2022-2-(7): Manuscript for presentation at IPSJ-SIGPRO, 28 7 2022.

Design and Implementation of a Parallel-native Programming Language

Ken Shibata[1,a)] Shinya Takamaeda[2,b)]

Abstract: Although multi-core CPUs have become mainstream in recent years, many of the current mainstream languages were designed when single-core CPUs were the mainstream, and often require special libraries and syntax to improve execution speed through parallel execution, making parallel execution not easy to achieve. We developed a parallel-oriented programming language, named Coa, which makes parallel execution the default. To avoid data races, it automatically detects variable dependencies and executes programs in parallel, just as out-of-order execution of CPUs detects dependencies among data. Therefore, although parallel execution is performed internally, the behavior seen from the outside is the same as sequential execution, and execution speed can be improved without increasing the complexity of the program. It also has a function that enables sequential execution, in case the processing unit is small and the overhead of sequential execution is large. The interpreter itself is written in Go, and source code written in Coa is executed in parallel in the interpreter using goroutine. Currently, the language can be used to implement basic programs. As a comparison, two experiments, downloading 13 files and computing to display the Mandelbrot set were executed in sequential and parallel processing. The execution time, number of lines of code, and result for each were compared, and it was confirmed that Coa can improve processing speed through parallel execution without increasing the complexity of the code and having consistent results.

Keywords: Parallel, Programming Language

1. Introduction

Due to recent advances in semiconductor technology, multi-core computers are becoming mainstream. While multi-core computers are becoming mainstream, the processing speed of individual cores has not improved much in recent years. Therefore, in order to improve the operating speed of programs, it is necessary to take advantage of multi-cores by executing programs in parallel.

Fig. 1 shows the number of cores in Apple workstations, desktops, and smartphones. It shows that multicores began to be used in the mid-2000s, and that multicores have become mainstream by 2020. The black triangles in Fig. 1 show the release years of major programming languages. Comparing this with the transition in the number of cores, we can see that all major programming languages except Go were designed in the single-core era.

Thus, traditional programming languages such as C and Java were designed in an era when single cores were the norm. In order to perform parallel processing we need to write programs tailored for parallel processing. In addition, bugs caused by race conditions of variables are often hard to reproduce, and debugging can also be difficult.

A comparison between programming languages focusing on the parallel execution capability of programs is shown in Fig. 2. In Java, threads are usually used for parallel execution; this requires additional code to create and manage threads. In addition, code such as locking variables to prevent data races is also required. Therefore, code for sequential execution and code for parallel execution are generally very different. Python, JavaScript, Ruby, etc. can use threads, and in some cases, features such as asynchronous execution are available. In addition, Go [1] and Elixir [2] uses models similar to CSP [3] or actor models, which make parallel execution easier. However, as with Java, sequential execution and parallel execution require different code in Go, Elixir, and OpenMP [4]. PaSh [5] performs parallel execution without special description by performing runtime analysis. However, since PaSh is a shell script and parallel execution is done on a per-execution-process basis, it does not have sufficient functionality to describe general programs.

To overcome these drawbacks, this paper proposes a parallel execution method that automatically resolves data dependencies and produces the same results as sequential execution, even though parallel execution is used. We implement this in the Coa programming language and evaluated it using several example programs. The Coa language performs parallel execution by default; since problems such as race condition between variables can occur without modification, we detect race condition between variables using the same mechanism as for out-of-order execution on a CPU. If a conflict between variables occurs, sequential execution is automatically performed, and if not, parallel execution is automatically performed. Therefore, parallel execution can be performed without changing the program description for sequential execution and parallel execution, and the advantages of multi-core can be utilized.

This paper describes the proposed programming language Coa

[1] William Lyon Mackenzie Collegiate Institute, Toronto, Ontario, Canada
[2] Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo, Hongo, Bunkyo-ku, Tokyo, 113–0033, Japan
[a)] kenxshibata@gmail.com
[b)] shinya@is.s.u-tokyo.ac.jp

in detail. Section 2 provides an overview of the Coa language, its syntax, and an algorithm for detecting variable conflicts. Section 3 describes the implementation of the Coa interpreter using the Go language. Section 4 compares, examines, and discusses the results of executing two different tasks using the Coa language. Finally, a summary and issues are discussed.

