Parallel

Introduction:
Parallel is a domain-specific language designed for straightforward and efficient parallel programming. **Parallel** combines features from Go and Rust to simplify syntax and thread management. **Parallel** introduces the concept of Tasks that functionally behave as Threads, but provide much better control of memory and locking. It also draws motivation from OpenMP style parallelism to write short code for quicker tasks.

Appendix:

General Syntax Constructs

Comments:
Single-line comments:
- Syntax: Start with two forward slashes `//`.
- Usage: Used to add brief notes or explanations on a single line.

```
int x = 10; // This is a single-line comment
```

Multi-line Comments (Block Comments)

- Syntax: Start with `/*` and end with `*/`.
- Usage: Used for longer explanations or to comment out blocks of code over multiple lines.

```
/*
This is a multi-line comment.
It can span multiple lines.
*/
```

Identifiers:

Identifiers are distinct names assigned to variables, structs, functions, and other entities within a program. They are used to uniquely distinguish each entity. In our DSL, each identifier must be unique; duplicates are not allowed in the same scope.

Rules for naming an identifier:

- **Start with a Letter or Underscore:** Identifiers must begin with a letter (a-z, A-Z) or an underscore (_). They cannot start with a digit.
- **Followed by Letters, Digits, or Underscores:** After the initial character, identifiers can include letters, digits (0-9), and underscores.
- **Case Sensitivity:** Identifiers are case-sensitive, meaning `Variable`, `variable`, and `VARIABLE` are considered distinct.
- **No Reserved Keywords:** Identifiers cannot be the same as reserved keywords or built-in names defined by the language.
- **No Special Characters:** Identifiers cannot include special characters such as @, $, %, etc.

Identifier Scope:

The scope of a variable refers to the region of the program where the variable is accessible.
Types of scope:

Local scope:

- **Definition**: Variables declared within a block (enclosed by curly braces `{}`) have local scope.
- **Access**: They are only accessible within that block and its nested blocks.

```
{
      int a = 2;
}
// accessing identifier 'a' over her will throw an error
```

```
int a = 10;
{
      int a = 5;  //local scope
}
// accessing 'a' over here will return 10
```

```
{
      // accessing 'a' over here will throw an error
      int a = 10;
}
```

Global Scope:

The global scope refers to the region outside any block or function. Global variables are visible in every part of the program.

Basic Data Types:

Type	Storage Size	Value range	Format specifier
char	1 byte	0 to 127 (ASCII format)	%c
int	4 byte	-2^{31} to $2^{31}-1$	%d
long	8 bytes	-2^{63} to $2^{63}-1$	%l
bool	1 byte	0 (false) or 1 (true)	%d
float	4 bytes	1.2×10^{-38} to	%f

		3.4×10^{38}	

char

- **Initialization**:

```
char c = 'a';
char c = 65;      //both statements initialise c as character 'a'
```

- **Explanation**: Represents a character using the ASCII standard. It is typically used for single characters, such as letters or digits.

int

- **Initialization**: `int x = 10;`
- **Explanation**: Represents a 32-bit signed integer, suitable for storing integers without decimal points.

long

- **Initialization**: `long num = 100000000;`
- **Explanation**: Represents a 64-bit signed integer, useful for storing large integers without decimal points.

bool

- **Initialization**:

```
 bool flag = true;
(or)
bool flag = 1;
```

- **Explanation**: Represents a Boolean value, which can be either `true` or `false`. Commonly used in logical expressions and conditional statements.

float

- **Initialization**: `float pi = 3.14;`
- **Explanation**: Represents a 32-bit floating-point number, used for storing real numbers with fractional parts. It provides a balance between range and precision for decimal values.

Type can be automatically inferred if constants are used or on function return types.
Example:

```
X = 5;
pi=3.14;
```

Note:
If a variable is neither initialized nor the data type is specified then an error will be thrown.
Example:

```
Y; // error
```

Operators:

`+`: Adds two operands.
`-`: Subtracts the second operand from the first.
`*`: Multiplies two operands.
`/`: Divides the first operand by the second.
`%`: Computes the remainder of division.
`==`: Checks if two operands are equal.
`!=`: Checks if two operands are not equal.
`>`: Checks if the first operand is greater than the second.
`>=`: Checks if the first operand is greater than or equal to the second.
`<`: Checks if the first operand is less than the second.
`<=`: Checks if the first operand is less than or equal to the second.
`&&`: Performs a logical AND operation.
`||`: Performs a logical OR operation.
`!`: Performs a logical NOT operation.
`+=`: Adds the right operand to the left operand and assigns the result to the left operand.
`-=`: Subtracts the right operand from the left operand and assigns the result to the left operand.
`*=`: Multiplies the left operand by the right operand and assigns the result to the left operand.
`/=`: Divides the left operand by the right operand and assigns the result to the left operand.
`%=`: Computes the remainder of dividing the left operand by the right operand and assigns the result to the left operand.
Unary `-`: Negates the value of the operand.

