scheduling

With the aforementioned dependency graph, we can optimize task scheduling by taking into account the interdependence and precedence of the operations. For instance, as shown in the `pure.hs` example in Section 3.1, `x`, `y`, and `z` do not have dependence on anything since they are just integer-valued variables, and the functions `add x y z` and `multiply x z` also just depend on these variables and nothing else. Hence, all of these can be executed in parallel on different workers. However, as the function `k `sub` y` depends on `k` which is in turn the output of the function `multiply x z`, it has to wait until `k` is ready.

Our solution works as follows. First, we will reverse the direction of all edges in the dependency graph so that each edge indicates precedence rather than dependence relation, and we will call this resulting graph the “precedence graph”. We also execute all the variable declarations on the master machine

when the master first starts. Then, we use a FIFO queue to keep track of the current list of operations that are ready to be dispatched to different workers. The queue is initialized to contain functions that only depend on the initially declared variables in the input program. As long as some workers are idle, we immediately pop tasks from the queue and assign them to any random worker. When a task T is completed, we store its result in a HashMap for future variable assignments that need the result, and remove the node T and its outgoing edges from the precedence graph. On each node and edge removal, we check the in-degree of the destination node. If the destination node has an in-degree of 0, which indicates all of its dependencies are ready, then we will append it to the queue as a new task. In this fashion, all of our operations will be completed. Our algorithm guarantees that at any time step, all tasks in the queue are independent and therefore can be assigned to random idle remote workers.

In our implementation, we define a process `assignJobs()` that runs continually on the master machine to dispatch jobs in the queue to remote workers until the queue is empty and all nodes have results. The process does not wait for the completion of remote execution. To listen for and handle response from remote workers, we implement `processReply()`. When a remote worker completes execution and returns a result, `processReply()` will update the result HashMap and decrement the in-degree of all the nodes that depend on this computation. The full implementation of can be found in `DependencyGraph/queue.hs`, and a pseudocode for `assignJobs()` and `processReply()` can be found in Appendix A.

3.4 Message Passing

We used Cloud Haskell [4] with the `distributed-process-simplelocalnet` backend. This library provides a message passing communication model, and serializes functional closures for transmission over the network.

In our implementation and experiments, each node is spawned as a single thread process. While all nodes are spawned on a single machine, they communicate via different ports on the machine through the local network. When the master is spawned, it first uses UDP multicasting to look for slaves on the local network. We propose that this simulates a data center environment, where nodes are weaved together via fast interconnects, yet not so fast as communicating via shared memory.

This subsection describes message passing as done in `DependencyGraph/queue.hs`.

3.4.1 Defining Messages and Remote Calls

We start by defining the data types for messages and remote calls:

```
data Message = Result String Double
  deriving (Typeable, Generic)

data RemoteCall = RemoteCall String ProcessId
    (HM.HashMap String Double)
    (HM.HashMap String [String])
  deriving (Show, Typeable, Generic)
```

The `Message` data type represents messages containing the results of computations. A `RemoteCall` is a data type containing the information required for a worker to perform a computation.

3.4.2 Remote Call Function

The worker processes execute the `remoteCall` function:

```
remoteCall :: RemoteCall -> Process ()
```

It takes a `RemoteCall` as an argument and pattern matches on the node type (function, variable, or argument) to perform the corresponding computation. Once the computation is completed, the worker sends the result back to the master process using the `send` function. Parameters to the execution are passed in via the input HashMap, which is sent through the network as part of the functional closure. In our prototype, this HashMap is very small, consisting of just a few floating point values as the inputs and outputs to our benchmark, so the overhead of passing the entire HashMap is low. For robustness, one can imagine implementing a functional closure where only the necessary values are sent between the master and the worker nodes.

3.4.3 Job Dispatching

The `dispatchJob` function is responsible for sending tasks to worker processes:

```
dispatchJob :: ProcessId -> NodeId -> String ->
    (HM.HashMap String Double) ->
    (HM.HashMap String [String]) -> Process ()
```

It takes the master process ID, worker node ID, task information, and the dependency graph as arguments. The function creates a `remoteCallClosure` and spawns a new process on the specified worker node using the `spawn` function.

3.4.4 Master Process

The master process is responsible for coordinating the entire computation. It starts by initializing data structures for the task queue, result storage, and worker node management. The `assignJobs` function iteratively assigns tasks to available worker nodes by updating the task queue and rotating the worker node list.

