P-lang语言设计说明书

一、概述

1.1 背景介绍

近几年来，高性能计算，尤其是并行计算越来越多地进入我们的视野，特别是在人工智能领域，模型的训练、推理离不开高度并行的GPU等硬件。即使是对于普通用户来说，并行计算也能极大地提升工作效率。

多线程是常用的并行计算手段，广泛使用的程序设计语言，如C/C++、Java、Python等都提供了对多线程的支持。但实际编码时发现，这些语言提供的线程操作往往很繁琐，对于初学者来说需要花费大量的时间去学习，并且开发难度也远远高于一般的程序。

因此我们希望设计一种支持并行（多线程）的语言，它具有语法简单、并行直观、功能强大的优点。即使是没有接触过多线程的初学者，在看完这篇说明后也能迅速学会常用操作，并编写出漂亮的多线程程序。

1.2 P-lang简介

我们的语言命名为P-lang，P取自并行（Parallel）的首字母。这门语言最突出的特点就是能用极其简单的语法实现线程操作，并且程序的可读性很高，能够轻易地分辨哪些部分是单线程，哪些是多线程。如下面简短的几行代码就能实现线程的创建和运行：

```
1   int index[3] = {1, 2, 3};
2   parallel (int x) in index {
3       printf("I'm thread %d\n", x);
4   }
```

上面的代码将 `index` 数组中的序号分别传入三个子线程，然后在各个子线程中打印序号。其中 `parallel` 关键字表明块中的语句是并行执行的，而 `parallel` 块之外的语句则是顺序单线程执行。从这里不难看出P-lang简洁和直观的优势。

我们的项目在github上开源，地址为 **https://github.com/Longxmas/ADSL_HW** ，其中提供了编译器和一些测试用例。

二、P-lang基础

在正式介绍文法和编译器实现之前，我们打算用一个章节来介绍P-lang的总体设计，使读者先对这门语言有一个大致的了解。

2.1 基础语法

P-lang的语句以 `;` 结尾。注释可以是单行的 `//` ，也可以是多行的 `/* */` 。主函数必须声明为 `void main() {}` ，不能有返回值。

2.2 数据类型

P-lang中数据分为常量和变量，常量必须在声明时赋值，变量可以在声明时赋值，也可以之后再赋值。常量用关键字 `const` 表示。

P-lang是强类型语言，共支持4种基本数据类型，其解释如下表所示。

类型	关键字	说明
布尔	`bool`	取值为 `true` 或 `false` 。
整型	`int`	32位有符号整数。
浮点	`float`	32位浮点数。
字符串	`str`	字符串常量，长度任意。

P-lang还支持数组类型，可以是任意一种基本数据类型的数组，最多**二维**。下面给出一些创建的示例：

```
const bool b = true;
int i = 5;
float f;
str s = "abc";
int arr[3] = {1, 2, 3};
```

2.3 运算符

P-lang支持如下运算符：

- **算术运算符**：`+`、`-`、`*`、`/`、`%`；
- **比较运算符**：`==`、`!=`、`>`、`>=`、`<`、`<=`；
- **逻辑运算符**：`&&`、`||`、`!`。

2.4 条件语句

P-lang支持if-else条件语句，语法如下：

```
if (<条件表达式>) {
    <语句>
} else {
    <语句>
}
```

其中 `else` 是可选部分，且如果 `{}` 中的语句只有一条则可以省略 `{}` 。

2.5 循环语句

P-lang支持两种循环语句。

- 一种是带有3个参数的 `for` 语句，其用法与C语言的 `for` 相同，三个表达式都是可选项，语法如下：

  ```
  for (<表达式1>; <条件表达式>; <表达式2>) {
      <语句>
  }
  ```

- 另一种是`for-in`语句，用于方便地遍历数组，语法如下：

```
for <变量> in <数组> {
    <语句>
}
```

其中的变量不需要提供类型，因为可以从数组的类型推断。

和C语言一样，循环中允许使用`break`和`continue`跳出循环和跳过本次循环。

2.6 函数

函数需要指定参数和返回值的数据类型，其定义语法如下：

```
def <返回值类型> <函数名>(<参数1类型> <参数1名>, ...) {
    <语句>
    return <返回值>;
}
```

返回值可以是4种基本数据类型，如果没有返回值，则类型指定为`void`。

2.7 并行语句块

这是P-lang最具特色的语法，通过该并行语句块可以实现多线程并行。其语法如下：

```
parallel (<参数1类型> <参数1名>, ...) in <数组1>, ... {
    <并行语句>
}
```

`parallel`关键字声明了该语句块为并行语句块。参数列表类似函数的参数列表，这些参数将会传递给各个线程。参数的值来自`in`后面的数组列表，数组和参数一一对应，且要求所有的数组长度相同。P-lang将会跟据数组的长度`len`创建`len`个线程，每个数组的第`i`个值将会传递给第`i`个线程的对应参数中。

需要指出的是并行语句块和主程序中的代码也是并行执行的，如果想让主程序等待各线程，则应该使用下面的管道进行阻塞。

2.8 管道

为了支持线程间传递数据我们设计了管道数据类型，管道类型不能是`const`，创建管道只需在基本数据类型前加上`pipe`关键字，如下面的例子：

```
pipe int ret;
```

这行代码定义了一个`int`类型的管道，它能容纳一个`int`类型的数据。管道数据类型不允许赋初值。

要对管道类型的数据进行读写需要使用专门的管道运算符，如下所示：

```
ret << a;
ret >> b;
```

`<<`表示将右边的数据写入管道，数据类型与管道的类型必须相同；`>>`表示将管道的数据读出至右边的变量，另外右边的变量可以不写，表示只将管道的数据读出但不赋给任何变量。

管道的行为往往伴随着阻塞，当一个线程向管道中写入一个数据，这个线程将被阻塞，直到有线程读出了这个数据；另一方面，只有管道中有数据时，读线程才能读出数据继续执行，否则也会被阻塞。

通过管道可以实现线程的同步和通信，如下面的例子通过管道实现了主线程等待子线程并接收返回值：

```
int index[3] = {0, 1, 2};
pipe int p[3];
parallel (int x, pipe int r) in index, ret {
    ...
    r << x;
}
int res[3];
for i in index {
    ret[i] >> res[i];
}
```

2.9 互斥语句块

为了实现对共享变量的正确访问，我们实现了互斥语句块，语法如下所示：

```
mutex <互斥语句块名> {
    <互斥语句>
}
```

`mutex` 关键字声明了该语句块为互斥语句块，线程在进入互斥语句块前会先进行加锁，保证同一时间只有一个线程能访问互斥语句块中的语句，执行完后再释放锁。 `mutex` 关键字后应指定互斥语句块的名称，以区分不同的互斥语句块。