Array:

An array is a collection of elements of the same type stored in contiguous memory locations. It allows efficient access and manipulation of a fixed-size sequence of elements.

Initialisation:

data_type <name_of_array>[size] = { <elements> };
data_type <name_of_array>[size];// mentioning elements are optional;
data_type <name_of_array>[size,value] ; // initialise all array elements with provided value.

Example:

int my_first_array[5]={1,2,3,4,5};
int my_second_array[5];
int my_third_array[5,10]; // All elements of array will be initialised with 10.

```
int arr[5]={1,2,3,4,5};
x= arr[0];// accessing the elements
arr[4]=6;// changing the elements
y=arr[-1]; // error out of bound
z=arr[5]; // error out of bound
/* can access elements only in range 0 to size-1*/
```

.. Construct:

- 1..n : creates an array which contains elements from 1 to n(n exclusive)

```
arr = 1..n;
```

- 1..=n : Here, n is inclusive as well.

String :

It is a sequence of characters for representing text.
Initialisation:
string <name_of_string> =”<content>”;
// content of the string should be written between string literals (““) .

Example:
string s=”How are you?”;

Characteristics:
- **Dynamic Size:** `string` automatically manages its size and grows as needed, unlike fixed-size character arrays.
- **Built-in Functions**: It comes with many built-in functions for common operations like concatenation, comparison and size.

Key Operations on string:

- **Concatenation**: Strings can be concatenated using the `+` operator.
- **Length**: Use `.length()` to get the length of the string.
- **Access Characters**: You can access individual characters using the `[]` operator, similar to arrays.
- **Modify**: Strings can be modified by directly accessing characters
- **Comparison**: Strings can be compared using `==`, `!=`, `<`, `>`, etc.

Functions:

A function is a block of code that executes only when it is invoked. You can pass information, referred to as parameters, into a function. Functions are crucial for code reuse, enabling you to execute the same set of instructions multiple times without duplicating the code.

In our DSL, a copy is made of the parameters passed in a function. Two functions cannot be given the same name.
main function is an entry of the program.

Function declaration:

```
 func <name_of_Function> <return_type> (Parameters separated by comma){
// code block
}
```

Example:

```
func example_function int (int a, int b){
      c = a + b;
      a = 5;
      b = 10;
      return c;
}

int main(){
      a = 1;
      b = 2;
      // calling the function
      X = example_function(a,b);    /* this will return 3
                              but values of a and b will remain 1 and 2 */
```

References:

When a variable is declared as a reference, it becomes an alternative name for an existing variable. A variable can be declared as a reference by putting '&' in the declaration.

Our DSL incorporates the concept of references to enable data manipulating and resource management.

```
int a = 10;
int& ref = a;
ref = 20;   // now the value of a is 20
```

Avoids Unnecessary Copying: When variables are passed by reference, the function operates directly on the original variable, avoiding the cost of creating a copy. This is particularly advantageous for large data structures like arrays or objects, where copying would incur significant overhead.

Struct:

'struct' is a keyword that allows grouping several variables that may be of different types into one single entity. In the structure, each variable is said to be a structure member or

element. While arrays can hold elements of only the same type, a structure can hold variables of different types: int, float, char etc.
// Define a struct for a person

```
struct Person {
    // Data members
        int age;
        float height;
};
int main(){
    //Create an instance of person
    Person person;
     //Accessing and modifying struct members
     person1.age = 30;
     Person1.height = 1.75;
     return 0;
}
```

Control flow:

Conditional branching statements (if - else if - else)

The `if - else if - else` statement is used to execute different blocks of code based on a condition. It provides a way to control the flow of execution depending on whether a condition is true or false.

```
if (condition) {
        // Code to execute if condition is true
 }
```

```
if (condition) {
    // Code to execute if condition is true
} else {
    // Code to execute if condition is false
}
```

```
if (condition1) {
    // Code to execute if condition1 is true
} else if (condition2) {
    // Code to execute if condition 2 is true and condition1 is false
} else {
    // Code to execute if neither condition1 nor condition2 is true
}
```

Loops:

Loops are programming elements that repeat a portion of code multiple times. ‘for’ is a keyword used for declaring a loop.