The `processReply` function listens for incoming results from worker processes. When a result is received, it updates the result storage and modifies the dependency graph accordingly. If new tasks become available, they are added to the task queue.

3.4.5 Main Function

The `main` function initializes the backend and starts either the master or slave processes based on the command line arguments.

```
main :: IO ()
```

It takes the command line arguments and initializes the backend with the specified host and port. The program can run in either master or slave mode, depending on the command line arguments.

In conclusion, our Haskell implementation demonstrates how to use message passing and distributed computing with the Cloud Haskell framework. The master process coordinates tasks among worker processes to achieve parallelism and efficient computation.

4 Evaluation

4.1 Workload Description

To evaluate the performance of Hoo-Ray, we decide to choose matrix binary operations. Our motivation is mainly twofold. First, we believe that tasks involving heavy workload will demonstrate the advantage of Hoo-Ray better, and the fact that matrix multiplication has a cubic time complexity makes it an appealing choice. Second, we realize the indispensable role of matrix multiplication in machine learning tasks, which are becoming increasingly popular and demanded today. We hope to demonstrate that the program execution speedup brought by Hoo-Ray has the potential to be applied to impactful real-world applications.

We design the workload such that every entry of every calculated matrix needs to contribute to the final output. This is because the Glasgow Haskell Compiler (GHC) already automates the computation process by performing lazy evaluation; unused intermediate values will not be calculated at all by default. The matrix operation workload is determined by five parameters l, m, n, p, r, and consists of the following steps. First, we generate a random matrix of size $m \times n$ and a random matrix of size $n \times p$, where every entry of the matrices is in the range $[-r, r]$. Then we multiply these two matrices to get a matrix of size $m \times p$, and calculate the sum of all mp entries. This is repeated for l times and generates l scalar values. Finally, we add all l scalar values to obtain the output. In our evaluation, we set $m = 100, n = 1000, p = 100, r = 1$, and vary the value of l, which we will refer to as "task size" in the following subsections. Note that the number of binary operations performed in the evaluation workload is linear in l. The workload generator is `Tests/matmul_test_gen.hs`.

4.2 Benchmarks

We choose single thread and Shared Memory Parallelism (SMP) as our benchmarks. Single thread simply means that the code execution is strictly sequential even though certain operations in our test workload can be paralleled. SMP is where multiple lightweight threads running on the same physical machine can access and modify the same memory locations. We expect SMP to be much faster because Haskell's SMP is built on top of its runtime system, which enables efficient parallel execution on multi-core processors without the overhead of creating many OS-level threads.

The evaluation program for single thread benchmark and SMP benchmark are named `matmul_ss_test.hs` and `matmul_ss_test.hs`, which will be automatically generated after running `Tests/matmul_test_gen.hs`. We assign variables using = to enforce single thread execution and using <- to allow SMP in accordance to Haskell syntax. Additionally, for SMP we use `par` from `Control.Parallel` to express parallelism between computations and run the code with the `RTS -N` flag, which signals GHC to run the code on more than one processor. To be clear, our SMP evaluation code works as follows when task size $l = 50$ (although variable names in our actual codebase might be different):

```
t1 <- calculateMatrix 100 1000 100 486490 584180 1.0
t2 <- calculateMatrix 100 1000 100 773021 990972 1.0
...
t50 <- calculateMatrix 100 1000 100 839071 269845 1.0
let tmp = [t1, t2, ..., t50]
let result = foldr1 (\acc x -> x `par` (acc + x)) tmp
print result
```

The usage of `par` indicates that each element in the list `tmp`, which corresponds to a bulky matrix multiplication and addition task, needs to be run in parallel as much as the physical machine permits.

4.3 Results

Figure 2 illustrates the evaluation results, where all time readings (in seconds) are wall time rounded to the nearest decimal place. Hoo-Ray consistently shows a significant improvement over single thread execution across all task size, and performs on-par with SMP when the task size is 50 and 100. Additionally, it can be observed that as the number of remote worker increases, Hoo-Ray becomes faster due to its ability to leverage the additional computing power provided by remote workers. Therefore, we believe that with a higher number of remote workers (e.g. when deployed with a large computing cluster where hundreds or thousands of remote workers are available), Hoo-Ray can eventually take advantage of distributed computing to outperform SMP on larger tasks, given

that the latter could be bottlenecked by the number of threads available on a given CPU.