三、语法语义

3.1 词法规则

3.1.1 关键字

本语言的关键字及其含义如下表所示：

关键字	意义	Token
const	定义常量	CONST
int	整数类型	INT
void	无返回值类型	VOID
float	浮点数类型	FLOAT
bool	布尔类型	BOOL
str	字符串类型	STR
true	布尔值真	TRUE
false	布尔值假	FALSE
return	返回语句	RETURN
if	条件判断语句	IF
else	条件判断分支语句	ELSE
for	循环语句	FOR
break	循环终止语句	BREAK
continue	继续循环语句	CONTINUE
scanf	格式化输入语句	SCANF
printf	格式化输出语句	PRINTF
parallel	并行执行语句	PARALLEL
main	主函数定义关键字	MAIN
in	用于指定范围或集合成员关系（如在循环中）	IN
def	函数定义关键字	DEF
mutex	互斥锁关键字	MUTEX
pipe	管道关键字（用于进程间通信或数据处理管道）	PIPE

3.1.2 操作符

支持的运算符及其对应的 token 类型如下：

运算符	Token
+	PLUS
-	MINUS
*	TIMES
/	DIVIDE
%	MOD
=	ASSIGN
==	EQUAL
!=	NOTEQUAL
<	LESS
<=	LESSEQUAL
>	GREATER
>=	GREATEREQUAL
&&	LOGICALAND
||	LOGICALOR
!	NOT
<<	LSHIFT（用于管道读）
>>	RSHIFT（用于管道写）
(	LPAREN
)	RPAREN
{	LBRACE
}	RBRACE
[	LBRACKET
]	RBRACKET
;	SEMICOLON
,	COMMA

3.1.3 标识符和字面量

- **标识符**：以字母或下划线开头，后跟字母、数字或下划线的字符序列，用于标识变量、函数等程序实体，在词法分析中匹配正则表达式 `[a-zA-Z_][a-zA-Z_0-9]*`。例如 `myVariable`、`func1` 等。
- **数值常量**：

    - **整数常量**：匹配正则表达式 `0|[1-9][0-9]*`，例如 `0`、`42`、`12345` 等，在词法分析时将其转换为整数值。
    - **浮点常量**：匹配正则表达式 `\d+\.\d+`，如 `3.14`、`2.5` 等，词法分析阶段转换为浮点数值。
- **字符串常量**：由双引号括起的字符序列，支持转义字符，匹配正则表达式 `"([^"\\]|\\.)*"`。例如 `"Hello, World!"`、`"This is a \"test\" string."`。在词法分析时去除双引号，保留转义字符对应的实际字符。
- **布尔常量**：`true` 和 `false`，分别表示布尔值真和假，词法分析时转换为相应的布尔值。

3.2 语法规则

采用 BNF 范式描述语法规则：

3.2.1 程序结构

- **CompUnit**（编译单元）：

```
CompUnit : Decls FuncDefs MainFuncDef
    | Decls MainFuncDef
    | FuncDefs MainFuncDef
    | MainFuncDef
```

  编译单元可以由声明部分（Decls）、函数定义部分（FuncDefs）和主函数定义（MainFuncDef）以不同组合构成。

- **Decls**（声明列表）：`Decls : Decl Decls | Decl`
  声明列表由一个或多个声明（Decl）组成。
- **Decl**（声明）：`Decl : ConstDecl | VarDecl`
  声明可以是常量声明（ConstDecl）或变量声明（VarDecl）。

3.2.2 常量声明相关

- **ConstDecl**（常量声明）：`ConstDecl : CONST BType ConstDefList SEMICOLON`
  常量声明以 `CONST` 关键字开头，后跟基本类型（BType）、常量定义列表（ConstDefList）和分号。
- **ConstDefList**（常量定义列表）：`ConstDefList : ConstDef | ConstDef COMMA ConstDefList`
  常量定义列表由一个或多个常量定义（ConstDef）组成，用逗号分隔。
- **ConstDef**（常量定义）：

```
ConstDef : IDENTIFIER ASSIGN ConstInitVal
    | IDENTIFIER LBRACKET ConstExp RBRACKET ASSIGN ConstInitVal
    | IDENTIFIER LBRACKET ConstExp RBRACKET LBRACKET ConstExp RBRACKET ASSIGN ConstInitVal
```

  常量定义可以是简单的标识符赋值常量初值（如 `x = 42`），也可以是数组形式的常量定义（如 `arr[3] = {1, 2, 3}`）。

- **ConstInitVal**（常量初值）：`ConstInitVal : ConstExp | LBRACE ConstInitValList RBRACE`
  常量初值可以是常量表达式（ConstExp）或由花括号括起的常量初值列表（ConstInitValList）。

- **ConstInitValList**（常量初值列表）：`ConstInitValList : ConstInitVal | ConstInitVal COMMA ConstInitValList`
  常量初值列表由一个或多个常量初值组成，用逗号分隔。
- **ConstExp**（常量表达式）：`ConstExp : AddExp`
  常量表达式主要由加法表达式（AddExp）构成，也可包含其他更复杂的表达式运算。

3.2.3 基本类型相关

- **BType**（基本类型）：`BType : INT | BOOL | FLOAT | STR | PIPE INT | PIPE BOOL | PIPE FLOAT`
  基本类型包括整数型（INT）、布尔型（BOOL）、浮点型（FLOAT）、字符串型（STR）以及用于管道操作的特定类型（如 `PIPE INT` 表示管道传输整数数据类型）。
- **FuncBType**（函数返回基本类型）：`FuncBType : INT | BOOL | FLOAT | STR | VOID`
  函数返回值的基本类型可以是整数型、布尔型、浮点型、字符串型或无返回值类型（VOID）。

3.2.4 变量声明相关

- **VarDecl**（变量声明）：`VarDecl : BType VarDefList SEMICOLON`
  变量声明以基本类型开头，后跟变量定义列表（VarDefList）和分号。
- **VarDefList**（变量定义列表）：`VarDefList : VarDef | VarDef COMMA VarDefList`
  变量定义列表由一个或多个变量定义（VarDef）组成，用逗号分隔。
- **VarDef**（变量定义）：

```
VarDef : IDENTIFIER
    | IDENTIFIER ArrayDimensions
    | IDENTIFIER ASSIGN InitVal
    | IDENTIFIER ArrayDimensions ASSIGN InitVal
```

变量定义可以是简单的标识符定义，也可以是带有数组维度定义（ArrayDimensions）或初始化值（InitVal）的形式，如 `x`、`arr[3]`、`y = 3.14`、`matrix[2][3] = {{1, 2}, {3, 4}}`。

