基于模型驱动的分治并行函数式程序生成及自动验证

王昌晶[1,2], 王忠文[1], 潘 丞[1], 黄 箐[1], 左正康[1]

[1]江西师范大学计算机信息工程学院 江西 中国 330022
[2]江西师范大学管理科学与工程研究中心 江西 中国 330022

摘要 并行计算作为人工智能发展的动力, 使得并行算法的可解释性和安全性成为人工智能领域重要研究方向。形式化方法以数理逻辑为基础, 已经成为复杂安全苛求系统可信构建的重要方法, 而函数式编程则在算法领域中具有更强的数学表达性。本文旨在提出一种基于模型驱动的分治并行函数式程序生成及自动验证方法, 融合形式化方法, 以解决目前分治并行程序生成和验证中缺乏可解释性、易错、低可信度等问题。首先, 采用分划递推法和循环不变式等新策略推导出串行算法; 然后, 利用辅助函数和算法连接函数将其提升为并行算法, 并使用我们提出的并行算法设计语言 Radl$^+$进行描述; 进而, 采用同态定理验证框架在 Isabelle 中验证算法连接函数满足同态定理, 即提升后的算法可并行化; 最后, 提出了 Radl$^+$→Haskell 转换规则, 设计了“Radl$^+$→Haskell 并行程序生成系统”软件原型。实验结果表明, 本文能够生成和验证一系列算法的并行函数式程序, 并且能够产生良好的加速比。本文方法不仅具有一定的可解释性, 而且自动验证减少了传统手工验证易错性和繁琐的工作量, 保证算法正确性和提高安全性, 对大幅度提升高可信并行函数式程序的开发效率具有重要意义。

关键词 模型驱动; 分治并行; 函数式程序; 程序生成; 自动验证
中图法分类号 TP311.5 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2023.05.07

Model-driven Divide-and-conquer Parallel Functional Program Generation and Automatic Verification

WANG Changjing[1,2], WANG Zhongwen[1], PAN Cheng[1], HUANG qing[1], ZUO Zhengkang[1]

[1] School of Computer and Information Engineering, Jiangxi Normal University, Jiangxi 330022, China
[2] Management science and Engineering Research Center, Jiangxi Normal University, Jiangxi 330022, China

Abstract Parallel computing as a driving force for the development of artificial intelligence has made the interpretability and security of parallel algorithms an important research direction in the field of artificial intelligence. Formal methods, based on mathematical logic, have become an important approach for the credible construction of complex secure caustic systems, while functional programming has a stronger mathematical expressiveness in the field of algorithms. The purpose of this paper is to propose a model-driven divide-and-conquer parallel functional program generation and automatic verification method by fusing formal methods to solve the current problems of lack of interpretability, error-prone, and low trustworthiness in divide-and-conquer parallel program generation and verification. First, a sequential algorithm is derived using the partition-and-recur method and the new strategy of loop invariant development; then, it is lifted to a parallel algorithm using auxiliary functions and algorithmic join functions and described using our proposed parallel algorithm design language Radl$^+$; further, the homomorphism theorem verification framework is used to verify in Isabelle that the algorithmic connectivity functions satisfy the homomorphism theorem, i.e., the elevated algorithm is parallelizable; and Finally, we propose the Radl$^+$→Haskell transformation rule and design a software prototype of “Radl$^+$→Haskell Parallel Program Generation System”. The experimental results show that this paper can generate and verify parallel functional programs for a series of algorithms, and can produce good speedup. The method not only has certain interpretability, but also reduces the error-prone and tedious workload of traditional manual verification, ensures the correctness of algorithms and improves security, which is of great significance to significantly improve the development efficiency of highly trusted parallel functional programs.

Key words model-driven; divide-and-conquer parallel; functional programing; program generation; automatic verification

通讯作者: 左正康, 博士, 副教授, zuo803@jxnu.edu.cn。
本课题得到国家自然科学基金项目(No. 62262031), 江西省教育厅科技重点项目(No. GJJ2200302, No. GJJ210307), 江西省研究生创新基金项目(No. YC2022-s349)资助。
收稿日期: 2022-09-11; 修改日期: 2022-12-12; 定稿日期: 2023-03-28

1 引言

随着大数据和人工智能的发展, 并行计算已经成为实现高性能计算的主要技术手段。分治法[1]作为一种通用的并行计算框架日益受到重视, 它是提高多核资源利用率和执行效率最有潜力的途径。分治法的主要思想是将一个复杂问题分解为规模更小、结构与原问题一致的子问题进行并行计算, 递归地求解子问题, 最终将子问题的解进行合并得到原问题的解, 它能够充分地发挥多核处理器性能, 高效利用计算机空间资源且提高任务的执行效率。

并行算法是并行计算的核心和瓶颈技术[2], 因此并行算法的正确性是提高网络环境下软件可靠性的关键。形式化方法是基于严格数学基础, 对计算机硬件和软件系统进行描述、开发和验证的技术[3], 是并行算法高可信的根本保证。大量的工业实践表明, 形式化方法已经成为复杂安全苛求系统可信构建的重要方法, 在处理软件开发复杂性和提高软件可靠性方面具有无可取代的潜力。随着形式化验证技术的发展, 在形式化研究领域出现了许多性能优良的程序证明器, 在数学定理证明、软/硬件验证等方面发挥着重要作用, 如 Coq[4]、Dafny[5]和 Isabelle[6]等。其中 Isabelle 是基于高阶逻辑的交互式定理证明器[7], 作为最具代表性之一的程序证明器, 它既支持多种对象逻辑又允许自定义新逻辑, 具有丰富的类型系统、强大的规则库和灵活高效的命令集, 同时能够充分利用多核处理器并行证明[8], 达到了“提高验证效率和保证算法程序的高可信”的目标。

高可信并行算法的生成及验证, 已经逐渐成为计算机科学非常热门的一个研究方向。现有的大多数并行算法生成研究, 串行算法是直接通过经验给出的, 算法连接函数和辅助函数的获得需要更多的信息, 如弱右逆[9]; 或通过特定的重写规则[10]或合成算法[11]。并行算法的验证, 需要证明串行算法提升至并行算法满足可并行性的同态定理, 现有的传统手工证明方法对其验证, 不仅易错, 且可信度低; 或采用验证工具如 Dafny, Dafny 的类与类之间仅支持有限的继承, 代码存在冗余, 通用性和可复用性较弱[12]; 且 Dafny 是一种命令式与函数式混合的描述语言, 在验证时探索前后置条件以及大量的中间断言是一段相当痛苦的过程[13]。

针对上述挑战, 融合形式化方法, 本文提出了一种基于模型驱动的分治并行函数式程序生成方法及自动验证方法。模型驱动开发是一种面向模型的新型软件设计方法[14]。在该方法中, 模型转换是软件开发的核心, 软件的各个方面由各种模型来表达, 最终的软件由模型通过模型转换来生成。该方法使得软件开发人员不必再去关心编程语言的细节和执行平台的特性, 提高了软件工程的抽象和自动化水平。

首先, 将问题抽象为功能规约, 并通过本团队提出的分划递推法[15]和循环不变式开发新策略[16]推导出用 Radl 描述的串行算法; 通过对循环不变式进行展开并归纳, 推导得到相应的辅助函数和算法连接函数, 将串行算法进行提升得到原问题用 $Radl^{+}$描述的并行算法; 进而, 利用 Isabelle 定理证明器来自动验证算法连接函数满足同态定理[17], 即提升后的算法满足可并行性。

函数式编程已经成为当前最流行的编程模式之一, 它作为一种采用函数进行程序设计的编程范式, 比命令式编程具有更强的数学表达性[18], 更加强调对函数的计算而不是对指令的运行, 能够极大减少代码量、软件开发时间及成本, 有利于降低软件复杂性。同时, 函数式编程不会产生副作用, 不需要考虑死锁, 利于部署多个线程的并行计算, 因此天然支持并行。Haskell[19]作为当前比较流行的一门通用的纯函数式编程语言, 具有简洁、表达力强的语法和丰富的内置数据类型, 提供了抽象的编程能力, 且易于维护。

最后, 通过本文设计的“$Radl^{+}$→Haskell 并行程序生成系统”软件原型将并行算法转换为 Haskell 并行函数式可执行程序。实验结果表明, 本文能够生成和验证数组最小和、最大基因序列等一系列算法的并行函数式程序。自动验证减少了传统手工验证易错性和繁琐的工作量, 保证了算法正确性, 对大幅度提升高可信并行函数式程序的开发效率具有重要意义。

本文提出一种基于模型驱动的分治并行程序生成及自动验证方法。本文创新点如下:

1) 提出一种基于模型驱动的方法将复杂算法问题生成分治并行函数式程序, 首次实现从抽象规约到具体分治并行函数式程序完整的生成过程。本文首先使用分划递推法推导出串行算法, 然后基于循环不变式的扩展与归纳来得到辅助函数和算法连接函数, 以将串行算法提升为并行算法。该方法的串行算法是通过推导得到的, 而非经验给出。算法连接函数和辅助函数生成的输入只需要串行算法, 生成过程更加系统和一般化。

2) 使用 Isabelle 定理证明器对提升后的算法满足同态定理自动验证, 证明该算法的可并行化, 保

障了并行算法的可靠性并提高了验证效率。使用 Isabelle 较 Dafny 通用性和复用性更强, 也无须在验证时探索前后置条件以及大量的中间断言。

3) 将验证成功的 Radl$^+$并行算法通过设计的软件原型和转换规则映射为具体的 Haskell 并行函数式程序, 生成的函数式程序具有良好的扩展性和天然的并行性。基于原型系统, 本文通过模型驱动方法对一系列典型案例生成了 Haskell 并行函数式程序。

本文的组织结构如下: 第 2 节阐述本文提出的模型驱动的分治并行程序设计及自动验证方法步骤; 第 3 节简述问题规约到串行算法的推导过程; 第 4 节阐述串行算法如何提升至并行算法; 第 5 节通过 Isabelle 验证并行算法的可并行性; 第 6 节介绍 Haskell 并行函数式程序的生成; 第 7 节通过实验验证了本文方法具有良好的加速比; 第 8 节介绍相关工作; 第 9 节总结全文并介绍未来工作展望。

2 基于模型驱动的分治并行函数式程序生成及自动验证方法

针对复杂算法问题分治并行函数式程序生成及验证的挑战, 本文将模型驱动方法与形式化方法结合, 研究关于复杂线性算法问题的并行程序生成及验证的完整过程。首先将问题规约进行变换至用 Radl[20] (Recurrence-based Algorithm Design Language) 描述的串行算法, 然后构造辅助函数和算法连接函数将算法提升至用 Radl$^+$描述的并行算法, 进而使用 Isabelle 对该并行算法满足同态定理进行自动验证, 最终将验证成功的可并行化算法映射至 Haskell 并行函数式程序。

本文主要针对在列表上的单路(Single-pass)函数, 且求解该函数的串行程序是单层循环(无嵌套循环)结构, 采用分治并行模式来设计和生成并行程序。假设类型 S 是一个列表的类型, 类型 *Sc* 是该列表元素的任一具体类型, 如 int, float, char 和 bool 等。下面给出单路函数的正式定义[21]。

定义 1.(左向函数, 右向函数) 函数 $h: S \to Sc$ 是左向函数, 当且仅当存在一个二元连接算子 $\oplus$, 使得 $h([a] \bullet x) = a \oplus h(x)$; 函数 h 是右向函数, 当且仅当存在一个二元连接算子 $\otimes$, 使得 $h(x \bullet [a]) = h(x) \otimes a$。

定义 2.(单路函数) 函数 $h: S \to Sc$ 是单路函数, 当且仅当 h 是左向或右向函数。

假设一个计算 $f(a)$的分治并行模式策略如下:

$$f(a) = f(\gamma_a.1) \odot f(\gamma_a.2) \odot \cdots \odot f(\gamma_a.n) \qquad (1)$$

