



Time efficiency analysis of parallel programs on Liquid Haskell

Yu DAIKI¹ and Shinya NISHIZAKI²

¹ Tokyo Institute of Technology, Ookayama, Meguro, Tokyo 152-8552, JAPAN
yu.daiki@lambda.cs.titech.ac.jp

² Tokyo Institute of Technology, Ookayama, Meguro, Tokyo 152-8552, JAPAN
nisizaki@cs.titech.ac.jp

Abstract. In this study, we focus on parallel programming in the purely functional language Haskell and try to analyze the execution time statically.

Liquid Haskell is a program verification tool that integrates refinement types into the Haskell programming language and relies on SMT solvers for verification. In this study, we will improve the existing method for analyzing sequential execution time by extending its capabilities to parallel programs using Liquid Haskell.

Specifically, we have implemented two execution time analysis functions in Liquid Haskell. One is a function that executes a two-element tuple sequentially or in parallel, depending on the number of processors. The other is a function that executes each element of a list in parallel.

We compare the results of parallel execution time inferences made in Parallel Haskell with the actual execution times of parallel runs on computers with multi-core CPUs.

Keywords: execution time analysis, functional programming language, Haskell, SMT solver

1 Introduction

Execution time and time complexity

In recent years, the advancement of single-core processors has plateaued, as they have reached their performance limits. As a result, there has been a growing emphasis on parallel computing acceleration using both multi-core and many-core processors to overcome these limitations. This shift towards parallelism aims to leverage the combined processing power of multiple cores to achieve higher computational efficiency. However, writing parallel programs that fulfill the desired performance criteria is a complex and challenging task. Without proper parallelization techniques and careful consideration of factors such as data dependencies, workload balance, and synchronization, the execution time of a parallel program may not show significant improvement over a sequential one. In some cases, improper parallelization can even result in slower execution times due to overheads such as thread creation and communication.

Measuring execution time is a valuable method for verifying the efficiency of a parallel program. However, ensuring that a parallel program consistently performs as expected can be challenging because execution time can vary depending on several factors. These include the environment in which the program is running (such as the number of processors) and the specific input provided. In addition, changes to certain parts of the source code may require remeasuring the execution time, further complicating the verification and optimization process.

Time complexity is a critical metric for evaluating the performance of any computer program, including parallel programs. In the context of parallel computing, time complexity not only measures the number of computational steps required for a given task, but also incorporates the number of processors as a variable. This unique consideration allows one to analyze how the actual number of computational steps can be reduced by increasing the number of processors, thereby achieving more efficient parallel execution.

The concept of time complexity goes beyond simply counting operations. It describes the asymptotic behavior of the program's execution time as the size of the input increases. By understanding the time complexity of a parallel program, developers can predict its performance over different input sizes and different numbers of processors. This understanding aids the optimization process, enabling the creation of more responsive and resource-efficient parallel applications.

Functional Programming Language Haskell

Haskell[1][2][3] is a purely functional programming language known for its clear syntax and strong type system. When discussing Haskell in the context of compiling and running code in this paper, we assume that we are referring to the Glasgow Haskell Compiler (GHC), the most widely used compiler for the language.

In addition to its functional nature, Haskell[4][5][6][7] also provides robust support for parallel programming. This capability allows developers to use multiple processors to perform tasks simultaneously, which can lead to improved performance. Unlike some other parallel programming environments, Haskell's design ensures that parallel programs are deterministic, meaning that they produce the same results on multiple runs. This avoids common parallel programming pitfalls such as race conditions and deadlocks, problems that can cause unpredictable behavior in parallel systems.

In Haskell[1], the complexities of synchronization and communication between parallel tasks are handled implicitly by the language and its runtime system. Rather than requiring the programmer to manually control these aspects, the developer simply indicates in the source code where parallelization is expected or desired. The runtime system then analyzes the code and automatically makes decisions about how to execute the parallel parts, taking care of the underlying details. This approach promotes a cleaner code base and allows the

programmer to focus more on logic and functionality, fostering a more efficient and error-free development process.

Type Classes in Haskell

A type class in Haskell represents a set of functions that defines a particular behavior and specification. Essentially, it specifies a set of methods that must be implemented for a type, allowing that type to be treated in a specific way. Type classes enable polymorphism, where different types can be treated uniformly based on shared behaviors rather than their specific implementations. When a type is made an instance of a type class, it means that the type implements the functions required by the type class. These implementations provide the actual behavior for the specific type, fulfilling the contract that the type class specifies.

In Haskell functions that take polymorphic types as arguments, type class constraints can be applied to ensure that the types conform to particular behaviors. By requiring that the type variable be an instance of a specific type class, the function can ensure that the type can perform the computations dictated by the type class. This enforces a level of type safety and allows for more abstract and reusable code.

For example, the `Num` type class in Haskell defines the behavior for numeric types, specifying operations such as addition (`+`), subtraction (`-`), and multiplication (`*`). Any type that is made an instance of `Num` must provide implementations for these operations. Common numeric types like the type of fixed-length integers such as 64bit integers `Int`, the type of floating-point numbers `Float`, and the type of multiple precision integers `Integer` are instances of the `Num` type class, and therefore, they support these operations.

In Figure 1, consider a function like `sum` that takes a list of type `a`, with the constraint that `a` satisfies the type class `Num`.

```

1 sum :: Num a => [a] -> a
2 sum [] = 0
3 sum (x:xs) = x + sum xs

```

Fig. 1. Function `sum`

This constraint ensures that the elements in the list can be added together, as the `Num` typeclass guarantees that the addition operation is defined for type `a`. By leveraging the type

Monad in Haskell

A monad[8] is a design pattern that represents a generalized way of structuring computations, including aspects such as the state of the computation, in-

put/output operations, and more. It's a powerful abstraction that allows computations to be combined in a structured and flexible way.

A monad has three essential components:

1. A data type that is the object or target of the computation, representing the underlying structure or context.
2. A function that produces computed values, often referred to as the **return** function. This function encapsulates a value within the monad, creating a basic computational context.
3. A function that composes computations, often referred to as a bind operation (**>>=**). This function allows for the chaining or sequencing of computations, transforming the output of one computation into the input of the next.

In Haskell, a monad is implemented as a type class that requires the definition of these two core functions: the return function for value generation and the bind (**>>=**) function for computation composition. By implementing these functions for a particular datatype and making that datatype an instance of the monad type class, calculations can be performed on that datatype in a monadic way. This allows complex computations to be constructed using simple, composable building blocks.

Figure 2 illustrates the typing of these functions, showing how they interact to form the computational structure of a monad. The monadic design pattern promotes more maintainable and understandable code by abstracting common computational patterns, making it a fundamental concept in functional programming, especially in Haskell.

```
1 return :: a -> m a
2 (>>=) :: forall a b. m a -> (a -> m b) -> m b
```

Fig. 2. Typing of **return** and (**>>=**)

The **return** function is a special function that takes a value and encapsulates it inside a monad, creating a monadic value from a regular value.

The bind function (**>>=**) is a central component of monads. It takes a value of type **a**, wrapped in a monad, along with a function that accepts a value of type **a** and returns a value of type **b**, wrapped in the same monad. The bind function combines these elements, resulting in a value of type **b** wrapped in the monad. This allows for sequencing of computations within the monadic context.

Common examples of monads in Haskell include the **Maybe** monad and the **IO** monad. The **Maybe** monad provides a way of dealing with computations that may fail. It adds an additional case to a value, representing failure or absence. When a function returns a value wrapped in the **Maybe** monad, it can represent either a successful computation resulting in a value (**Just**) or a failure (**Nothing**).

The bind function (`>>=`) is essential when working with the `Maybe` monad. It allows you to concatenate computations that may result in failure. When performing a series of computations, if any step fails, the entire computation will result in `Nothing`. The final value will be `Just` if and only if all computations in the series succeed, allowing elegant handling of potentially failing computations.

```
1 data Maybe a = Nothing | Just a
```

Fig. 3. Maybe monad

The `IO` monad in Haskell is a special construct that encapsulates side effects and allows for controlled interaction with the outside world. Since Haskell functions are inherently pure, meaning they have no side effects, the `IO` monad provides a means to perform impure operations such as reading from or writing to files, interacting with the network, or accessing user input. By encapsulating these side effects within the `IO` monad, Haskell maintains its functional purity while still allowing necessary interactions with external systems.

Refinement type

Refinement Types[9] are types that classify values such that they satisfy a particular predicate. The following examples are refinement types of non-zero integers and even integers, respectively.

$$\{v \in \mathbb{Z} \mid v \neq 0\} \text{ and } \{v \in \mathbb{Z} \mid \exists w \in \mathbb{Z}(v = 2w)\}$$

The parts to the right of the vertical bar, $v \neq 0$ and $\exists w \in \mathbb{Z}(v = 2w)$, are the predicate.

SMT solver

SMT (Satisfiability Modulo Theories)[10] refers to the task of determining the satisfiability of logical formulas in the context of certain mathematical theories. These formulas are expressed using first-order predicate logic and can represent constraints related to various domains, such as arithmetic, arrays, or data structures.