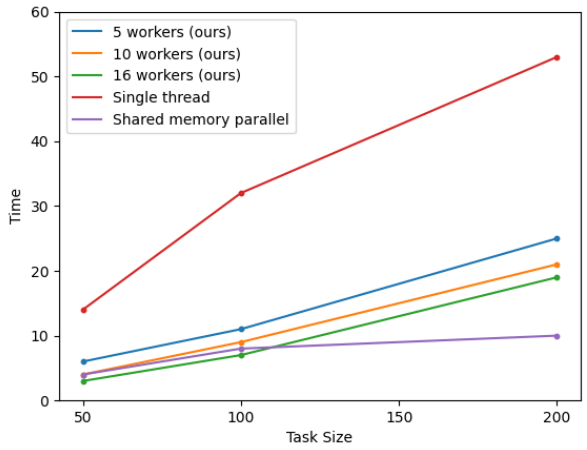


Figure 2: Hoo-Ray vs Benchmarks. We set task size $l = 50, 100, 200$ and measure the performance of Hoo-Ray with 5, 10, and 16 remote workers, respectively, against single thread and SMP benchmark.

5 Future Work

While Hoo-Ray successfully demonstrates the feasibility of distributing workloads for parallel execution in a functional programming language, there is still room for improvement in terms of fault tolerance and task scheduling optimization. These areas will be the focus of our future work.

To improve fault tolerance, Hoo-Ray needs to be able to handle common exceptions in distributed systems, such as network unreliability (e.g. dropped or duplicated messages), worker failure, and even master failure. One possible solution to address network unreliability is to add retry logic and timeouts. Retry logic involves automatically attempting to resend a message or request when it is not acknowledged or responded to, while timeouts set an upper limit on how long a message or request can go unanswered before it is considered lost. To handle worker failure, we can consider adding a heartbeat to remote workers. If two consecutive heartbeats are missed, the worker is flagged as crashed and a new worker can be spawned to take its place. We also recognize the potential for master failure, which could be mitigated with checkpointing (as used in Tensorflow [1]) or a Raft [6] cluster to replicate the state of the master machine across multiple systems.

In addition to fault tolerance, task scheduling in Hoo-Ray can be further optimized. Inspired by MapReduce, we believe Hoo-Ray would benefit from a straggler mitigation strategy, such as re-running slow jobs on a different worker. For example, if a job running on a remote worker takes longer than expected and the queue is already empty, the scheduler can dispatch the same job to a different worker and use whichever result arrives first. Another approach is locality-aware scheduling, which is used in Ray to reduce data transmission latency and network traffic between remote nodes. We consider this important if Hoo-Ray scales up and gets deployed to work with larger-scale data storage systems that may span over multiple nodes and geographic locations.

6 Conclusion

We have described and implemented Hoo-Ray, a distributed execution engine that enables the parallelization of Haskell programs. By automating the process of building a computational graph and distributing workload across multiple remote workers through a greedy scheduler, Hoo-Ray combines the advantages of MapReduce and Ray to speed up program execution in Haskell.

Our experiments on bulky mathematical operations, such as matrix multiplication, demonstrate that Hoo-Ray is capable of effectively capitalizing on parallelizable components in the computational graph of input programs. The results show that Hoo-Ray outperforms the single thread baseline by over 100% and scales well with the number of remote workers. These findings highlight the potential of Hoo-Ray to significantly improve the performance of complex programs in Haskell.

Overall, Hoo-Ray represents a valuable contribution to the field of distributed computing, providing a powerful and efficient tool for parallelizing the workload of Haskell programs. With further improvements in fault tolerance and task scheduling optimization, Hoo-Ray has the potential to become an important asset for researchers and practitioners seeking to optimize large computational work.

References

[1] Martın Abadi et al. “Tensorflow: a system for large-scale machine learning.” In: *Osdi*. Vol. 16. 2016. Savannah, GA, USA. 2016, pp. 265–283.

[2] Manuel MT Chakravarty et al. “Data Parallel Haskell: a status report”. In: *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*. 2007, pp. 10–18.

[3] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. OSDI’04. 2004, pp. 137–150. URL: https://www.usenix.org/legacy/publications/library/proceedings/osdi04/tech/full_papers/dean/dean.pdf.

[4] Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. “Towards Haskell in the cloud”. In: *Proceedings of the 4th ACM symposium on Haskell*. 2011, pp. 118–129.

[5] Philipp Moritz et al. “Ray: A Distributed Framework for Emerging AI Applications”. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. OSDI’18. 2018, pp. 561–577. URL: https://www.usenix.org/system/files/osdi18-moritz.pdf.

[6] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 305–319.

