

一种含共享变量维持声明的并行程序语言

汪 晨^{1,2}, 张 昱^{1,2}, 付小鹏^{1,2}, 张 伟^{1,2}

¹ (中国科学技术大学计算机科学与技术学院, 安徽 合肥 230027)

² (中国科学技术大学苏州研究院软件安全实验室, 江苏 苏州 215123)

E-mail : yuzhang@ustc.edu.cn

摘 要: 现今的并行编程实践多采用锁来同步对共享资源的访问, 编程难且易出错; 新引入的原子区构造虽简化了编程, 但支持其实现的软硬件技术尚不令人满意。本文就同步提出一种新的语言级抽象——共享变量维持声明, 它允许程序员从局部于线程的观点声明当前线程对某共享变量 s 访问的维持需求, 即声明当前线程在运行时从上次访问 s 到这次访问 s 期间不允许其他线程访问 s 。从而, 程序员无须考虑该如何使用锁等具体机制来同步对共享变量的访问, 也可以避免或解决原子区所面临的一些问题。本文给出了共享变量维持声明的语法和语义描述, 讨论了由这种声明信息生成共享变量访问控制代码的方法。

关键词: 并行程序语言; 同步; 共享变量维持; 访问控制

中图分类号: TP

文献标识码: A

文章编号: 1000-1220 (2010) 02--

A Parallel Programming Language With Shared Variable Holding Declaration

Wang Chen^{1,2}, Zhang Yu^{1,2}, Fu Xiao-peng^{1,2}, Zhang Wei^{1,2}

¹ (School of Computer Science & Technology, University of Science & Technology of China, Hefei 230027, China)

² (Software Security Lab., Suzhou Institute for Advanced Study, University of Science & Technology of China, Suzhou, 215123)

Abstract: Nowadays the way to synchronously access the shared resources with lock in parallel programming is difficult to write and error-prone. The newly language construct of atomic section promises to make parallel programming easier, but the software and hardware technology to support atomic sections is not satisfied. This paper presents a new language abstract with synchronization mechanisms, i.e. holding declaration of shared variables. It allows user to declare the local holding requirement of some shared variable s in current thread, e.g. declaring that it is forbidden to access the shared variable s from other threads since the latest access to s in current thread. Therefore, programmers need not consider how to access shared variables with concrete synchronization mechanisms, such as using lock and so on, and can avoid or solve some problems confronted by atomic sections. In the paper, the syntax and the semantic description of the shared variable holding declaration are presented, and methods for generating the shared variable access control code from such declaration information are discussed.

Key words: parallel programming language; synchronization; shared memory holding; access control

1 引言

现今流行的并行编程实践多采用锁、信号量等较低级的同步机制来控制对共享资源的访问。采用这种机制的编程比较复杂, 容易出错, 并且由程序员对共享变量访问控制直接编程会使代码发挥底层并行结构优势的可伸缩性 (scalability) 不好。例如, 当使用锁进行同步控制时, 程序员需要决定用什么锁来保护哪个 (哪些) 共享变量, 并确定在程序中的何处安插获得锁和释放锁的代码; 能否有效控制

程序对共享变量的访问以及程序的并发度完全由程序员的编程决定。

正在研究的高产能并行编程语言 IBM 的 X10^[1,2]、Cray 的 Chapel^[3]和 Sun 的 Fortress^[4]都引入原子区这种语言构造, 原子区的一次执行称为一个事务。事务具有其代码要么全都执行、要么全不执行的原子性, 以及执行中不受其他事务干扰的隔离性。原子区的引入确实简化了对共享资源访问控制的编程^[5], 但是目前支持原子区的软硬件技术尚不令人满意。支持原子区的一种方式是使用事务内存技术^[6,7], 但是事务内存系统的运行开销普遍较高, 此外还有很多问题, 如系统调用和 I/O 操作可否出现在事务中、事务可否嵌套等,

尚在讨论和解决中。支持原子区的另一种方式是由编译器将原子区转换成基于锁的代码^[8,9],编译器需要通过全程序分析来识别共享资源并为之分配锁,确定在程序中加锁和解锁的位置。

本文就同步提出一种新的语言级抽象——共享变量维持声明,它允许程序员从局部于线程的观点使用 `last` 修饰共享变量 s ,来声明当前线程对 s 访问的维持需求,即声明当前线程在运行时从上次访问 s 到这次访问 s 期间不允许其他线程访问 s 。在所设计的并行小语言 SPC (Shared variable holding declarations based Parallel C-style language) 中,还引入一些具有原子执行性质的原子代码段(称为原子命令)。

根据原子命令的原子性以及 SPC 程序中的共享变量维持声明信息,编译器就可以通过静态分析获得线程对共享变量的维持信息:线程中每条语句执行前需要获得维持的共享变量集合及执行后需要释放维持的共享变量集合。依据维持信息不仅可以给出语言的操作语义并能进一步为 SPC 程序生成采用某种具体同步控制机制的共享变量访问控制代码。

图 1 给出了分别使用锁、原子区及共享变量维持声明编写的并行计算 Fibonacci 数程序代码片段。程序中的 t 为局部变量, pre 和 cur 为共享变量。并行执行的各个线程将循环地执行图 1 中的某一列代码。

```

lock lcur;
t=cur;      atomic{
lock lpre;      t=cur;      t=cur;
cur=cur+pre;   cur=cur+pre;   cur=last cur+pre;
unlock lcur;   pre=t;      last pre=t;
pre=t;      }
unlock lpre;
(a)          (b)          (c)