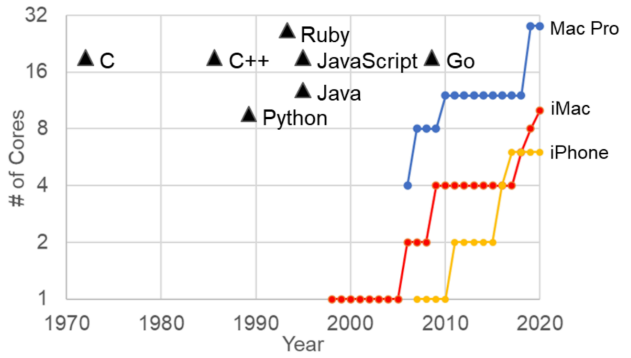

Fig. 1 Number of cores vs. year.

	Typical Language Support	Complexity of Parallel Code	Unit of Parallelization	Can Use Multiple Cores?
C	No language support (platform-dependent)	High (no dependency detection)	Per-process/ thread	Yes
Java	Thread (platform-independent)	High (no dependency detection)	Per-thread	Yes
Python	Async-await/ green threads	Moderate (no dependency detection)	Per-function/ thread	Yes
JavaScript	Async-await	Moderate (no dependency detection)	Per-function	No
Go [1]	Goroutine & channel (actor-like)	Moderate (no dependency detection)	Per-actor	Yes
Elixir [2]	Native (actor-like)	Moderate	Per-process/actor	Yes
OpenMP [3]	Native (threads)	Moderate	Per-thread	Yes
PaSh [4]	Native (actor)	Minimum (automatic dependency detection)	Per-process	Yes
Coa	Native (async-await-like)	Minimum (automatic dependency detection)	Per-function	Yes

Fig. 2 Comparison table.

2. Programming Language Coa

2.1 Coa Language Overview

We designed the Coa language and Coa interpreter to test the usefulness and scope of a language model that automatically analyses dependencies between variables and automatically determines sequential and parallel execution based on the analysis. Following are the summary of the features of the Coa:

- Has a Lisp-like syntax based on S-expressions
- Defaults to parallel execution (executes the top-level compound statement in a block in parallel)
- Has dependency analysis between variables is performed at runtime to prevent data races
- Automatically performs parallel or sequential execution so that the result is the same as if it were processed sequentially.

More details about each of these are described below.

2.2 Coa Language Syntax

Similar to Scheme and Common Lisp, the Coa language employs S-expressions as its syntax. Basic language features such as arithmetic, assignment, conditional branching, repetition, function definitions, array, and block definitions (anonymous functions) are implemented. The following is a code example in contrast to the C language.

```
# Assignment
C:   int i = 10;
Coa: (@def i 10)

# Arithmetic
C:   int a = b + c;
Coa: (@def a (@add b c ))

# Conditional Branching
C:   if (a == 0) a() else b()
Coa: (@if (@eq a 0) (a) (b))

# Looping
C:   for (int i = 0; i < 10; i++) {}
Coa: (@for (@def i 0) (@lt i 10) ())

# Function Definition
C:
  int div2(x int) {
    return x/2;
  }
Coa:
  (@def div2 {(@def x $0)
    (@div x 2)
  })

# Array Definitions and Accesses
C:
  int numbers[] = {1, 2, 3, 4, 5}
  for (int i = 0; i < 5; i ++) {
    int number = numbers[i];
    div2(number);
  }
Coa:
  (@def numbers [1, 2, 3, 4, 5])
  (@map numbers {(@def number $1)
    (div2 number)
  })

# Input / Output
C:   printf("Hello world: %d", x)
Coa: (@io_outln "Hello world: $x")
```

In addition, similarly to other languages in the Lisp family such as Scheme, there are higher-level functions such as `@map` and `@foldr`.

As an example of a more complex Coa program, the following is an example of a FizzBuzz program. For every number from 0 to 100, it outputs ”Fizz” for a multiple of 3, ”Buzz” for a multiple of 5, ”Fizz Buzz” for both multiples (multiples of 15), and the number if it is neither.

```
(@io_outln (@foldr @add
  (@for (@def i 0) (@lt i 101) (@mod i (@add i 1)) {
    (@def line (@add
      (@if (@eq (@rem i 3) 0) "Fizz" "")
      (@if (@eq (@rem i 5) 0) "Buzz" "")
    ))
    (@add (@if (@eq line "") (@string i) line) "\n")
  })
))
```

Appendix A.1 lists the built-in operators. Appendix A.2 shows some sample code in the Coa language.