上述$f(a)$表示函数f作用于列表a, 分划算子 γ通过一个分划函数将 a 分划为 n 个子列表且并行计算它们的结果, 连接算子$\odot$利用算法连接函数合并子列表的值, 直至最终合并为一个结果, 其中每一层的合并均可并行执行。基于分治并行模式驱动的函数 f 计算过程如图 1 所示:

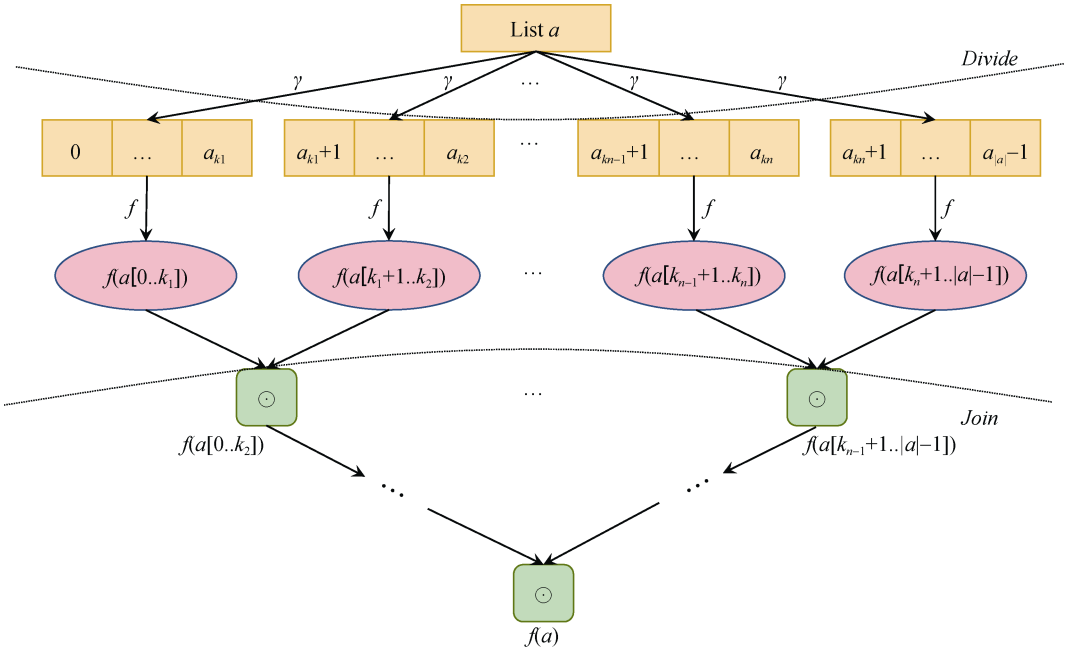

图 1 函数 f 的分治并行计算

Figure 1 D&C parallel computing of function f

模型驱动的分治并行函数式程序生成及自动验证方法步骤如图 2 所示, 分为如下四个阶段:

第一阶段, 从问题描述出发, 用 Radl 语言精确地描述求解问题的功能规约, 正确使用分划递推法和量词特性对其进行规约变换, 构造问题求解的递推关系, 并根据本团队提出循环不变式的开发新策略来构造问题的循环不变式, 基于所得的递推关系和循环不变式进行算法构造, 得到求解问题的 Radl 串行算法;

第二阶段, 对循环不变式进行展开并归纳, 推导得到需要添加的辅助函数(目标函数不可直接并行时)和算法连接函数。将辅助函数和算法连接函数组合构造得到分治并行算法。为 Radl 语言添加并行性描述, 扩充为用于描述分治并行算法的 Radl$^+$语言, 对提升得到的并行算法进行描述;

第三阶段, 从定理的角度去证明满足同态定理性质的函数一定是可并行的, 通过定理证明器 Isabelle 定义相关函数, 创建算法连接函数满足同态定理的相关引理, 自动验证并行算法的可并行性;

第四阶段, 通过设计的“Radl$^+$→Haskell 并行程序生成系统”软件原型和提出的“Radl$^+$→Haskell 转换规则”将 Radl$^+$算法映射为具体可执行的 Haskell 并行函数式程序, 并放至 GHC 平台[22]运行

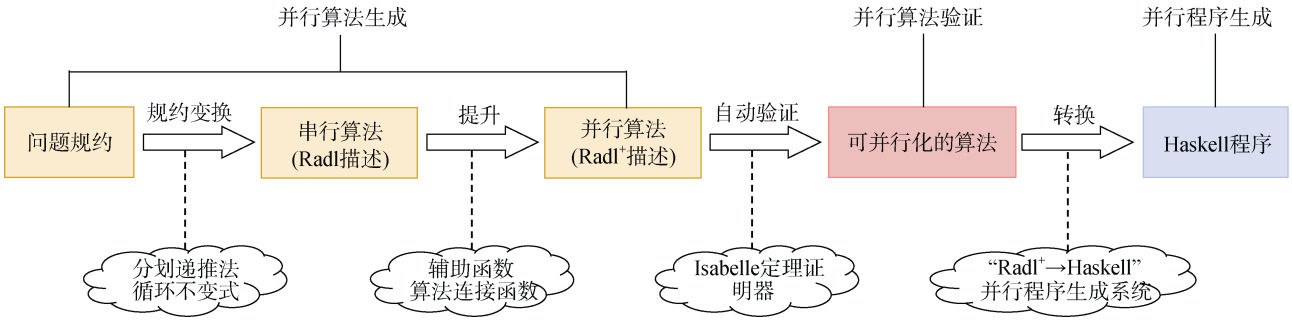

图 2　模型驱动的分治并行函数式程序生成及自动验证方法步骤

Figure 2　Model-driven D&C parallel functional program generation and automatic verification method steps

3　串行算法生成

本文首先从问题规约出发, 使用本研究团队提出的分划递推法[15], 基于规约变换规则构造求解问题的所有递推关系, 并通过循环不变式的开发策略推导求解问题的循环不变式, 最终得到求解问题的具有单重循环结构的串行算法, 并使用 Radl 语言描述。

3.1　分划递推法

为精确地刻画问题的功能规约, 令 q 表示满足交换律和结合律的二元运算符, 量词 Q 表示 q 对任何集合的应用, 则量词 Q(除 N 外)是 q 的一般化, 并进行如下约定: $\forall$ 代表全称量词、$\exists$ 代表存在量词、Σ 代表求和量词、$\prod$ 代表求积量词、N 代表计数量词、MAX 和 MIN 分别代表极大和极小量词等。即 $\forall$ 是逻辑运算符 $\wedge$ 的一般化, $\exists$ 是 $\vee$ 的一般化, Σ 是+的一般化, $\prod$ 是*的一般化, MAX 和 MIN 分别是 max 和 min 的一般化。

在本文中, 我们以 AQ 表示问题的前置条件, AR 表示问题的后置条件, S 表示求解问题的程序。根据 Hoare 逻辑得到以下 Hoare 三元组[23,24]规约:

$$\{AQ\} \quad S \quad \{AR\} \tag{2}$$

式(2)表示 S 在执行前满足条件 AQ, 在终止后满足条件 AR。同时 AR 可以定义为以下形式:

$$AR \triangleq (Qi : r(i) : f(i)) \tag{3}$$

式(3)表示在范围 $r(i)$上 q 对函数 $f(i)$进行 q 运算所得的量, 其中 i 表示约束于 Q 的变量, q 是 Q 的一般化, 表示满足交换律和结合律的二元运算符。

功能规约和普通程序的一个显著区别是可以对功能规约进行数学推演(公式推演), 使其从一种形式变换为另一种形式。功能规约有两个作用: 第一, 判断程序的多种规范形式是否等价; 第二, 变换功能规约, 寻找求解问题的算法。本文主要涉及第二个作用, 具体规则如下:

规则 1. 单点范围　$(Qi : i = k : f(i)) = f(k)$

规则 2. 范围分裂

$$(Qi : r(i) : f(i)) = (Qi : r(i) \wedge b(i) : f(i)) \; q \; (Qi : r(i) \wedge \neg b(i) : f(i))$$

规则 3. 交叉积

$$(Qi, J : r(i) \wedge s(i, J) : f(i, J)) = (Qi : r(i) : (QJ : s(i, J) : f(i, J)))$$

其中 J 表示任何约束变量的非空集合。

规则 4. 一般分配律　如果 $\odot$ 对 q 满足分配律, i 不是函数 g 的自由变量, 则有:

$$(Qi:r(i):g\odot f(i))=g\odot(Qi:r(i):f(i))$$

分划递推法的主要思想是应用数学工具、程序设计知识和领域知识推导产生问题求解的设计思想和方法并给出它的形式规约表达, 功能规约通过上述规则 1-4 可以进行等价变换, 将复杂问题分划为一系列更简单的子问题, 并通过不断地推导可以得到问题求解的递推关系, 有利于解决复杂问题, 减小问题功能规约向算法程序转变的难度, 对软件自动化具有重要意义。分划递推法的定义如下:

定义 3.(分划递推法) 假设问题 P 的解可分划为由 n 个计算步产生的结果序列 AR_1, AR_2,…, AR_n,且 $\forall i,j:AR_i\cap AR_j=\varnothing$, 其中 $1\leqslant i<j\leqslant n$, 则每一 AR_i 都是问题 P 的子解, 等式 $AR_j=F(\overline{AR_i})$ 表示 AR_j 是其子解 $\overline{AR_i}$ 的函数, 其中 $\overline{AR_i}$ 表示多个子解 AR_i 的序列, 则 AR_n 是问题 P 的解, 称 $AR_j=F(\overline{AR_i})$ 为求解问题 P 的递推关系。通过上述分划和递归过程得到问题 P 的递推关系的方法, 称为分划递推法。

下面以数组最小和问题作为全文的运行实例来详细说明本文提出的方法。本节先给出通过分划递归法推导得到该问题的两个递归关系:

数组最小和实例 给定一串已知长度的实数序列并将元素存放于数组 $a[0{:}n]$, 计算 a 中相邻元素的最小和 mss。则该问题的前置条件 AQ 为 $n\geqslant 0$, 后置条件 AR 为 $mss(n)\equiv(MINi,j:0\leqslant i\leqslant j\leqslant n:sum(i,j))$, 其中 $sum(i,j)=\sum k:i\leqslant k\leqslant j:a[k]$, 表示 a 的子段元素 $a[i]$~$a[j]$之和, 函数 mss 称为求解实例的目标函数。

下面通过规则 1-4 对 AR 进行规约变换:

$$\begin{aligned}
&mss(n)=(MINi,j:0\leqslant i\leqslant j\leqslant n:sum(i,j))\\
&=\{\text{规则 3.交叉积}\}\\
&\quad MINj:0\leqslant j\leqslant n:(MINi:0\leqslant i\leqslant j:sum(i,j))\\
&=\{\text{引入新定义}mts(j)=(MINi:0\leqslant i\leqslant j:sum(i,j)),\\
&\quad \text{表示}a\text{以}a[j]\text{为终点的最小子段和}\}\\
&\quad MINj:0\leqslant j\leqslant n:mts(j)\\
&=\{\text{规则 2.范围分裂与规则 1.单点范围}\}\\
&\quad min((MINj:0\leqslant j<n:mts(j)),mts(n))\\
&\quad =\{\text{mss}(n)\text{定义折叠}\}\\
&\quad min(mss(a[0:n-1]),mts(n))
\end{aligned}\tag{4}$$