```

图 1 并行计算 Fibonacci 数的代码片段

Fig. 1 Code segment computing Fibonacci numbers in parallel

图 1(a)为使用锁的方式,程序员分别为 pre 和 cur 分配锁 $lpre$ 和 $lcur$,且在访问它们之前加锁,访问之后解锁。图 1(b)为使用原子区方式,程序员只需将共享变量访问代码置于 `atomic` 块中。对程序员来说,编程简单了,但编译器负担却加重了,因为对共享变量没有任何提示,且原子区中共享变量访问控制的粒度受到限制。图 1(c)为 SPC 编写的程序,一个赋值语句是一个原子执行的原子命令, `last` 要求后续的那个变量从上次访问所在原子命令之前的程序点到本次访问所在原子命令执行后的程序点期间不允许被其他线程访问,这允许程序员以不依赖于具体同步控制机制的形式来描述对共享变量的访问需求。与图 1(b)的原子区方式相比,使用 SPC 可以给出共享变量访问约束的准确信息,如本例中,在对 cur 赋值结束后,变量 cur 就可以被其他线程访问,这使得编译器有可能利用这些信息产生较优的代码。

和已有的共享资源访问的同步结构/应用编程接口相比,本文所提出的共享变量维持声明有以下特色:

(1) 程序员只需声明对共享变量的维持需求,而不需要采用具体的同步技术来对共享变量的访问控制直接编程。

(2) 利用程序员给出的共享变量维持声明信息,编译器能较容易地获得对共享变量的维持信息。

(3) 利用共享变量的维持信息,编译器可以结合底层结

构特征,采用不同策略灵活生成共享变量访问控制代码。

本文第 2 节介绍 SPC 的语法。第 3 节首先简要介绍 SPC 的控制流;然后重点介绍 SPC 的维持分析。第 4 节给出 SPC 操作语义。第 5 节介绍为 SPC 生成访问控制代码。第 6 节为结论和进一步工作。

2 SPC 语法

图 2 给出了 SPC 语言的抽象语法。如图所示, x 为变量(本文仅考虑整数类型), s 为共享变量, n 为常数, E 为算术表达式, B 为布尔表达式,用作 `if` 语句和 `while` 语句的条件, op_a 为算术操作符(如“+”), op_b 为布尔操作符(如“!”), op_r 为关系操作符(如“==”)。语法中的 \tilde{x} 表示实际编程中的 `last x`, \tilde{x} 表示 x 或 \tilde{x} 。关键字 `last` 用于修饰共享变量以声明该变量从上次访问所在原子命令之前的程序点到本次访问所在原子命令执行后的程序点期间禁止被其他线程访问。

$$\begin{aligned}
 (Stmt) \quad S \in \mathcal{S}_{stmt} &::= [\text{skip}]^l \mid [\tilde{x} = E]^l \\
 &\quad \mid [(\tilde{x}_1, \dots, \tilde{x}_m)]^l \\
 &\quad \mid S_1; S_2 \mid \text{if } ([B]^l) S_1 \text{ else } S_2 \\
 &\quad \mid \text{while } ([B]^l) S \mid S_1 \parallel \dots \parallel S_n \\
 (Atom) \quad A \in \mathcal{A} &::= [\text{skip}]^l \mid [\tilde{x} = E]^l \mid [B]^l \\
 &\quad \mid [(\tilde{x}_1, \dots, \tilde{x}_m)]^l \\
 (AExp) \quad E \in \mathcal{E} &::= x \mid \tilde{x} \mid n \mid (E) \mid -E \\
 &\quad \mid E_1 \text{ op}_a E_2 \\
 (BExp) \quad B \in \mathcal{B} &::= (B) \mid \neg B \mid E_1 \text{ op}_r E_2 \\
 &\quad \mid B_1 \text{ op}_b B_2 \\
 (Lab) \quad l \in \mathcal{L} &::= l_1 \mid l_2 \mid \dots \mid l_n \mid \dots \\
 (Var) \quad x \in \mathcal{V} & \\
 (SVar) \quad s \in \mathcal{V}_s \subseteq \mathcal{V} & \\
 (Const) \quad n \in \mathcal{C} &
 \end{aligned}$$

图 2 SPC 抽象语法

Fig. 2 Abstract syntax of the SPC language

S 为语句,它包含有:空语句 `[skip]l`,它只在抽象语法中存在; `[($\tilde{x}_1, \dots, \tilde{x}_m$)]l` 语句(实际编程时写成 `skip last $x_1, \dots, \text{last } x_m$`),它不执行任何操作而只是用于表达共享变量的维持声明; $S_1 \parallel \dots \parallel S_n$ 为并行语句,并行语句采用 fork-join 形式;条件语句统一为 `if([B]l) S1 else S2`,若 `else` 分支没有语句,用 `[skip]l` 替代; `while([B]l) S` 为循环语句; `[$\tilde{x} = E$]l` 为赋值语句。 $S_1; S_2$ 为语句的连接。

A 为原子命令。在 SPC 中,原子命令为执行的基本单位,在将要讨论的流图上,表达一个结点。原子命令有三类,用 $A_{tag} = \{\perp, C, L\}$ 表示,其中 C 表示 `if` 语句的条件表达式, L 表示 `while` 语句的条件表达式,其他的用 \perp 表示。

l 为原子命令的标签,是为便于描述程序分析而引入的。

语句 `[($\tilde{x}_1, \dots, \tilde{x}_m$)]l` 用于解决当共享变量 s 要维持到某程序点结束,但该程序点之后的原子命令没有对 s 的访问情况,此时,在该程序点处增加 `[(\tilde{s})]l` 语句即可。

线程在执行时对共享变量的维持体现在以下两种情况。一是线程在执行原子命令时,原子命令中的所有共享变量都需要被维持。二是通过 `last` 修饰符声明的共享变量在原子命令之间的维持需求。

3 共享变量的维持分析

共享变量的维持分析通过分析 SPC 程序中的共享变量

信息(每条原子命令所访问的共享变量及包含 **last** 修饰的共享变量)来获得线程对共享变量的维持信息: 每条原子命令执行前需要获得维持的共享变量集合和执行后需要释放维持的共享变量集合。