- **ArrayDimensions**（数组维度）：`ArrayDimensions : LBRACKET ConstExp RBRACKET | ArrayDimensions LBRACKET ConstExp RBRACKET`
  数组维度定义通过方括号内的常量表达式（ConstExp）指定，支持多维数组定义，如 `[3]`（一维数组长度为 3）、`[2][3]`（二维数组，第一维长度为 2，第二维长度为 3）。
- **InitVal**（变量初值）：`InitVal : Exp | LBRACE InitValList RBRACE`
  变量初值可以是表达式（Exp）或由花括号括起的初值列表（InitValList），用于数组等复合类型的初始化。
- **InitValList**（变量初值列表）：`InitValList : InitVal | InitVal COMMA InitValList`
  变量初值列表由一个或多个初值组成，用逗号分隔。

3.2.5 表达式相关

- **Exp**（表达式）：`Exp : AddExp`
  表达式以加法表达式（AddExp）为基础构建更复杂的表达式。
- **LOrExp**（逻辑或表达式）：`LOrExp : LAndExp | LOrExp LOGICALOR LAndExp`
  逻辑或表达式由逻辑与表达式（LAndExp）通过逻辑或运算符（LOGICALOR）组合而成。
- **LAndExp**（逻辑与表达式）：`LAndExp : EqExp | LAndExp LOGICALAND EqExp`
  逻辑与表达式由相等性表达式（EqExp）通过逻辑与运算符（LOGICALAND）组合而成。

- **AddExp**（加法表达式）：`AddExp : MulExp | AddExp PLUS MulExp | AddExp MINUS MulExp`
  加法表达式由乘法表达式（MulExp）通过加法或减法运算符构建。
- **MulExp**（乘法表达式）：`MulExp : UnaryExp | MulExp TIMES UnaryExp | MulExp DIVIDE UnaryExp | MulExp MOD UnaryExp`
  乘法表达式由一元表达式（UnaryExp）通过乘法、除法或取模运算符构建。
- **EqExp**（相等性表达式）：`EqExp : RelExp | EqExp EQUAL RelExp | EqExp NOTEQUAL RelExp`
  相等性表达式由关系表达式（RelExp）通过相等或不等运算符构建。
- **RelExp**（关系表达式）：`RelExp : AddExp | RelExp LESS AddExp | RelExp LESSEQUAL AddExp | RelExp GREATER AddExp | RelExp GREATEREQUAL AddExp`
  关系表达式通过加法表达式和关系运算符构建，用于比较大小关系。
- **UnaryExp**（一元表达式）：`UnaryExp : PrimaryExp | IDENTIFIER LPAREN FuncRParams RPAREN | UnaryOp UnaryExp`
  一元表达式可以是基本表达式（PrimaryExp）、函数调用形式（以标识符开头后跟括号及参数列表）或由一元运算符（UnaryOp）作用于一元表达式构成。
- **UnaryOp**（一元运算符）：`UnaryOp : PLUS | MINUS | NOT`
  一元运算符包括正号、负号和逻辑非运算符。
- **PrimaryExp**（基本表达式）：`PrimaryExp : LPAREN Exp RPAREN | LVal | INTCONST | FLOATCONST | STRCONST | TRUE | FALSE`
  基本表达式可以是括号括起的表达式、左值表达式（LVal）、常量值（整数、浮点、字符串、布尔常量）。
- **LVal**（左值表达式）：`LVal : IDENTIFIER | IDENTIFIER ArrayDimensions`
  左值表达式可以是简单标识符或带有数组维度的标识符，用于表示可赋值的变量或数组元素。

3.2.6 函数相关

- **FuncDefs**（函数定义列表）：`FuncDefs : FuncDef | FuncDef FuncDefs`
  函数定义列表由一个或多个函数定义（FuncDef）组成。
- **FuncDef**（函数定义）：`FuncDef : DEF FuncBType IDENTIFIER LPAREN FuncFParams RPAREN Block | DEF FuncBType IDENTIFIER LPAREN RPAREN Block`
  函数定义以 `DEF` 关键字开头，后跟返回类型（FuncBType）、函数名（IDENTIFIER）、参数列表（FuncFParams 或为空括号）和函数体（Block）。
- **FuncFParams**（函数形参列表）：`FuncFParams : FuncFParam | FuncFParam COMMA FuncFParams`
  函数形参列表由一个或多个形参（FuncFParam）组成，用逗号分隔。
- **FuncFParam**（函数形参）：`FuncFParam : BType IDENTIFIER | BType IDENTIFIER LBRACKET ConstExp RBRACKET | BType IDENTIFIER LBRACKET ConstExp RBRACKET LBRACKET ConstExp RBRACKET`
  函数形参可以是基本类型后跟标识符的简单形式，也可以是带有数组维度定义的形式，用于函数参数的定义，如 `int x`、`float arr[3]`、`str matrix[2][3]`。
- **FuncRParams**（函数实参）：`FuncRParams : Exp | Exp COMMA FuncRParams`
  函数实参列表由一个或多个表达式（Exp）组成，用逗号分隔，用于函数调用时传递参数。

3.2.7 语句相关

- **Stmt**（语句）：

```
Stmt : LVal ASSIGN Exp SEMICOLON
| Exp SEMICOLON
| SEMICOLON | Block
| IF LPAREN Cond RPAREN Stmt ELSE Stmt
| IF LPAREN Cond RPAREN Stmt
| FOR IDENTIFIER IN IDENTIFIER Stmt
| FOR LPAREN ForExp SEMICOLON Cond SEMICOLON ForExp RPAREN Stmt
| FOR LPAREN ForExp SEMICOLON SEMICOLON ForExp RPAREN Stmt
| FOR LPAREN ForExp SEMICOLON Cond SEMICOLON RPAREN Stmt
| FOR LPAREN SEMICOLON Cond SEMICOLON ForExp RPAREN Stmt
| BREAK SEMICOLON | CONTINUE SEMICOLON
| RETURN Exp SEMICOLON | RETURN SEMICOLON
| LVal LSHIFT Exp SEMICOLON | LVal RSHIFT Exp SEMICOLON | LVal RSHIFT
SEMICOLON
| PRINTF LPAREN STRCONST RPAREN SEMICOLON
| PRINTF LPAREN STRCONST PRINTFParams RPAREN SEMICOLON
| SCANF LPAREN STRCONST PRINTFParams RPAREN SEMICOLON
| PARALLEL LPAREN FuncFParams RPAREN IN ParallelRealList Block
```

语句可以是赋值语句（如 `x = 3`）、表达式语句（如 `3 + 4;`）、空语句（`;`）、语句块（Block）、条件语句（if-else）、循环语句（for 循环的多种形式）、跳转语句（break、continue、return）、移位语句（如 `x << 2;`）、输入输出语句（printf、scanf）、并行执行语句（parallel）等。