式(4)已经得到了 mss 的递推关系式, 其包含了函数 mts, 继续通过规则 1~4 对 $mts(n)$进行转换:

$$\begin{aligned}
&mts(n)=(MINi:0\leqslant i\leqslant n:sum(i,n))\\
&=\{sum(i,j)\text{定义展开}\}\\
&\quad MINi:0\leqslant i\leqslant n:(\sum k:i\leqslant k\leqslant n:a[k])\\
&=\{\text{规则2.范围分裂与规则1.单点范围}\}\\
&\quad MINi:0\leqslant i\leqslant n:((\sum k:i\leqslant k<n:a[k])+a[n])\\
&=\{\text{规则4.一般分配率与}sum(i,j)\text{ 定义折叠}\}\\
&\quad (MINi:0\leqslant i\leqslant n:sum(i,n-1))+a[n]\\
&=\{\text{规则2.范围分裂与规则1.单点范围}\}\\
&\quad min((MINi:0\leqslant i<n:sum(i,n-1)),sum(n,n-1))+\\
&\qquad a[n])\\
&=\{mts(j)\text{定义折叠}\}\\
&\quad min(mts(n-1),0)+a[n]\\
&=\{+\text{对 }min\text{ 的分配率}\}\\
&\quad min(mts(n-1)+a[n],a[n])
\end{aligned}\tag{5}$$

通过上述转换, 最终得到原问题的全部递推关系, 如式(6)所示:

$$\begin{cases}mts(n)=min(mts(n-1)+a[n],a[n])\\ mss(n)=min(mss(n-1),mts(n))\end{cases}\tag{6}$$

3.2 开发循环不变式

循环不变式是推导、开发和证明算法程序的基础和关键, 体现了循环程序的本质特征, 在形式化方法中占有重要的地位。传统的循环不变式定义和开发策略只能推导一些简单问题的循环不变式, 而对于相对复杂的算法问题很难得到满意的结果。

本研究团队在文献[16]中已经提出循环不变式的新定义和开发策略:

定义 4(循环不变式) 给定循环 DO 和它的所有循环变量的集合 A, 一个反映 A 中每一元素的变化规律并在每次循环执行前后均为真的谓词称作循环 DO 的循环不变式。

令 I 表示 DO 的循环不变式, 由定义 4 定义的 I 表示 A 的函数 $I(A)$。集合 A 由若干个子集组成: 循环变量集合 X、AR 中出现的循环变量 Y 和集合 Z, 其中 $Z=A-X-Y$。这样 DO 的前置断言用 $AQ(A)$表示, 后置断言用 $AR(Y)$表示。

策略 1(适用于现存的算法程序) 以循环程序正确性验证为基准, 考察循环初始条件及循环结束所得的信息, 分析程序所解问题的实际背景、数学性质和程序特征, 通过归纳推理找出所有循环变量的变化规律, 即为所求循环不变式。

策略 2(适用于待开发的算法程序) 考察被求解问题的前后置断言和数学特性, 利用行之有效的算法设计方法确定解问题的总策略(在很多情况下是确定问题求解序列的递推关系)和所需的全部循环变量, 用谓词描述每一变量的变化规律, 即为所求循环不变式; 若递推关系中子解数超过 1, 则还需引进一集合变量或一起堆栈作用的序列变量, 递归定义序列中的内容。

数组最小和实例 根据定义 4 和策略 2(本实例属于待开发问题), 首先通过归纳推理找出所有循环变量的变化规律, 分别用状态变量 t 和 s 存放递推关系中 $mts(i)$和 $mss(i)$的值, 并约束循环控制变量 i 的变化范围, 然后用谓词精确表达上述循环变量的变化规律, 从而得到原问题的循环不变式 inv 为:

$$inv: t = mts(i) \wedge s = mss(i) \wedge 0 \leqslant i \leqslant n \tag{7}$$

3.3 串行算法生成

Radl 是一种基于递推关系的算法设计语言[20], 它是为实现算法程序形式化和半自动开发的分划递推方法而定义的, 其主要功能是描述问题规约、规约变换规则和算法。用 Radl 描述串行算法的一般形式如下:

Radl ALGORITHM:<*algorithm name*>
|[*Initialized definitions of operational variables*]|
{*AQ^AR*}
BEGIN:< *variables that control the recursion* >
< *initialization of operational variables* >
TERMINATION:< *conditions for recursive termina-tions* >
RECUR: < *recursion relations* >
END

一个给定问题的串行算法设计可以分为以下三个步骤:

1) 用Radl语言精确地形式化描述问题的功能规约, 即算法要完成的工作;

2) 通过分划递推法将问题按照 3.1 节的各规则分划为若干与原问题结构相同但规模更小的子问题, 然后对子问题继续分划, 直至每一个子问题都能直接求解(最小子问题), 构造出求解问题的递推关系;

3) 通过 3.2 节提出的策略推导出原问题的循环不变式, 确定递推关系中变量的初值和递推的终止条件, 最终组合构造得到原问题的串行算法。

数组最小和实例 根据式(6)和式(7), 可以得到如下数组最小和的串行算法(由 Radl 语言描述), 其中 t 表示第 i 次循环时 a 以 $a[i]$为终点的最小子段和, s 表示第 i 次循环时子段 $a[0..i]$中相邻元素的最小和, i=1++1 表示变量 i 的初始值为 1, 每次循环增长 1。

ALGORITHM 1.*mss* 串行算法.

|[*In n*:*integer* , *a*:*array*[0..*n*, *integer*];*Aux i*:*integer*; *Out s* , *c*:*integer*]|
{*AQ*} ∧ {*AR*}
BEGIN : *t*, *s*= *a*[0], *a*[0] ; *i* = 1++1
TERMINATION : *i*=*n*+1
RECUR :
t= *min*(*t*+*a*[i] , *a*[*i*])
s = *min*(*s*, *t*)
END

4 串行算法提升

当串行算法可直接并行时, 可用目标函数对子问题并行求解, 然后采用算法连接函数对其直接合并。而当串行算法不可直接并行时, 可通过本文提出的策略 3 推导出辅助函数, 将其与目标函数一并添加至算法连接函数的参数中, 提升至能有效进行分治连接的并行算法, 并使用 Radl$^+$语言描述。其中本文针对两种情形下的算法连接函数构造提出了开发方法(策略 4)。

4.1 辅助函数

本文的辅助函数是由不可直接并行的串行算法提升为并行算法所需要额外构造的函数, 它对算法提升起辅助作用。假设类型 S 是一个列表的类型, 类型 Sc 是该列表元素的任一具体类型, 如 int, float, char 和 bool 等。下面给出可直接并行函数和辅助函数的定义。

定义 5. (可直接并行函数) 函数 $h: S \to Sc$ 是可直接并行函数, 当且仅当 $h(x \bullet y) = h(x) \odot h(y)$, 其中 $\bullet: S \times S \to S$ 是 S 上的列表拼接运算, $\odot: Sc \times Sc \to Sc$ 是 Sc 上的连接运算。

定义 6. (辅助函数) 若函数 $h: S \to Sc$ 作用在列表 $x \bullet y$ 上的解可通过 $h(x \bullet y) = (f(x), \varphi(x)) \odot (f(y), \varphi(y))$ 得到, 且 $\varphi \neq 0$, 其中 $h = (f, \varphi)$, f 是问题求解的目标函数, φ 是计算 f 的辅助函数。

根据定义 5 可知, 若函数 h 可以通过一个连接算子 $\odot$ 直接合并其左子序列 x 和右子序列 y 的分治计算结果, 则称 h 是可直接并行的, 此时串行算法在不需要辅助函数的情况下可直接提升为并行算法, 如图 3 所示。

在算法设计中许多基于列表的函数是不可直接并行的, 即大部分复杂算法问题在分划成子问题后分治计算得到的结果不能轻易地直接合并, 此时需

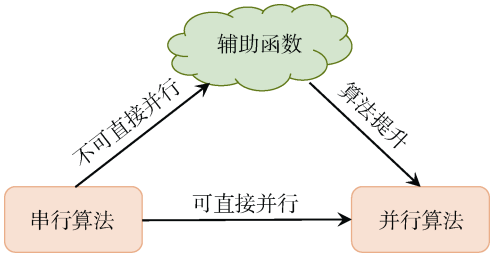

图 3　并行算法生成步骤

Figure 3　Parallel algorithm generation steps

构造合适的辅助函数 φ 并将其添加到 h 的算法连接函数中, 才能提升为并行算法。

根据定义 6, 我们提出了辅助函数的开发策略, 如策略 3 所示。

策略 3. (辅助函数开发策略) 当函数不可直接并行时, 通过归纳法将问题的循环不变式进行展开, 然后将原问题上的解分为两个部分(即目标函数 f 分别作用在左子序列和右子序列上的解), 在连接算子 $\odot$ 的参数列表中(或称 h 的元组中)将目标函数剔除, 剩余函数部分 φ 即为所需额外构造的辅助函数。

数组最小和实例 以 t_i, s_i 分别记录状态变量 t, s 在第 i 次循环的状态。为便于观察分治过程, 以 $t_{i..j}$, $s_{i..j}$ 分别记录 $mts(i..j)$, $mss(i..j)$的值, 表示 mts, mss 作用在子列表 $a[i:j]$上的结果。对 3.2 节得到的循环不变式即式(7)进行展开, 可得。

$$\begin{aligned} t_{i+1} &= mts(i+1) \\ &= min(mts(i)+a[i+1],a[i+1]) \\ &= min(t_i+sum(i+1,i+1),mts(i+1..i+1)) \\ &= min(t_{0..i}+m_{i+1..i+1},t_{i+1..i+1}) \end{aligned} \tag{8}$$

$$\begin{aligned} s_{i+1} &= mss(i+1) \\ &= min(mss(i),mts(i+1)) \\ &= min(mss(i),min(a[i+1],\mathrm{mts}(i)+a[i+1])) \\ &= min(mss(i),min(mss(i+1..i+1),mts(i)+ \\ &\qquad mps(i+1..i+1))) \\ &= min(s_{0..i},s_{i+1..i+1},t_{0..i}+p_{i+1..i+1}) \end{aligned} \tag{9}$$

式(8)中以 $m_{i..j}$ 记录 $sum(i,j)$的值, 表示 sum 作用在子列表 $a[i:j]$上的结果。与式(4)函数 $mts(j)$表示以 $a[j]$为终点的最小子段和类似, 式(9)中 $mps(n)=(MINi:0\leqslant i\leqslant n:sum(0,i))$ 表示以 $a[0]$为起点的最小子段和, 且以 $p_{i..j}$ 表示 mps 作用在子列表 $a[i:j]$上的结果。对式(8)和式(9)继续展开:

$$\begin{aligned} t_{i+2} &= mts(i+2) \\ &= min(mts(i+1)+a[i+2],a[i+2]) \\ &= min(min(mts(i)+a[i+1],a[i+1])+a[i+2],a[i+2]) \\ &= min(min(mts(i)+a[i+1]+a[i+2],a[i+1]+a[i+2]), \\ &\qquad a[i+2]) \\ &= min(mts(i)+a[i+1]+a[i+2],min(a[i+1]+a[i+2], \\ &\qquad a[i+2])) \\ &= min(t_{0..i}+m_{i+1..i+2},t_{i+1..i+2}) \end{aligned} \tag{10}$$

$$\begin{aligned} s_{i+2} &= mss(i+2) \\ &= min(mss(i+1),mts(i+2)) \\ &= min(min(mss(i),mts(i+1)),mts(i+2)) \\ &= min(mss(i),min(mts(i+1),mts(i+2))) \\ &= min(mss(i),min(min(mts(i)+a[i+1],a[i+1]), \\ &\qquad min(mts(i)+a[i+1]+a[i+2],a[i+1]+a[i+2]))) \\ &= min(mss(i),min(a[i+1],a[i+1]+a[i+2]),mts(i) \\ &\qquad +min(a[i+1],a[i+1]+a[i+2])) \\ &= min(s_i,mss(i+1..i+2),mts(i)+mps(i+1..i+2))) \\ &= min(s_{0..i},s_{i+1..i+2},t_{0..i}+p_{i+1..i+2}) \end{aligned} \tag{11}$$