[7] Alex Schwendner. “Distributed functional programming in Scheme”. MA thesis. Massachusetts Institute of Technology, 2010. URL: https://groups.csail.mit.edu/commit/papers/2010/alexrs-meng-thesis.pdf.

Appendix

A Master machine implementation

Algorithm 1 Assign Jobs

function ASSIGNJOBS
  $queue \leftarrow$ readMVar($queueVar$)
  $results \leftarrow$ readMVar($resultsVar$)
  if *queue* is empty and length of *results* is equal to length of *nodeList* **then**
    print "Final result: " + sum of *results*
    return
  else
    $workerList \leftarrow$ readMVar($workerListVar$)
    $job \leftarrow queue[0]$
    $newQueue \leftarrow$ remove first job from $queue$
    $worker \leftarrow$ first worker in $workerList$
    modifyMVar($queueVar$, return $newQueue$)
    modifyMVar($workerListVar$, return rotated $workerList$)
    dispatchJob($masterPid, worker, job, results, depGraph$)
    call ASSIGNJOBS()
  end if
end function

AssignJobs will continually dispatch jobs from the queue to remote workers to be executed until all nodes in the dependency graph have been calculated.

Algorithm 2 Process Reply

function PROCESSREPLY
  $results \leftarrow$ read $resultsVar$
  if length($results$) == length($nodeList$) **then**
    return
  else
    $(node, result) \leftarrow$ expect
    $visit \leftarrow$ read $visitVar$
    $visit' \leftarrow$ delete node from $visit$
    update $visitVar$ with $visit'$
    $results' \leftarrow$ insert $(node, result)$ into $results$
    update $resultsVar$ with $results'$
    $indegrees \leftarrow$ read $indegreeMapVar$
    $updatedIndegrees \leftarrow$
      updateIndegreeMap($node, reversedGraph$)
    update $indegreeMapVar$ with $updatedIndegrees$
    $queue \leftarrow$ read $queueVar$
    $newNodes \leftarrow$
      findNewNodes($updatedIndegrees, visit'$)
    update $queueVar$ with $(queue + newNodes)$
    call PROCESSREPLY()
  end if
end function

ProcessReply will listen for response from remote workers and update result hashmap and decrement the in-degree of

nodes pointed to by the executed task upon receiving worker response.

B Team Member Contributions

Jaden Long

- **Project planning (everything not reflected in GitHub commits)**
  - Came up with the initial draft with the idea.
  - Arranged meetings with team members to furnish this initial idea.
  - Drafted the project proposal document.
  - In the project checkpoint 1 document, Drafted sections *High level Design* and *Member Responsibilities*. In the project checkpoint 2 document, drafted sections *Message Passing with Cloud Haskell* and *Member Responsibilities*.
  - Designed roadmap for our project and assigned weekly tasks to other team members (with regards to both learning Haskell and researching for the project). Checked up with team members weekly about their progress.
  - Created the introduction slides (slides 1 through 19) for our final presentation.
  - In this report, drafted the subsection *Message Passing*.
- **Code**
  - Researched the use of Cabal, the package manager for Haskell, and initialized our package structure with it.
  - Documentation about installation and setup of Haskell and the environment.
  - Read the book *Parallel and Concurrent Programming in Haskell* concerning distributed programming. Researched the use of the *distributed-process* package.
  - Met with CS dept IT Joe Shamblin when noticing issue with UDP multicasting as required by the *distributed-process-simplelocalnet* package. Added UDP multicast testing resources.
  - Implemented a remote add application as prototype for computation distribution (but no scheduler yet!).
  - Wrote Python script for visualizing computational graph.
  - Wrote benchmark script for concurrent programming with `Par` monad (not used in final report in favor of the SMP module).
  - Wrote `queue.hs` with Dennis and Tony (it was a marathon spanning two days!).
  - Benchmarked `queue.hs` with baseline methods (single thread and SMP) using 3, 5, 10 workers.