- **ForExp**（for 循环初始化表达式）：`ForExp : LVal ASSIGN Exp`
  用于 for 循环初始化部分，如 `i = 0`。
- **PRINTFParams**（printf 函数参数列表）：`PRINTFParams : COMMA Exp | COMMA Exp PRINTFParams`
  printf 函数的参数列表由一个或多个表达式组成，用逗号分隔，用于指定输出格式和内容，值得注意的是，由于传参形式相同，PRINTParams同样也可以作为SCANF语句的参数。
- **ParallelRealList**（并行执行实参列表）：`ParallelRealList : LVal | LVal COMMA ParallelRealList`
  并行执行语句中的实参列表由一个或多个左值表达式组成，用逗号分隔，用于指定并行操作的对象。
- **Block**（语句块）：`Block : LBRACE RBRACE | LBRACE BlockItems RBRACE | MUTEX IDENTIFIER LBRACE BlockItems RBRACE`
  语句块可以是空块（`{}`）、由多个语句项（BlockItems）组成的块或带有互斥锁（mutex）保护的语句块，用于组织语句的执行顺序和作用域控制。
- **BlockItems**（语句块项）：`BlockItems : BlockItem | BlockItem BlockItems`
  语句块项由一个或多个语句块项组成，可以是声明（Decl）或语句（Stmt）。
- **BlockItem**（语句块子项）：`BlockItem : Decl | Stmt`
  语句块子项可以是声明或语句，用于填充语句块内容。
- **Cond**（条件表达式）：`Cond : LOrExp`
  条件表达式由逻辑或表达式构成，用于条件语句和循环语句中的条件判断。

3.3 指称语义

3.3.1 语义域

1. **Value域**（`Value`）

该域包含程序中所有可能的值类型，包括：

- **Integer**：整数值。
- **Float**：浮点数值。
- **String**：字符串值。
- **Bool**：布尔值（`true` 或 `false`）。
- **Array_Value**：数组值。
- **Function**：函数值，表示一个函数的引用。
- **Mutex**：互斥锁值，用于表示并行任务之间的同步。
- **Pipe**：管道值，用于并行任务之间的数据通信。

Value域的指称语义：值是表达式的结果，可以是任意类型的数据，表示程序的计算结果。

2. **存储域**（`Store`）

存储域定义了内存中变量或数据的位置：

- `Location → (stored Storable + undefined + unused)`：每个存储位置可以存储某些值（`Storable`），或者处于“未定义”或“未使用”状态。

Store的指称语义：存储域决定了程序中变量的位置，并且每个存储位置包含一个实际值，表示内存中的数据。

3. **绑定域**（`Bindable`）

绑定域将值与变量的存储位置进行绑定：

- **Value**：表示具体的值（如整数、浮点数等）。
- **Location**：表示变量或数据的存储位置。

Bindable的指称语义：绑定域将标识符（如变量名）与存储位置或具体的值相关联。变量名和存储位置是绑定关系的核心。

4. **环境域**（`Environ`）

环境域将标识符（变量名）与其绑定的值或位置关联：

- `identifier → (bound Bindable + unbound)`：每个标识符可能与某个存储位置或值绑定，或者未绑定。

Environ的指称语义：环境域表示程序的上下文，定义了标识符与其对应的存储位置或值之间的映射。每次查找标识符时，都会在环境中找到相应的绑定信息。

3.3.2 辅助函数

1. 环境域的辅助函数：

- **empty_environ**：返回一个空的环境。
- **bind**：将标识符与可绑定的值或位置绑定。

- **overlay**：将两个环境合并，产生新的环境。
- **find**：在环境中查找标识符对应的绑定值。

2. 存储域的辅助函数：

- **empty_store**：返回一个空的存储。
- **allocate**：为新的变量或数据分配存储空间。
- **deallocate**：释放指定位置的存储空间。
- **update**：更新存储中某个位置的值。
- **fetch**：从存储中提取值。
- **coerce**：将某个绑定值转换为相应的值。

3.3.3 变量定义的指称语义

1. 变量定义的辅助函数：

- **elaborate**：将变量定义转化为环境和存储操作，生成新的环境和存储。
- **get_type**：获取变量的类型。
- **allocate_base**：为基础类型变量分配存储空间。
- **allocate_array**：为数组分配存储空间。

2. 变量定义指称语义过程：
3. **评估表达式**：首先评估赋值语句右侧的表达式（`evaluate expression`），计算出值。
4. **获取类型**：通过 `get_type(val)` 获得表达式的值类型。
5. **分配存储空间**：通过 `allocate` 为变量分配存储空间。
6. **绑定标识符**：通过 `bind(ID, variable loc)` 将标识符与存储位置绑定。
7. **更新存储**：使用 `update(sto, loc, val)` 将计算得到的值存储到相应的位置。

3.3.4 表达式指称语义

辅助函数：

- **sum**：对两个数值进行加法计算。
- **mul**：对两个数值进行乘法计算。
- **evaluate**：对表达式进行求值，返回一个值。
- **evaluate_array**：对数组表达式进行求值，返回数组值。
- **compare_type**：比较两个变量的类型是否相同。

`for` 循环表达式：

1. `for` **循环执行**：在我们的设计中，`for` 循环会依据循环初始化、条件和步进表达式来执行，并且支持并行计算。以下是对 `for` 循环的指称语义描述：

```
execute[for IDENTIFIER in range_expression stmt] =
    let range = evaluate range_expression env sto in
    let (sto', loc) = allocate(sto) in
    let (sto'', var) = bind(ID, variable loc), sto' in
    execute_for_loop range stmt sto''
```

具体执行步骤如下：

- **评估范围表达式**：`evaluate range_expression` 计算出循环范围，并得到对应的值（如从一个数组或数值区间中获取）。
- **分配存储空间**：为循环变量分配存储空间，`allocate(sto)` 为变量分配内存位置。
- **绑定循环变量**：通过 `bind(ID, variable loc)` 将循环变量与存储位置绑定。
- **执行循环体**：调用 `execute_for_loop` 来执行循环体。该函数通过并行计算（如需要）执行循环中的语句。

2. `for` **循环的并行计算**： 如果 `for` 循环中包含 `PARALLEL` 语句，指称语义会确保循环体中的任务并行执行。例如：

```
execute[for IDENTIFIER in range_expression parallel stmt] =
    let range = evaluate range_expression env sto in
    let (sto', loc) = allocate(sto) in
    let (sto'', var) = bind(ID, variable loc), sto' in
    execute_parallel_for_loop range stmt sto''
```

在这种情况下，`execute_parallel_for_loop` 会确保并行执行每次循环的操作。

3. `for` **循环内的并行任务调度**： 循环中的任务可能会在多个线程或进程中并行执行，具体的任务调度和同步依赖于 `PARALLEL` 语句的实现：
   - 在每次迭代中，任务的输入参数（如 `FuncFParams`）会被计算并传递给并行任务。
   - `PARALLEL` 语句通过 `IN` 关键字与并行执行的任务语句块相关联。