维持分析通过两步来获得每条原子命令执行前需要获得维持的共享变量集合和执行后需要释放维持的共享变量集合: 首先通过逆向数据流分析收集由程序中的 **last** 修饰而建立的共享变量在原子命令之间的维持信息, 该过程简称 **last** 维持分析; 然后结合原子命令本身对共享变量的维持要求计算出每条原子命令执行前需要获得维持和执行后需要释放维持的共享变量集合。

本节先给出 SPC 程序的流图, 然后介绍 **last** 维持分析, 最后计算每条原子命令执行前需要获得维持和执行后需要释放维持的共享变量集合信息。

3.1 流图

为了进行维持分析, 为 SPC 程序中的每个线程分别构造独立的流图。流图中每个结点是一个原子命令, 以原子命令的标签为结点编号, 结点之间的控制流关系表示成边, 且边有三类, 用 $E_{tag} = \{\perp, T, F\}$ 表示, 其中 **T** 和 **F** 分别为分支的 true 和 false 边, \perp 为其他边。下面给出文中使用到的控制流函数及其功能, 设计细节见技术报告^[10]。

$$\begin{aligned} \text{init} &: S_{tmt} \rightarrow \mathbb{L} \\ \text{final} &: S_{tmt} \rightarrow \mathcal{P}(\mathbb{L}) \\ \text{atoms} &: S_{tmt} \rightarrow \mathcal{P}(A \times A_{tag}) \\ \text{flow} &: S_{tmt} \rightarrow \mathcal{P}(\mathbb{L} \times \mathbb{L} \times E_{tag}) \end{aligned}$$

函数 **init** 返回语句中最先执行的原子命令标签, **final** 返回语句中最后执行的原子命令的标签集合。atoms 返回语句对应的原子命令及原子命令的类型, 它用来描述语句与原子命令之间的关系, 因为控制流中的边使用两个原子命令的标签和边的类别来表示。flow 返回语句的控制流, 控制流体现为所有原子命令之间的关系。

3.2 原子命令上的 last 维持分析

3.2.1 last 维持

辅助函数 **svars** 和 **islasted** 用于描述共享变量与原子命令之间的关系:

$$\begin{aligned} \text{svars} &: \mathbb{L} \rightarrow \mathcal{P}(V_s) \\ \text{islasted} &: V_s \times \mathbb{L} \rightarrow \text{Bool} \end{aligned}$$

函数 **svars** 返回标签为 l 的原子命令 $[A]^l$ 中访问的所有共享变量集合, **islasted** 返回标签为 l 的原子命令 $[A]^l$ 中是否有对共享变量的 $\overline{\text{last}}$ 访问, 如有, 返回 true, 否则返回 false。Bool 表示布尔类型。

定义 1 (原子命令序列) 线程 T 执行的代码 S 中有原子命令 $[A_1]^{l_1}, \dots, [A_n]^{l_n}$ 且 $\forall 1 \leq i \leq n-1, (l_i, l_{i+1}) \in \text{flow}(S)$, 则称 $[A_1]^{l_1}, \dots, [A_n]^{l_n}$ 为连续执行的原子命令序列。

下面给出 **last** 维持的定义, 并依据 **last** 维持的定义收集原子命令的入口及出口上的共享变量的 **last** 维持集合。

定义 2 (last 维持) 如果线程 T 执行的代码 S 中存在原子命令序列 $[A_1]^{l_1}, \dots, [A_n]^{l_n}$, 且对共享变量 s 有 $\text{islasted}(s, l_n) \wedge s \in \text{svars}(l_1) \wedge \neg \text{islasted}(s, l_1) \wedge (\forall 2 \leq i \leq n-1. s \notin \text{svars}(l_i) \vee \text{islasted}(s, l_i))$

则将 s 在 A_1 的出口至 A_n 的入口之间的维持称为 **last** 维持。

3.2.2 last 维持分析

由定义 2 可知, 当共享变量 s 在原子命令序列 $[A_1]^{l_1}, \dots, [A_n]^{l_n}$ 上被 **last** 维持时, 则在 A_1 的出口、 $A_i (2 \leq i \leq n-1)$ 的入口和出口、以及 A_n 的入口均被 **last** 维持。依据定义 2, 图 3 给出了逆向收集每一原子命令入口和出口的共享变量的 **last** 维持集合信息的数据流方程。函数

$$\text{kill}_{LH} : A \rightarrow \mathcal{P}(V_s)$$

产生原子命令不再需要 **last** 维持的共享变量 s 的集合, 即 **kill** 的结果为原子命令中所访问的不含 **last** 修饰的共享变量集合。函数

$$\text{gen}_{LH} : A \rightarrow \mathcal{P}(V_s)$$

产生原子命令上需要 **last** 维持的共享变量集合。即 **gen** 的结果为原子命令中所访问的含 **last** 修饰的共享变量集合。函数

$$LH_{in}, LH_{out} : \mathbb{L} \rightarrow \mathcal{P}(V_s)$$

分别为原子命令入口和出口需要 **last** 维持的共享变量集合。

依据 **last** 维持的定义可知, 线程最后执行的原子命令的 LH_{out} 为空。对任意在线程中的其他原子命令 A , A 的 LH_{out} 为 A 的直接后继原子命令的 LH_{in} 的并集, A 的 LH_{in} 为 A 的 LH_{out} 去掉在 A 中 **kill** 掉的共享变量, 再并上在 A 上 **gen** 的共享变量。