2.3 Parallel Execution by Default

In Coa, the block definition operator { } is used to execute compound statements equivalent to `lambda` and `begin` in Scheme and other languages as shown in Fig. 3. Top-level expressions defined in blocks are executed in parallel by default.

```
{
  (func1)
  (func2)
}
```

In this example, `func1` and `func2` has no dependencies or side effects, so `func2` is executed without waiting for `func1` to finish.

However, if the overhead of parallel execution is large, such as when the execution unit is small, sequential execution can also be performed using the sequential execution block operator {% } as shown below.

```
{%
  (func1)
  (func2)
}
```

In this example, `func2` is executed after `func1` is executed, just as in a normal language. By adopting an execution model that defaults to this kind of parallel execution, it is possible to perform parallel execution that takes advantage of multiple cores with only minor changes to the source code. However, as one can easily imagine, if there are dependencies between variables, the execution results may differ depending on the order of execution. This problem and its solution in the Coa language will be explained in the next section.

2.4 Data Races

This section describes the automatic parallel execution feature, which is a feature of the Coa language. The Coa language performs parallel execution by default, However, simply executing all compound statements in parallel causes a problem in which the execution result could differ from that of sequential execution, depending on the order of execution. For example:

```
{
  (@def x 1)
  (@def y (@add x 1))
}
```

When the above code is executed, the value of variable y after execution is 2 in sequential execution. However, if the code is

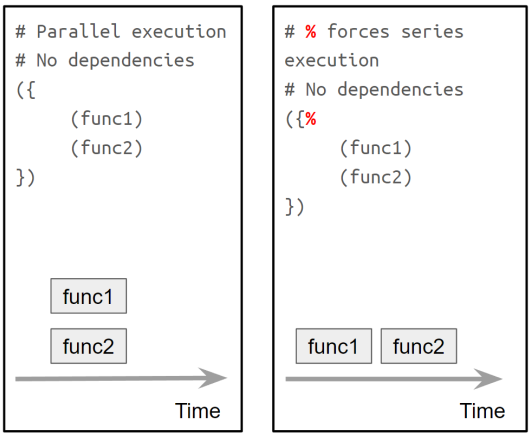

Fig. 3 Parallel execution and serial execution.

simply executed in parallel, the value of variable y after execution is 2, depending on the order of execution. or an undefined reference error to x occurs.

To prevent errors due to such data races, Coa analyzes the dependencies between data at runtime and controls the order of execution. In this example, x is updated in line 2, and the calculation dependent on x is performed in line 3. Therefore, line 3 is evaluated after line 2 has finished evaluating.

In Fig. 4, since `(func3 var1 va2)` (line 2) uses `var1` and `var2`, this would be run in the following order: `func1`, `func2`, and `func3`. `func4` would be run in parallel as it is independent.

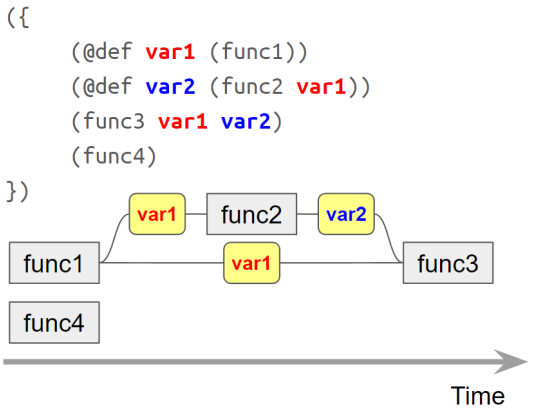

Fig. 4 Parallel execution example.

The Coa language performs dependency analysis between data at runtime. In order to prevent possible analysis omissions, restrictions have been introduced into the language. One such restriction is the elimination of pointers. Pointers allow data to be referenced implicitly, making dependency analysis difficult. For the same reason, arrays are immutable. Coa also tracks whether the functions are pure or having side effects. When an expression includes a function having side effects, the expression are executed in series.

3. Coa Interpreter

3.1 Interpreter Overview

The Coa language interpreter was implemented in the Go language. The implementation of the Coa interpreter is almost the same as that of a general AST-traversing interpreter except for

how the AST is executed as shown in Fig. 5. At runtime, the source code is converted to an AST by the lexer and parser. The interpreter executes the program following the AST. Within the interpreter, the following additional three steps: ”data dependency detection”, ”instruction assignment” and ”parallel execution” are added in Coa, and those are explained in details in the following sections.

