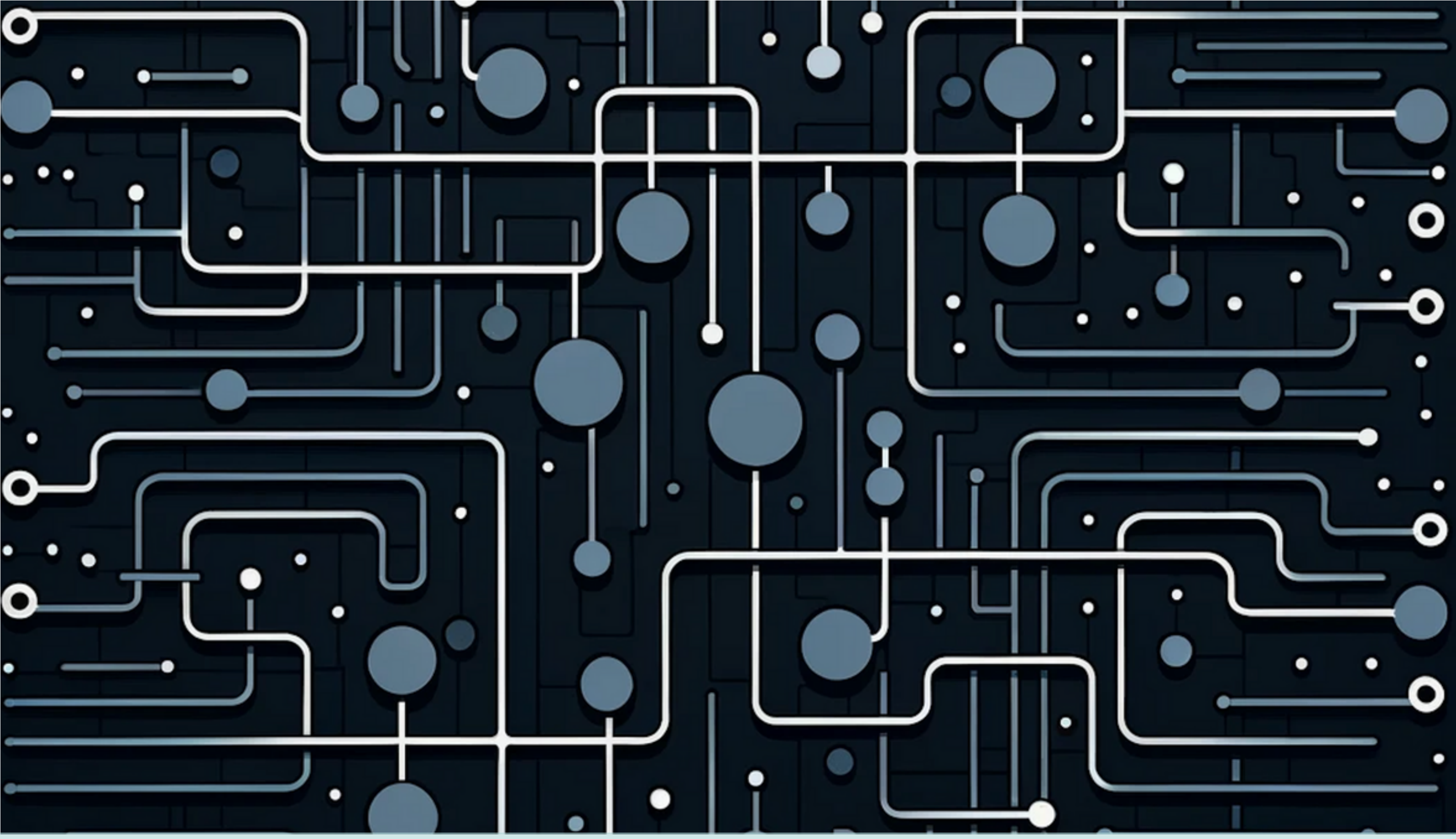