Syntax of a for loop:

```
for (initialization; condition; update) {
    // Code to be executed in each iteration
}
```

- **Initialization**: This is executed once at the beginning of the loop. It is typically used to initialise the loop control variable.
- **Condition**: This is evaluated before each iteration of the loop. If it evaluates to true (non-zero), the loop body executes. If it evaluates to false (zero), the loop terminates.
- **Update**: This is executed after each iteration of the loop body. It is usually used to update the loop control variable.

Syntax for iterating over an array or string has been made concise by providing an additional ‘for’ loop syntax:

```
for (i in <array_name>){
     // in each iteration i becomes a copy of array element
}
for(i in <string_name>){
     // in each iteration i becomes a copy of a character of string
}
for( &i in <array_name>){
     //in each iteration i becomes a reference for an array element
}
for( &i in <string_name>){
     // in each iteration i becomes a reference for character of
string
}
```

Thinking in terms of Tasks and TaskGroups

Tasks

A **Task** represents a distinct unit of execution, encapsulating a block of code that is explicitly executed on a separate thread. This abstraction simplifies thread management by allowing developers to focus on task-level concurrency without needing to handle low-level thread management directly.

TaskGroups

A **TaskGroup** is a collection of tasks, providing a higher level of control over their execution. TaskGroups allow developers to group related tasks and manage their behaviour collectively, such as specifying execution order or defining synchronization constraints. The example below demonstrates how to define tasks within a **TaskGroup**.

```
@TaskGroup g1
{
   @Task t1
   {
       for (int i = 0; i < 10; i++)
       {
       }
   }

   @Task t2
```

```
    {
        for (int i = 0; i < 10; i++)
        {
        }
    }
}
```

Defining Tasks under TaskGroups

Here, we define a task group `g1` that contains two tasks: `t1` and `t2`. Both tasks execute a loop from 0 to 10. The tasks run independently, each on its own thread, but their execution can be collectively managed and coordinated through the **TaskGroup**.

Controlling Behaviour of Tasks and @Properties Directive

Within a **TaskGroup**, we introduce the `@Properties` block to control the behaviour of tasks such as ordering, memory access scopes, and more.

```
@TaskGroup g1
{
    @Task t1
    {}
    @Task t2
    {}
    @Task t3
    {}
    @Properties
    {
        ...
    }
}
```

The `@Properties` block within TaskGroup `g1` contains directives that control the behavior and execution of the tasks (`t1`, `t2`, `t3`, etc.). We use multiple directives within the `@Properties` block to define the desired behavior of tasks within the specific TaskGroup.

@Order Directive

The **@Order** directive is used to specify the execution order of tasks and determines how tasks are executed with respect to each other.

```
@Order

{

    t1 -> t2, t3; // rule within @Order directive

}
```

In this example, the `@Order` directive uses the `->` operator to enforce strict ordering. Here, `t1` is executed first, followed by the concurrent execution of `t2` and `t3` on separate threads. Depending on optimization, one of them might run on the same thread as `t1` once it finishes.

We can concatenate multiple `->` operators to define a sequence of tasks. For example, `t1 -> t2 -> t3` specifies that `t1` runs first, followed by `t2`, and then `t3`.

`all` Keyword

The `all` keyword is a useful shorthand to reference all tasks within a TaskGroup. It can **only** be placed on the leftmost side or the rightmost side in a rule. Precisely, it refers to all tasks that are not mentioned in the current rule.

Example:

```
t1, t2 -> all;  // All tasks except t1, t2 will run after t1 and t2 are
completed.

all -> t3, t4;  // t3 and t4 will execute after all tasks except t3 and t4
have been completed.
```

Conflict Detection

We perform compile-time checks to detect and resolve any conflicting order dependencies that may arise within the **@Order** directive. If a conflict is detected, such as circular dependencies or self-references (`t1 -> t1`), a compile-time error is generated to ensure correct task execution behaviour.