3.3.5 函数指称语义

函数声明与定义：

1. **函数声明**：

```
elaborate[func ID(FP) [generics -type-list] E] env =
    if has_gen_list == truth_value true
    then env' = allocate_gen(sto, env) in
    else env' = env
    let func arg =
        let parenv = bind_parameter FP arg in
        evaluate E(overlay(parenv, env'))
    in
    bind(ID, function func)
```

2. **函数调用**：

```
evaluate[ID(AP)] env =
    let function func = find(env, ID) in
    let arg = give_argument AP env in
    func arg
```

3.3.6 并行计算的指称语义

并行语句（`PARALLEL`）：

1. `PARALLEL` 语句

：执行多个函数并行计算。

```
IN
```

关键字用于并行任务之间的通信或数据交换。

```
elaborate[parallel FuncFParams IN ParallelRealList Block] env sto =
    let parallel_func arg =
        evaluate Block (overlay(parenv, env))
    in
    let (sto', loc) = allocate(sto) in
    bind(ID, function parallel_func), sto'
```

互斥锁（`mutex`）：

1. 互斥锁的语义

   ：使用互斥锁来确保在并行任务中对共享资源的独占访问。

```
elaborate[mutex ID LBRACE BlockItems RBRACE] env sto =
    let (sto', loc) = allocate(sto) in
    bind(ID, variable loc), sto'
```

管道（`pipe`）：

1. 管道的语义

   ：通过管道在并行任务之间传递数据。

```
elaborate[pipe ID] env sto =
    let (sto', loc) = allocate(sto) in
    bind(ID, pipe loc), sto'
```

四、编译器实现

4.1 词法分析

我们使用了**PLY**（Python Lex-Yacc）库来实现词法分析器，基于正则表达式的方式来识别输入源代码中的词法单元（tokens）。下面是具体的实现方式：

1. **定义词法规则**

在词法分析器中，首先定义了各种**词法规则**，这些规则通过正则表达式来匹配输入源代码中的不同词法单元（如标识符、常量、操作符等）。例如，对于浮点数的定义：

```
def t_FLOATCONST(t):
    r'\d+\.\d+'  # 匹配浮点数的正则表达式
    t.value = float(t.value)  # 将匹配到的字符串转换为浮点数
    return t
```

这种方式允许灵活地定义词法单元，通过正则表达式来指定各类标识符、常量（如整数、浮点数、布尔值、字符串常量）和操作符（如`+`，`-`，`*`等）。

2. **构建有限状态机（FSM）**

我们选择通过 PLY自动根据上述定义的词法规则（正则表达式）生成一个有限状态机（FSM），来处理源代码中的字符流。词法分析器会根据当前状态和输入字符，查找状态转换规则，完成词法单元的识别。

3. **执行状态转换**

在状态转换部分，通过 `lexer.input(text)` 将源代码传递给词法分析器，词法分析器会根据定义好的规则，通过自动生成的有限状态机逐步匹配字符并识别出相应的词法单元。每次匹配成功后，返回相应的 token。

```
def lex_input(text):
    lexer.input(text)  # 将源代码传递给词法分析器
    tokens = []
    while True:
        token = lexer.token()  # 获取下一个token
        if not token:
            break
        tokens.append(token)  # 将token添加到tokens列表
    return tokens
```

4. **识别词法单元**

当词法分析器匹配到一个完整的词法单元时，它会返回一个 `token` 对象，包含了词法单元的类型和对应的值。比如，对于浮点数 `3.14`，词法分析器会返回一个 `FLOATCONST` 类型的 token，值为 `3.14`。

5. **错误处理**

词法分析部分通过 `t_error` 函数实现了错误处理。当词法分析器遇到无法识别的字符时，它会调用 `t_error` 函数，并输出错误信息：

```
def t_error(t):
    print(f"Illegal character '{t.value[0]}' at line {t.lineno}")
    t.lexer.skip(1)
```

这会在源代码中发现非法字符时，打印出错误信息并跳过该字符继续分析。

4.2 语法分析

语法分析部分使用了 PLY 的递归下降解析法来构建抽象语法树（AST）。这段代码中定义了多种语法规则，并通过递归函数来匹配输入的令牌流，从而构建语法树。

1. **定义语法规则**

通过定义一系列函数（每个函数代表一个语法规则），这些函数的名称与语法规则中的非终结符对应。例如，`p_CompUnit` 处理编译单元（`CompUnit`）的语法规则：

```
def p_CompUnit(p):
    '''CompUnit : Decls FuncDefs MainFuncDef
                | Decls MainFuncDef
                | FuncDefs MainFuncDef
                | MainFuncDef'''
    if len(p) == 4:
        p[0] = ASTNode('CompUnit', [p[1], p[2], p[3]])
    elif len(p) == 3:
        p[0] = ASTNode('CompUnit', [p[1], p[2]])
    else:
        p[0] = ASTNode('CompUnit', [p[1]])
```

这里的 `CompUnit` 是语言中的一个非终结符，它可以由声明（`Decls`）、函数定义（`FuncDefs`）和主函数定义（`MainFuncDef`）组成。根据语法规则的不同形式，函数会将其子节点构建成一个 `ASTNode`。

2. **实现语法分析器**

在语法分析器中，每个语法规则函数都会使用 `p`（参数传递）来表示当前规则匹配到的输入。`p[0]` 是当前规则匹配的结果，它通常被赋值为一个 `ASTNode`，代表这一语法结构在抽象语法树中的一个节点。

例如，`p_Decl` 规则用于处理变量声明：

```
def p_Decl(p):
    '''Decl : ConstDecl
            | VarDecl'''
    p[0] = ASTNode('Decl', [p[1]])
```

`p[0]` 被赋值为一个 `ASTNode`，该节点包含一个子节点，表示当前语法结构（声明）对应的具体类型（`ConstDecl` 或 `VarDecl`）。

3. **构建抽象语法树（AST）**

语法分析器将通过递归调用语法规则函数，逐步构建整个抽象语法树（AST）。每个 AST 节点代表源代码中某一语法结构的一个实例，并通过子节点表示语法结构中的嵌套关系。例如，`if` 语句的 AST 节点结构如下：

```
def p_if_stmt(p):
    '''IfStmt : IF LPAREN Cond RPAREN Stmt'''
    p[0] = ASTNode('IfStmt', [p[3], p[5]])
```