图 3 方程的输入为每一原子命令的 kill_{LH} 和 gen_{LH} , 通过逆向数据流迭代计算后, 输出为每一原子命令的 LH_{in} 和 LH_{out} 。

$$\begin{array}{c} \text{kill and gen functions} \\ \hline \text{kill}_{LH}([A]^l) = \{s \mid s \in \text{svars}(l) \wedge \neg \text{islasted}(s, l)\} \\ \text{gen}_{LH}([A]^l) = \{s \mid \text{islasted}(s, l)\} \\ \hline \text{data flow equations: LH=} \\ \hline LH_{out}(l) = \begin{cases} \emptyset & \text{if } l \in \text{final}(S), \\ \bigcup \{LH_{in}(l') \mid (l, l', -) \in \text{flow}(S)\} & \text{otherwise} \end{cases} \\ LH_{in}(l) = (LH_{out}(l) \setminus \text{kill}_{LH}(A^l)) \cup \text{gen}_{LH}(A^l) \\ \text{where } A^l \in \text{atoms}(S) \end{array}$$

图 3 计算 **last** 维持的数据流方程

Fig. 3 Data flow equation for computing the **last** holding

3.3 维持信息求解

函数

$$\mathcal{H}_a : \mathbb{L} \rightarrow \mathcal{P}(V_s)$$

$$\mathcal{H}_r : \mathbb{L} \rightarrow \mathcal{P}(V_s \times E_{tag})$$

分别表示原子命令在执行之前需要获得维持的共享变量集合和执行之后需要释放维持的共享变量集合。

对原子命令为 $[B]^l$, 它有两个不同的分支, 在 $[B]^l$ 上被线程维持的共享变量 s , 在 $[B]^l$ 执行后对不同分支可能存在有三种不同释放对 s 的维持需求: 一是两分支都将 s 释放; 二是在 $[B]^l$ 的 true 分支上 s 继续被维持, 而在 false 分支上释放对 s 的维持; 三是在 $[B]^l$ 的 false 分支上 s 继续被维持, 而在 true 分支上释放对 s 的维持。为此, $\mathcal{H}_r(l)$ 以二元组 (s, t) 形式给出标签为 l 的原子命令在类别为 t 的出口处需释放对共享变量 s 的维持, 当为情况一时, (s, T) 和 (s, F) 都在 $\mathcal{H}_r(l)$ 中, 当为情况二时, (s, F) 在 $\mathcal{H}_r(l)$ 中; 当为情况三时, (s, T) 在 $\mathcal{H}_r(l)$

中。当原子命令不为 $[B]^l$ 时,且 s 在 $[B]^l$ 执行之后要释放维持,则有 (s, \perp) 在 $\mathcal{H}_r(l)$ 中。

在收集到原子命令入口和出口的 last 维持信息后,依据 last 维持定义和原子命令本身对共享变量的维持要求,原子命令 $[A]^l$ 在执行之前需要获得维持的共享变量集合 $\mathcal{H}_a(l)$ 为:一是原子命令上的任意没有 last 修饰的共享变量 s (非 \overleftarrow{s}),二是 $\text{LH}_{in}(l)$ 中没有 s 且 $\text{LH}_{out}(l)$ 中有 s 。原子命令 $[A]^l$ 在执行之后需要释放维持的共享变量集合 $\mathcal{H}_r(l)$ 为:一是原子命令上任意没有被 last 修饰的共享变量 s (非 \overleftarrow{s})且在 $\text{LH}_{out}(l)$ 中没有 s ,二是 $\text{LH}_{in}(l)$ 中有 s 且 $\text{LH}_{out}(l)$ 中没有 s ;三是针对分支情况,如 $[A]^l$ 的有两个直接后继分支原子命令 A_t^l 和 A_f^l ,其中 A_t^l 为 true 分支后继, A_f^l 为 false 分支后继,则当 $\text{LH}_{in}(l)$ 和 $\text{LH}_{out}(l)$ 中都有 s , $\text{LH}_{in}(l)$ 中没有 s 且 $\text{LH}_{in}(l')$ 中有 s ,则 (s, F) 在 $\mathcal{H}_r(l)$ 中,当 $\text{LH}_{in}(l)$ 和 $\text{LH}_{out}(l)$ 中都有 s , $\text{LH}_{in}(l')$ 中有 s 且 $\text{LH}_{in}(l')$ 中没有 s ,则 (s, F) 在 $\mathcal{H}_r(l)$ 中。

图 4 方程为在收集到线程的每一原子命令的 LH_{in} 和 LH_{out} 后,计算线程中每一原子命令要获得维持的共享变量集合 \mathcal{H}_a 与要释放维持的共享变量集合 \mathcal{H}_r 。

$$\mathcal{H}_a(l) = \{v | v \notin \text{LH}_{in}(l) \wedge (v \in \text{LH}_{out}(l) \vee v \in \text{svars}(l))\}$$

$$\mathcal{H}_r(l) = \{(v, t) | (v \in \text{LH}_{in}(l) \vee v \in \text{svars}(l)) \wedge ((t = \perp \wedge (l, l', t) \in \text{flow}(S) \wedge v \notin \text{LH}_{out}(l)) \vee (t = T | F \wedge (l, l', t) \in \text{flow}(S) \wedge v \notin \text{LH}_{in}(l'))))\}$$

图 4 计算 \mathcal{H}_a 和 \mathcal{H}_r 方程

Fig. 4 The equation to compute \mathcal{H}_a and \mathcal{H}_r .