An SMT solver is a specialized tool that solves SMT problems by combining the capabilities of two types of solvers: a SAT (Boolean Satisfiability) solver, which handles the propositional logic aspects, and a solver dedicated to the particular theory, such as integer arithmetic, for handling domain-specific constraints.

A prominent example of an SMT solver is Z3[11], which is widely used in various fields, including formal verification and symbolic execution. To illustrate,

consider a system of simultaneous integer equations, such as $x + y = 10$ and $x + 2y = 20$. By appropriately encoding these constraints and providing them as inputs to an SMT solver such as Z3, one can automatically determine the values of the variables x and y that satisfy the given equations. The encoding for Z3 is shown in Figure 4.

```

1 (declare-const x Int)
2 (declare-const y Int)
3 (assert (= (+ x y) 10))
4 (assert (= (+ x (* 2 y)) 20))
5 (check-sat)
6 (get-model)

```

Fig. 4. Code for solving a linear system of equations

Liquid Haskell

Liquid Haskell[12] is an extension to the Haskell programming language that introduces refinement types to facilitate static verification using SMT solvers. These refinement types allow for more precise constraints on values, thereby increasing the correctness of the code. Programs successfully verified by Liquid Haskell are guaranteed to be compilable with GHC (Glasgow Haskell Compiler).

The verification process in Liquid Haskell consists of a few key steps. First, a subtype relation is extracted from the refinement type specified in the code. This relation is then transformed into a logical expression suitable for analysis. The SMT solver then verifies the truth of this logical expression, confirming that the constraints imposed by the refinement type are met.

In Liquid Haskell, refinement types are often expressed as annotations within comments in the Haskell source code, maintaining compatibility with standard Haskell tools. As an illustrative example, consider the `safeDiv` function, which is designed to perform division in a way that avoids division by zero. This safe division can be defined and used through refinement types, with the specifics outlined in Figure 5.

```

1 {-@ safeDiv :: Int -> { v:Int | v /= 0 } -> Int @-}
2 safeDiv x y = x `div` y
3 main = print $ safeDiv 3 0

```

Fig. 5. Function `safeDiv`

Liquid Haskell reports the given program as unsafe due to the refinement type constraint that requires the second argument to be nonzero.

In Liquid Haskell, you can express a function's preconditions by adding a predicate to the type of its arguments. This predicate helps define the expected behavior of the function by stipulating conditions that must be met before execution. Similarly, predicates can be appended to a function's return type to detail the postconditions, or the expected state after the function's execution. Additionally, by associating a predicate with the type of a data structure, an invariant condition that must always hold for that data structure can be enforced.

Consequently, programs verified by Liquid Haskell are ensured to meet these specified preconditions, postconditions, and invariant conditions, providing a robust and clear definition of program behavior. However, it's important to note that limitations in the SMT solver's inference capabilities might occasionally lead to a program being judged as unsafe, even when it conforms to the refinement type constraints. This highlights the complexity of static analysis and the ongoing challenges in fully automating the verification process.

Efficiency Analysis using Liquid Haskell

The paper [13] by Handley et al. introduced a novel method for static analysis of program efficiency using Liquid Haskell. This method uses a unique data structure called a *tick monad* to monitor resource usage during computation.

As shown in Figure 6, a tick monad serves as a monad specifically designed to store both the number of operation steps (representing the cost of computation) and the actual value resulting from the computation. Within this monad, `tcost` denotes the number of operation steps, while `tval` encapsulates the value of the computed result.

When an operation is performed within the context of a tick monad, the number of steps required for that particular operation can be systematically recorded. This is accomplished by incrementing the value of `tcost` accordingly, which accurately represents the cumulative number of operation steps. Through this methodology, the tick monad provides a structured way to analyze the time complexity of computations in a way that integrates seamlessly with Liquid Haskell's verification framework.

```

1 {-@ data Tick a = Tick { tcost :: Int, tval :: a } @-}
2 data Tick a = Tick { tcost :: Int, tval :: a }

```

Fig. 6. Type of Tick Monad

The definitions of the `return` and `(>=)` functions that make up the tick monad are shown in Figure 7.

```

1 {-@ reflect return @-}
2 {-@ return :: x:a
3   -> { t:Tick a / x == tval t && 0 == tcost t } @-}
4 return :: a -> Tick a
5 return x = Tick 0 x
6
7 {-@ reflect >=> @-}
8 {-@ (>=>) :: t1:Tick a -> f:(a -> Tick b)
9   -> { t:Tick b /
10     tval (f (tval t1)) == tval t &&
11     tcost t1 + tcost (f (tval t1)) == tcost t }
12 @-}
13 infixl 4 >=>
14 (>=>) :: Tick a -> (a -> Tick b) -> Tick b
15 Tick m x >=> f = let Tick n y = f x in Tick (m + n) y

```

Fig. 7. Functions `return` and `(>=>)` of Tick Monad

In Line 4 and 5 of Figure 7, the `return` function takes a value and returns a Tick monad with `tcost` of 0.

In Line 15 of Figure 7, the `(>=>)` function takes the `tcost` of the returned Tick monad as `(m+n)`, the result of adding `m`, the `tcost` of the first argument, and `n`, the `tcost` of the result of applying the value `x` of the first argument to the second argument `f`.

There are two prevalent methods for analyzing timing in Liquid Haskell:

- one that relies on automated reasoning within the Liquid Haskell framework, and
- the other that relies on manual external theorem proving through the use of proof combinators.

The advantage of the automated reasoning approach is that it eliminates the need for manual proofs, thus streamlining the verification process. However, this method is not without its challenges. Automated proofs require type-level computations, which means that inferences cannot be made simply by using the `(>=>)` function. Instead, the inference process requires the use of the `eqBind` function, as shown in Figure 8. Within this function, `tcost (f (tval t1))` is strategically replaced with an integer argument, providing a mechanism for encoding complex logical relationships.

```

1 {-@ eqBind :: n:Int -> t1:Tick a
2     -> f:(a -> { tf:Tick b | n == tcost tf })
3     -> { t:Tick b |
4         tval (f (tval t1)) == tval t &&
5         tcost t1 + n == tcost t }
6 @-}
7 eqBind :: Int -> Tick a -> (a -> Tick b) -> Tick b

```

Fig. 8. Function eqBind

2 Research Purpose

In this study, we introduce an implementation for analyzing time efficiency in parallel programs, utilizing the Liquid Haskell framework as initially proposed by Handley et al. Leveraging this new implementation, we showcase its application across several parallel programs, conducting detailed analyses of their execution times.

Additionally, we execute these parallel programs on a computer equipped with a multi-core CPU to gauge the real-world elapsed time. This empirical measurement is then compared with the theoretically computed number of operation steps from our analysis. This juxtaposition not only validates our implementation but also provides insightful observations on the time efficiency of parallel programs when analyzed through Liquid Haskell.

Shiromizu et al. [14] introduced an extension of the computational complexity analysis method to parallel programs using the Coq theorem proving system [15]. The monad \mathbf{C} , shown in Figure 9, is composed of a value of type \mathbf{A} and a predicate P , which yields a proposition characterizing computational complexity. The \mathbf{C} is described in the usual mathematical style as follows: for a set A and a predicate P on A and \mathbb{N} , C is defined as

$$\{a \in A \mid \exists m \in \mathbb{N} \ P(a, m)\},$$

where a natural number m is a computational costs and an output value a is related to m through the predicate P . This complexity is defined by both a value of type \mathbf{A} and the number of processors involved.

In this parallel adaptation, the methodology integrates several new components: an argument representing the logarithm of the number of processors to indicate the degree of parallelism, a special format to describe parallel calls, and an evaluation of the computational load relative to the degree of parallelism.

This approach allows the operations relevant to computational complexity to be described within the type itself. As a result, code can be extracted from Coq and translated to OCaml while preserving the essential features of computational complexity. It's worth noting, however, that proofs in Coq still have to be done manually.

```

1 Definition C (A:Set) (P:A -> nat -> Prop) : Set :=
2   {a : A | exists (m:nat), (P a m)}.

```

Fig. 9. Definition of monad C

Hoffmann et al. [16] introduced a method for static and automated analysis of the costs associated with parallel programs. This approach is based on the principle that the cost of a parallel program can be evaluated using two key metrics: *work*, which represents the total computation time, and *depth*, which represents the duration of the longest computation. Using these criteria, they developed a consistent analysis method that has been implemented in Resource Aware ML (RaML), where its practical applicability has been confirmed.

3 Time efficiency analysis of parallel programs

In this section, we begin by exploring the computational model underlying a parallel program. We then introduce a function library that facilitates execution time inference in Liquid Haskell. Building on the function library, we then implement specific parallel programs using the function library within Liquid Haskell. Finally, we analyze the execution time of the programs, providing insight into their performance characteristics.