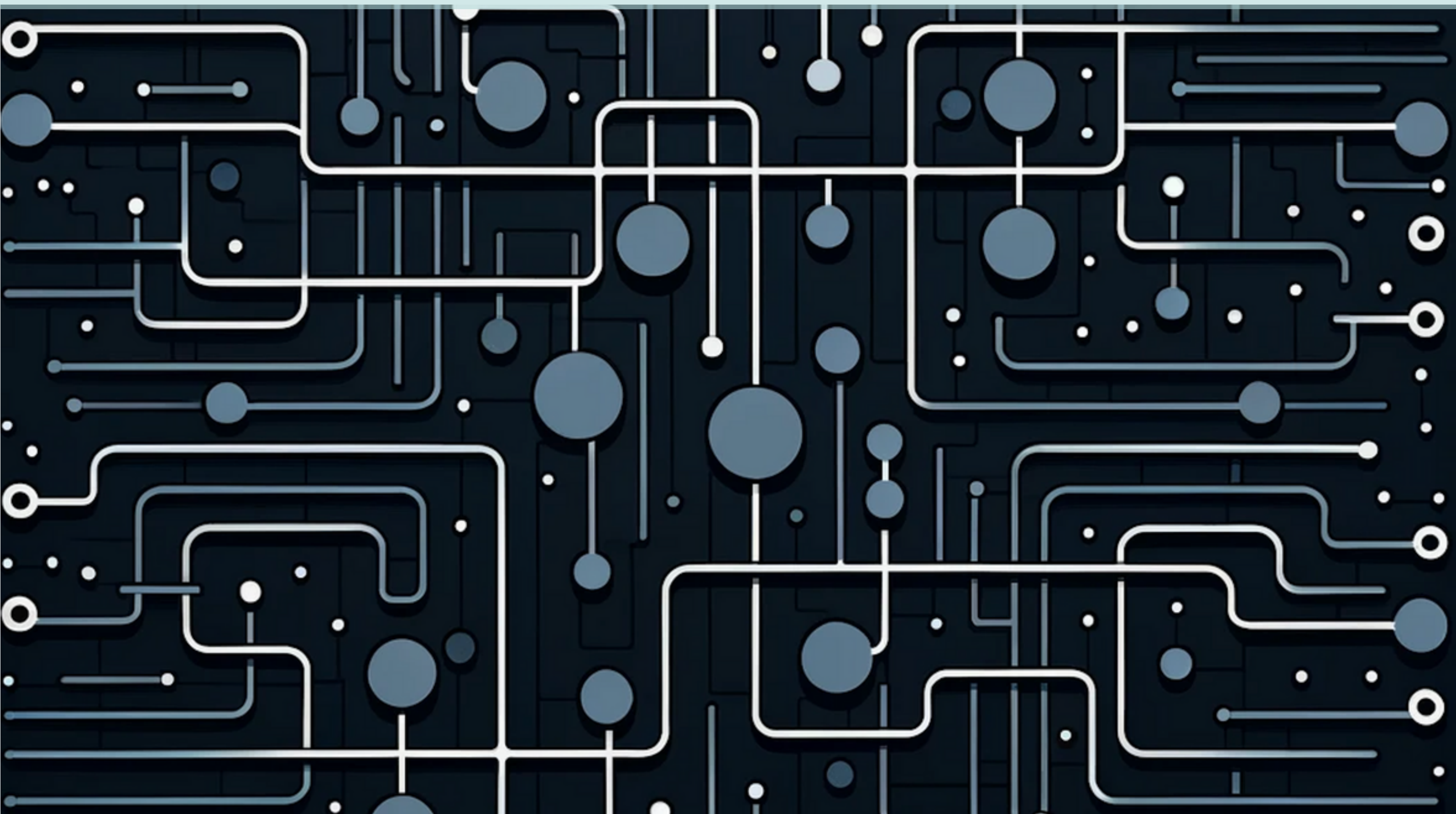