根据策略 3, 从式(8)和式(10)可推导出 sum 是计算 mts 的辅助函数, 从式(9)和式(11)可推导出 mts 和 mps 是计算 mss 的辅助函数。

4.2　算法连接函数

算法连接函数是在原问题分划成子问题后进行合并的函数, 即连接算子 $\odot$ 对应的函数。当问题采用分治并行模式求解时, 该问题将被分解成若干子问题并行求解, 最终需推导合适的算法连接函数将这些子问题的解合并为原问题的解。基于策略 3, 本文针对两种情形下的算法连接函数构造, 提出了以下算法连接函数的开发策略:

策略 4. (算法连接函数开发策略) 当目标函数 f 作用在列表 $x\bullet y$ 上可直接并行时, 算法连接函数的参数由目标函数分别作用在左、右子序列的值 f_x 和 f_y 组成; 否则, 需通过策略 2 对循环不变式展开并归纳, 根据归纳结果构造辅助函数作用在子序列上的值 φ_x 或 φ_y, 再将该值与 f_x, f_y 添加到算法连接函数的参数中。

数组最小和实例 假设数组 a 分划为 $a[0..i]$和 $a[i+1..n]$两个子段, 若采用两个线程分别计算它们的函数值, 如图 4 所示:

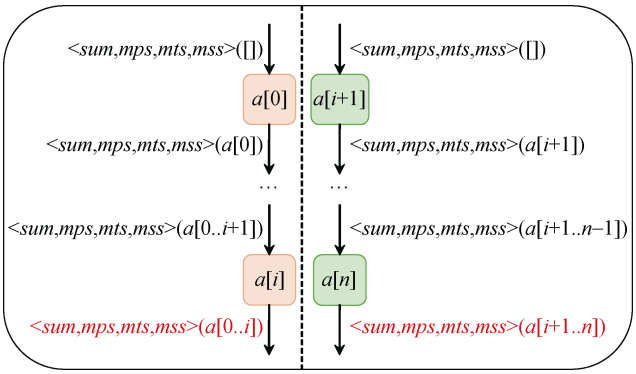

图 4　两个线程的计算过程

Figure 4　The calculation process of two threads

显然,这两个线程是并行执行的, 但它们各自内部是串行计算的。执行结束后需找到合适的算法连接函数将它们的结果进行合并, 此时对式(8)和(10)、式(9)和(11)分别进行归纳, 可得:

$$\begin{cases} mts(n)=min(mts[0..i]+sum[i+1..n],\ mts[i+1..n]) \\ mss(n)=min(mss[0..i],\ mss[i+1..n],\ mts[0..i]+mps[i+1..n]) \end{cases} \tag{12}$$

mps 的算法连接函数与函数 *mts* 归纳过程同理, 再根据定义 4 可得函数 *sum* 是可直接并行的, 它们的算法连接函数很容易得到, 限于篇幅此处省略。最终, 采用如下算法连接函数来合并两个线程的计算结果, 其中 sum_l 表示函数 *sum* 作用在 *a*[0..*i*]上, sum_r 表示函数 *sum* 作用在 *a*[*i*+1..*n*]上, 其余函数同理。

$$\begin{cases} sum=sum_l+sum_r \\ mps=min(mps_l,\ sum_l+mps_r) \\ mts=min(mts_l+sum_r,\ mts_r) \\ mss=min(mss_l,\ mss_r,\ mts_l+mps_r) \end{cases} \tag{13}$$

4.3　算法提升

算法提升是指串行算法转换为并行算法的过程。当串行算法可直接并行时, 可推导出合适的算法连接函数将其提升为并行算法, 否则在提升过程中需要向该算法添加辅助函数, 并使其算法连接函数满足同态定理(见 5.1 节的详细定义)。

Radl 语言具有引用透明性, 易于数学推导, 但不具有并行规约描述, 只适用于描述串行算法, 因此本文受并行算法导论 PRAM 模型[25]启发, 定义了 Radl$^+$作为本文的并行算法设计语言, 它在 Radl 的基础上增加了并行算法的定义和调用。首先在 ALGORITHM 中初始化全局变量, 然后在 BEGIN 中调用分划程序和合并程序, 程序均用 PROCEDURE 定义, 循环体用 RECUR 和 RECUR_parallel 两种形式表示, 两者可嵌套使用, 其中 RECUR 表示以串行方式依次计算每个循环过程, 而 RECUR_parallel … END_parallel 表示以并行方式将多个循环过程同时计算。Radl$^+$的一个典型嵌套结构如下所示:

Radl$^+$ ALGORITHM:<algorithm name>
|[*Initialized definitions of operational variables*]|
{*AQ^AR*}
BEGIN:<*invoked the procedure of divide*>
　　<*invoked the procedure of join*>
END

PROCEDURE:<*procedure name*>
BEGIN: <*variables that control the recursion of RECUR*>
　　<*initialization of operational variables in the RECUR*>
TERMINATION:< *termination conditions of RECUR*>
RECUR:
BEGIN:<*variables that control the recursion of RECUR_parallel*>
　　<*initialization of operational variables in the RECUR_parallel* >
TERMINATION:< *termination conditions of RECUR_parallel*>
RECUR_parallel:
　　<*recursion relations*>
END_parallel
END

数组最小和实例　根据 4.1 节和 4.2 节推导出的辅助函数和算法连接函数(式(13)), 可以得到如下数组最小和的并行算法 **ALGORITHM 2**(由 Radl$^+$语言描述)。

在 **ALGORITHM 2** 中, 函数 *Mss_divide*(*n*)将序列 *a* 的元素划分为 log*n* 个元素为一组, 然后并行求得每组元素的 *sum*, *mps*, *mts* 和 *mss* 值, 此过程可用(*n*/log*n*)个处理器在 *O*(log*n*)时间内完成。*Mss_join*(*p*)函数在每次循环中都并行地将第 2*i*-1 组和第 2*i* 组(*i*=0 to *p*/2)的 *sum*, *mps*, *mts* 和 *mss* 值合并为一个组, 直至最终只有一组, 该组的 *mss* 值即为原问题的解, 此过程可用(*p*/2)个处理器在 *O*(log*p*)时间内完成。为了简便, 假定(*n*/log*n*)、log*n* 和 log*p* 是整数。

5　可并行性

本文使用同态定理来验证提升后的并行算法正确性, 即可并行性。同态定理约定了两个线程可并行求解, 且互不干扰, 最终通过连接操作符将其进行合并, 因此满足同态定理的并行算法具有良好的并行性。同时, 使用 Isabelle 定义了同态定理统一验证

框架并进行自动证明, 其证明步骤可分为算法函数定义和同态定理验证。

5.1 同态定理

同态函数是一个代数结构到同类代数结构的映射, 它保持所有相关的结构不变。在本文中我们研究一类特殊的同态, 即一组列表上的同态。假设类型 S 是一个列表的类型, 类型 Sc 是该列表元素的任一具体类型, 如 int, float, char 和 bool 等。下面给出同态函数的正式定义。

定义 7($\odot$-同态函数) 若函数 $h:S\to Sc$ 对于任意列表 $x,y\in S$, 存在一个二元运算符 $\odot$ 能够使得 $h(x\bullet y)=h(x)\odot h(y)$, 则称 h 满足同态定理, 其中 h 为 $\odot$-同态函数。

ALGORITHM 2.*mss* 并行算法

|[*In n*:*integer* , *a*:*array*[0..*n*,*integer*]; *Aux i* , *p*:*integer*; *Out sum* , *mps* , *mts* , *mss*: *integer*]|

{*AQ*} ∧ {*AR*}

BEGIN :

Mss_divide(*n*)

Mss_join(*n*/log*n*)

END

PROCEDURE *Mss_divide*(*n*)

BEGIN : *sum*(0) , *mps*(0) , *mts*(0) , *mss*(0) = *a*[0] , *i* = 1++1

TERMINATION : *i* = *n*/log*n*

RECUR_parallel :

sum((*i*-1)log*n*+1) = *sum*((*i*-1)log*n*) + *a*[(*i*-1)log*n*+1]

mps((*i*-1)log*n*+1) = *min*(*mps*((*i*-1)log*n*) , *sum*((*i*-1)log*n*) + *a*[(*i*-1)log*n*+1])

mts((*i*-1)log*n*+1) = *min*(*mts*((*i*-1)log*n*) + *a*[(*i*-1)log*n*+1] , *a*[(*i*-1)log*n*+1])

mss((*i*-1)log*n*+1) = *min*(*mss*((*i*-1)log*n*) , *a*[(*i*-1)log*n*+1] , *mts*((*i*-1)log*n*+1))

END_parallel

PROCEDURE *Mss_join*(*p*)

BEGIN : *sum*(*i*) = *sum*((*i*-1)log*n*+1) , *mps*(*i*) = *mps*((*i*-1)log*n*+1) ,

mts(*i*) = *mts*((*i*-1)log*n*+1) , *mss*(*i*) = *mss*((*i*-1)log*n*+1) , (*i* : *integer* = 1 to *p*)

TERMINATION : *p* = 0

RECUR :

BEGIN : *i* = 1

TERMINATION : *i* = *p*/2

RECUR_parallel :

sum(*i*) = *sum*(2*i*-1) + *sum*(2*i*)

mps(*i*) = *min*(*mps*(2*i*-1) , *sum*(2*i*-1) + *mps*(2*i*))

mts(*i*) = *min*(*mts*(2*i*-1) + *sum*(2*i*) , *mts*(2*i*))

mss(*i*) = *min*(*mss*(2*i*-1) , *mss*(2*i*) , *mts*(2*i*-1) + *mps*(2*i*))

END_parallel

p = *p*/2

END

显然, 定义 7 中的函数 h 作用在列表上是可直接并行的, 此时称 h 是列表同态函数, 它是一类在算法设计中普遍存在的函数, 如求和函数 *sum*, 长度函数 *length*, 最大值函数 *max* 等。

文献[26]给出第一同态定理及其证明。

定理 1(第一同态定理) 函数 $h:S\to Sc$ 是同态的, 当且仅当 h 是一个 *map* 函数和一个 *reduce* 函数的复合运算, 即 h=*reduce*($\odot$)$\circ$*map* f , 其中函数 *map* 和 *reduce* 分别为:

$$\begin{aligned} &map\ f\ [x_1,x_2,\ldots,x_n]=[f\ (x_1),f(x_2),\ldots,f(x_n)]\\ &reduce\ (\odot)\ [x_1,x_2,\ldots,x_n]=x_1\odot x_2\odot\ldots\odot x_n \end{aligned} \tag{14}$$

定理 1 表明, 当一个函数是 *map* 和 *reduce* 函数的复合形式时, 其并行性是显而易见的, 因此它一定是同态的。上述同态都是函数可直接并行时存在的同态。然而, 大部分复杂算法问题的目标函数都是不可直接并行的, 此时应该修改代码使其满足同态, 具体分析过程如下。

文献[27]给出了定理 2 及其证明。

定理 2(列表同态定理) 若函数 $h:S\to Sc$ 是 $\odot$-同态的, 则它的左向和右向函数具有同样的连接算子 $\odot$。若函数 h 仅是左向或右向函数, 且存在对应的二元连接算子 $\oplus$ 或 $\otimes$, 则 h 是 $\oplus$-或 $\otimes$-同态函数。定理 2 表明当一个函数无论是左向或右向函数(即它是单路函数), 它都是列表同态的。

根据定义 6 和定理 1,2, 我们得到如下推论:

推论 1 若函数 $h:S\to Sc$ 可写成元组形式 $h=(f,\varphi)$, 且 $h(x\bullet y)=(f(x),\varphi(x))\odot(f(y),\varphi(y))$, 其中 φ 是计算 f 的辅助函数, 则 h 是同态的, 且原问题的解可由 f = *fst*$\circ$*reduce*($\odot$)$\circ$*map* h 投影得到, 其中 *fst* 函数返回元组的第一个元素。

证明:

$$\begin{aligned} h(x\bullet y)&=(f(x),\varphi(x))\odot(f(y),\varphi(y))\\ &=(f,\varphi)[x]\odot(f,\varphi)[y]=h(x)\bullet h(y) \end{aligned} \tag{15}$$

式(15)显然满足同态定理, 即函数 h 是同态的。

$$\begin{aligned} &fst\circ reduce(\odot)\circ map\ h\ [x\bullet y]\\ &=fst\circ reduce(\odot)\circ map\ (f,\varphi)\ [x\bullet y]\\ &=fst\circ reduce(\odot)\circ[(f(x),\varphi(x)),(f(y),\varphi(y))] \end{aligned}$$

$$\begin{aligned}&= fst \circ ((f(x),\varphi(x)) \odot (f(y),\varphi(y)))\\&= fst \circ ((f(x) \odot f(y)),(\varphi(x),\varphi(y)))\\&= f(x) \odot f(y)\\&= f(x \bullet y)\end{aligned} \tag{16}$$

推论 1 指出, 当目标函数 f 直接作用在列表 $x \bullet y$ 上不可并行时, 可添加辅助函数 φ 改写为元组形式 h, 使得算法连接函数满足同态定理, 然后通过算法连接函数来合并元组, 最终投影得到目标函数在该列表上的解。

数组最小和实例 根据式(13)中原问题的辅助函数和算法连接函数, 并结合推论 2, *mss* 与辅助函数 *mts*、*mps*、*sum* 一起构成元组 $h=(mss, mts, mps, sum)$, 则 $h(x \bullet y)$ 可扩展为以下形式:

$$\begin{aligned}&h(x \bullet y)\\&= (mss\ x, mts\ x, mps\ x, sum\ x) \odot (mss\ y, mts\ y, mps\ y, sum\ y)\\&= (mss\ x \odot mss\ y, mts\ x \odot mts\ y, mps\ x \odot mps\ y, sum\ x \odot sum\ y)\\&= (min(mss\ x, mss\ y, mts\ x + mps\ y), min(mts\ x + sum\ y, mts\ y), min(mps\ x, sum\ x + mps\ y), sum\ x + sum\ y)\end{aligned}$$

此时根据推论 1, 函数 h 显然是同态的, 最终可根据 $mss = fst \circ reduce(\odot) \circ map\ h$ 执行并行计算、连接和投影得到原问题的解。

5.2 Isabelle 自动验证

对于 4.3 节的 **ALGORITHM 2**, 采用 Isabelle 定理证明器来证明其满足同态定理, 即证明该算法中的算法连接函数可直接并行时满足定义 7, 不可直接并行时满足推论 1。

Isabelle 的证明过程是一个不断创建新理论文件(T.thy)的过程, 理论 T 既可以继承父理论 $S_1, S_2, \ldots, S_n$, 又可以定义新的类型、函数等, 并创建新的定理或引理进行证明。thy 文件由 ML 语言编写, 其基本结构如图 5 所示。

```
Theory T
    imports S1,S2,..,Sn
    begin
        Declaration  //声明部分
        Definition  //定义部分
        Proofs    //证明部分
end
```

图 5 thy 文件的基本结构

Figure 5 The basic structure of thy file

本文中, Declaration 用于声明同态定理的统一验证框架, Definition 用于定义求解问题需用到的相关算法函数, Proofs 用于验证目标函数的算法连接函数满足同态定理。

5.2.1 同态定理统一验证框架

Isabelle 标准库的区域(Locale)提供了一种程序的模块化和参数化机制, 它可以灵活地定义一个参数化的程序块, 且这些参数满足一定的逻辑条件[28]。因此, 本文使用区域技术在 Isabelle 的声明部分中定义了同态定理的统一验证框架, 描述同态定理验证的一般统一形式, 如图 6 所示。

```
locale BaseLoc =
  fixes Obj :: "'a list ⇒ 'a"
    and Aux :: "'a list ⇒ 'a"
    and ObjJoin :: "'a ⇒ 'a ⇒ 'a ⇒ 'a"
  assumes  HomObj : "Obj (a_l @ a_r) = ObjJoin (Aux a_l) (Obj a_l) (Aux a_r) (Obj a_r)"
```

图 6 同态定理统一验证框架的区域定义

Figure 6 The locale definition of unified verification framework for homomorphism theorem

图 6 定义了一个 *BaseLoc* 区域, 关键字 fixes 声明局部参数 *Obj*(目标函数)、*Aux*(辅助函数)和 *ObjJoin*(目标函数的算法连接函数), 5.2.2 节对其进行了具体实现。关键字 assumes 定义局部参数需满足的逻辑规约 *HomObj*, 即 *ObjJoin* 满足同态定理, 5.2.3 节给出了其具体验证过程。

实际上, 对于不同算法的 *Obj*、*Aux* 和 *ObjJoin*, 其参数类型、个数可能不一致, 而区域扩展和区域继承不支持对父区域 fix 定义函数的参数进行修改。但是, 对于具体问题的求解, 可使用区域继承技术对 *BaseLoc* 区域增加求解问题所需的局部参数和逻辑规约。

数组最小和实例 通过区域继承, 在 *BaseLoc* 基础上增加具体的目标函数 *Mss*, 辅助函数 *Mps*、*Mts* 和算法连接函数 *MssJoin*, 并增加新的逻辑规约 *HomMss*, 即 *MssJoin* 满足同态定理。最终得到新的区域 *MssLoc*, 如图 7 所示。

```
locale MssLoc = BaseLoc Obj Aux ObjJoin
  for Obj :: "'a list ⇒ 'a"
    and Aux :: "'a list ⇒ 'a"
    and ObjJoin :: "'a ⇒ 'a ⇒ 'a ⇒ 'a"
  +
  fixes Mss :: "int list ⇒ int"
    and Mps :: "int list ⇒ int"
    and Mts :: "int list ⇒ int"
    and MssJoin :: "int ⇒ int ⇒ int ⇒ int ⇒ int"
  assumes HomMss : "Mss (a_l @ a_r) = MssJoin (Mts a_l) (Mss a_l) (Mps a_r) (Mss a_r)"
```

图 7 *mss* 同态定理验证的区域定义

Figure 7 The locale definition of *mss* homomorphism theorem verification

5.2.2 算法函数定义

根据具体算法问题进行相应的区域继承, 需对该区域新增的局部参数和逻辑规约进行实例化, 之

后本节将对局部参数进行具体实现。

Isabelle标准库提供了递归函数定义命令primrec和非递归函数定义命令definition。因此, 在Isabelle的定义部分用这些命令实现新增局部参数(即具体问题的目标函数 f、辅助函数 φ 和算法连接函数 $\odot$)的功能。

数组最小和实例 根据图 7 的 *MssLoc* 区域, 对新增的局部参数 *Mss*、*Mps*、*Mts* 和 *MssJoin* 进行实例化, 具体实现步骤如下:

1) 用 definition 和 primrec 命令定义函数 *Mps*, *Mts* 和 *Mss*, 并定义其它相关函数 *Min* 和 *Sum*, 如图 8 所示;

```
definition Min :: "int ⇒ int ⇒ int"
  where "Min x y ≡ if x≤y then x else y"                                   (a)

primrec Sum :: "int list ⇒ int"
  where "Sum [] = 0" |
        "Sum (x#xs) = x + Sum xs"                                          (b)

primrec Mps :: "int list ⇒ int"
  where "Mps [] = 0" |
        "Mps (x#xs)
         = Min 0 (x + Mps xs)"                                             (c)

primrec Mts :: "int list ⇒ int"
  where "Mts [] = 0" |
        "Mts (x#xs)
         = Min (x + Sum xs) (Mts xs)"                                      (d)

primrec Mss :: "int list ⇒ int"
  where "Mss [] = 0" |
        "Mss (x#xs) = Min (x + Mps xs) (Mss xs)"                           (e)
```

图 8 相关函数定义

Figure 8 The definitions of related function

2) 根据 *MssLoc* 区域新增的局部参数 *MssJoin*, 用 definition 命令定义与其等价的算法连接函数 *MssJoin*。根据 4.2 节式(13)及 4.3 节 **ALGORITHM 2** 中 *Mss_join* 的 RECUR_parallel 部分, 再定义 *Sum*、*Mps* 和 *Mts* 的算法连接函数 *SumJoin*, *MpsJoin* 和 *MtsJoin*, 如图 9 所示。

```
definition SumJoin :: "int ⇒ int ⇒ int"
  where "SumJoin sum_l sum_r
         ≡ sum_l + sum_r"                                                  (a)

definition MpsJoin :: "int ⇒ int ⇒ int ⇒ int"
  where "MpsJoin sum_l mps_l mps_r
         ≡ Min (sum_l + mps_r) mps_l"                                      (b)

definition MtsJoin :: "int ⇒ int ⇒ int ⇒ int"
  where "MtsJoin mts_l sum_r mts_r ≡ Min (mts_l + sum_r) mts_r"            (c)

definition MssJoin :: "int ⇒ int ⇒ int ⇒ int ⇒ int"
  where "MssJoin mts_l mss_l mps_r mss_r ≡ Min (mts_l + mps_r) (Min mss_l mss_r)"  (d)
```

图 9 算法连接函数定义

Figure 9 The definitions of algorithmic join function

5.2.3 同态定理验证

定义算法函数之后, 需在 Isabelle 的证明部分对目标函数的算法连接函数满足同态定理(其数学形式为 $h(x \bullet y) = (f(x), \varphi(x)) \odot (f(y), \varphi(y))$)进行正确性证明, 即证明 *BaseLoc* 区域的逻辑规约 *HomObj* 的正确性。Isabelle 中采用 theorem 命令创建与 *HomObj* 等价的同态定理, 并通过如下策略完成证明[29]:

策略 5. 使用 apply 命令来引入 Isabelle 内置规则库的规则来进行化简。

策略 6. 创建相应的中间引理并进行证明,然后使该引理作为化简规则来进行化简。

策略 7. 使用 Sledgehammer 工具来调用其他定理证明器来辅助证明, 如 CVC4[30], SPASS[31]和 Z3[32]等。

数组最小和实例 根据 *MssLoc* 区域定义的逻辑规约 *HomMss*, 在 Isabelle 创建与其等价的同态定理 HomMss, 并证明其正确性。为此, 先根据策略 6 定义相应的辅助引理 HomSum, HomMps 和 HomMts, 再根据策略5和策略7, 使用apply命令和Sledgehammer工具对这些引理和定理进行结构化方式证明。具体的验证结构如图 10 所示。

图 10 定理和每个引理的结构化证明分为基础情况证明(*Nil*)和归纳情况(*Cons*)证明。对于基础情况, 另外创建引理 BasecaseSum, BasecaseMps, BasecaseMts 和 BasecaseMss 并对它们进行证明; 归纳情况可借由基础情况和其它辅助引理完成证明。图 11 给出了定理与最重要的辅助引理之间的依赖关系。

最终成功验证完定理和所有引理, 下面仅列出最为重要的算法连接函数 *MssJoin* 满足同态定理的结构化证明脚本, 限于篇幅其它从略。HomMss 作为需要验证的定理, 基础情况证明依赖于中间引理