3.1 Our parallel computation model: PRAM

When analyzing time and space computations, it is essential to understand the underlying computer model. For sequential programs, models such as Turing machines and random access machines (RAMs) are commonly used.

Parallel random access machines (PRAMs) [17][18] are the dominant model for parallel computing. A PRAM consists of a series of processors, each of which is a RAM with local memory and shared memory accessible to all processors. In the PRAM model, each processor executes each instruction (or step) in a unit of time, and the cycles of instruction execution are synchronously coordinated. When the PRAM stops, it means that all processors have stopped working.

Thus, the time complexity of a parallel program is defined as the elapsed time until the PRAM stops. Since each step is executed in a single unit of time, this complexity can also be interpreted as the number of steps when the entire program is considered as a single processor.

Conversely, the workload of a parallel program refers to the cumulative number of instructions executed by all processors within the given program. The relationship between workload and time is known as the work-time paradigm.

In the following discussion of a parallel algorithm that takes n inputs, we will denote the number of steps as $T(p, n)$ and the work as $W(n)$, where p is the number of processors.

In general, the relationship between work and time is given by

$$W(n) \leq pT(p, n).$$

The equality holds in cases where all processors execute instructions until the PRAM stops. When $p = 1$, that is, when the parallel program is executed sequentially, the relationship simplifies to $W(n) = T(1, n)$.

3.2 Parallel Execution Functions

In this study, the method of Handley et al. [13] is extended so that the definition of the tick monad remains identical. Here, `tval` refers to a type value, while `tcost` represents the time complexity of the parallel program as defined earlier in this study. Although the terms “number of steps” and “time complexity” are often used interchangeably, in the context of additive processing it is more natural to interpret them as the number of steps. Therefore, they are primarily considered as such throughout the process.

During the course of this study, two important functions for parallel execution were carefully designed and implemented. This section serves only as a brief introduction, with full details provided in the later section on implementation.

```

1 {-@ evalPair :: t12:(Tick a, Tick b) -> lbp:Nat
2   -> { t:Tick (a, b) |
3     tcost t == if lbp == 0
4               then (tcost (fst t12) + tcost (snd t12))
5               else tcost (fst t12) &&
6     tval t == (tval (fst t12), tval (snd t12)) }
7   @-}
8 evalPair :: (NFData a, NFData b) =>
9   (Tick a, Tick b) -> Int -> Tick (a, b)

```

Fig. 10. Function `evalPair`

Function `evalPar` The `evalPair` function accepts a tuple consisting of two values of type `Tick`, specifically `(Tick a, Tick b)`, along with information about the logarithm of the number of processors `lbp`. It returns a value of type `Tick` encapsulating the tuple `(a, b)`.

In this storyline, each of the two tick types within the argument tuple has a `tcost`, which represents the number of steps required to execute or evaluate a single tick value.

The `evalPair` function takes the logarithm of the number of processors as an argument. If the number of processors is 1, a sequential computation is performed. If the number of processors is greater than 1, a parallel computation is performed.

In sequential computation, as shown in Figure 11, the first and second elements are evaluated sequentially by the same processor. Consequently, the `tcost`, which represents the number of steps for both elements, is accumulated.

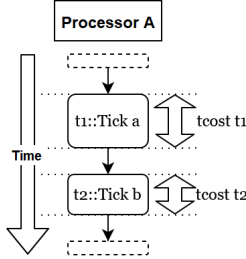


Fig. 11. Sequential Computation of Pair

Unlike sequential computation, parallel computation evaluates the processes simultaneously, as shown in Figure 12. Specifically, the first process is assigned to an available processor while the second process runs on the original processor. Only after both processes are completed does the next process begin.

When evaluation and synchronization are performed in parallel, the total number of steps is determined by the greater of the step counts of the two processors. However, it's assumed that the first element is passed with a higher `tcost`. See the implementation chapter for more details.

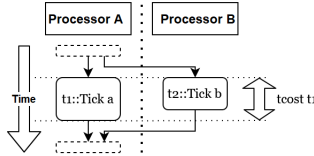


Fig. 12. Parallel Computation of Pair

The `(>>=)` function shown in Figure 13 is defined as a helper function. While its arguments are identical to those of the `bind` operator `(>>=)`, the constraints imposed by the refinement type are different. Within the `(>>=)` function, the specific constraint is that the `tcost` of the tick in the return value of the argument `f` must be equal to 1. This particular constraint allows the inference of `tcost (f (tval t1))` to be replaced by 1, thus facilitating inference within the SMT solver.

```

1 {-@ (>=)/ :: t1:Tick a
2     -> f:(a -> { tf:Tick b | tcost tf == 1 })
3     -> { t:Tick b |
4         tval (f (tval t1)) == tval t &&
5         tcost t1 + 1 == tcost t }
6 @-}
7 infixl 4 >= /
8 (>=)/ :: Tick a -> (a -> Tick b) -> Tick b

```

Fig. 13. Function EqBind

```

1 {-@ parList :: c:Nat
2     -> { xs:[{ t1:Tick a | tcost t1 <= c }] | xs /= [] }
3     -> { p:Nat | p >= length xs }
4     -> { t:Tick {
5         os:[a] | length xs == length os
6         } | tcost t == c }
7 @-}
8 parList :: (NFData a) =>
9   Int -> [Tick a] -> Int -> Tick [a]

```

Fig. 14. Function parList

Function parList The `parList` function accepts a list of type `Tick [Tick a]`, a cost parameter, and the number of processors. It then returns a value of type `Tick [a]`, with the list encapsulated within the `Tick` type.

As depicted in Figure 15, the `parList` function evaluates each element of the list in parallel.

The `parList` function imposes certain refinement type constraints. The total `tcost` of the list of ticks received as an argument must be less than or equal to the specified cost, and the number of processors received must be greater than or equal to the length of the array. These constraints ensure that each element of the array can be evaluated in parallel using the processors provided. Consequently, the total `tcost` can be equal to the cost received as an argument.

To facilitate the use of the `parList` function, additional functions `chunk` (see Figure 16) and `mapParList` have been implemented. The `chunk` function takes a number representing the desired list division and returns a segmented list by dividing the input list accordingly. A brief overview of these utility functions is also provided.

The `mapParList` function performs the functions passed to it in parallel, using the previously defined `parList` function to achieve this parallelization.

The following section uses these functions to analyze the execution time of a parallel program.

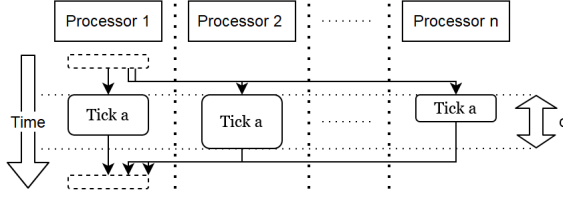


Fig. 15. Parallel Computation on Lists

```

1 {-@ chunk :: { n:Nat | n /= 0 } -> { m:Nat | m /= 0 }
2   -> { xs:[a] | length xs == n * m }
3   -> { yss:[ { ys:[a] | length ys == n } ] |
4     length yss == m }
5   / [m]
6   @-}
7 chunk :: Int -> Int -> [a] -> [[a]]

```

Fig. 16. Function chunk

4 Exection time analysis in parallel execution for specific programs

4.1 Parallel Fibonacci Function

The Fibonacci sequence is a sequence of numbers defined by the following recurrence formulas.

$$\begin{aligned}
 F_0 &= 1 \\
 F_1 &= 1 \\
 F_{n+2} &= F_{n+1} + F_n \quad (n \geq 0)
 \end{aligned}$$

```

1 {-@ mapParList :: { n:Nat | n > 0 }
2   -> { xs:[a] | length xs == n }
3   -> c:Nat
4   -> (a -> { t:Tick b | tcost t <= c })
5   -> { t:Tick { os:[b] | length xs == length os } |
6     tcost t == c }
7   @-}
8 mapParList :: (NFData b) =>
9   Int -> [a] -> Int -> (a -> Tick b) -> Tick [b]

```

Fig. 17. Function mapParList

Function `fib` defined in Figure 18 calculates the n th number in the Fibonacci sequence.

```

1 {-@ fib :: n:Nat -> { v:Nat / v >= n } @-}
2 fib :: Int -> Int
3 fib 0 = 1
4 fib 1 = 1
5 fib n = fib (n - 1) + fib (n - 2)

```

Fig. 18. Function `fib`

The `fib` function shown in Figure 18 can be conveniently parallelized by modifying the part that makes recursive calls to itself. We can consider a parallelized version called `fibPar`, which preserves the time complexity as `tcost`.