The Go language implements a parallel execution mechanism called goroutine, and the goroutine is used for Coa's parallel execution. In implementing the interpreter, the lexer and parser used Participle [6]. The entire source code is approximately 3000 lines and is available at [*1].

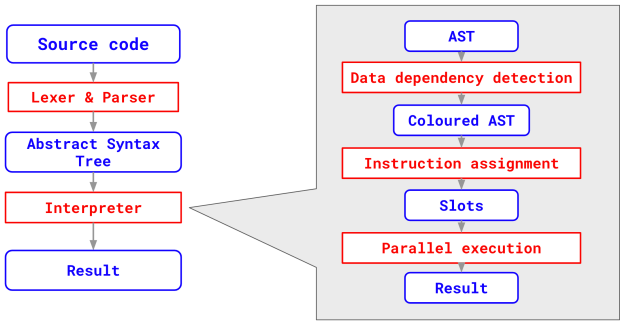

Fig. 5 Coa interpreter.

3.2 Dependency Detection

The dependency detection procedure is explained using the following Coa code as an example:

```
({
  (@def a 1)
  (@mod a (@add a 1))
  (@def b 3)
  (func1 a)
  (func2 b)
})
```

To create a dependency graph, the AST is traversed and the following is performed. First, it gets the uses of all variables that are not builtin and not from an outer scope as they always defined, and cannot be re-defined. In the example, line 3 and 5 uses `a`, and line 6 uses `b`. Second, the interpreter gets where variables are defined by finding calls to `@def` and `@mod`. In the example, line 2 defines, line 3 re-defines `a`, and line 4 defines `b`. Third, the interpreter maps variables to the latest definition. In the example, The use of `a` in line 3 is mapped to line 2, while the use in line 5 is mapped to line 3 as that is the latest definition. Similarly, the use of `b` in line 6 is mapped to line 4. The result is shown in Fig. 6. By following the above procedure, the dependency graph is created, and the generated graph is provided to the next instruction assignment step.

3.3 Instruction Assignment and Parallel Execution

Within the interpreter, there are the following goroutines. The main goroutine first performs data dependency analysis. Next, the

[*1] https://gitlab.com/coalang/go-coa/-/archive/bland/go-coa-bland.tar.gz

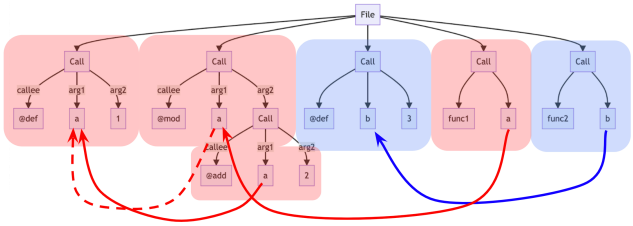

Fig. 6 Dependency graph generation and AST coloring.

AST is separated according to the results of the data dependency analysis and placed in multiple execution ”slots”. Each slot has a list of AST nodes to evaluate, and a list of other slots on which it depends on.

The example in section 3.2 will be used to explain the procedure. The resulting slots are shown in Fig. 7. Since line 3 uses `a` (defined in line 2), it will be evaluated after line 2. In addition, line 5 uses `a` but it will be evaluated after line 3, as line 2 re-defines `a`. Lines 2, 3, and 5 can be executed sequentially, and form one slot. Similarly, and since line 6 uses `b` (defined in line 4), it will be evaluated after line 4. Therefore, lines 4 and 6 form another slot. However, since lines 2, 3, and 5 and lines 4 and 6 are completely independent, they can be executed in parallel. Therefore, each slot has no dependencies.

The main goroutine creates a sub-goroutine for each slot which does not have any dependencies and initiates parallel execution. Each sub-goroutine may also create other sub-goroutines to execute slots that depend on itself. For example, if slot C depends on both slot A and B, when slot A and B finish evaluation, one of the sub-goroutines evaluating the slot may create a new sub-goroutine to run slot C. If the sub-goroutine does not have any other sub-goroutines to create, it will run slot C without creating a new sub-goroutine.

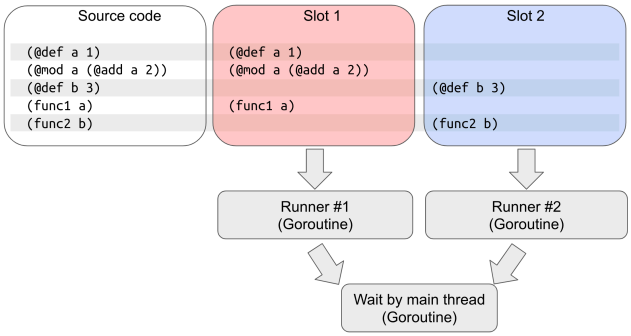

Fig. 7 Instruction assignment and parallel execution by goroutine.