3.4 维持分析实例

图 5 给出了计算 Fibonacci 数的线程代码及其流程图:(a)为线程代码,其中 t 为局部变量,其他为共享变量,多个线程并发执行这段代码;(b)为线程的流程图。图 6 为图 5(a)的分析结果。以共享变量 num 为例, num 在标签为 1 和 5 的语句上被访问,且在 5 上有 last 声明,所以它出现在 $\text{kill}_{LH}(1)$ 、 $\text{kill}_{LH}(6)$ 及 $\text{gen}_{LH}(6)$ 中;通过图 3 数据流方程迭代计算后,在 $\text{LH}_{out}(1)$ 、 $\text{LH}_{in}(2)$ 、 $\text{LH}_{out}(2)$ 、 $\text{LH}_{in}(3)$ 、 $\text{LH}_{out}(3)$ 、 $\text{LH}_{in}(4)$ 、 $\text{LH}_{out}(4)$ 及 $\text{LH}_{in}(5)$ 中有 num ,即 num 在标签为 1 的语句出口到标签为 5 的语句入口被 last 维持;然后通过图 4 方程得,标签为 1 的语句要获得对 num 的维持($num \in \mathcal{H}_a(1)$),标签为 5 的语句要释放对 num 的维持($(num, \perp) \in \mathcal{H}_r(5)$),由于在标签为 1 的语句上有另一出口(while 条件为 false 时),且该出口不需要对 num 维持,所以 1 的出口对应 while 条件为 false 时,也要释放对 num 的维持($(num, F) \in \mathcal{H}_r(1)$)。

l	kill_{LH}	gen_{LH}	$\text{LH}_{in}^{(1)}$	$\text{LH}_{out}^{(1)}$	LH_{in}	LH_{out}	\mathcal{H}_a	\mathcal{H}_r
1	$\{num\}$	\emptyset	\emptyset	$\{num\}$	\emptyset	$\{num\}$	$\{num\}$	$\{(num, F)\}$
2	$\{cur\}$	\emptyset	$\{num\}$	$\{num, cur\}$	$\{num\}$	$\{num, cur\}$	$\{cur\}$	\emptyset
3	$\{cur, pre\}$	$\{cur\}$	$\{num, cur\}$	$\{num, pre\}$	$\{num, cur\}$	$\{num, pre\}$	$\{pre\}$	$\{(cur, \perp)\}$
4	\emptyset	$\{pre\}$	$\{num, pre\}$	$\{num\}$	$\{num, pre\}$	$\{num\}$	\emptyset	$\{(pre, \perp)\}$
5	$\{num\}$	$\{num\}$	$\{num\}$	\emptyset	$\{num\}$	\emptyset	\emptyset	$\{(num, \perp)\}$

图 6 Fibonacci 实例分析结果

Fig. 6 The analysis result of the example for the simple Fibonacci program

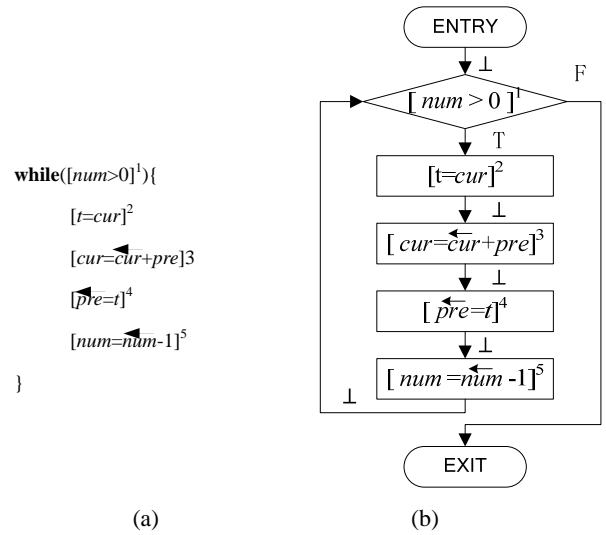


图 5 Fibonacci 代码实例与控制流

Fig. 5 The code and flow graph of a simple Fibonacci program

4 SPC 操作语义

在获得线程中每一原子命令上要获得维持与释放维持的共享变量集合后,就可给出 SPC 的操作语义了。在 SPC 中,线程执行到原子命令时,需保证原子命令上要获得维持的共享变量没有被其他线程维持,执行原子命令时,要具有对这些共享变量的维持,原子命令执行完成后,还需释放那些不再要维持的共享变量。

首先定义运行 SPC 程序的抽象机。如图 7 所示,抽象机状态空间 \mathbb{W} 由存储状态 \mathbb{S} ,维持集合 \mathbb{HD} 及线程集合 \mathbb{TS} 三部分组成。 \mathbb{S} 记录了 SPC 中所有变量及其值的关系,为了描述简化假设 SPC 的变量都不同名。 \mathbb{HD} 描述当前各线程对共享变量的维持状态,如 x_i 被线程号为 tid 的线程维持,则 \mathbb{HD} 中有 $[x_i, tid]$ 项。 \mathbb{TS} 为线程序列,它由一组线程 T 组成。 \mathbb{T} 为 \mathbb{W} 的简化,用于在转换规则中描述某个线程执行时状态空间中的 \mathbb{S} 和 \mathbb{HD} 的变换。 T 为线程,它由线程号 tid ,线程代码 S 和子线程集合 \mathbb{TS} 构成。 tid 为线程号,每个线程号对应一个线程。 S 为线程代码,它为图 2 中的语句,规定抽象机在执行原子命令时,不允许被其他线程干扰。

抽象机上定义了两种状态变换:

$\mathbb{W} \Rightarrow \mathbb{W}'$: 抽象机的状态变换,由任意线程的执行引起。

$T \Rightarrow_{\text{thrd}} T'$: 某个线程执行一步而引起抽象机上的状态变换。

图 9 为具体操作语义变换规则。 $\mathbb{S} \vdash E \triangleright v$ 或 $\mathbb{S} \vdash B \triangleright v$ 表示在存储状态 \mathbb{S} 下原子命令中的表达式 E 或 B 计算为 v 。谓词 mayHd 表示线程 tid 在执行到标签为 l 的原子命令时检查当前的维持状态 \mathbb{HD} 以确定该原子命令上要维持的共享变

量集合 $\mathcal{H}_a(l)$ 中的共享变量没有被其他线程维持。谓词 updHd 、 updHdT 和 updHdF 表示线程 tid 在执行标签为 l 的原子命令后释放该原子命令上要释放维持的共享变量集合 $\mathcal{H}_r(l)$ ，以更新维持状态 HD 。 updHd 用于非 $[B]^l$ 的原子命令上， updHdT 和 updHdF 分别用于 $[B]^l$ 上，且依据 B 计算的值为 1 和 0 时使用。它们的定义见图 8。