ConcurX

A Parallel Computing Language

N. Sai Vamsi   Perala Anudeep Rao   Varshini Jonnala   J. Tushita Sharva



Contents

Introduction

1. About the Project

1.1 Motivation

The motivation behind the development of this language is to address the challenges of parallel and concurrent programming. Writing efficient and correct parallel code can be complex and error-prone. Therefore, the language's motivation is to simplify this process and provide developers with the necessary tools to maximize the power of multi-threading and concurrency.

1.2 Goal

We aimed to design a language simplifies parallel and concurrent software programming. The programmers will have flexibility to work with both lock-based and lock-free data structures.

Additionally, our language places a strong emphasis on **performance analysis**. The integrated tools and constructs of this language enable developers to measure and analyze thread performance, execution times, and program efficiency. This helps one to identify bottlenecks and areas for improvement, ensuring that their parallel code runs efficiently.

Furthermore, the language implements **comprehensive logging capabilities**. Efficient logging mechanisms have been implemented using lock-free queues. This feature helps developers track and analyze program behavior, making it easier to diagnose issues and monitor the execution of parallel code.

The language also encourages the development of parallel algorithms and the creation of parallel versions of common algorithms.

II

About the Language

2. Lexical Conventions

2.1 Comments

Comments are non-executable annotations within the source code that help programmers understand and document their code. Both single line and multi-line comments are supported in this language.

```
$$  I am a single line comment

$*
    I am a Multi line comment
*$
```

2.2 Reserved Keywords

These are predefined words with specific meanings, and they cannot be used as identifiers (variable or function names) in the program, as they have a role in the programming language's syntax.

`number`	`main`	`ip`	`get`	`when`	`invoke`	`decimal`	`task`	`void`
`letter`	`default`	`op`	`for`	`else when`	`type`	`repeat`	`atomic`	`array`
`text`	`analyze`	`func`	`bool`	`make parallel`	`return`	`break`	`continue`	`time`

2.3 Punctuation

These are characters used to structure and delimit code, aiding in the syntactic analysis of programming languages.

```
, $$ comma
: $$ colon
; $$ semicolon
```

2.4 Identifiers

We have adopted a naming convention for identifiers in our project that closely follows the established naming convention used in the C++ programming language.

2.5 Operators

These symbols perform specific operations on operands, such as addition or comparison, and are used to manipulate data during program execution.

Arithmetic	+, -, /, *, ^, %
Access	$\rightarrow$
Assignment	=, +=, -=, /=, *=, ^=, %=
Logical	&&, ||
Relational	<=, >=, >, <, ==, !=, $\sim$

2.5.1 Precedence of Operators

Operator	**Description**	**Associative**
->	Access Operator	Left to right
$\sim$	Negation	Left to Right
^	Exponent	Left to Right
%	Modulo	Left to Right
*	Multiplication	Left to Right
+	Addition	Left to Right
-	Subtraction	Left to Right
==, !=	Relational Operators	Left to Right
>=, >, <=, <	Relational operators	Left to Right
&&, ||	Logical operators	Left to Right
=, *=, +=	Assignment Operators	Left to Right

2.6 Special Symbols

```
For beginning and ending of scopes:  << >>
For predicates:  [ ]
```

2.7 Constants

2.7.1 Real constants

Real constants are numeric values, either integers or floating-point numbers, that represent continuous quantities with or without decimal points.

```
   1. INTEGER CONSTANTS
      Examples:  3, -5
   2. DECIMAL CONSTANTS
      Examples:  3.14, -12.7
```

2.7.2 String literals

String literals are sequences of characters enclosed in double quotation marks (" ") and they are used to represent textual data.

```
A string literal can be defined as "<text>" where <text> can include
letters, numbers, symbols, and escape sequences.
Examples:  "", "Compilers2", "123", "\n\t"
```

2.7.3 Character literals

Character literals are individual characters enclosed in single quotation marks (‘ ’).

```
A character literal can be typically expressed as ‘<char>’ where <char> is
a single character that can be letter, digit, symbol or escape sequence.
Examples:  ‘’, ‘a’, ‘3’, ‘\n’
```

2.7.4 Boolean literals

Boolean literals are used for logical conditions and are not typically enclosed in quotes or symbols.

```
They are binary values that represent two logical states:
   • true
   • false
```

2.8 Datatypes

2.8.1 Built-in Datatypes

Primitive Datatypes	Atomic Datatypes
number	atomic number
letter	atomic letter
text	atomic text
decimal	atomic decimal
boolean	atomic boolean
void	atomic void
array	atomic array

2.8.2 User-defined Datatypes

There are two types of declarations a user can perform:

A. Regular Declaration: - `type`

- The TYPE declaration defines a user-defined data type, indicating the creation of a custom data structure or class.
- Within TYPE, variable declarations should be written with no access specifiers.
- Any function declarations or implementations must be contained within the scope of this TYPE, encapsulating data and behavior within a single entity.

B. Atomic Declaration

We are not actively working on this feature at the moment.

3. Syntax

3.1 Code Space

The entire program is structured using `type, functions`, and `task` declarations. Each of these components serves a specific purpose in organizing and encapsulating code for various functionalities within the program. The main function serves as the entry point for the program. It arranges the execution by creating objects from classes, invoking functions to perform tasks, and coordinating the flow of the program.

3.2 Types

This a concept that is similar to classes in C++. Types define the kind of data a variable can hold, specifying the set of values and operations that are valid for that data. In short, they define structure and behaviors of the data in a more general sense.