4. Experiment and Results

The Coa language allows parallel and sequential execution with nearly identical code without causing data races. Furthermore, it is proposed that parallel execution can improve speed. To confirm these two points, two sample programs were developed in Coa, Python, and Go, and the results of the parallel and sequential programs were compared for consistency. We evaluated the consistency of the program results, the change in the number of program lines, and the change in execution time between parallel and sequential programs.

4.1 File Downloads

We evaluated a program that performs multiple file downloads. The downloading of multiple files can be done independently, making it suitable for parallel processing. In addition, since file downloads are limited by the bandwidth of the network, the We used this to compare the parallelism of interpreted and compiled languages in the same way. Programs with similar behavior were implemented in Coa, Go, and Python, and were run in parallel. We compared whether the same results could be obtained by sequential execution and parallel execution, the change in the number of lines of code, and the change in the execution time. As an experiment, we downloaded 10 to 15 files of several 10 MB to several GB in size at the same time, mainly from Wikipedia, and measured the time until completion. The execution environment used was an Intel Core i5 CPU with 4 cores and 8 threads running Linux.

We implemented the code for sequential and parallel execution using Coa, Go, and Python, respectively. First, we executed the programs and confirmed that the same files could be downloaded in sequential and parallel execution, and that there was no difference in the operation results. Fig. 8 shows a comparison of the number of lines of sequential and parallel execution code in each implementation language of the code. In Coa, sequential and parallel codes can be implemented with the same number of lines of code, while in Go and Python, parallel codes can be implemented with the same number of lines of code. Go and Python, the number of lines of parallel execution code increases by a factor of 1.2 to 1.3. This figure shows that Coa can implement sequential and parallel execution with the same number of lines of code, and that the goal of parallel execution without increasing the complexity of the code has been confirmed.

It can also be confirmed that the program is more concise in Coa than in Go and Python. Go has ”import” statements, which increase the number of lines. Python uses indentation to write block statements, which is thought to be less dense than Coa's S-expression statements.

Fig. 9 shows a comparison of execution times. It was confirmed that parallel execution reduced execution time by a factor of 0.78 to 0.53 compared to sequential execution for all languages. The results show that parallel execution in any language reduces execution time from 0.78 times to 0.53 times faster than sequential execution. Using the Coa language in this example, we can see that At least for tasks with large granularity, such as file downloads. parallel processing without increasing the complexity of the code. execution time can be reduced without increasing the complexity of the code.

4.2 Mandelbrot Set

The second experiment was an evaluation of a program that computes a Mandelbrot set for a given range in the complex plane. To find the Mandelbrot set, the following formula is calculated for a point c in a complex plane, and the number of iterations until divergence until convergence is obtained.

$$z_{n+1} = z_n + c \tag{1}$$

$$z_0 = 0 \tag{2}$$

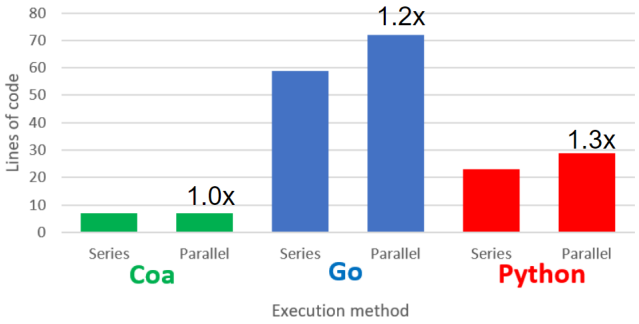

Fig. 8 File downloads code lines.

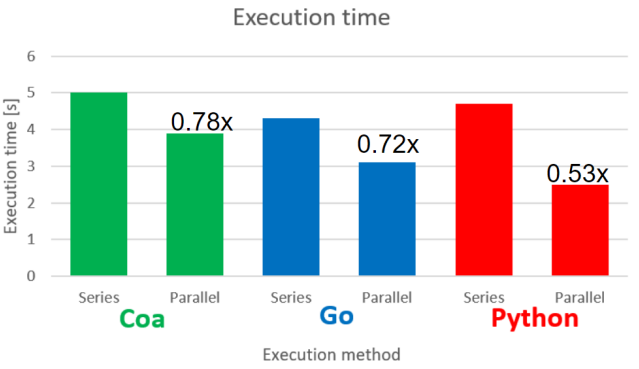

Fig. 9 File downloads execute time.