WORLD 规则描述整个抽象机状态的变换，整个状态的变换由任意线程的一步执行而引起。ASS 规则描述线程在执行赋值语句时，首先检查语句上要获得维持的共享变量没有被其他线程维持(使用 mayHd 函数)，语句执行完后，存储状态和维持集合都有所改变，维持集合的改变由 updHdT 来完成。IF1、IF2 规则描述了线程在执行 **if** 语句时由于条件表达式计算结果的不同，而走不同的分支，在计算条件之前需要做类似于 ASS 规则的 mayHd 操作，当条件为 true 时(条件结果为 1)，执行结果会进入 **if** 语句的 **then** 分支，当条件为 false 时(条件表达式结果为 0)，执行结果进入 **else** 分支。IF1 和 IF2 的不同，还在于对维持状态 HD 更新时，IF1 使用 updHdT ，IF2 使用 updHdF 。LOOP1 和 LOOP2 描述了线程在 **while** 语句时，根据条件表达式的计算结果不同，而走不同的分支，在计算条件之前需要做类似于 ASS 规则的 mayHd 操作，当条件为 true 时，首先进入循环体，然后还要增加下一次循环，当条件为 false 时，退出循环，在更新维持状态 HD 时类似 IF 规则。IF 和 LOOP 规则都没有改变

$W \Rightarrow W'$ (World Transition)

$$\frac{(\mathcal{S}, \text{HD}, T_i) \Rightarrow_{\text{thrd}} (\mathcal{S}', \text{HD}', T'_i)}{(\mathcal{S}, \text{HD}, (T_1, \dots, T_i, \dots, T_n)) \Rightarrow_{\text{thrd}} (\mathcal{S}', \text{HD}', (T_1, \dots, T'_i, \dots, T_n))} \text{WORLD}$$

$T \Rightarrow_{\text{thrd}} T'$ (Thread Transition)

$$\frac{(\mathcal{S}, \text{HD}, (tid, S_1, \emptyset)) \Rightarrow_{\text{thrd}} (\mathcal{S}', \text{HD}', (tid, S'_1, \emptyset))}{(\mathcal{S}, \text{HD}, (tid, S_1; S_2, \emptyset)) \Rightarrow_{\text{thrd}} (\mathcal{S}', \text{HD}', (tid, S'_1; S_2, \emptyset))} \text{SEQ1}$$

$$\frac{}{(\mathcal{S}, \text{HD}, (tid, \text{skip}; S_2, \emptyset)) \Rightarrow_{\text{thrd}} (\mathcal{S}, \text{HD}, (tid, S_2, \emptyset))} \text{SEQ2}$$

$$\frac{\text{mayHd}(\text{HD}, tid, l) \quad \text{HD}' = \text{updHd}(\text{HD}, tid, l)}{(\mathcal{S}, \text{HD}, (tid, [(x_1, \dots, x_m)]^l, \emptyset)) \Rightarrow_{\text{thrd}} (\mathcal{S}, \text{HD}', (tid, \emptyset))} \text{HOLD}$$

$$\frac{\text{mayHd}(\text{HD}, tid, l) \quad \mathcal{S} \vdash E \triangleright v \quad \text{HD}' = \text{updHd}(\text{HD}, tid, l) \quad \mathcal{S}' = \mathcal{S}\{x \rightsquigarrow v\}}{(\mathcal{S}, \text{HD}, (tid, [x = E]^l, \emptyset)) \Rightarrow_{\text{thrd}} (\mathcal{S}', \text{HD}', (tid, \emptyset))} \text{ASS}$$

$$\frac{\text{mayHd}(\text{HD}, tid, l) \quad \mathcal{S} \vdash B \triangleright 1 \quad \text{HD}' = \text{updHdT}(\text{HD}, tid, l)}{(\mathcal{S}, \text{HD}, (tid, \text{if}([B]^l)S_1 \text{ else } S_2, \emptyset)) \Rightarrow_{\text{thrd}} (\mathcal{S}, \text{HD}', (tid, S_1, \emptyset))} \text{IF1}$$

$$\frac{\text{mayHd}(\text{HD}, tid, l) \quad \mathcal{S} \vdash B \triangleright 0 \quad \text{HD}' = \text{updHdF}(\text{HD}, tid, l)}{(\mathcal{S}, \text{HD}, (tid, \text{if}([B]^l)S_1 \text{ else } S_2, \emptyset)) \Rightarrow_{\text{thrd}} (\mathcal{S}, \text{HD}', (tid, S_2, \emptyset))} \text{IF2}$$

$$\frac{\text{mayHd}(\text{HD}, tid, l) \quad \mathcal{S} \vdash B \triangleright 1 \quad \text{HD}' = \text{updHdT}(\text{HD}, tid, l)}{(\mathcal{S}, \text{HD}, (tid, \text{while}([B]^l)S, \emptyset)) \Rightarrow_{\text{thrd}} (\mathcal{S}, \text{HD}', (tid, S; \text{while}([B]^l)S, \emptyset))} \text{LOOP1}$$

$$\frac{\text{mayHd}(\text{HD}, tid, l) \quad \mathcal{S} \vdash B \triangleright 0 \quad \text{HD}' = \text{updHdF}(\text{HD}, tid, l)}{(\mathcal{S}, \text{HD}, (tid, \text{while}([B]^l)S, \emptyset)) \Rightarrow_{\text{thrd}} (\mathcal{S}, \text{HD}', (tid, \text{skip}, \emptyset))} \text{LOOP2}$$

$$\frac{}{(\mathcal{S}, \text{HD}, (tid, S_1 || \dots || S_n; S, \emptyset)) \Rightarrow_{\text{thrd}} (\mathcal{S}, \text{HD}, (tid, \text{skip}; S, (\text{spawn}(S_1), \dots, \text{spawn}(S_n)))} \text{PAR}$$