@Mem Directive

The **@Mem** directive is used to control memory access within tasks. The **@Mem** block allows us to explicitly specify which memory locations are accessible to each task, ensuring restricted and safe access to shared variables.

```
@Mem
{
    R2 -> t2;          // read-only access for t2
    R3 -> t3 mut;      // write access for t3
}
```

Defining Memory Access

The **@Mem** directive defines the memory locations that are allowed to be used by specific tasks. The arrow (`->`) operator assigns memory access to tasks.

In the example above:

- `R2 -> t2`: Task `t2` is granted **read-only** access to memory location `R2` (Type-1 access).
- `R3 -> t3 mut`: Task `t3` is granted **write** access to memory location `R3` (Type-2 access) by adding `mut` keyword.

Memory Access Rights

The memory access permissions in the DSL can be classified into three types:

1. **Access with Copy (Default)**: If a variable is not specified in the **@Mem** directive, a copy of the variable is created for the task. This is the default behaviour. The task is provided with a private copy of the variable, which is isolated from the main thread and other tasks.

2. **Read-Only Access to Existing Memory (Type-1)**: A task can reference the existing memory location of the main thread but can only read the variable. If the task attempts to write to this memory, a compile-time error is generated. Example:
   R2 -> t2; // read-only access to R2 for thread t2

3. **Write Access to Existing Memory (Type-2)**: A task can read and write to the shared memory location. This is achieved using the `mut` keyword. It grants the task **mutable** (i.e. both read and write) access to memory. Example:
   R3 -> t3 mut; // write access to R2 for thread t2

`unsafe` Keyword

The `unsafe` keyword can be used to bypass the default behaviour of copying memory. It grants **Type-1 access** to all variables, meaning that instead of creating a copy, the task references the original memory location directly, but only for read access. We can still use `@Mem` directive to grant write access to tasks.

@Shared Directive

The **@Shared** directive is used to define shared memory locations accessible by multiple tasks. These shared memory variables are thread-safe and allow controlled access to specific tasks.

Rules:

- Shared variables must not start with an underscore (`_`).
- They are accessed within tasks by prepending an underscore (`_`) to their name.
- The names must also follow standard naming conventions.

Example:

```
@Task t1
{
    _sh;
}
@Task t2
{
    _sh;
}
@Shared
{
    sh: int[2] a{0} -> t1, t2;  // Shared variable 'sh' (array) accessible
by t1 and t2.
    // can be used for synchronization like Peterson's lock.
}
```

The `@Shared` directive declares shared memory that can be used by multiple tasks. The arrow (`->`) operator assigns access rights from a shared memory variable to tasks. In the example above, `sh` is a shared array of integers, accessible by both `t1` and `t2` tasks.

@Supervisor Directive

The **@Supervisor** directive defines special threads that have the ability to control the execution of tasks manually, overriding any predefined task ordering. When a `@Supervisor` is specified, the `@Order` directive is ignored, and the `Supervisor` can control and call tasks using the `call` keyword.

Rules:

- The **Supervisor** controls the invocation of tasks by calling them with the `call` keyword.
- When a task is called, it returns a **Task Object**, which can be joined to ensure proper synchronisation between threads.
- If the `join` method is not explicitly called on a task object, it will automatically be called at the end of the Supervisor's execution.
- Note that Memory usage for the tasks still needs to be explicitly defined even under **Supervisor** control.

Example:

```
@Supervisor s1
{
    i1 = call t1;    // calling task1 and storing the task object in i1
    i1.join();       // wait for task1 to complete
}
```

Here, Supervisor `s1` calls task `t1` using the `call` keyword, which returns a **Task Object** (`i1`). The `join()` method is then called on the task object to ensure that the Supervisor waits for the task's completion.

Conditional Signals and Returns (Inspired by Go)

The **Conditional Signals and Returns** feature allows tasks to communicate and synchronise based on specific conditions. A task can signal another task once certain preconditions are fulfilled, allowing for efficient coordination between tasks. Also, return values can be passed between tasks.

Syntax:

```
@TaskGroup
{
    @Task t1
    {
        ...// pre-conditions for t1
        .ct <- 5;  // send 5 upon completion of preconditions to channel 'ct'
        ...
        .ct;       // signal completion to other tasks(no return value)
    }
    @Task t2
    {
        .wt{t1, 1} -> x;  // wait for signal from t1, capture sent value into 'x'
    }
}
```

Explanation:

- Task t2 waits for a signal from t1. The signal is only sent after the preconditions in t1 are executed.
- In this example, task t1 sends the value 5 into the channel .ct once its preconditions are complete.
- Task t2 uses the .wt{} directive to wait for the signal from t1. The return value from the first signal from t1 is captured into the variable x.