Since the computation of each point on the complex plane is independent, the computation can proceed in parallel. This was used as a benchmark for fine-grained parallel computation.

We were able to confirm that the calculation results are the same for sequential and parallel processing, even though the parallel execution is performed by default. Fig. 10 compares the number of lines of code between the Coa and Go implementations. While Go increases the number of lines by approximately 10%, Coa increases the number of lines by approximately the same amount. Coa has the same number of rows, confirming that parallel execution is achieved without increasing complexity.

Fig. 11 shows a comparison of execution times for the Coa, Go, and Python implementations. As in the file download experiment, a 4-core, 8-thread Intel Core i5 was used as the execution environment. The parallel execution in Coa reduced the execution time by a factor of 0.94 compared to sequential execution, and in Go, 0.37 times faster than sequential execution. Although parallel execution reduced execution time, 0.94 × was not sufficient considering the 4-core execution environment. A possible reason for this is that Coa is an interpreter, and dependency analysis is performed only on the main goroutine. Therefore, the dependency analysis becomes a bottleneck in such fine-grained calculations. The current implementation does not allow the calculation to proceed with sufficient parallelism.

5. Conclusions

We proposed a method that automatically resolves data dependencies to obtain the same results as sequential execution, even if parallel execution is used as the default execution method. We implemented such a method as a programming language Coa and evaluated it on several example programs. The evaluation results

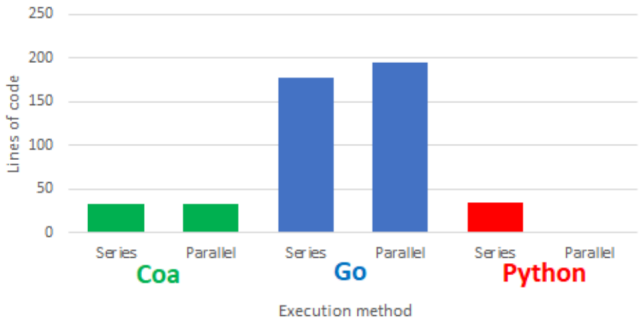

Fig. 10 Mandelbrot set code lines.

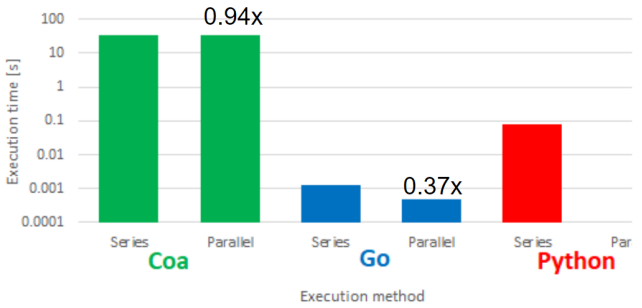

Fig. 11 Mandelbrot set execute time.

showed that the parallel execution was almost identical to the sequential execution (starting a block with {% instead of {), and the same results could be obtained. We also showed that parallel execution can speed up the execution time. However, there are some issues to be addressed in terms of speedup through parallel execution. The interpreter itself is slow, and the speedup was not proportional to the number of cores due to the following reasons.

In the future, we would like to speed up the interpreter as a whole by, for example, detecting dependencies before execution, as explained earlier, and compiling the program into bytecode [7] [8]. Next, we would like to increase what can be done with Coa by making Go's libraries available for Coa as well.

Acknowledgments This work was supported by the JST Global Science Campus Experts in Information Science program hosted by the National Institute in Informatics, Information Processing Society of Japan, and the Japanese Committee for the International Olympiad in Informatics. We appreciate the mentors and students in the program for their useful feedback. Financial support was provided by the Masason Foundation.