The `fibPar` function takes two inputs: `n`, and `lbp`, which is the logarithm to the base 2 of the number of processors. It then returns a `Tick` type where the value is the n th Fibonacci number, F_n . The implementation of the `fibPar` function is shown in Figure 33.

```

1 {-@ fibPar :: n:Nat -> lbp:Nat
2   -> { t:Tick Int / (tcost t == fibCost n lbp) } @-}
3 fibPar :: Int -> Int -> Tick Int
4 fibPar 0 _ = wait 1
5 fibPar 1 _ = wait 1
6 fibPar n 0 =
7   ((x, y) `evalPair` 0)
8   >>= / (\(a, b) -> wait (a + b))
9   where
10     x = fibPar (n - 1) 0
11     y = fibPar (n - 2) 0
12 fibPar n lbp =
13   ((x, y) `evalPair` lbp)
14   >>= / (\(a, b) -> wait (a + b))
15   where
16     x = fibPar (n - 1) (lbp - 1)
17     y = fibPar (n - 2) (lbp - 1)

```

Fig. 19. Function `fibPar`

If the input is either 0 or 1, the value returned is 1. The `wait` function takes a value and returns a tick with `tcost` set to 1.

For an input of $n > 1$, we want to compute the sum of `fib(n-1)` and `fib(n-2)`. So we assign the result of the call to the `fibPar` function so that `fib(n-1)` is computed for `x` and `fib(n-2)` is computed for `y`. If the number of processors is 2 or more (i.e., the base-2 logarithm of the number of processors is 1 or more), then the base-2 logarithm of the number of processors minus 1 is passed as an argument to the `fibPar` function. At this stage, `x` and `y` are not evaluated.

Next, the tuple `(x,y)` and the two base-2 logarithms of the number of processors are passed to the `evalPair` function. If two or more processors are available, the `evalPair` function evaluates the operation in parallel; if only one processor is available, it evaluates the operation sequentially. Since `x` and `y` are calls to the `fibPar` function, this operation is repeated recursively.

The result of `evalPair` becomes the second argument to the `(>>=)` function for both sequential and parallel processing. The `(>>=)` function passes the pairs of `x` and `y` values to the `wait` function, which takes the sum of the pairs and returns it to the `wait` function as a `Tick` type representing the result of the sum. Since the `wait` function specifies 1 as `tcost`, the total `tcost` is the sum of the `tcost` of the `Tick` type resulting from the `evalPair` function and 1.

However, due to Liquid Haskell's inference specification, it is necessary to explicitly distinguish between the cases where `lbp = 0` and where `lbp > 0`.

Static Analysis of Execution Time of Parallel Fibonacci Function Function `fibCost` is defined to represent the execution time of `fibPar`, as illustrated in Figure 20. Using the inputs n and lbp , the value returned by `fibCost` can be

```

1 {-@ fibCost :: n:Nat -> lbp:Nat -> Nat @-}
2 fibCost :: Int -> Int -> Int
3 fibCost 0 _ = 1
4 fibCost 1 _ = 1
5 fibCost n lbp
6   | n <= lbp + 1 = n
7   | otherwise = 2 * fib (n - lbp) - 1 + lbp

```

Fig. 20. Function `fibCost`

expressed as $T(n, lbp)$ as follows.

$$\begin{aligned}
 T(0, lbp) &= 1, \\
 T(1, lbp) &= 1, \\
 T(n, lbp) &= n, & (n \leq lbp + 1) \\
 T(n, lbp) &= 2 \cdot \text{fib}(n - lbp) - 1 + lbp, & (\text{otherwise}).
 \end{aligned}$$

Thus, if lbp is $lbp < n$ for sufficiently large n , then we have

$$T(n, lbp) = O\left(\left(\frac{1 + \sqrt{5}}{2}\right)^{n-lbp}\right);$$

if lbp is $n \leq lbp + 1$ for sufficiently large n , then we have

$$T(n, lbp) = O(n)$$

The results show that when the number of parallel processors is limited, the computation time grows exponentially. However, when there is sufficient parallelism, the computation is completed in linear time. In contrast, when lbp exceeds n , it implies that the logarithm of the number of processors is greater than n , a scenario that is impractical since it would require approximately 2^n processors. Below is a brief explanation that confirms the correctness of the above time complexity for each scenario.

Case (either $n = 0$ or $n = 1$). The case where $n \leq lbp + 1$ occurs when there are enough processors to allow for as much parallelization as desired. When the `fibPar` function receives n , it passes $n - 1$ and $n - 2$ to subsequent calls of the `fibpar` function, executing them in parallel. The case with the larger number of steps involves passing $n - 1$. If the function continues to follow the path with the large number of steps, it will eventually stop at $n = 1$. The number of steps from n to 2 is increased by one by the `wait` function then adding, and again at 1 when returning, ensuring that the number of steps aligns with the value n for `tcost`.

On the other hand, for the cases where $1 < n$ and $lbp \leq n$, we divide the cases into those with $lbp = 0$ and those without $lbp = 0$.

Case ($1 < n$ and $lbp = 0$.) In this case, the number of processors is 1. Except for the cases $n = 0$ and $n = 1$, it holds that $n > 1$. Since $n \leq lbp + 1$ is not satisfied, this is the fourth case of the definition of $T(n, lbp)$. Substituting lbp for 0, we get that the cost is $2 \times fib(n) - 1$.

There is an intuitive explanation for this value. When $lbp = 0$ and $n > 1$, the result of $fib(n)$ is the sum of $fib(0)$ or $fib(1)$ as follows.

$$\begin{aligned} fib(2) &= fib(1) + fib(0) = 2 \\ fib(3) &= fib(2) + fib(1) \\ &= fib(1) + fib(0) + fib(1) = 3 \\ fib(4) &= fib(3) + fib(2) \\ &= fib(2) + fib(1) + fib(1) + fib(0) + fib(1) \\ fib(n) &= \overbrace{fib(1) + \dots + fib(1)}^{fib(n)} \end{aligned}$$

Since $fib(0) = fib(1) = 1$, the sum of $fib(0)$ and $fib(1)$, which make up $fib(n)$, is $fib(n)$. When $fib(0)$ or $fib(1)$ is returned in the *FibPar* calculation, **tcost** is set to 1. 1 is also added to **tcost** when $fib(n-1)$ and $fib(n-2)$ are added. Therefore, since the number of steps $fib(n)$ is added by the wait function when returning the value and the number of steps $(fib(n)-1)$ is added when adding, the total number of steps is $2fib(n)-1$.

Case ($1 < n$ and $0 < lbp \leq n$).

In this case, the calculation is parallelized up to the point where there is a surplus of processors and parallelization is possible, and sequential processing is performed once the tasks have been assigned to each processor.

As an example, Figure 33 shows the execution of **fibPar**(5, 2).

The **fibPar**(5, 2) performs the computation of $fib(5)$ on 2^2 processors. **fibPar**(5, 2) calls **fibPar**(4, 1) and **fibPar**(3, 1) in parallel. In **fibPar**(4, 1), **fibPar**(3, 0) and **fibPar**(2, 0) are called in parallel, and these $fib(3)$ and $fib(2)$ computations are done in parallel on each processor. The same is true for **fibPar**(3, 1), where the calculations of $fib(2)$ and $fib(1)$ are performed in parallel on each processor. The largest number of steps is in $fib(3)$, and as mentioned above, it is $2fib(3)-1$. Therefore, the number of steps in **fibPar**(4, 1) is $2 \times fib(3)$, which is the number of steps plus 1 in the addition process. Therefore, the number of steps in **fibPar**(5, 2) is

$$2fib(3) + 1 = 2fib(5-2) - 1 + 2$$

, which is the number of steps plus 1 in the addition process.

The above is an analysis of the time complexity of the parallel fib function and an intuitive explanation of it.

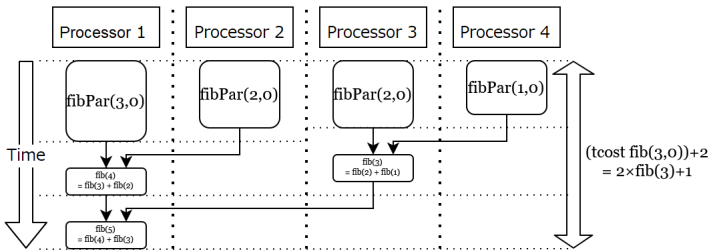


Fig. 21. Execution of $fib(5, 2)$

4.2 Power function 2^n in parallel execution

A **pow2** function means a function that computes 2^n for a given n . The following is an example implementation of the **pow2** function. Although more efficient

```

1 {-@ pow2 :: n:Nat -> { v:Nat | v >= n } @-}
2 pow2 :: Int -> Int
3 pow2 0 = 1
4 pow2 n = pow2 (n - 1) + pow2 (n - 1)

```

Fig. 22. Function `pow2`

implementations exist, for the purpose of this parallelization introduction, the implementation shown in Figure 22.