在这种情况下，`IfStmt` 节点有两个子节点，分别是条件表达式（`Cond`）和执行语句（`Stmt`）。

其中ASTNode的定义如下：

```
class ASTNode:
    """
    parent_node: 父节点
    is_terminal: 是否是终结符
    node_type: is_terminal == false时才有意义，表示当前非终结符类型
    child_nodes: is_terminal == false时才有意义，表示子节点
    word_type: is_terminal == true时才有意义，表示终结符类型（即lexer.py中的tokens）
    word_value: is_terminal == true时才有意义，表示终结符对应的值（关键字、数字、字符串）
    """
    def __init__(self, type, children=None, value=None):
        self.parent_node = None
        self.is_terminal = (children == None)
        if isinstance(children, ASTNode):
            self.child_nodes = [children]
        elif isinstance(children, list):
            self.child_nodes = children
        else:
            self.child_nodes = []
        self.node_type = type
        self.word_type = type if self.is_terminal else None
        self.word_value = value
```

每个节点记录其父节点和子节点，对于终结符节点，还会记录其相应的值，包括关键字、数字、字符串等。除此之外，由于ply语法解析过程没有记录ASTNode节点之间的父子关系。为此，实现了基于DFS构建节点间父子关系的遍历函数：

```
def build(self):
    """
    遍历当前节点及其所有子节点，为所有节点构建父子关系
    """
```

```
    def visitor(node, parent=None):
        if not node:
            return

        node.parent_node = parent

        if not node.is_terminal and node.child_nodes:
            for child in node.child_nodes:
                if isinstance(child, ASTNode):
                    visitor(child, node)
                elif isinstance(child, list):
                    for c in child:
                        visitor(c, node)

    visitor(self)
```

在构建完AST后调用build函数即可构建节点之间的父子关系，便于后续代码生成使用。

4. **生成语法分析树**

通过调用 `yacc.parse()` 方法，可以将通过词法分析器生成的令牌序列传递给语法分析器，构建出语法分析树（AST）。这段代码的主要任务就是根据输入的令牌流生成抽象语法树，并在 AST 中反映出程序的结构。

```
parser = yacc.yacc(debug=True, debuglog=log)
```

这行代码使用 `PLY` 的 `yacc` 来创建语法分析器，并且可以通过调试日志来跟踪解析过程中的各个步骤。

4.3 代码生成

为了方便地实现线程操作，我们选择将代码翻译为go语言，综合考虑如下：

- go的线程操作相对简洁，同步和互斥较容易实现；
- go的线程执行效率高，python由于有全局解释器锁的限制同一时间只能解释一个线程的代码，想要高效地实现并行只能使用多进程，但进程的创建开销过大；
- go是强类型语言，与P-lang十分契合，相比弱类型语言在运行效率上更有优势。

4.3.1 整体逻辑

代码生成即根据语法分析构建的AST来生成目标代码。生成时，需要为每种节点编写对应的代码生成子程序，然后从AST的根节点开始，自顶向下遍历每个节点。下面是常量声明语句 `ConstDecl` 代码生成子程序的一个例子：

```
def g_ConstDecl(self, node: ASTNode):
    '''ConstDecl : CONST BType ConstDefList SEMICOLON'''
    children = node.child_nodes
    assert equals_T(children[0], 'BType')
    btype = children[0]
    assert equals_NT(children[1], 'ConstDefList')
    self.g_ConstDefList(children[1], btype)
```

该子程序根据文法的结构进行处理，对每个非终结符依次进入其代码生成子程序，注意到该子程序也包含了错误检测，以保证AST的结构正确，这对于调试很有帮助。由于代码生成较为繁琐，下面仅提供了几类非并行语法的处理，并行语法的代码生成将在下面三章叙述。

- 对于表达式语句，P-lang与go的语法几乎一致，因此只需简单根据AST的结构生成代码即可；
- 对于非数组的常量和变量的声明和赋初值，P-lang与go的参数位置不一样，在生成代码时需要调整生成顺序和添加必要的关键字；
- 对于数组的赋初值，go要求在初值列表前提供参数类型，因此在编写代码生成子程序时，需要将数组的类型传入初值列表的生成子程序；
- 对于输入输出语句，我们延用了C语言中的`printf`和`scanf`，这在go中也有对应的实现，方便起见，`scanf`语句中的`&`由编译器辅助添加，使得代码更加简洁；
- P-lang的语句分隔符只支持`;`，在生成时，考虑到代码的可读性，我们将`;`转换为go的换行符；
- P-lang与go在括号的使用上也略有差别，P-lang的`if`和`for`语句（不包括`for-in`）要求后面的表达式带括号，而go不需要，当`if`和`for`的语句块中只有一条语句时`{}`可以省略，但go必须有`{}`，对于这些细节，我们都进行了判断以生成正确的代码。

4.3.2 线程实现

对于`parallel`语句块的实现，我们利用了go中的`goroutines`轻量级线程。将`parallel`块中的语句封装成函数，然后循环使用`go`语句创建多个线程，如下面的`parallel`语句：

```
int index[3] = {1, 2, 3};
parallel (int x) in index {
    <语句>
}
```

将被翻译为如下的go语言代码：

```
var index [3]int = [3]int{1, 2, 3}
for _i := 0; _i < len(index); _i++ {
    go parallel_1(index[_i])
}

func parallel_1 (x int) {
    <语句>
}
```

4.3.3 管道实现

管道类型`pipe`使用了go中的`Channel`数据类型，在使用`Channel`类型前必须对其初始化。如下面创建`pipe`类型数组的语句：

```
pipe bool ret[3];
```

将被翻译为：

```
var ret [3]chan bool
for _i := 0; _i < 3; _i++ { ret[_i] = make(chan bool) }
```

在循环中我们使用`make(chan <type>)`来初始化每一个管道类型变量。

4.3.4 互斥实现

互斥语句块的实现使用了go中`sync.Mutex`库提供的互斥操作，互斥语句块中的互斥语句块名将被翻译为一个同名的互斥锁。如下面的互斥语句块：

```
mutex m1 {
    <互斥语句>
}
```

将被翻译为：

```
var m1 sync.Mutex

m1.Lock()
<互斥语句>
m1.Unlock()
```

五、验证与测试

为验证P-lang的词法语法分析和代码生成的正确性，我们进行了一系列的测试。

5.1 基础功能

该测试旨在验证P-lang除并行以外功能的正确性，我们的测试用例覆盖了尽可能多的情况。输入以下命令即可运行基础功能测试：

```
$ python main.py full
```

测试代码如下：

```
// 变量声明和赋值
int a = 4;
int b = 5;
float d = 3.14;
bool x = false;
bool c[3] = {true, true, false};
str name = "自带并行の语言！";

// 函数定义与调用
/* 注释的测试 */
def int sum(int x, int y) {
    return x + y;
}

void main() {
    // 字符串输出
    printf("这是一个%s \n", name);

    // 控制流：条件判断
    if (a > 5) {
        printf("a is greater than 5");
    } else {
        if (a == 5) {
            printf("a is equal to 5");
        } else {
            printf("a is less than 5");
        }
    }