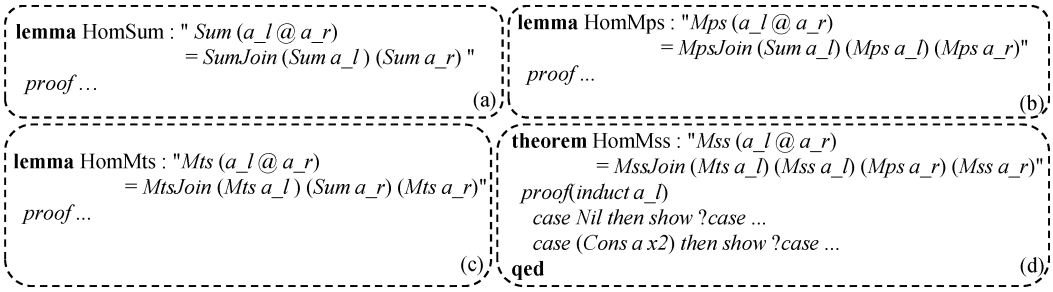

图 10 *mss* 的同态验证结构

Figure 10 The homomorphic verification structure of *mss*

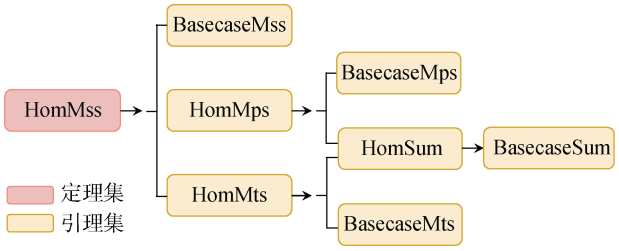

图 11 定理与辅助引理的依赖关系

Figure 11 Dependencies of theorems and auxiliary lemmas

BasecaseMss, 归纳情况证明需引入前面的函数定义(*MssJoin*, *MpsJoin*, *Min*)和引理(HomMps, HomMts), 如图 12 所示。

6 并行程序生成

为生成 Haskell 并行函数式程序, 本文提出了由 Radl$^+$并行算法转换为 Haskell 并行程序的转换规则, 并基于此规则设计了“Radl$^+$→Haskell 并行程序生成系统”的软件原型, 最终将映射生成的 Haskell 程序放至 GHC 平台运行成功。

6.1 Radl$^+$→Haskell 并行程序生成系统

至此, 我们已经实现了从问题的规约推导至 Radl$^+$并行算法。为了实现从抽象规约到具体程序的完整生成过程, 本文设计了一个“Radl$^+$→Haskell 并行程序生成系统”的软件原型, 将 Radl$^+$算法映射为 Haskell 并行函数式程序。软件原型系统的结构图如图 13 所示:

该软件原型的输入为 Radl$^+$描述的分治并行算法, 输出为 Haskell 并行函数式程序, 两者在数学上和逻辑上是等价的, 中间的转换过程基于如下转换规则:

转换规则(Radl$^+$→Haskell 转换规则)

- 使用 Radl$^+$描述的 ALGORITHM 转换为 Haskell

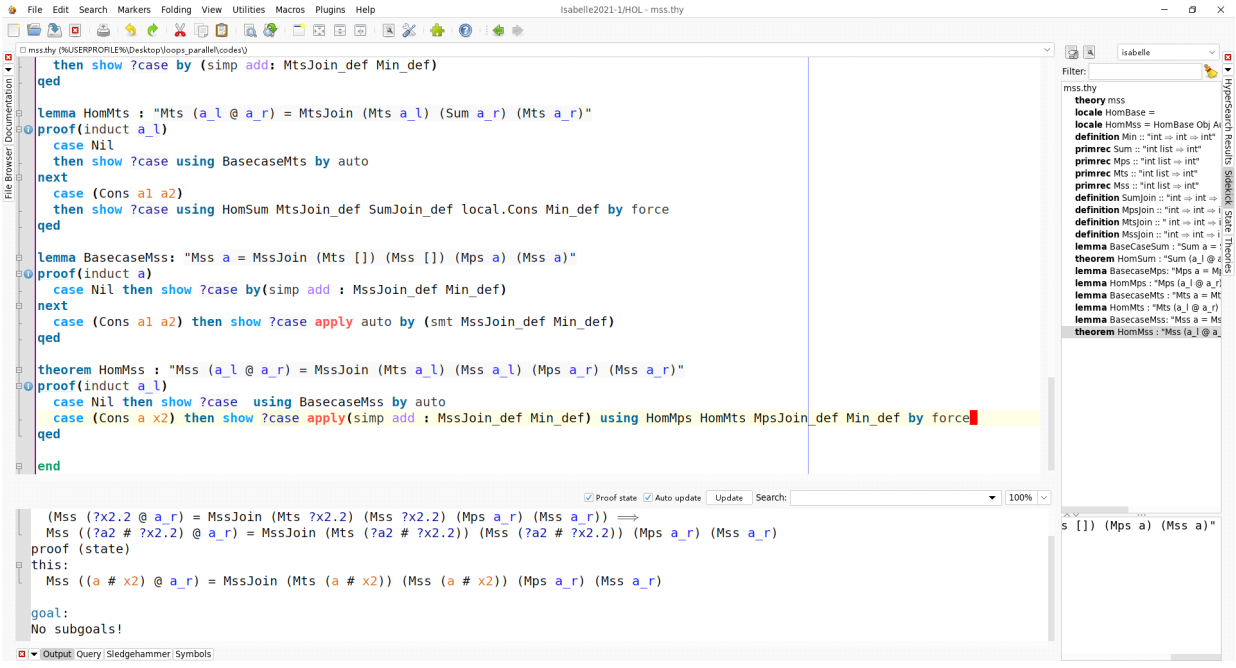

图 12 *mss* 的同态验证过程

Figure 12 The homomorphic verification process of *mss*

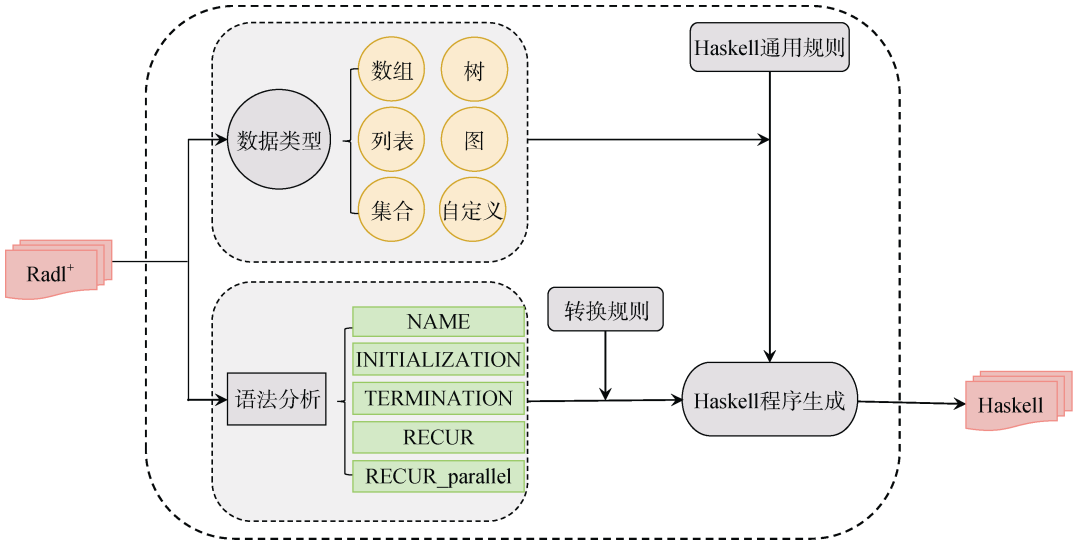

图 13 “Radl$^+$→Haskell 并行程序生成系统”结构图

Figure 13 The structure diagram of “Radl$^+$→Haskell Parallel Program Generation System”

的主程序 Main::IO(), 它是程序执行的起点;

- 使用 Radl$^+$描述的分划函数转换为 Haskell 的 *par* 函数, 表示两个线程的并行计算。如 *l* ‘*par*’ *r* 表示程序的主任务分成线程 *l* 和 *r* 并行计算, 且 *l* 和 *r* 分别递归产生更多的子线程;
- 使用 Radl$^+$描述的合并函数转换为 Haskell 的 *pseq* 函数, 表示一个线程的连续计算。当 *par* 运行完成后, 用 *pseq* 合并函数结果来进行同步。

数组最小和实例 使用 Radl$^+$描述的 **ALGORITHM 2** 作为“Radl$^+$→Haskell 并行程序生成系统”的输入, 在生成系统中进行语法分析, 其中四个函数 *sum*、*mps*、*mts* 和 *mss* 根据转换规则映射得到的 Haskell 函数定义, 如图 14 所示。Haskell 分治并行计算包 *Control.Parallel* 增加了 *par* 和 *pseq* 函数来声明函数的并行性。

最终经软件原型生成的 Haskell 并行函数式程序如图 15 所示。

6.2 Haskell 并行程序实现

GHC 作为 Haskell 的编译器, 支持在对称的共享内存多处理器(Symmetric Multi-Processing, SPM)上并行运行 Haskell 程序, 特别适合于分治并行。Haskell 程序默认在一个处理器上(即单线程)线性运行, 因此编写并行程序需向 GHC 声明并行性。一种方法是使用并行 Haskell 来分叉线程, 即使用多线程来执行程序任务。更简单的机制是从纯代码中提取并行性, 它使用一对组合函数 *par* 和 *pseq*[33], 组合语法是 *par l* (*pseq r* (*l* ⊙ *r*)) 或 *l'par'r'pseq'* (*l* ⊙ *r*)。GHC 在编译 Haskell 文件时需要-threaded

```
sum :: [Int] -> Int
sum [] = 0
sum [x] = x
sum a = (sum l) 'par' (sum r) 'pseq' (sum l + sum r)
 where
   mid = div (length a) 2
   l = take mid a
   r = drop mid a
```
(a)

```
mps :: [Int] -> Int
mps [] = 0
mps [x] = x
mps a = (mps l) 'par' (mps r) 'pseq' (minimum [mps l , sum l + mps r])
 where
   mid = div (length a) 2
   l = take mid a
   r = drop mid a
```
(b)

```
mts :: [Int] -> Int
mts [] = 0
mts [x] = x
mts a = (mts l) 'par' (mts r) 'pseq' (minimum [mts l + sum r , mts r])
 where
   mid = div (length a) 2
   l = take mid a
   r = drop mid a
```
(c)

```
mss :: [Int] -> Int
mss [] = 0
mss [x] = x
mss a = (mss l) 'par' (mss r) 'pseq' (minimum [mss l , mss r ,
          mts l + mps r])
 where
   mid = div (length a) 2
   l = take mid a
   r = drop mid a
```
(d)