Similar to the `fibPar` function, the `pow2Par` function, which computes `pow2` in parallel, can be defined as shown in Figure 23 by parallelizing the process of recursively calling it.

```

1 {-@ pow2Par :: n:Nat -> lbp:Nat
2   -> { t:Tick Int | (tcost t == pow2Cost n lbp) } @-}
3 pow2Par :: Int -> Int -> Tick Int

```

Fig. 23. Function `pow2Par`

The `pow2Cost` function is a function that represents the execution time of the `pow2Par` function and is defined as shown in Figure 24.

```

1 {-@ reflect pow2Cost @-}
2 {-@ pow2Cost :: Nat -> lbp:Nat -> {v:Nat | v >= 1} @-}
3 pow2Cost :: Int -> Int -> Int
4 pow2Cost 0 _ = 1
5 pow2Cost n lbp
6   | n <= lbp = n + 1
7   | otherwise = pow2 (n + 1 - lbp) - 1 + lbp

```

Fig. 24. Function `pow2Cost`

The value returned by `pow2Cost` can be expressed by the following equation using the input n and lbp .

$$T(n, lbp) = \begin{cases} 1 = O(1) & (n = 0) \\ n + 1 = O(n) & (0 < n \wedge n \leq lbp) \\ 2^{n+1-lbp} - 1 + lbp = O(2^{n-lbp}) & (\text{Otherwise}) \end{cases}$$

The correctness of the time complexity of the `pow2Par` function can be seen by considering it in the same way as for the `fibPar` function. This result is consistent with the results of the time complexity analysis of the parallel `pow2` function in the paper [14]. Note, however, that the GHC optimization does not have the same time complexity as the `pow2` function, since the two values to be added are the same expression.

4.3 Parallel traversal of complete binary tree

The height of a tree refers to the greatest distance from the root node to any leaf node in the tree. A perfect binary tree, also known as a complete binary tree, is characterized by having the heights of the left and right subtrees of each node exactly equal.

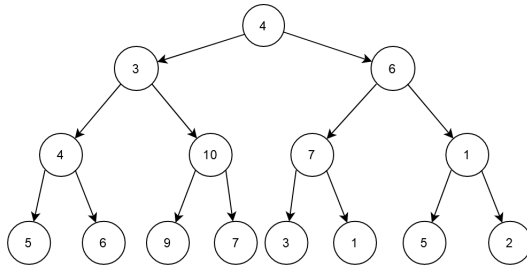


Fig. 25. Example of a perfect binary tree

In Liquid Haskell, a perfect binary tree can be defined as in Figure 26.

`PBT` is a datatype in Haskell that signifies a complete binary tree. While it essentially represents a binary tree with both height and key attributes, its completeness is assured through the use of a refinement type constraint. The function `getHeight` is designed to retrieve the height of the given binary tree. This datatype adheres to a specific constraint, requiring that the heights of both left and right subtrees be exactly one less than the overall height h of the binary tree, thus ensuring that `PBT` is a complete binary tree. It's important to note, though, that under this definition, the leaves of the tree do not contain values

```

1  {-@ measure getHeight @-}
2  {-@ getHeight :: PBT a -> Nat @-}
3  getHeight :: PBT a -> Int
4  getHeight Leaf = 0
5  getHeight (Node _ h _ _) = h
6
7  {-@
8    data PBT a
9      = Leaf
10     | Node
11       { key :: a,
12         h :: Nat,
13         l :: { v:PBT a | getHeight v == h - 1 },
14         r :: { v:PBT a | getHeight v == h - 1 }
15       }
16   @-}
17 data PBT a
18   = Leaf
19   | Node
20     { key :: a,
21       h :: Int,
22       l :: PBT a,
23       r :: PBT a
24     }
25 deriving (Show)

```

Fig. 26. Definition of Perfect Binary Tree

```

1  {-@ sumPBT :: { p:PBT Nat | getHeight p /= 0 }
2    -> lbp:Nat
3    -> { t:Tick Int /
4      (tcost t == pow2Cost (getHeight p - 1) lbp) }
5    @-}
6  sumPBT :: PBT Int -> Int -> Tick Int
7  sumPBT (Node key 1 _ _) _ = wait key
8  sumPBT (Node key _ 1 r) 0 =
9    ((x, y) `evalPair` 0)
10   >>= / (\(a, b) -> wait (a + b + key))
11   where
12     x = sumPBT l 0
13     y = sumPBT r 0
14  sumPBT (Node key _ 1 r) lbp =
15    ((x, y) `evalPair` lbp)
16    >>= / (\(a, b) -> wait (a + b + key))
17    where
18      x = sumPBT l (lbp - 1)
19      y = sumPBT r (lbp - 1)

```

Fig. 27. Function `sumPBT`

The parallel time complexity of the `sumPBT` function can be expressed by the function `pow2Cost`, which expresses the parallel time complexity of the function `pow2Par`. This is because the `pow2Par` function, which takes two parallel calls with a small value as an argument, and the `sumPBT` function, which takes two parallel calls with a small height complete binary tree as an argument, have the same structure. The `pow2Cost` function takes `getHeight p - 1` as an argument. In this definition, the recursion ends when the full binary tree reaches height 1, so the height of the full binary tree minus 1 is passed.

Calculating the Total Amount of Values of All Nodes When computing the sum of all nodes in a tree, it is essential to traverse each node. This process can be greatly accelerated by parallelization.

The `sumPBT` function is designed to perform this task by calculating the sum of all nodes in parallel. Its definition is shown in the figure 28. Similar to the parallel fib function and the parallel `pow2` function, the part of the process that involves recursive function calls is parallelized, delegating the work to another available processor.

The parallel execution time of the `sumPBT` function can be represented by the `pow2Cost` function, which also describes the parallel execution time of the `pow2Par` function. This correspondence arises because both the `pow2Par` function, taking two parallel calls with one smaller value, and the `sumPBT` function, taking two parallel calls with a complete binary tree of reduced height, share the same structural design. Within this definition, the `pow2Cost` function accepts

```

1 {-@ sumPBT :: { p:PBT Nat | getHeight p /= 0 }
2   -> lbp:Nat
3   -> { t:Tick Int |
4     (tcost t == pow2Cost (getHeight p - 1) lbp) }
5   @-}
6 sumPBT :: PBT Int -> Int -> Tick Int
7 sumPBT (Node key l _ r) _ = wait key
8 sumPBT (Node key _ l r) 0 =
9   ((x, y) `evalPair` 0)
10  >>= / (\(a, b) -> wait (a + b + key))
11  where
12    x = sumPBT l 0
13    y = sumPBT r 0
14 sumPBT (Node key _ l r) lbp =
15   ((x, y) `evalPair` lbp)
16   >>= / (\(a, b) -> wait (a + b + key))
17   where
18     x = sumPBT l (lbp - 1)
19     y = sumPBT r (lbp - 1)

```

Fig. 28. Function `sumPBT`

`getHeight(p) - 1` as an argument, since the recursion terminates when the full binary tree attains a height of 1, meaning the height of the full binary tree minus one is passed.

Computing the maximum among all nodes in parallel The `maxPBT` function, which computes the maximum value of all nodes in parallel, can be defined in a manner analogous to the `sumPBT` function, as shown in Figure 4.3. Like summation, maximum computation requires traversing all nodes, but parallelization allows the left and right subtrees to be processed by different processors. The time complexity for the `maxPBT` function is identical to that of the `sumPBT` function.

```

1 {-@ maxPBT :: { p:PBT Nat | getHeight p /= 0 }
2   -> lbp:Nat
3   -> { t:Tick Int |
4     (tcost t == pow2Cost (getHeight p - 1) lbp) }
5   @-}
6 maxPBT :: PBT Int -> Int -> Tick Int

```

Fig. 29. Function `maxPBT`

4.4 Parallel division of list processing

Next, consider a case where a long list is to be processed, and the process can be divided and parallelized to speed up the process.

The `countSolve` function takes a list of Sudoku problems consisting of strings and counts the number of solutions obtained. The number of steps required to solve a Sudoku problem is `calcCost`, and the total number of steps is `calcCost` multiplied by the length of the list. The implementation of the `countSolve` function is not covered here.

```

1  {-@ countSolve :: { xs:[String] | length xs /= 0 }
2    -> { t:Tick Int | tcost t == calcCost * length xs }
3    / [length xs]
4  @-}
5  countSolve :: [String] -> Tick Int
6
7  {-@ countSolveList :: { n:Nat | n > 0 }
8    -> { m:Nat | m > 0 }
9    -> { xs:[String] | length xs == n * m }
10   -> { t:Tick [Int] | tcost t == calcCost * n }
11 @-}
12 countSolveList :: Int -> Int
13   -> [String] -> Tick [Int]
14 countSolveList n m xs =
15   mapParList m ys (calcCost * n) countSolve
16 where
17   ys = chunk n m xs
18   c = calcCost * n

```

Fig. 30. Function `countSolve` and function `countSolveList`

When the `countSolve` function is applied to a list, it yields a `Tick Int` result from sequential processing. The `countSolveList` function is a parallelized version of this process. During parallelization, the `mapParList` and `chunk` functions can be employed.

First, the `chunk` function divides the given list into m lists, each of length n , with the number of processors (or fewer) passed as m . The result, `ys`, is of type `[[String]]`, representing a list of strings; it's then passed to the `mapParList` function, and each element `[String]` is handed to the `countSolve` function for parallel evaluation.