```

```
    // 循环：for 循环
    int i = 0;
    printf("\n测试第一种for循环: for ;;;\n");
    for (i = 0; i < 10; i = i + 1) {
        printf("i:%d, ", i);
    }


    printf("\n测试第二种for循环: for x in\n");
    bool boo;
    for boo in c {
        printf("boo:%t, ", boo);
    }

    printf("\n");
    int result = sum(5, 8);
    printf("The sum is: %d\n", result);

    // 数组和访问
    float arr[2][3] = {{1.0, 2.5, 3.6}, {4.6, 5.7, 6.8}};
    printf("The first element of arr is: %f\n", arr[1][1]);

    // 数组遍历
    int k = 0;
    int j = 0;
    for (k = 0; k < 2; k = k + 1) {
        for (j = 0; j < 3; j = j + 1) {
            printf("arr[%d][%d] = %f, ", k, j, arr[k][j]);
        }
        printf("\n");
    }

    printf("testing if for\n");
    a = 10;
    // 嵌套控制流
    if (a > 5) {
        for (i = 0; i < 3; i = i + 1) {
            printf("Nested loop, i = %d, ", i);
        }
        printf("\n");
    } else {
        printf("Outer condition failed.\n");
    }

    // 运算符使用
    int sum2 = a + b;
    int product = a * b;
    printf("Sum: %d, Product: %d", sum2, product);
}
```

运行结果如下：

```
这是一个自带并行的语言!
a is less than 5
测试第一种for循环: for ;;;
i:0, i:1, i:2, i:3, i:4, i:5, i:6, i:7, i:8, i:9,
测试第二种for循环: for x in
boo:true, boo:true, boo:false,
The sum is: 13
The first element of arr is: 5.700000
arr[0][0] = 1.000000, arr[0][1] = 2.500000, arr[0][2] = 3.600000,
arr[1][0] = 4.600000, arr[1][1] = 5.700000, arr[1][2] = 6.800000,
testing if for
Nested loop, i = 0, Nested loop, i = 1, Nested loop, i = 2,
Sum: 15, Product: 50
```

5.2 并行功能

5.2.1 通信、同步、互斥

该测试旨在验证P-lang并行功能的正确性，包括对并行语句块、线程同步、线程通信、互斥访问的测试。输入以下命令即可运行并行功能测试：

```
$ python main.py parallel
```

测试代码如下：

```
int value1 = 0;
int value2 = 0;

pipe int   p12;     // 线程1向线程2发送int
pipe float p23;     // 线程2向线程3发送float
pipe bool  p31;     // 线程3向线程1发送bool

void main() {
    int index[3] = {1, 2, 3};   // 线程编号
    pipe bool ret[3];           // 用于返回数据，同时阻塞主线程
    int i;

    /************** 演示线程同步与通信 **************/
    parallel (int x, pipe bool r) in index, ret {
        if (x == 1) {
            int send = 123;
            bool receive;
            p31 >> receive;
            printf("线程1接收bool: %v\n", receive);
            p12 << send;
            printf("线程1发送int: %d\n", send);
        } else {
            if (x == 2) {
                float send = 3.14;
                int receive;
                p23 << send;
                printf("线程2发送float: %f\n", send);
                p12 >> receive;
                printf("线程2接收int: %d\n", receive);
            } else {
                bool send = true;
```

```
                float receive;
                p31 << send;
                printf("线程3发送bool: %v\n", send);
                p23 >> receive;
                printf("线程3接收float: %f\n", receive);
            }
        }
        r << true;
    }
    for i in index {
        ret[i - 1] >>;      // 主线程阻塞，等待子线程结束
    }

    /*************** 演示共享变量互斥访问 ***************/
    parallel (pipe bool r) in ret {
        int i;
        for (i = 0; i < 10000; i = i + 1) {
            value1 = value1 + 1;
            mutex m1 {      // 互斥访问语句块
                value2 = value2 + 1;
            }
        }
        r << true;
    }
    for i in index {
        ret[i - 1] >>;      // 主线程阻塞，等待子线程结束
    }
    printf("value1: %d\n", value1);     // value1访问没有互斥，因此小于30000
    printf("value2: %d\n", value2);     // value2等于30000
}
```

运行结果如下：

```
线程1接收bool: true
线程3发送bool: true
线程3接收float: 3.140000
线程2发送float: 3.140000
线程2接收int: 123
线程1发送int: 123
value1: 29866
value2: 30000
```

上述测试代码通过三个子线程互相发送数据验证了使用`pipe`实现通信的正确性；主线程阻塞等待子线程验证了同步的正确性；对value2的互斥访问使得其求和的结果正好为30000，而在互斥语句块外求和的value1结果小于30000，说明了互斥语句块功能的正确性。

5.2.2 嵌套并行

另外我们的`parallel`语句块还支持嵌套使用，这更加说明了P-lang的灵活性和鲁棒性。输入以下命令即可运行嵌套功能测试：

```
$ python main.py nest
```

测试代码如下：

```
void main() {
    int outer_index[3] = {1, 2, 3};   // 外层线程编号
```

```
    pipe bool ret[3];

    parallel (int x, pipe bool r) in outer_index, ret {
        int outer_index[3] = {x, x, x};      // 记录外层编号
        int inner_index[3] = {1, 2, 3};      // 内层线程编号
        pipe bool rett[3];

        parallel (int x, int y, pipe bool r) in outer_index, inner_index, rett {
            printf("我是子线程%d的子线程%d\n", x, y);
            r << true;
        }
        int i;
        for i in inner_index {
            rett[i - 1] >>;
        }
        r << true;
    }
    int i;
    for i in outer_index {
        ret[i - 1] >>;
    }
}
```

运行结果如下：

```
我是子线程3的子线程3
我是子线程1的子线程3
我是子线程1的子线程1
我是子线程1的子线程2
我是子线程2的子线程3
我是子线程2的子线程1
我是子线程2的子线程2
我是子线程3的子线程1
我是子线程3的子线程2
```

5.3 归并排序

我们用P-lang编写了并行的归并排序代码，排序时的二分操作将分为两个线程分别执行。输入以下命令即可运行归并排序测试：

```
$ python main.py msort
```