图 14 Haskell 函数定义

Figure 14 Haskell function definitions

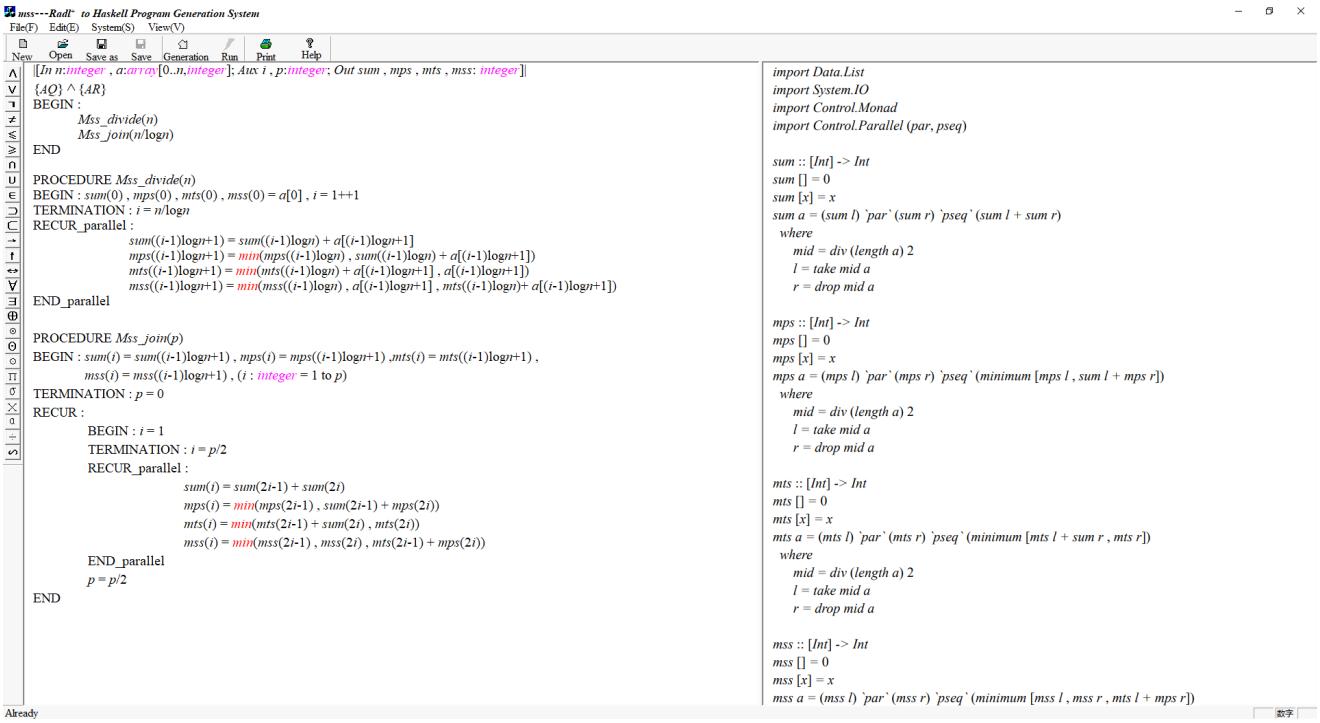

图 15　Haskell 并行程序生成

Figure 15　Haskell parallel program generation

参数才能支持并行, 生成可执行文件, 并通过参数 -N 来设置运行该文件的线程数量, 参数-s 使系统输出文件运行的统计结果。

数组最小和实例　将上一节生成的 Haskell 并行函数式程序放至 GHC 平台编译并运行, 对该算法输入一个包含 100 万个元素的列表, 使用两个线程 2536.486 秒就得到了该列表最小和的计算结果, 如图 16 所示。

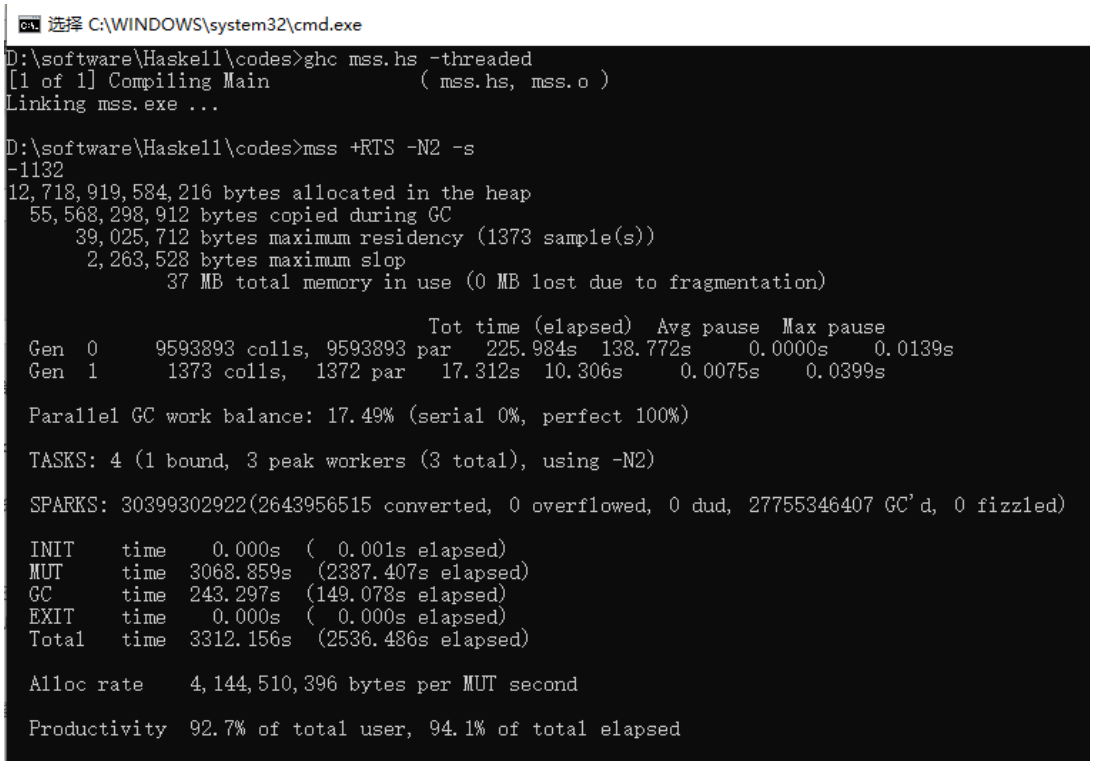

图 16　*mss* 的 Haskell 并行实现

Figure 16　Haskell parallel implementation of *mss*

7 实验

本节将通过软件原型生成的 Haskell 并行函数式程序放至 GHC 平台运行, 与串行 Haskell 程序的执行时间进行对比, 证明本文提出的方法具有良好的加速比。

7.1 实验数据与环境

根据第 2 节提出的模型驱动的分治并行程序生成及自动验证方法步骤, 我们对一系列复杂算法问题(包括可直接并行函数和不可直接并行函数)进行了程序生成和 Isabelle 验证, 其程序生成和自动验证方法与数组最小和案例一致, 限于篇幅不再赘述。这些复杂算法中包含的函数如下:

1) *min*, *max*, *sum*, *length*: 这些函数均为常见结构简单的可直接并行函数。其中 *min* 用于求解列表所有元素的最小值; *max* 用于求解列表所有元素的最大值; *sum* 用于求解列表所有元素的和; *length* 用于求解列表的长度, 即元素的个数。

2) *fac*, *poly*: 这些函数用于多项式求值问题。其中 *fac* 求解给定元素的 *length* 次幂, 它是并行计算 *poly* 的辅助函数, 它是可直接并行的; *poly* 用于列表的多项式求值, 它是不可直接并行的。

3) *mps*, *mts*, *mss*: 这些函数用于求解数组最小和问题, 它们都是不可直接并行的。其中 *mps* 计算列表的最小前缀和; *mts* 计算列表的最小后缀和; *mss* 计算列表所有子段的最大和。

4) *bal*, *mp*, *mt*, *mg*: 这些函数用于求解最大基因序列问题, 除 *bal* 外均为不可直接并行函数。其中 *bal* 判断序列是否为连续基因, 它是 *mp* 和 *mt* 的辅助函数; *mp* 标志以序列第一个元素开始的基因最长连续长度, 它是 *mt* 和 *mg* 的辅助函数; *mt* 标志以序列最后一个元素结束的基因最长连续长度, 它是 *mg* 的辅助函数; *mg* 标志整个序列中的基因最长连续长度。

在上述算法的函数中, 输入均采用文件导入方式, 该文件为包含 2 亿个随机数的 int 型列表。本文实验采用的是 Intel(R) Core(TM) CPU i9-9900K 服务器, 显卡内存为 32GB, 最大线程数量为 16, 操作系统为 Win10, 在 GHC 平台(版本为 9.2.2)上编译并运行经软件原型生成的 Haskell 并行程序。

7.2 实验结果与分析

本文在 GHC 平台上通过设置不同线程数量运行 Haskell 并行程序, 与 Haskell 串行程序(即一个线程)比较得到不同线程情况下运行时间的加速比, 如图 17 所示:

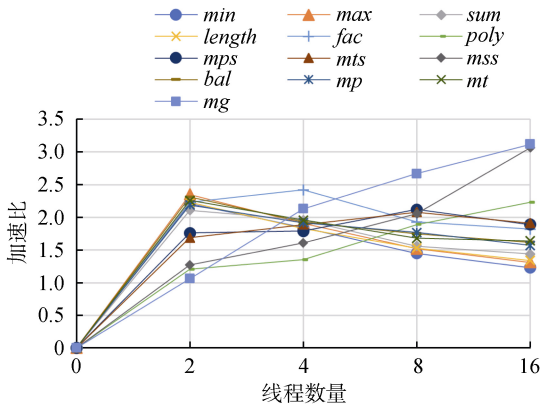

图 17 相对于串行程序的加速比

Figure 17 Speedups relative to the sequential program

理论上并行程序中利用的线程越多, 加速比越高。但实际并非完全如此, 根据图 17, 我们可以分两类进行讨论:

1) 常见函数(如 *min*、*max* 等)和一些辅助函数(如 *bal*、*mp* 等)使用 2 线程时加速比略高于理想状态下的加速比 2, 这种超线性加速比成因是数据量较大, 需要较大的内存, 而程序运行时选择多核处理器上的高速缓存, 其读写性能高于内存, 可以产生额外的加速效果。在 4 线程时并行性能较为稳定几乎不变, 之后在 8~16 线程时缓慢下降。这是因为这些函数的结构和计算过程较为简单, 高性能电脑能够以较少的线程数进行快速计算, 若线程数过多, 反而会造成多线程合并计算结果的时间过久, 加长线程之间的通信时间, 同时会增加大量计算内存空间, 从而使得加速比降低。

2) 对于三个复杂算法问题的目标函数 *mss*、*poly* 和 *mg*, 加速比一直在随着线程数量而增长, 在 16 线程时甚至达到了 3 倍。这是因为这些函数本身结构较为复杂, 调用的辅助函数过多(*mss* 和 *mg* 都有 3 个)或本身运算量大(*poly* 一直在做幂运算, 使得数值增长速度不断加快), 采用的线程越多, 才能加快多线程对函数的计算时间和合并时间, 使得加速比不断增加。

8 相关工作

并行程序的设计与验证已经有较多工作进行了深入研究, 本文通过以下两个方面与较为经典的相关工作进行比较:

分治并行程序设计: Gorlatch S[34]提出了一种系统的 *cons*&*snoc* 并行化方法, 将问题抽象为函数并给

定其列表同态[35]表示, 该方法可成功应用于已知的所有同态, 但辅助函数直接给出, 没有推导过程, 且生成的并行算法不完整。Kazutaka Morita 等人[9]基于第三同态定理[36], 通过函数的弱右逆实现来自动推导出代价最优的列表同态函数, 简化了列表同态的推导, 最终生成可执行的C++并行程序, 但弱右逆函数的推导过程过于复杂, 没有统一的推导规则。Azadeh Farzan 等人[37-39]利用语法引导合成(Syntax-guided synthesis, 简称 SyGuS)的方法, 研究了对单重和双重循环结构的串行程序进行分治并行化, 最终生成C++并行程序, 并提出了使用Dafny程序证明器验证并行程序满足同态定理的方法, 但该工作的串行程序都是通过经验直接给出, 辅助函数的推导过程也过于简略。熊英飞等人[40-41]基于演绎法与归纳法实现了算法合成工具 *AutoLifter*, 不仅能够自动合成分治并行算法, 还可以推广到其它算法策略(如动态规划), 在 *AutoLifter* 合成算法正确性方面, 采用 Occam 求解器进行概率保证。