Since the processing occurs on the subdivided list in parallel, the total number of steps corresponds to the number of steps performed by the `countSolve` function on a single processor, that is, `calcCost` multiplied by n .

As demonstrated, functions that process arrays and return ticks can be parallelized with relative ease by dividing the array by the number of processors.

5 Implementation of functions for parallel execution time analysis

5.1 Parallel Haskell

There are several parallelization methods in Haskell. Among them, the method using the package `parallel` is described. The simplest way to achieve parallelism is to use the `par` and `pseq` functions in the package `Control.Parallel`.

```
1 par :: a -> b -> b
2 pseq :: a -> b -> b
```

Fig. 31. Function `par` and `pseq`

The `par` function indicates that it is useful to evaluate the first and second arguments in parallel and returns the second argument. A Spark corresponding to the first argument is generated and added to the spark pool. The sparks are computations waiting to be executed, and the spark pool is the place where they are stored. A spark added to the spark pool is evaluated and allocated to the remaining processor using a method called work stealing.

The `pseq` function evaluates the first argument in positive form and returns the second argument. However, it does not evaluate to the canonical form, so if you want to evaluate to the canonical form, use the `deepseq` function.

Parallelism can be described using the above two functions. However, the `par` and `pseq` functions must be written in the middle of an expression, making parallelism algorithm-dependent. Strategies, which are included in the same package. (See Figure 32.)

```
1 infixl 0 `using`
2 using :: a -> Strategy a -> a
3
4 rdeepseq :: NFData a => Strategy a
5 rpar :: Strategy a
6
7 rparWith :: Strategy a -> Strategy a
8
9 evalTuple2 :: Strategy a -> Strategy b
10   -> Strategy (a, b)
11 parList :: Strategy a -> Strategy [a]
```

Fig. 32. `Control.Parallel.Strategies`

The `Strategy` type indicates which evaluation strategy is used. A `using` function is a function that takes a value and an evaluation strategy, and evaluates the value using that strategy. The `using` type can be used as an infix operator. For example, you can write `(f x) 'using' rpar` to separate the value, the algorithm to compute it, and the evaluation strategy. The `rdeepseq` function evaluates to normal form, and the `rpar` function returns a parallel evaluation strategy. The `rparWith` function adds evaluation in addition to parallel evaluation. For example, passing the `rdeepseq` function to the `rparWith` function allows parallel evaluation up to normal form. The `evalTuple2` function takes two evaluation strategies and returns a strategy that applies the strategy to the first and second elements. In the following sections, these functions are combined to implement parallel processing.

5.2 Naive implementation of parallel execution analysis

Let us consider extending time complexity analysis in Liquid Haskell to parallel processing. First, as a concrete situation, we will attempt an execution time analysis of a parallel Fibonacci function.

As a simple implementation, we choose to directly extract `tval` and `tcost` from the `Tick` monad and perform appropriate operations on them. The definition of the function `fibPar`, implemented as a parallel Fibonacci function, is shown in Figure 33.

The result itself is identical to the parallel Fibonacci function introduced in the Section 4.1. The sequential processing for $lbp = 0$ and the parallel processing for $lbp > 0$ are also identical.

On the other hand, it is possible to reason about `tval` compared to the case of Section 4.1. `tval t == fib n` guarantees as a refinement type constraint that the returned value is equal to the same input `n` given to the Fibonacci function in sequential processing.

However, the inference fails if the type returned by the `fibPar` function is `Tick Nat`. It is expected that we should be able to infer that the value of the Fibonacci function is `Nat`, i.e., a non-negative integer. The problem here, however, is that the arguments of the `rdeepseq` function are required to satisfy the type class `NFData`. Since the type `Nat` is an annotated type in Liquid Haskell and does not satisfy the type class `NFData`, the return type of the `rdeepseq` function is interpreted as `Int`, not `Nat`, and therefore `Strategy Int` is returned. `Int` is returned through `usingStrategy`.

The `usingStrategy` function is a function that evaluates the first argument based on the evaluation strategy of the second argument and returns the same value as the argument.

5.3 Toward generalization of the implementation

In the previous example of the parallel Fibonacci function, `tcost` was manipulated directly in ticks. However, directly manipulating `tcost` can inadvertently

```

1 {-@ fibPar :: n:Nat -> lbp:Nat ->
2   { t:Tick Int |
3     (tcost t == fibCost n lbp) && (tval t == fib n) }
4   @-}
5 fibPar :: Int -> Int -> Tick Int
6 fibPar 0 _ = wait 1
7 fibPar 1 _ = wait 1
8 fibPar n 0 = Tick (c1 + c2 + c3) v3
9   where
10     x0 = fibPar (n -1) 0
11     y0 = fibPar (n -2) 0
12     Tick c1 v1 = x0 `usingStrategy` rdeepseq
13     Tick c2 v2 = y0 `usingStrategy` rdeepseq
14     Tick c3 v3 = wait (v1 + v2)
15 fibPar n lbp = Tick (c1 + c3) v3
16   where
17     x0 = fibPar (n -1) (lbp-1)
18     y0 = fibPar (n -2) (lbp-1)
19     Tick c1 v1 = x0 `usingStrategy` rparWith rdeepseq
20     Tick c2 v2 = y0 `usingStrategy` rdeepseq
21     Tick c3 v3 = wait (v1 + v2)

```

Fig. 33. Function `fibPar`

```

1 {-@ usingStrategy :: x:a
2   -> Strategy a -> {v:a | v == x } @-}
3 usingStrategy :: a -> Strategy a -> a

```

Fig. 34. Function `usingStrategy`

increase or decrease the number of steps. Here we consider defining and invoking a parallel processing function that handles the number of steps appropriately.

First, we try to generalize a function that returns a `Tick` value as an argument. Specifically, we take two functions that take a value and return a `Tick` value, pass arguments to them, and actually use them sequentially or in parallel to return a `Tick` value with `tcost` appropriately added. However, obtaining the `tcost` when applied to a function that returns a tick requires higher-level reasoning, and for the same reason that the `bind` function cannot be reasoned about, it did not work.

So we decided to pass the `Tick` value itself instead of the function. Specifically, we consider a function that takes a pair of `Tick` values and returns a `Tick` value with the pair as an element. This is the `evalPair` function, one of the functions implemented in this study.

5.4 Functions to process in parallel

In this section, we explain how to implement functions to process in parallel.

Function `evalPair` The `evalPair` function is introduced in the previous section. Here we discuss its internal implementation. The `evalPair` function calls the sequential processing function `seqPair` if the logarithm of the processor's 2, `lbp`, is 0, and the parallel processing function `parPair` if it is positive.

Due to the nature of the Parallel Haskell, we want to guarantee that the results do not change when parallelized. However, due to the limitations of the SMT solver's reasoning, the `evalPair` function is not able to reason about this in actual use, although it is included in the refinement type constraints.

```

1 {-@ evalPair :: t12:(Tick a, Tick b) -> lbp:Nat
2   -> { t:Tick (a, b) |
3     tcost t ==
4       if lbp == 0
5         then (tcost (fst t12) + tcost (snd t12))
6         else tcost (fst t12) &&
7       tval t == (tval (fst t12), tval (snd t12)) }
8   @-}
9 evalPair :: (NFData a, NFData b)
10   => (Tick a, Tick b)
11   -> Int -> Tick (a, b)
12 t12 `evalPair` 0 = seqPair t12
13 t12 `evalPair` lbp = parPair t12 lbp

```

Fig. 35. Function `evalPair`

Function seqPair The definition of the `seqPair` function is shown in the Figure 36. The `seqPair` function takes a pair `xy0` consisting of ticks as argument and returns the result of the sequential evaluation of the pair.

The `evalTuple2` function takes the first and second evaluation strategies of a two-element tuple and returns the evaluation strategy of the two-element tuple. Assuming that the first and second are evaluated to normal form by `rdeepseq`, the elements of the pair are evaluated sequentially. The evaluation results `t1` and `t2` are each of type `Tick`, so they are decomposed, their `tcosts` are added, and `tval` is returned as a pair.

```

1 {-@ seqPair :: t12:(Tick a, Tick b)
2   -> { t:Tick (a, b) |
3     tcost t == tcost (fst t12) + tcost (snd t12) &&
4     tval t == (tval (fst t12), tval (snd t12)) }
5 @-}
6 seqPair :: (NFData a, NFData b)
7   => (Tick a, Tick b) -> Tick (a, b)
8 seqPair xy0 =
9   Tick (tcost t1 + tcost t2) (tval t1, tval t2)
10  where
11    (t1, t2) =
12      xy0 `usingStrategy`
13      evalTuple2 rdeepseq rdeepseq

```

Fig. 36. Function `seqPair`

Function parPair The definition of the `parPair` function is shown in Figure 37. The `parPair` function takes as arguments a pair `xy1` consisting of ticks and `lbp`, the logarithm of the number of processors, and evaluates them in parallel, returning the result. The logarithm of the number of processors is used to verify that it is positive, i.e., that there are available processors, and is not used for processing. It has much in common with the `seqPair` function, except that it performs parallel processing by adding `rparWith` before `rdeepseq` as a strategy for evaluating the first element, and it uses only `t1` to calculate `tcost`.