References

[1] The Go Authors, "The Go Programming Language," Retrieved September 19, 2021. [Online]. Available: https://golang.org
[2] The Elixir Team, "Getting started: Processes," Retrieved July 24, 2022. [Online]. Available: https://elixir-lang.org/getting-started/processes.html
[3] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, p. 666–677, aug 1978. [Online]. Available: https://doi.org/10.1145/359576.359585
[4] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," 1998.
[5] S. Handa, K. Kallas, N. Vasilakis, and M. C. Rinard, "An order-aware dataflow model for parallel unix pipelines," *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, aug 2021. [Online]. Available: https://doi.org/10.1145/3473570
[6] A. Thomas, "alecthomas/participle: A parser library for go." [Online]. Available: https://github.com/alecthomas/participle
[7] T. Ball, "Writing a Compiler in Go."
[8] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.
[9] University of Waterloo, "CCC 2012 Problems, Tests and Solutions." [Online]. Available: https://www.cemc.uwaterloo.ca/contests/computing/2012/index.html

Appendix

A.1 Coa system commands and expressions

The following is a list of Coa language constants, built-in functions, etc.

```
# constants
@true # boolean true
@false # boolean false

# time
(@time_now) # get current time
(@time_sleep time) # return after waiting time
seconds

# outside system
@sys_os # name of os (e.g. windows, linux)
@sys_arch # name of CPU architecture (e.g. x86,
amd64)
@sys_args # list of arguments
(@sys_exit exit_code) # exit with exit_code
@sys_env # map of environment variables

# testing
(@assert assertion name) # assert that assertion
is @true. (if not, raises an error)

# filtering
(@filter list filter) # filter list using filter
(@glob pattern) # make a glob filter with pattern
(@regex pattern) # make a regex filter with
pattern

# evaluation control
(@error content) # raise an error with content
(@continue) # skip the current loop
(@break) # stop the loop
(@return returned) # exit current block with
return value returned

# scope
(@use name) # mark an unused variable name as
used
(@def name content) # define a variable name with
content
(@mod name content) # modify the innerest scope
variable with name name to content

# loops
(@for init cond iter callable)
# run init once and then callable and iter until
```

```
cond is @false
(@while cond callable) # run callable until cond
is @false

# control
(@if [cond value]... [else])
# if cond is @true, evaluate and return the value
next to
# If all conds are @false, evaluate and return
else.

# mapping
(@map list callable) # run callable with each
key/index and value of list
(@mapnokey list callable) # run callable with
each value of list

# lists
(@split list splitter) # split list with splitter
(@has_prefix list prefix) # return @true if list
has prefix prefix
(@trim_prefix list prefix) # remove prefix prefix
if list has prefix prefix
(@has_suffix list suffix) # return @true if list
has suffix suffix
(@trim_suffix list suffix) # remove suffix suffix
if list has suffix suffix
(@len list) # return length of list

# folding
(@foldl callable list) # fold (left) list using
callable
(@foldr callable list) # fold (right) list using
callable

# comparisons
(@lt a b) # a < b
(@le a b) # a <= b
(@gt a b) # a > b
(@ge a b) # a >= b
(@eq a b) # returns whether contents of (@inspect
a) and (@inspect b) are equal
(@or a b) # a ∨ b
(@and a b) # a ∧ b
(@not a) # ¬ a

# arithmetic
(@concat a b) # concatenate string a with b
(@add a b) # a + b
(@sub a b) # a - b
(@mul a b) # a × b
(@div a b) # a ÷ b
(@rem a b) # a mod b

(@http_get url) # return body of HTTP GET request
sent with URL url

(@file_write path content) # write content to
file path
(@file_read path) # return content of file path
(@file_remove path) # remove file path
(@file_list path) # return list of files in
directory path

(@io_out content) # print content to stdout
(@io_outln content) # print content and ASCII
code 10 (decimal) to stdout
(@io_in delim) # return content read until delim
from stdin (returned doesn't contain delim)

(@complex real imag) # make a new complex number

(@complex_to input) # convert input to a complex
number
(@int input) # convert input to an integer
(@uint input) # convert input to an unsigned
integer
(@float input) # convert input to a
floating-point number
(@string input) # convert input to a string
(@inspect input) # convert input to a string
representation

(@json_to input) # convert input to JSON format
(@json_from input) # convert input to native Coa
data

(@get_try map key fallback) # try to get value of
key key from map map. If it doesn't exist, return
fallback
(@get map key) # return value of key key from map
map
(@set map key value) # set key key to value value
in map map
(@keys map) # get unordered keys of map

(@select list index) # return index-th value of
list
(@take_from list index) # return list from index
(@take_to list index) # return list to index
(@take list start end) # return list from start
to end

# utils
(@label label content) # returns content; use to
label nodes
```

A.2 Sample codes

Some sample programs are shown to illustrate the use of the Coa language.