测试代码如下：

```
int arr_size = 8;
int arr[8] = {5, 11, 9, 4, 12, 6, 7, 1};
int temp[8];

// 归并排序函数
def void msort(int s, int t)
{
    if (s == t)
        return;
    int mid = (s + t) / 2;
    int begin[2] = {s, mid + 1};
    int end[2] = {mid, t};
```

```
    pipe bool ret[2];

    // 分两个线程对二分的数组排序
    parallel (int x, int y, pipe bool r) in begin, end, ret {
        msort(x, y);
        r << true;
    }

    ret[0] >>; ret[1] >>;
    int i = s, j = mid + 1, k = s;
    for (i = s; i <= mid && j <= t; )
    {
        if (arr[i] <= arr[j])
        {
            temp[k] = arr[i];
            k = k + 1;
            i = i + 1;
        }
        else
        {
            temp[k] = arr[j];
            k = k + 1;
            j = j + 1;
        }
    }
    for (; i <= mid; i = i + 1)
    {
        temp[k] = arr[i];
        k = k + 1;
    }
    for (; j <= t; j = j + 1)
    {
        temp[k] = arr[j];
        k = k + 1;
    }
    for (i = s; i <= t; i = i + 1)
        arr[i] = temp[i];
}

// 打印数组
def void printArray(int arr[8], int size)
{
    int i;
    for (i = 0; i < size; i = i + 1)
        printf("%d ", arr[i]);
    printf("\n");
}

void main()
{
    printf("排序前的数组: \n");
    printArray(arr, arr_size);

    msort(0, arr_size - 1); // 调用排序函数

    printf("排序后的数组: \n");
```

```
    printArray(arr, arr_size);
}
```

运行结果如下：

```
排序前的数组：
5 11 9 4 12 6 7 1
排序后的数组：
1 4 5 6 7 9 11 12
```

5.4 GEMV

这一小节主要介绍GEMV测试的验证和实现方式。

首先我们实现了一个python脚本，通过 `numpy` 初始化了一个 `5*10` 的矩阵A和一个 `10*1` 的向量x，并调用 `numpy` 的矩阵向量乘方法获取 `A*x` 的正确结果。

```
# 构造一个5 * 10的随机浮点矩阵
matrix_5x10 = np.random.rand(5, 10).astype(float)
# 构造一个1 * 10的随机浮点向量
vector_1x10 = np.random.rand(1, 10).astype(float)

# 进行矩阵向量乘法
result_vector = np.dot(rounded_matrix_5x10, rounded_vector_1x10.reshape(-1,
1)).flatten()
```

同时，将矩阵和向量的值输入到文本文件里供后续编译运行使用

```
with open('gemv.txt', 'w') as f:
    # 先写入5 * 10的矩阵数据
    for row in rounded_matrix_5x10:
        row_str = " ".join(map(str, row))
        f.write(row_str + ' ')
    # 再写入1 * 10的向量数据
    vector_str = " ".join(map(str, rounded_vector_1x10))
    f.write(vector_str + "\n")
    # 写入矩阵向量乘法结果
    result_str = " ".join(map(str, rounded_result_vector))
    f.write(result_str + "\n")
```

在测试代码方面，主要逻辑为，初始化 `5*10` 矩阵A以及 `10*1` 向量x，然后通过scanf语句输入数据：

```
float A[5][10];   // 5x10矩阵
float x[10];                          // 10x1向量
float y[5] = {0.0, 0.0, 0.0, 0.0, 0.0};     // 5x1结果向量
for (i = 0; i < 5; i = i + 1){
    for (j = 0; j < 10; j = j + 1) {
        scanf("%f", A[i][j]);
    }
}
for (i = 0; i < 10; i = i + 1) {
    scanf("%f", x[i]);
}
```

通过parallel语句进行并行化地计算：

```
parallel (int i, float row[10], pipe bool r) in index, A, ret {
    int j;
    for (j = 0; j < 10; j = j + 1) {
        y[i] = y[i] + row[j] * x[j];
    }
    r << true;
}

for i in index {
    ret[i] >>;      // 主线程阻塞，等待子线程结束
}
```

最终将结果输出：

```
for (i = 0; i < 5; i = i + 1) {
    printf("y[%d] = %f, ", i, y[i]);
}
```

值得一提的是，测评过程中用subprocess库实现了测试过程全自动化执行。

```
# 在gemv测试模式下，使用subprocess模块运行python get_input.py命令，并等待其执行完成
try:
    process = subprocess.Popen(["python", "get_input.py"], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    stdout, stderr = process.communicate()
    if process.returncode!= 0:
        print(f"运行python get_input.py出现错误，错误信息如下：\n{stderr.decode('utf-8')}")
        return
except FileNotFoundError:
    print("python get_input.py文件不存在，请确保该文件已存在")
    return

# 尝试从gemv.txt中读取第一行作为后续go命令的输入
try:
    with open('gemv.txt', 'r', encoding='utf-8') as f:
        first_line = f.readline().strip()
        second_line = f.readline().strip()
except FileNotFoundError:
    print("gemv.txt文件不存在，请确保该文件已生成")
    return

# 从gemv.txt读取的第一行作为输入传入go run code_output.go命令
process = subprocess.Popen(
    ['go', 'run', 'code_output.go'],  # Go 命令和文件
    stdin=subprocess.PIPE,  # 启用标准输入管道
    stdout=subprocess.PIPE,  # 获取标准输出
    stderr=subprocess.PIPE,  # 获取标准错误输出
)

# 将 first_line 变量传递给 Go 程序
stdout, stderr = process.communicate(input=first_line.encode())  # 传递的输入需要编码成字节

# 打印 Go 程序的输出
print(stdout.decode())  # 获取并打印 Go 程序的输出
```

```
# 然后输出第二行表示正确结果
print("GEMV正确结果为: " + second_line)
```

5.5 性能测试

基于5.3节的归并排序代码，我们对并行语句块的性能进行了测试，即比较是否使用并行语句块对运行时间的影响。详细的代码见 `/testcase/perf.p`，其中不使用并行语句块的核心代码如下：

```
def void msort_normal(int s, int t)
{
    ...
    msort_normal(s, mid);
    msort_normal(mid + 1, t);
    ...
}
```

使用并行语句块的核心代码如下：

```
def void msort_parallel(int s, int t)
{
    ...
    if (t - s > 10000) {
        int begin[2] = {s, mid + 1};
        int end[2] = {mid, t};
        pipe bool ret[2];
        parallel (int x, int y, pipe bool r) in begin, end, ret {
            msort_parallel(x, y);
            r << true;
        }
        ret[0] >>; ret[1] >>;
    } else {
        msort_normal(s, mid);
        msort_normal(mid + 1, t);
    }
    ...
}
```

上述代码对数据量进行了判断，当排序数组的长度小于等于 `10000` 时使用普通排序，这是为了避免频繁创建线程带来过大开销。我们的测试数据量为 `1000000`，使用python生成随机数据然后输入给程序，运行结果如下：

```
normal msort:
81.7834ms
parallel msort:
20.0138ms
```

从上述运行时间可以得出，使用并行语句块的归并排序比普通排序的运行时间减少了约 `75%`，从而验证了并行语句块的高效性。