The `parPair` function assumes that the `tcost` of the first element of the pair is greater than the second element as an implicit constraint that is not a refinement type constraint. This is to simplify inference by assigning the first `tcost` as the resulting `tcost`.

We will consider the case where the implementation is such that the one with the larger `tcost` is adopted. Even with such an implementation, inferences can be made when the left and right `tcosts` match, as in the case of the `pow2Par` function. On the other hand, in the case of the `fibPar` function, the left and right `tcosts` do not match, and furthermore, the SMT solver cannot infer which

of the left and right `tcosts` is larger. Therefore, as a temporary measure, we chose to set an implicit constraint.

```

1 {-@ parPair :: t12:(Tick a, Tick b)
2   -> { lbp:Nat / lbp > 0 }
3   -> { t:Tick (a, b) /
4     tcost t == tcost (fst t12) &&
5     tval t == (tval (fst t12), tval (snd t12)) }
6   @-}
7 parPair :: (NFData a, NFData b)
8   => (Tick a, Tick b) -> Int -> Tick (a, b)
9 parPair xyl _ = Tick (tcost t1) (tval t1, tval t2)
10 where
11   (t1, t2) =
12     xyl `usingStrategy`
13     evalTuple2 (rparWith rdeepseq) rdeepseq

```

Fig. 37. Function `parPair`

5.5 Function for List processing in parallel

Similar to `parPair`, which performs parallel processing on pairs, consider a function that performs parallel processing on lists.

Function `parList` The `parList` function is introduced in the previous section. Here, we discuss the internal implementation.

The `parList` function specifies `c` as the cost of the tick and the value as the result of executing the list `xs` in parallel. The `usingStrategy` function, the `parList` function, and the `rdeepseq` function are used to evaluate the list `xs` in parallel up to the normalized form. The `go` function is an auxiliary function that retrieves a list from a list of ticks.

The cost `c` is not necessary if the largest `tcost` is taken from a list `xs` of `Tick` types. However, in Liquid Haskell reasoning, it is difficult to take such a value and use it as a constraint. Although it would be more natural to use `xs` as the first argument, we use this order because the cost `c` must be passed first in order to specify that all the elements of `xs` satisfy the condition.

For example, this problem could be solved if the list to be received were a data structure representing a list of `Tick` types below a certain value. However, the current form, which simply accepts a list of costs and `Tick` types, is more convenient when combined with other operations.

```

1 {-@ parList :: c:Nat
2   -> { xs:[{ t1:Tick a | tcost t1 <= c }] | xs /= [] }
3   -> { p:Nat | p >= length xs }
4   -> { t:Tick { os:[a] | length xs == length os } |
5     tcost t == c }
6   @-}
7 parList :: (NFData a)
8   => Int -> [Tick a] -> Int -> Tick [a]
9 parList c xs p =
10   Tick c (go xs `usingStrategy` parList rdeepseq)
11   where
12     {-@ go :: xs:[Tick a]
13       -> { ys:[a] | length xs == length ys } @-}
14     go :: [Tick a] -> [a]
15     go [] = []
16     go (x : xs) = cons (tval x) (go xs)

```

Fig. 38. Function parList

Function chunk The `chunk` function takes as input a positive number `n` and `m` and a list `xs` of length `n` and returns a list consisting of `m` lists of length `n`. The refinement type is used to specify the length of the list to be a specified length. Here, the list is returned as a simple list of lists that are not wrapped in a `Tick` type. However, it is possible that `tcost` should be recorded and returned wrapped in a `Tick` type, since the parallel partitioning is part of the process.

```

1 {-@ chunk :: { n:Nat | n /= 0 }
2   -> { m:Nat | m /= 0 }
3   -> { xs:[a] | length xs == n * m }
4   -> { yss:[ { ys:[a] | length ys == n } ] |
5     length yss == m }
6   / [m]
7   @-}
8 chunk :: Int -> Int -> [a] -> [[a]]
9 chunk _ 1 xs = [xs]
10 chunk n m xs = cons as (chunk n (m - 1) bs)
11   where
12     as = take n xs
13     bs = drop n xs

```

Fig. 39. Function chunk

Function map The `map` function takes a cost `c`, a function `f` that returns a `Tick` value, and a list `xs`; the function returns an array of `Tick` values as a result of applying `f` to `xs`. Note that this function simply applies the processing of the function `f` to each element of the list `xs`, and does not perform any parallel processing based behavior. Passing a cost `c` guarantees that the `tcost` of the resulting `Tick` type is less than or equal to `c` as a refinement type constraint.

```

1 {-@ map :: c:Nat
2   -> (a -> { t:Tick b | tcost t <= c })
3   -> xs:[a]
4   -> { ys:[{ t:Tick b | tcost t <= c }] /
5     length ys == length xs }
6 @-}
7 map :: Int -> (a -> Tick b) -> [a] -> [Tick b]
8 map c f [] = []
9 map c f (x : xs) = cons (f x) (map c f xs)

```

Fig. 40. Function `map`

Function mapParList The `mapParList` function is a function that uses the `parList` and `map` functions internally. The function takes an positive integer `n`, a list `xs` of length `n`, a cost `c`, and a function `f` that returns the `Tick` type of cost `c`. The function applies the function `f` to each element of `xs` in parallel and returns a `Tick` value such that the list is of type `tval`. The `tcost` of the returned `Tick` type is guaranteed to be `c`.

The second argument of the `parList` function is the list of ticks after the `map` function has been applied. In other words, the way they are evaluated is not determined at the stage of passing the list to the `map` function, but it is determined that they will be evaluated in parallel by passing them as arguments to the `parList` function. The `tcost` return value of the `map` function is discarded.

6 Validity based on parallel execution time analysis results

In this study, we introduced a method for statically analyzing the number of steps within a parallel program. To validate this method, we measured the actual execution time of a parallel program and compared it to the number of steps determined by our analysis. The term “execution time” in this context refers to the elapsed time from the beginning to the end of the target computation within the given execution environment. The execution environment is as follows. The CPU is AMD Ryzen Threadripper 3970X, which is a 32-core processor, the size of the RAM is 32 GB, and the operating system Ubuntu 20.04.2 LTS.

```

1 {-@ mapParList :: { n:Nat | n > 0 }
2   -> { xs:[a] | length xs == n }
3   -> c:Nat
4   -> (a -> { t:Tick b | tcost t <= c })
5   -> { t:Tick { os:[b] | length xs == length os } |
6     tcost t == c }
7   @-}
8 mapParList :: (NFData b)
9   => Int -> [a] -> Int -> (a -> Tick b) -> Tick [b]
10 mapParList n xs c f = parList c (map c f xs) n

```

Fig. 41. Function `mapParList`

Cores	1st(s)	2nd(s)	3rd(s)	Average(s)	Analyzed Steps
1 = 2 ⁰	10.987433	10.996685	11.023951	11.00269	134,217,727
2 = 2 ¹	7.094920	7.064267	7.036276	7.065154	67,108,864
4 = 2 ²	4.063124	3.993696	4.005592	4.020804	33,554,433
8 = 2 ³	2.457884	2.472770	2.487182	2.472612	16,777,218
16 = 2 ⁴	1.869392	1.882695	1.882147	1.878078	8,388,611
32 = 2 ⁵	2.216256	2.229249	2.196343	2.213949	4,194,308

Table 1. Execution times and analyzed the numbers of steps for a perfect binary tree traversal

In GHC(Glasgow Haskell Compiler), the number of threads that can be run in parallel at the same time can be specified with the runtime option `-N`. It is recommended that this value be equal to the number of CPU cores. Since a 32-core CPU is used in this study, the value specified in `-N`, i.e., the assumed number of processors, is varied from 1 to 32, and the execution time of each is measured.

6.1 Case of Perfect binary tree traversal

The table 1 compares the three execution times, their average and the number of analyzed steps in the traversal of a complete binary tree.

Here, the measurement covers the calculation of the `maxPBT` function in Section 4.3 with the argument $n = 27$ and the argument `lbp` as the logarithm of 2 for the number of processors. However, the generation of the perfect binary tree was done before and is not included in the execution time.

The number of processors was specified as 1, 2, 4, 8, 16, or 32,, and the argument `lbp` was specified as 0, 1, 2, 3, 4, or 5 accordingly.

A line graph with the number of processors on the horizontal axis and the execution time and number of steps analyzed on the vertical axis is shown in Figure 42.

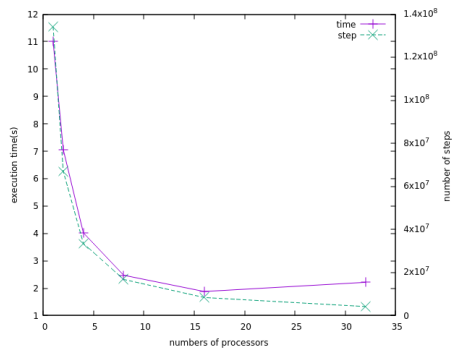


Fig. 42. Comparison between Execution times and analyzed the numbers of steps

The graph shows that as the number of processors (**numbers of processors**) is increased, the execution time (**time**) tends to decrease as well as the number of analyzed steps (**step**).