$$\frac{(\mathcal{S}, \text{HD}, T_i) \Rightarrow_{\text{thrd}} (\mathcal{S}', \text{HD}', T'_i)}{(\mathcal{S}, \text{HD}, (tid, S, (T_1, \dots, T_i, \dots, T_k))) \Rightarrow_{\text{thrd}} (\mathcal{S}', \text{HD}', (tid, S, (T_1, \dots, T'_i, \dots, T_k)))} \text{SUB-THRD1}$$

$$\frac{}{(\mathcal{S}, \text{HD}, (tid, S, ((tid_1, \text{skip}, \emptyset), \dots, (tid_k, \text{skip}, \emptyset)))) \Rightarrow_{\text{thrd}} (\mathcal{S}, \text{HD}, (tid, S, \emptyset))} \text{SUB-THRD2}$$

图 9 SPC 的操作语义

Fig. 9 The operational semantics of SPC

存储状态。PAR 描述线程执行到并行语句时，会创建子线程。SUB-THRD1 描述父线程创建子线程后，父线程需等待子线程的执行。SUB-THRD2 描述父线程的所以子线程执行完后，线程继续执行。其他规则的含义都很比较直接易懂。

$$\begin{aligned} (\text{World}) \quad W &::= (\mathcal{S}, \text{HD}, \text{TS}) \\ (\text{ThreadSet}) \quad \text{TS} &::= \emptyset \mid (T_1, \dots, T_n) \\ (\text{Store}) \quad \mathcal{S} &::= \{x \rightsquigarrow tid\}^* \\ (\text{ThreadState}) \quad T &::= (\mathcal{S}, \text{HD}, T) \\ (\text{Thread}) \quad T_i &::= (tid, \mathcal{S}, \text{TS}) \\ (\text{ThrdID}) \quad tid &::= m \text{ (nat nums, and } m > 0) \\ (\text{HoldingSet}) \quad \text{HD} &::= \{x \rightsquigarrow tid\}^* \end{aligned}$$

图 7 SPC 抽象机

Fig. 7 SPC abstract machine

$$\begin{aligned} \text{mayHd}(\text{HD}, tid, l) &\stackrel{\text{def}}{=} \forall s \in \mathcal{H}_a(l). s \notin \text{dom}(\text{HD}) \\ &\quad \vee (s, tid) \in \text{HD} \\ \text{updHd}(\text{HD}, tid, l) &\stackrel{\text{def}}{=} \text{HD} \cup \{(s, tid) \mid s \in \mathcal{H}_a(l)\} \\ &\quad - \{(s, tid) \mid (s, \perp) \in \mathcal{H}_r(l)\} \\ \text{updHdT}(\text{HD}, tid, l) &\stackrel{\text{def}}{=} \cup \{(s, tid) \mid s \in \mathcal{H}_a(l)\} \\ &\quad - \{(s, tid) \mid (s, \mathbf{T}) \in \mathcal{H}_r(l)\} \\ \text{updHdF}(\text{HD}, tid, l) &\stackrel{\text{def}}{=} \text{HD} \cup \{(s, tid) \mid s \in \mathcal{H}_a(l)\} \\ &\quad - \{(s, tid) \mid (s, \mathbf{F}) \in \mathcal{H}_r(l)\} \end{aligned}$$

图 8 操作语义辅助函数

Fig. 8 The auxiliary functions used in operational semantics

5 访问控制代码生成

在获得线程每一原子命令执行之前需要获得维持的共享变量集合和执行之后需要释放维持的共享变量集合后，编译器就可为共享变量增加具体的访问控制代码。本节先介绍生成基于锁的访问控制代码的方法，然后简要讨论生成到其他形式的访问控制代码的方法。

5.1 生成到锁方式的访问控制代码

我们已经实现将 SPC 源语言转换到锁方式的访问控制方式的目标语言上^[11]。目标语言见图 10。与源语言相比，多了 **lock** 加锁和 **unlock** 解锁操作，而去掉了含 **last** 修饰的表达式和语句，**if** 语句恢复到允许没有 **else** 分支的情况。将 SPC 转换到锁方式的目标语言的算法见图 11，算法的输入为 SPC 源程序和共享变量维持信息 \mathcal{H}_a 、 \mathcal{H}_r ，输出为带锁控制方式的目标语言。

$$\begin{aligned}
 (Stmt) \quad S &::= x = E \mid S_1; S_2 \mid S_1 \parallel \dots \parallel S_n \\
 &\quad \mid \text{if } (B) \ S \mid \text{if } (B) \ S_1 \ \text{else } S_2 \\
 &\quad \mid \text{while } (B) \ S \\
 &\quad \mid \text{lock}(x) \mid \text{unlock}(x) \\
 (AExp) \quad E &::= x \mid \bar{x} \mid n \mid (E) \mid -E \mid E_1 \ \text{op}_a \ E_2 \\
 (BExp) \quad B &::= (B) \mid \neg B \mid E_1 \ \text{op}_r \ E_2 \mid B_1 \ \text{op}_b \ B_2
 \end{aligned}$$

图 10 目标语言

Fig. 10 The object language

根据图 11 的算法，图 12(a)为图 5(a)转换到锁方式的代码实例。

```

for ([A]l ∈ S) do
  /* Acquire lock operation */
  for (x ∈ Ha(l)) do
    for ((l', l) ∈ flow(S)) do
      add a lock(x) between l' and l
    /* Release lock operation */
  for ((x, -) ∈ Hr(l)) do begin
    if ((x, ⊥) ∈ Hr(l)) do
      add an unlock(x) after the l
    if ((x, T) ∈ Hr(l)) do
      add an unlock(x) after true successor of the l
    if ((x, F) ∈ Hr(l)) do
      add an unlock(x) after false successor of the l
  end