Working:

- The symbol <- is used to send values from one task to another via channels. Task t1 sends the value 5 into the .ct channel.
- .wt{} allows task t2 to wait for the first signal from t1. Once t1 completes, the value sent (in this case, 5) is assigned to x.

This mechanism synchronises tasks based on conditions and facilitates passing values along channels, similar to Go's communication model.

Important Points

- The all keyword can turn a conditional signal into a broadcast-type signal. When a signal is sent using the .ct(all) syntax, it broadcasts the signal to all waiting tasks for the corresponding signal rather than a single task.
- The value sent through a channel is passed to all tasks or the specific task that resumes from a sleep state due to the fulfilled condition. When a task returns from its wait state due to a signal, it receives the value sent through the channel.

Additional Features

Logging

The **Logging** feature allows TaskGroups to generate logs that record the start and end times of tasks. This feature is useful for monitoring, debugging, identifying bottlenecks and improving the performance of the program.

Syntax:

```
@TaskGroup (log="file_name")
{
    ...
}
```

- `log="file_name"`: Specifies that a log file with the name `file_name` will be created to record timestamps for when each task starts and finishes.

Thread Pool

The **Thread Pools** feature allows for more controlled execution of tasks by creating a pool of threads. Instead of executing all tasks concurrently, tasks are distributed among a limited number of threads (saving overhead of creating and destroying threads).

Syntax:

```
@TaskGroup g(k=4)
{
    @Task t1(k=2)
    {
        ...
    }
}
```

- `k=4`: Specifies that the TaskGroup `g` will be executed using at most 4 threads concurrently. The remaining tasks will be managed in a thread pool.

- `k=2`: Specifies that task `t1` will be executed by at most 2 threads concurrently. This provides more control over the concurrency of individual tasks.

Example:

```
@TaskGroup g(k=4)
{
    @Task t1(k=2)
    {}
    @Task t2
    {}
}
```

Here, TaskGroup `g` will use up to 4 threads to execute its tasks. Task `t1` will be restricted to a maximum of 2 threads running concurrently, while other tasks within the TaskGroup can utilise the remaining threads up to the limit of 4.

Parallel Execution Using the `@parallel` Construct

The `@parallel` construct facilitates parallel execution of both loops and code blocks, offering dynamic control over thread allocation, iteration chunking, and execution strategy (similar to openMP).

Parallel Looping

- **Syntax**:

```
@parallel(shared=[var1, var2], private=[var3], reduction=[+: x, y; *: z],
threads=num_threads, schedule='static'|'dynamic')

for i in start..end
{
    // Loop body
}
```

- **Parameters**:
    - `shared`: Specifies a list of variables that are shared among all threads.
    - `private`: Specifies a list of variables that are private to each thread, ensuring no interference between threads.
    - `reduction`: Defines reduction operations, where `+: var1, var2` represents addition reductions, and `*: var3` represents multiplication reductions.
    - `threads`: (Optional) Specifies the number of threads to be used for parallel execution. If not provided, the system defaults to a number based on available hardware.
    - `schedule`: Defines how loop iterations are distributed among threads:
        - `'static'` (default): Iterations are divided evenly among threads before execution.
        - `'dynamic'`: Iterations are assigned to threads dynamically during execution, which is beneficial for non-uniform workloads.

Example:

```
total_sum = 0

@parallel(shared=[data], private=[temp], reduction=[+: total_sum],
threads=8, schedule='dynamic')

for i in range(0, len(data))

{
    temp = compute(data[i]);
    total_sum += temp;
}
```

In this example, the `total_sum` variable is shared among threads, `temp` is private to each thread, and the loop uses dynamic scheduling with 8 threads.

Parallel Code Block

The `@parallel` construct also supports parallel execution of code blocks.

- **Syntax**:

```
@parallel(threads=num_threads)
{
    // Code block to run in parallel
}
```

- **Parameters**:
    - `threads`: (Optional) Specifies the number of threads for parallel execution of the code block. The system uses a default number based on available hardware if not provided.

Example:

```
@parallel(threads=4)
{
    // Parallel code block execution
}
```

In this example, the code block is executed in parallel using 4 threads.