3.2.1 Scope of Type Declaration

The scope of the type declarations can have

- Declarative Statements
- Function Statements

Each of them have been discussed in detail in upcoming sections. For now, one should note that in this language, one must write the function body also inside the class itself. In other words, to specify the behavior of a particular type, programmers must define functions within the type declaration itself.

3.2.2 Types of Classes

There can be two types of declaration of classes. They are:

- **General declaration:** This type adheres to the conventional class declaration methodology, offering standard features and behaviors.
- **Atomic declaration** Whenever an object of this instance is created, it is by default has atomic properties. This means that any given property of the object can be manipulated only by one

thread at a time. Multiple threads can modify multiple the same object, given that they are modifying different properties.

SYNTAX

```
type type_name
<<
$$  Declarative statements

$*
    Function statements
*$
>>

atomic type type_name
<<
$$  Declarative statements

$*
    Function statements
*$
>>
```

```
Eg:

  1. type teacher
     <<
         text class_name;

         func attendance :  text class_name :  number
         <<
           $* Statements *$
         >>
     >>

  2. atomic type student
     <<
         number roll_number;

         func attendance :  number roll_number :  number
         <<
           $* Statements *$
         >>
     >>
```

3.3 Functions

Functions serve as modular, reusable blocks of code, similar in concept to C++ functions. They encapsulate specific functionality, and allow you to break down complex tasks into smaller, manageable units.

3.3.1 Scope of a Function

The scope of a function can contain all the types of statements except class definitions, task definitions and function definitions.

3.3.2 Types of Functions

We have introduced two distinct types of function declarations to cater to specific use cases:

Atomic Functions:

- **Purpose:** Atomic functions are specifically designed for operations involving atomic data types. They are essential in scenarios where concurrent operations on shared data need careful coordination to avoid data races.
- **Use Cases:** Working with shared counters, flags, or memory allocation operations to prevent interference from other threads.

General Functions:

- **Purpose:** Functions are standard building blocks of your code. They offer modularity and facilitate the organization of your program.
- **Use Cases:** Same as general purpose functions.

SYNTAX

```
func func_name :  datatype id1, ....  :  return_type
<<
    $* Statements *$
>>

atomic func func_name :  datatype id1, ....  :  return_type
<<
    $* Statements *$
>>
```

```
Eg:
  1. func attendance :  text class_name :  bool
     <<
         $* Statements *$
     >>
     Eg:  2.

  2. func attendance :  number roll_number :  bool
     <<
        $* Statements *$
     >>
```

3.4 Tasks

Task is construct designed to help programmers analyze the performance of multi-threaded applications. They serve as a way to execute a block of code in parallel, allowing developers to gather performance metrics and optimize the execution of that code. Unlike traditional functions, tasks are primarily focused on performance analysis and are used to optimize parallel algorithms.

3.5 Difference between Function and Task

Function:

- A function is a fundamental programming construct that defines a block of code that can be executed by calling it. They are used to encapsulate a piece of functionality, take input, perform operations, and return results.

- Functions can be used to create parallel algorithms and data structures by dividing the work into smaller tasks or subroutines that can be executed concurrently.
- Functions can encompass a wide range of tasks, from performing computations to managing data structures, and can be used for various purposes in a parallel program.

Task:

- A task is mainly used for analyzing the algorithm's execution time when threading is implemented.
- Tasks are more focused on measuring and profiling the behavior of parallel code i.e., they are typically used for performance analysis and profiling purposes.
- Tasks are often used to measure and understand how different parts of an algorithm run in parallel, helping to optimize performance.

3.6 Statements

Syntactical units representing a specific action or operation in a programming language, which are terminated by a semicolon

3.6.1 Declaration statement

A statement in a programming language that introduces a new variable type to the compiler. Declarations give the information about the name and data type of the identifier, allowing the compiler to allocate memory and validate its usage in the program.
SYNTAX

```
datatype identifier;
atomic datatype identifier;
user_defined_datatype identifier;
```

```
Eg:
    • number a;
    • atomic text tf;
    • teacher Anudeeperala;
```

3.6.2 Assignment statement

Used to assign values to variables or attributes of data structure.
SYNTAX

```
identifier assignment_operator RHS;
datatype identifier assignment_operator RHS;
```

```
Eg:

    • a = 10;
    • text s = "VamsiNS";
    • k = invoke f:;
```