```

图 11 SPC 转换到锁方式算法

Fig. 11 The algorithm of transforming SPC program to lock-based program

5.2 其他控制方式的生成

编译器通过分析原子命令在执行时需要获得维持和执行之后需要释放维持的共享变量集合，很容易分析得到共享变量的开始维持程序点和结束维持程序点，然后通过分析各个共享变量的维持期间是否存在包含关系来调整共享变量与保护它的锁之间的分配关系，例如，让一组维持期间基本一致的共享变量共享一个锁，让一组维持期间互不相交的共享变量分享一个锁等等。此外，还可以结合目标机器的特征以及程序特征，将源程序翻译到使用不同访问控制方式的代码上；甚至可以依据维持信息来分析出可能在访问时发生死锁的位置等。

```

lock(num);
while(num>0){
  lock(cur);
  t=cur;
  lock(pre);
  cur=cur+pre;
  unlock(cur);
  pre=t;
  unlock(pre);
  num=num-1;
  unlock(num);
  lock(num);
}
unlock(num);
(a)

lock(num);
while(num>0){
  t=cur;
  cur=cur+pre;
  pre=t;
  num=num-1;
  unlock(num);
  lock(num);
  unlock(num);
}
(b)

stm_start(num,cur,pre);
while(num>0){
  t=cur;
  cur=cur+pre;
  pre=t;
  num=num-1;
  stm_end();
  stm_start(num,cur,pre);
}
stm_end();
(c)

```

图 12 具体访问控制代码生成

Fig. 12 The generated concrete access control code

对图 5 的程序实例，通过图 6 的分析结果，编译器可以分析出共享变量 num 被维持在语句标签为 1 到 5 之间，而共享变量 cur 被维持在 2 和 3 之间， pre 被维持在 3 和 4 之间，依据上面的分析，可以使用一个锁在标签为 1 到 5 的原子集合之间来保护这三个共享变量的访问。从而减少锁资源的分配，转换结果如图 12(b)所示。

同样，经过分析图 6 的结果，可以考虑将图 5 程序中语句标签为 1 到 5 之间的代码作为一个事务来处理，这样，共享变量 num 、 cur 和 pre 在事务中被访问，即线程在执行这段代码时，先执行，执行到标签为 5 的语句后，通过事务的提交来检查是否发生了冲突，如果没有发生冲突，继续执行下次循环，如果发生冲突，则回滚到当前循环的 1 处重新执行。翻译到事务访问控制方式的结果如图 12(c)所示。

6 小结

本文阐述了基于共享变量维持声明的并行程序小语言 SPC 的设计和实现，它引入 **last** 修饰符允许程序员描述跨原子命令之间的共享变量维持需求，编译器将依据程序中的 **last** 维持信息和原子命令本身对共享变量的维持需求，来分析得到每条原子命令执行之前需要获得维持的共享变量集合以及执行之后需要释放维持的共享变量集合，并根据此给出 SPC 的操作语义，最后讨论如何利用维持信息为 SPC 程序自动增加访问控制代码。

由于本文的重点旨在阐述共享变量维持声明及其语义和到访问控制代码的自动生成，故为简化起见在语言中仅考虑整型变量的使用。在我们实现的编译器中，目前给出了整型的实现，在编译器为代码自动增加访问控制代码时，使用了互斥锁的方式。本文的后继工作将完善 SPC：增加对动态分配的共享数据结构的支持、对函数的支持和并行嵌套的支持；在翻译到具体的访问控制代码上，通过分析维持信息，考虑优化为共享变量分配锁，以减少分配锁时的锁资源，以及考虑翻译到事务方式上等；还可以尝试去分析死锁的可能性，在引入 I/O 等在事务中不可回滚的操作后，可以尝试将访问控制方式翻译到锁与事务相结合的访问控制方式上。

References:

- [1] Vijay Saraswat. Report on the experimental language X10. Version 2.01, Jan. 2010. <http://x10-lang.org/>.

- [2] Jonathan Lee and Jens Palsberg. Featherweight X10: A core calculus for async-finish parallelism In *Proceedings of 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*, Bangalore, India, pages 25-36, Jan. 2010.
- [3] Cray Inc. Chapel language specification 0.795. Apr. 2010. <http://chapel.cray.com/spec/spec-0.795.pdf>
- [4] Eric Allen, David Chase, Joe Hallett et al. The Fortress language specification. Version1.0. Sun Microsystems, Inc., Mar. 2008. <http://projectfortress.sun.com/Projects/Community/>.
- [5] Christopher J. Rossbach, Owen S. Hofmann and Emmett Witchel. Is transactional programming actually easier? [A]. In *Proceedings of 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming(PPoPP'10)*, Bangalore, India, pages 47-56, ACM, 2010.
- [6] J. R. Larus and R. Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*. Morgan & Claypool Publishers, Jan. 2007.
- [7] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46-58, 2008.
- [8] Yuan Zhang Joseph Bryant, Manzano Franco and Guang R. Gao. Atomic section: Concept and implementation. In *Proceedings of Mid-Atlantic Student Workshop on Programming Languages and Systems (MASPLAS'05)*, Newark, Delaware, USA, April 30, 2005.
- [9] Sigmund Cherem, Trishul Chilimbi and Sumit Gulwani. Inferring locks for atomic sections [A]. In *Proceedings of ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI'08)*, Tucson, Arizona, pages 304-315, Jun. 2008.
- [10] Zhang Yu, Wang Chen, Zhang Wei, Fu XiaoPeng [R]. A Parallel Programming Language with Resource Holding Declaration. University of Science and Technology of China: Technical Report, 2010. <http://sug.ustcsz.edu.cn/sites/default/files/intlang.pdf>
- [11] The HomePage of the *Parallel Programming Language with Shared Resource Specification*. May, 2009. <http://sug.ustcsz.edu.cn/pplrs>