A.2.1 FizzBuzz

Example code for a FizzBuzz question.

```
# FizzBuzz
(@io_outln (@foldr @add (@for (@def i 0) (@lt i
101) (@mod i (@add i 1)) {
    (@def line (@add
    (@if (@eq (@rem i 3) 0) "Fizz" "")
    (@if (@eq (@rem i 5) 0) "Buzz" "")))
    (@add (@if (@eq line "") (@string i) line)
"\n")
})))
```

A.2.2 Internet Access

Example of accessing a Mediawiki instance to perform a search.

```
# MediaWiki Search
(@def search {
    (@def name $0)
    (@def base_url $1)
    (@def url "https://$base_url/w/api.php?action=
query&list=search&srsearch=$name&format=json")
    (@def data (@get (@get (@from_json (@http_get
url)) "query") "search"))
    (@map data {
        (@def key $0)
        (@def value $1)
        [(@get value "title") (@get value
"pageid")]
    })
})

(@def search_and_save {
    (@def term $0)
    (@def base_url $1)
    (@file_write "search_results_$term.md" (@add
"# Search Results for $term ($base_url)\n\n"
(@foldr @add (@map (search term base_url) {
        (@def num (@add $0 1))
        (@def title (@select $1 0))
        (@def pageid (@select $1 1))
        "$num. [$title](https://$base_url/?curid=
        $pageid)\n"
    }))))
})

({% # percent means "allow running this
concurrently"
    (search_and_save "concurrency"
"en.wikipedia.org")
    (search_and_save "good" "en.wiktionary.org")
})
```

A.2.3 Canadian Computer Contest 2012-s2

Example of Canadian Computer Contest 2012 S2 problem solved in Coa. The problem is explained in [9].

```
# compute a number format where arabic and roman
numbers are used together:
# 3M1D2C = 3 * 1000 + 1 * 500 + 2 * 100 = 3700
# ccc12s2
(@def inp (@io_in '\n'))
#(@def inp "2I3I2X9V1X")
(@def roman_numerals [m 'I' 1 'V' 5 'X' 10 'L' 50
'C' 100 'D' 500 'M' 1000])
(@def prev_r 0)
(@def prev_a 0)
(@def total 0)

(@for (@def i 0) (@lt i (@len inp)) (@mod i (@add
i 2)) {
    (@def now_a (@int (@select inp i)))
    (@def now_r (@get roman_numerals
        (@select inp (@add i 1))))
    (@def value (@mul now_a now_r))
    (@if (@gt now_r prev_r) ({
        (@mod total (@sub total
            (@foldr @mul [2 prev_a prev_r])))
    }))
    (@mod prev_a now_a)
    (@mod prev_r now_r)
    (@mod total (@add total value))
})
(@io_outln (@string total))
```

A.2.4 Canadian Computer Contest 2021-s1

Example of Canadian Computer Contest 2022 S1 problem solved in Coa. The problem is explained in [9].

```
# ccc21s1
(@io_in '\n')
(@def heights (@mapnokey (@split (@io_in '\n') '
') @int))
(@def widths (@mapnokey (@split (@io_in '\n') '
') @int))
(@def area 0)
(@map widths {
    (@def i $0)
    (@def width $1)
    (@mod area (@add area (@mul width (@add
(@select heights i) (@select heights (@add i
1))))))
})
(@mod area (@div area 2))
(@io_outln (@string area))
```

A.2.5 Mandelbrot Set

```
(@def get_pixel {
    (@def x $0)
    (@def y $1)
    (@def c1 (@complex x y))
    (@def c2 (@complex 0 0))
    (@map (@range 1 900) {
```

```
        (@def n $1)
        (@if (@gt (@abs c2) 2) (@return_len n 2)
(@mod c2 (@add (@pow c2 2) c1)))
    })
    (@return 0)
})

(@def width (@int (@select @sys_args 2)))
(@def height (@int (@select @sys_args 3)))
(@io_outln "abc $width")
(@io_outln "def")
(@io_outln "ghi")

(@map (@range width) {%
    (@def x $1)
    (@map (@range height) {
        (@def y $1)
        (@def value (get_pixel
            (@div (@sub x (@mul 0.75 width))
(@mul 0.25 width))
            (@div (@sub y (@add (@mul 0.25
height) (@mul 0.25 width))) (@mul 0.25 width))
        ))
        (@io_outln "$x $y $value")
    })
})
```