RHS

RHS of an expression can be an identifier, a constant, a call or a predicate.

```
Eg:  10;  a > 5;  f();  a+2−b;

    • 10 is a constant
    • a > 5 is a predicate
    • invoke tushita :  varshini; is a function call
    • a+2−b is an expression
```

3.6.3 Loop statement

Used to execute a block of code repeatedly. The are two types of loops supported in the language: **for**, **while** loops. For loop executes a specific number of times. While loop executes a code as long as the given condition is true. As of now, step is an assignment statements, unary operators are not supported.
SYNTAX

```
$$ WHILE LOOP:
repeat[predicate]
<<
    $* Statements *$
>>

$$ FOR LOOP:
for[ assignment | predicate | step ]
<<
    $* Statements *$
>>
```

```
Eg:
  1. repeat[a < 10]
     <<
         a+ = 1;
     >>
  2. for[i = 15 | i > 5 | i/ = 3] <<
         a+ = 1;
     >>
```

3.6.4 Predicate

A condition or an expression that evaluates to true or false, which will be used in control flow statements (if-else) or loops to make decisions during program execution. They can be an identifier, constant, an expression, a call, and predicates nested using && or || logical operators.

```
Eg:   a == 10;  b > 10;
```

3.6.5 Conditional statement

Control the flow of a program based on specified conditions. The language supports the constructs: *when*, *else when* and *default*.
SYNTAX

```
when[predicate]
<<
     $* Statements *$
>>
else when[predicate]
<<
     $* Statements *$
>>

default
<<
     $* Statements *$
>>
```

```
Eg:
when[b > 15]
<<
     $* Statements *$
>>
else when[b < 5]
<<
     $* Statements *$
>>
default
<<
     $* Statements *$
>>
```

3.6.6 Call statement

Function call

Function call allows you to invoke or execute a function or method in the following way:

```
invoke func_name :  function_arguments;    $*separated by comma*$
```

```
Eg:
invoke attendance_repo:  class,student_number, bool
```

Task call

Task call typically refers to the execution or invocation of a task which can be executed concurrently on multiple processors or threads inorder to achieve parallelism.

```
make parallel task_name :  number_of_threads_to_be_used :  runs;
```

```
Eg:
make parallel find :  20 :  5
```

Here, the term *runs* represents the number of times the program is set to run.

3.6.7 Print statement

The below statement prints the string to be printed to the output terminal.
SYNTAX

```
op:  ‘‘string for printing to standard output’’; $* default to standard
output *$
```

The below statement prints the string to be printed to the file name.
SYNTAX

```
op:  "string to be printed" -> file_name;
```

```
Eg:
  1. op :  "hello, world!" -> "output.txt";
  2. op :  "hello!";
```

3.6.8 Scan statement

The scan statement is used for input operations, reading data from either the standard input or a specified file. SYNTAX

```
ip:  var1,var2,...........; $*Default input from standard input*$
ip → file_name:  var1,var2,.....; $*Default input from input file*$
```

```
Eg:
  1. ip:  a,b,c;
  2. ip −> "input.txt" :  A, B, C;
```

3.6.9 Return statement

The *result* statement is used to specify the value to be returned by a function or method in this programming language.
SYNTAX

```
return VALUE;
```

```
Eg:
return "Harishhh";
```

Here, VALUE includes *function calls, predicates, identifiers, constants*.

3.7 Task Parallelization

Parallelization is done on the tasks
Tasks invokes the functions, that we are interested in making parallel
To invoke a function, we use the following syntax

```
invoke func_name :  function_arguments   $*separated by comma*$
```

Each thread runs the task certain number of times, and we log the data which would be used or is required for the retrieval of data

```
make parallel task_name : number_of_threads_to_be_used : runs
```

3.7.1 Get

It is a special data structure which stores float values. When parallelizing a task using the "make parallel" command with a specified number of runs, the task is executed repeatedly, and it returns the average time taken for the threads to complete their execution. When we invoke "get→time," it retrieves and stores that time as a floating-point value in the get data structure. This get is then sent to analysis for plotting.
SYNTAX

```
get->time;
```

3.7.2 Analysis

We can analyze the impact of varying thread numbers on task performance by establishing an evaluation metric for comparison by plotting graphs
The syntax for the same is:

analyze y_{label} : x_{label} : x_{coord} : $data_1$: $data_2$: .....

- y_{label} is the label name on the Y-axis
- x_{label} is the label name on the X-axis
- x_{coord} is an array of x-coordinates
- $data_1$, $data_2$,.... are inside an array $data_i$, which is updated with data stored in **get** call.