On the other hand, it is observed that the execution time for the case with 32 processors is not shorter than that for the case with 16 processors.

Furthermore, a scatter diagram is shown in which a regression line is drawn with the number of steps analyzed on the horizontal axis and the execution time on the vertical axis 43

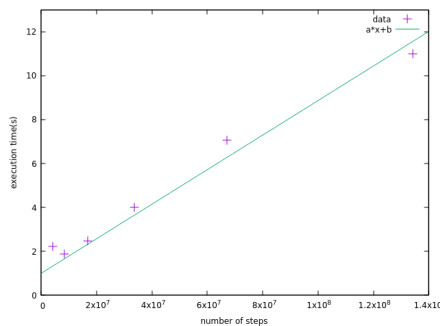


Fig. 43. Scatter diagram of execution times and analyzed the numbers of steps

The correlation coefficient between execution time and the number of steps is 0.9939287316. However, caution must be exercised in interpreting this result, as the small sample size (only six data points) may not provide sufficient evidence to establish immediate practical relevance or significance.

Cores	1st(s)	2nd(s)	3rd(s)	Average(s)	Analyzed Steps
1	13.727178	14.910259	13.647531	14.09499	1.6×10^9
2	7.563414	7.460836	7.429231	7.484494	8.0×10^8
4	4.087452	4.191272	4.068193	4.115639	4.0×10^8
5	3.335491	3.307790	3.386718	3.343333	3.2×10^8
8	2.330277	2.402992	2.358420	2.363896	2.0×10^8
10	1.926612	1.933136	1.911500	1.923749	1.6×10^8
16	1.356350	1.326441	1.350727	1.344506	1.0×10^8
20	1.289420	1.195688	1.184411	1.223173	8.0×10^7
32	1.223408	0.965334	1.026178	1.071640	5.0×10^7

Table 2. Execution times and analyzed the numbers of steps for parallel partitioning of list processing

6.2 Case of parallel partitioning of list processing

The table 2 compares the execution time in parallel partitioning of list processing with the number of analyzed steps.

Here we measured the computation of solving 16,000 Sudoku problems by dividing them by the number of processors using the `mapParList` function in the 4.4 section. The number of processors was set to 1, 2, 4, 5, 8, 10, 16, 20, and 32, respectively, so that the number of processors could be evenly distributed on each processor.

The line graph with the number of processors on the horizontal axis and the execution time and the number of steps analyzed on the vertical axis is shown in Figure 44.

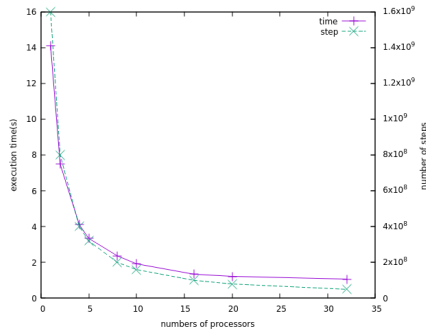


Fig. 44. Comparison between Execution times and analyzed the numbers of steps

Furthermore, the scatter diagram in which the regression line is drawn with the number of steps analyzed on the horizontal axis and the execution time on the vertical axis is shown in Figure 45. The correlation coefficient between execution time and number of steps was 0.9998209612. As in the aforementioned case, the

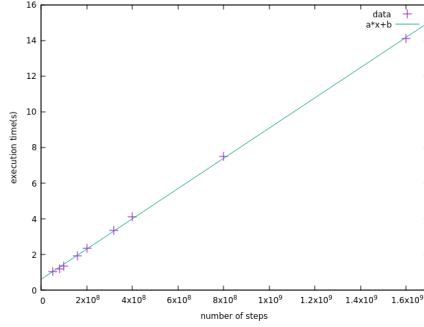


Fig. 45. Scatter diagram of execution times and analyzed the numbers of steps

number of data is also small, so caution is required before directly linking the correlation coefficient to usefulness.

6.3 Discussion on the appropriateness of adopting the PRAM model

In this study, the Parallel Random-Access Machine (PRAM) was chosen as the computational model for parallel computing. PRAM serves as an abstract model that contemplates both time and space complexity of parallel algorithms by simulating the concurrent operation of multiple Random-Access Machines (RAMs). RAM is a well-known computational model that permits constant-time access to any memory location.

Contrastingly, Haskell’s memory model does not allow for constant-time random access, differentiating it from the RAM model.

Another model worth mentioning in the context of parallel computation is the directed acyclic graph (DAG) model. Similar to Parallel Haskell, the DAG model abstracts the complex problem of scheduling tasks to processors. However, this abstraction complicates the analysis of an algorithm’s time complexity.

While the PRAM model was selected in this study to simplify the analysis of time complexity, there remains an opportunity to explore the applicability and relevance of other computational models in future research.

7 Conclusion

In this study, we developed and implemented a method for statically analyzing the execution time of parallel programs using Liquid Haskell. We applied this new method to analyze the execution time of several parallel programs. Furthermore, we executed these parallel programs and measured the elapsed time. The results were reasonably consistent with the number of steps we analyzed, validating our approach to some extent.

Future work in this area includes applying the method proposed in this study to an extended set of parallel program examples. This could range from typical algorithms, such as fast sorting and merge sorting, to more practical, real-world applications.

In addition, Liquid Haskell's ability to facilitate manual proofs, unrestricted by the limitations of automatic reasoning using proof combinators, provides an opportunity for further exploration. In particular, an analysis of the timing dynamics of parallel programs when using proof combinators remains an area for future investigation.

Furthermore, as highlighted in the Discussion section, there is potential to consider the adoption of computational models other than PRAM. This opens up avenues for exploration and potential enhancement of the current approach, providing a more robust and flexible basis for parallel computing analysis.

Acknowledgement.

This work was supported by JSPS KAKENHI Grant Number JP20K11743.

References

1. Jones, S.P.: Haskell 98 language and libraries: the revised report. Cambridge University Press (2003)
2. Haskell 2010 language report (2010), available on: <https://www.haskell.org/onlinereport/haskell2010>
3. Haskell – an advanced, purely functional programming language, <https://www.haskell.org/>.
4. Loidl, H.W.: Granularity in Large-scale Parallel Functional Programming. Ph.D. thesis, University of Glasgow (1998)
5. Trinder, P.W., Hammond, K., Loidl, H.W., Jones, S.L.P.: Algorithm + strategy = parallelism. *Journal of Functional Programming* 8(1), 23–60 (Jan 1998)
6. Marlow, S., Maier, P., Loidl, H.W., Aswad, M.K., Trinder, P.: Seq no more: better strategies for parallel haskell. *ACM SIGPLAN Notice* 45(11), 91–102 (Sep 2010), <https://doi.org/10.1145/2088456.1863535>
7. Marlow, S.: *Parallel and Concurrent Programming in Haskell*. O'Reilly Media, Inc. (2013)
8. Moggi, E.: Notions of computation and monads. *Information and Computation* 93(1), 55–92 (1991), <https://www.sciencedirect.com/science/article/pii/0890540191900524>, selections from 1989 IEEE Symposium on Logic in Computer Science
9. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement types for haskell. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. pp. 269–282. ICFP '14, Association for Computing Machinery, New York, NY, USA (2014)
10. Biere, A., Heule, M., van Maaren, H., Walsh, T.: *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, NLD (2009)
11. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

12. LiquidHaskell, <https://ucsd-progsys.github.io/liquidhaskell-blog/>.
13. Handley, M.A.T., Vazou, N., Hutton, G.: Liquidate your assets: Reasoning about resource usage in liquid haskell. *Proc. ACM Program. Lang.* 4(POPL) (Dec 2019), <https://doi.org/10.1145/3371092>
14. Shiromizu, S., Emoto, K.: Towards Systematic Proof of Parallel Running-time on Coq. *Proceedings of the 34th JSSST Annual Conference* 34, 141–151 (September 2017), <https://ci.nii.ac.jp/naid/40021461942/>, written in Japanese
15. McCarthy, J., Fetscher, B., New, M.S., Feltey, D., Findler, R.B.: A coq library for internal verification of running-times. *Science of Computer Programming* 164, 49 – 65 (2018), special issue of selected papers from FLOPS 2016
16. Hoffmann, J., Shao, Z.: Automatic Static Cost Analysis for Parallel Programs. In: Vitek, J. (ed.) *Programming Languages and Systems*. pp. 132–157. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
17. Wyllie, J.C.: *The Complexity of Parallel Computations*. Tech. rep., Cornell University (1979)
18. JáJá, J.: *An Introduction to Parallel Algorithms*. Addison-Wesley Professional (1992)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License (<http://creativecommons.org/licenses/by-nc/4.0/>), which permits any noncommercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