Dennis Xu

- Proposed the design details of the dependency graph parser, researched relevant libraries, and implemented the dependency graph parser (`graph.hs`) using Haskell's AST parsing libraries.
- Wrote helper functions (`extract_parens.hs`) to recursively extract parenthetical expressions into a series of `let` statements for easier dependency graph parsing.
- Implemented the final benchmark modules (`arithmetic_test_gen.hs`, `list_test_gen.hs`, `matmul.hs`, and `matmul_test_gen.hs`) to generate test files consisting of simple arithmetic, list (ex. summation, product), and matrix (summation, multiplication) operations. Multiple test files are generated for the same task for purposes of single-thread, multi-thread executions as well as easy dependency graph parsing. Documented usage (command line flags) and examples in `README.md`.
- Proposed the scheduler design and implemented it alongside Jaden and Tony in `queue.hs`.
- Wrote and edited sections 1.1, 2, 3.3 in this final report, and drafted the corresponding sections on *Background and Related Work* and Scheduling in checkpoints 1 and 2.
- Created slides 20-22, 28-29 in the final presentation on implementation details and benchmarking.
- Came up with the name for our project: Hoo-Ray.

Tony Wu

- Wrote the initial version of Hoo-Ray's scheduling algorithm in `queue.hs`, and collaborated with Jaden and Dennis to debug and finalize the code.
- Wrote Python code for visualizing the evaluation results (Figure 2) and improved code for visualizing dependency graph (Figure 1).
- Maintained codebase documentation for better readability and added exhaustive in-line comments for all major Haskell code files (`graph.hs`, `queue.hs`, `matmul_test_gen.hs`, etc).

- Worked on package dependency management and version control for both Haskell and Python code and removed unused dependencies and outdated files to improve code efficiency.
- Wrote Section 1.2, Section 3.1, Section 3.3, the entire Section 4, and majority of Section 5 and 6 in the final report, covering Hoo-Ray's dependency graph construction, scheduling, evaluation, and possible directions for future work.
- Condensed write-ups and weekly tasks from previous checkpoints into Appendix C as a way of documenting our progress on the project throughout the semester.
- Researched the use of Stack, an alternative to Cabal (the package manager for Haskell used in this project).
- Created slides on Hoo-Ray's workflow and sample run trace-through (slides 23 to 27) in our final presentation.

Jerry Hou

- Set up unit testing and benchmarking framework and developed exhaustive tests for functions related to dependency graph generation and expression parsing.
- Modularized functions in Hoo-Ray codebase to interface with unit and benchmark tests, and to ensure better readability and maintainability.
- Explored solutions for handling exceptions, such as network unreliability and worker failure, and suggested strategies for straggler mitigation and locality-aware scheduling.
- Researched GHC compiler's existing optimization techniques and explored alternatives to matrix multiplication for benchmarking.
- Wrote parts of Section 3.1, Section 3.2, parts of Section 5, and Section 6 in the final report, which covers the implementation details of Hoo-Ray's dependency graph generation, load balancing considerations, and directions for future works.
- Wrote Appendix A, which provides additional details on Hoo-Ray's master-machine implementation.
- Created slides 30-34 on summary of Hoo-Ray and possible directions for future work.

C Project Progress Log

This section is dedicated to documenting our progress on this project before Checkpoint 2. The following contains a list of tasks that each team member received between checkpoint 1 and 2.

To-do's for the week Mar 8 - Mar 21:

This is about the amount of task a week distributed over two weeks due to spring break. Since project checkpoint 2 is due immediately after spring break, we'll have to learn Haskell quick. We'll have to divide into two pairs:

Tasks for Pair 1: Server Implementation and Message Passing

1. Read Towards Haskell in the Cloud, specifically noting section 5.
2. Read Master-Slave, Work-Stealing and Work-Pushing, which should be rather relevant.
3. Read Chapter 12 of Samuli Thomasson's *Haskell High Performance Programming* and implement what the author wrote in that chapter.
4. Explore the two libraries mentioned in the previous reading in Cloud Haskell. i.e.
   - `distributed-process`
   - `distributed-process-simplelocalnet`
5. Set up test environment (emulated master and slave servers).

Tasks for Pair 2: Generating Dependency Graph on Function Call

1. A brief search led me to this post. The SourceGraph package is our only lead but it seems to be deprecated since 2018. The task will be to first understand how this package functions by looking at its source code.
2. Try to somehow install this package, and let it generate the graph for an example program (the one we mentioned in our checkpoint 1 manuscript should work).

Note: Project checkpoint 2 due on Mar 24

Description: *This is a five-page write-up containing a detailed description of the design and preliminary results, as well as an update on team member progress and future responsibilities.*

To-do's for the week Mar 2 - Mar 7:

1. Get the notebook version of *Learn You a Haskell for Great Good!*. We will refer to this book as the "textbook" from this point. Follow the installation instructions with Docker here.
2. Go through the first 5 chapters of the textbook.
3. Go through the first 20 problems of 99 Haskell Problems.