并行程序验证: 主要分为两类, 一类是基于模型检验的方法, 如基于 Petri 网的验证方法[42]、模型检测方法(如 SPIN[43]、SMV[44]等)等。Petri 网能够直接反映死锁和互斥性质, 但关于内容的形式化表示有限,如缺乏描述变量变化的方式[45]; 模型检测方法在面对复杂问题时往往会产生状态爆炸问题。另一类是基于定理证明的方法, 如 Rely-Guarantee 推理方法[46]和基于并发分离逻辑的验证方法[47]等。Rely-Guarantee 推理方法是一种容错(Fault-tolerance)的验证方法, 但证明过程复杂, 验证时所需的依赖-卫式条件难以获得; 并发分离逻辑方法是一种验证并行程序性质较为经典的方法, 它是一种避错(Fault-avoidance)的验证方法, 限制条件高, 由于可能涉及到指针操作, 验证抽象粒度小, 验证效率低。行为时序逻辑 TLA$^+$方法[48]既支持模型检验, 也支持定理证明, 但定理证明方面目前仅支持安全性的验证, 不支持活性的验证。

本文首次实现了从问题规约推导至串行算法, 再提升至并行算法并通过 Isabelle 验证其满足同态定理的可并行性, 最终生成分治并行 Haskell 函数式程序的完整过程。与现有的主要研究工作进行对比, 在生成方面, 该方法的串行算法是通过推导得到的, 而非经验给出, 算法连接函数和辅助函数生成的输入只需要串行算法, 生成过程更加系统和一般化; 在验证方面, 使用 Isabelle 定理证明器对提升后的算法满足同态定理自动验证, 证明该算法的可并行性, 保障了并行算法的可靠性和正确性。使用定理证明器 Isabelle 较 Dafny 通用性和复用性更强, 也无须在验证时探索前后置条件以及大量的中间断言。最终生成的是 Haskell 并行函数式程序, 更加简洁、清晰, 不会产生副作用, 天然支持并行。

9 总结与展望

随着大数据和人工智能的发展, 高可信并行算法的生成及验证已经逐渐成为计算机科学非常热门的一个研究方向。本文采用分治并行模式, 并结合形式化推导方法提出了一种基于模型驱动的分治并行函数式程序生成及自动验证方法。(1)首先, 从问题描述出发, 使用分划递推法来寻找函数之间的递推关系及循环不变式, 将递推关系和循环不变式组合构造串行算法; (2)其次, 扩展循环不变式且采用归纳法推导出辅助函数和算法连接函数来将其提升为并行算法, 并使用一种新的并行算法设计语言 Radl$^+$来描述; (3)然后, 使用 Isabelle 定理证明器对算法连接函数满足同态定理进行证明, 保障了并行算法的可靠性和正确性; (4)最后, 基于提出的“Radl$^+$→Haskell 转换规则”, 将验证成功的 Radl$^+$并行算法通过设计的“Radl$^+$→Haskell 并行程序生成系统”软件原型映射为具体的 Haskell 并行函数式程序。实验表明, 本文提出的方法能够对多项式求值、最大基因序列等一系列算法问题生成并行函数式程序, 并具有良好的加速比。

在并行程序生成方面, 本文是通过形式化方法来进行推导的, 生成过程更加系统和一般化, 同时生成的函数式程序具有良好的扩展性和天然的并行性; 在并行程序验证方面, 通过定理证明器自动验证, 保障了生成的并行程序的可靠性并提高了验证效率。使用 Isabelle 定理证明器验证算法连接函数满足同态定理, 即提升后的串行算法满足可并行性来实现并行程序的验证, 是一种新的验证思路, 验证相对更加简单。

在未来工作中, 一是深入挖掘整体函数同态性与单步递推计算函数的结合律、单位元性质之间的联系, 探索某特定子类问题验证的统一求解策略; 二是考虑将本文方法扩展到双重循环程序, 使其能够生成更多复杂算法问题的并行程序; 三是将本文方法运用到真实工业应用场景, 使其能够生成和验证更有意义的并行算法, 如大数据的分布式计算框架 MapReduce[49]和 Spark[50]。

参考文献

[1] Bentley J L. Multidimensional Divide-and-Conquer[J]. *Communi-*

cations of the ACM, 1980, 23(4): 214-229.

[2] Valiant L G. A bridging model for parallel computation[J]. *Communications of the ACM*, 1990, 33(8): 103-111.

[3] Wang J, Zhan N J, Feng X Y, et al. Overview of Formal Methods[J]. *Journal of Software*, 2019, 30(1): 33-61.
(王戟, 詹乃军, 冯新宇, 等. 形式化方法概貌[J]. *软件学报*, 2019, 30(1): 33-61.)

[4] Huet G, Kahn G, Paulin-Mohring C. The coq proof assistant a tutorial[J]. *Rapport Technique*, 1997, 178.

[5] Leino K R M. Developing Verified Programs with Dafny[C]. *2013 35th International Conference on Software Engineering*, 2013: 1488-1490.

[6] The Isabelle proof assistant. https://isabelle.in.tum.de/.

[7] Jiang N, Li QA, Wang LM, at el. Overview on mechanized theorem proving[J]. *Ruan Jian Xue Bao/Journal of Software*, 2020,31(1):82-112.
(江南, 李清安, 汪吕蒙, 等. 机械化定理证明研究综述[J]. *软件学报*, 2020, 31(1): 82-112.)

[8] Wenzel M. Shared-Memory Multiprocessing for Interactive Theorem Proving[C]. *International Conference on Interactive Theorem Proving*, 2013: 418-434.

[9] Morita K, Morihata A, Matsuzaki K, et al. Automatic Inversion Generates Divide-and-Conquer Parallel Programs[J]. *ACM SIGPLAN Notices*, 2007, 42(6): 146-155.

[10] Radoi C, Fink S J, Rabbah R, et al. Translating Imperative Code to MapReduce[C]. *The 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014: 909–927.

[11] Smith C, Albarghouthi A. MapReduce Program Synthesis[J]. *ACM SIGPLAN Notices*, 2016, 51(6): 326-340.

[12] Ahmadi R, Nummenmaa J. Automatic Verification of Dafny Programs with Traits[C]. *The 17th Workshop on Formal Techniques for Java-like Programs*, 2015: 1–5.

[13] Peña R. An Assertional Proof of Red–Black Trees Using Dafny[J]. *Journal of Automated Reasoning*, 2020, 64(4): 767-791.

[14] Pastor O, España S, Panach J I, et al. Model-Driven Development[J]. *Informatik-Spektrum*, 2008, 31(5): 394-407.

[15] Xue J Y, Zheng Y J, Hu Q M, et al. PAR: A Practicable Formal Method and Its Supporting Platform[C]. *International Conference on Formal Engineering Methods*, 2018: 70-86.

[16] Xue J Y. Two New Strategies for Developing Loop Invariants and Their Applications[J]. *Journal of Computer Science and Technology*, 1993, 8(2): 147-154.

[17] Geser A, Gorlatch S. Parallelizing Functional Programs by Generalization[J]. *Journal of Functional Programming*, 1999, 9(6): 649–673.

[18] Wang Y H, Li X. Neural-Guided Inductive Synthesis of Functional Programs on List Manipulation by Offline Supervised Learning[J]. *IEEE Access*, 2021, 9: 71521-71534.

[19] Bird R. Introduction to functional programming using haskell[M]. 2nd ed. London: Prentice Hall Europe, 1998.

[20] Wang C J, Xue J Y. Research on Relative Correctness of Radl Formal Specification[J]. *Journal of Software*, 2013, 24(4): 715-729.
(王昌晶, 薛锦云. Radl 形式规格说明相对正确性研究[J]. *软件学报*, 2013, 24(4): 715-729.)

[21] Hu Z J, Iwasaki H, Takechi M. Formal Derivation of Efficient Parallel Programs by Construction of List Homomorphisms[J]. *ACM Transactions on Programming Languages and Systems*, 1997, 19(3): 444–461.

[22] Nita S L, Mihailescu M. GHC[M]. Haskell Quick Syntax Reference. Berkeley, CA: Apress, 2019: 13-18.

[23] Floyd R W. Assigning Meanings to Programs[M]. Colburn TR, Fetzer JH, Rankin TL. Program Verification. Dordrecht: Springer, 1993: 65-81.

[24] Hoare C A R. An Axiomatic Basis for Computer Programming[J]. *Communications of the ACM*, 1969, 12(10): 576-580.

[25] Xavier C, Iyengar S S. Introduction to Parallel Algorithms[M]. New York: Wiley, 1998.

[26] Bird R S. An Introduction to the Theory of Lists[C]. *Broy M. Logic of Programming and Calculi of Discrete Design*, 1987: 5-42.

[27] Gorlatch S. Extracting and Implementing List Homomorphisms in Parallel Program Development[J]. *Science of Computer Programming*, 1999, 33(1): 1-27.

[28] Zhao YW. Functional Programming and Proof. Electronic textbook, 2021. https://www.yuque.com/zhaoyongwang/fpp/.

[29] Nipkow T, Paulson L C, Wenzel M. Isabelle/HOL: a proof assistant for higher-order logic[M]. Berlin: Springer, 2002.

[30] CVC4. http://cvc4.cs.stanford.edu/web/.

[31] Teucke A, Weidenbach C. SPASS-AR: A First-Order Theorem Prover Based on Approximation-Refinement into the Monadic Shallow Linear Fragment[J]. *Journal of Automated Reasoning*, 2020, 64(3): 611-640.

[32] Z3. https://github.com/Z3Prover/z3/.

[33] Loogen R. Eden – Parallel Functional Programming with Haskell[M]. Central European Functional Programming School. Berlin, Heidelberg: Springer, 2012: 142-206.

[34] Gorlatch S. Systematic Extraction and Implementation of Divide-and-Conquer Parallelism[M]. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996: 274-288.

[35] Cole M I. Parallel Programming with List Homomorphisms[J]. *Parallel Processing Letters*, 1995, 5(2): 191-203.

[36] Gibbons J. Functional Pearls[J]. *Journal of Functional Programming*, 1996, 6(4): 657-665.

[37] Farzan A, Nicolet V. Synthesis of Divide and Conquer Parallelism for Loops[J]. *ACM SIGPLAN Notices*, 2017, 52(6): 540-555.

[38] Farzan A, Nicolet V. Modular Divide-and-Conquer Parallelization of Nested Loops[C]. *The 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019: 610–624.

[39] Farzan A, Nicolet V. Phased Synthesis of Divide and Conquer Programs[C]. *The 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021: 974–986.

[40] Ji R Y, Xiong Y F, Hu Z J. Black-Box Algorithm Synthesis — Divide-and-Conquer and more[EB/OL]. 2022: arXiv: 2202.12193. https://arxiv.org/abs/2202.12193.

[41] Ji R Y, Zhu T R, Xiong Y F, et al. Synthesizing Efficient Dynamic Programming Algorithms[EB/OL]. 2022: arXiv: 2202.12208. https://arxiv.org/abs/2202.12208.

[42] Ding L, Dong L D, Piao Y. Deadlock prevention policy of concurrent programming base on Petri net[J]. *Journal of Zhejiang University*(*Science Edition*), 2012, 39(1):43-26.

[43] Holzmann G J. The Model Checker SPIN[J]. *IEEE Transactions on Software Engineering*, 1997, 23(5): 279–295.

[44] Witkowski T, Blanc N, Kroening D, et al. Model Checking Concurrent Linux Device Drivers[C]. *The 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007: 501–504.

[45] Petri C, Reisig W. Petri Net[J]. *Scholarpedia*, 2008, 3(4): 6477.

[46] Jones C B. Tentative Steps Toward a Development Method for Interfering Programs[J]. *ACM Transactions on Programming Languages and Systems*, 1983, 5(4): 596–619.

[47] Brookes S, O'Hearn P W. Concurrent Separation Logic[J]. *ACM SIGLOG News*, 2016, 3(3): 47-65.

[48] Cousineau D, Doligez D, Lamport L, et al. TLA+ Proofs[C]. *International Symposium on Formal Methods*, 2012: 147-154.

[49] Dean J, Ghemawat S. MapReduce[J]. *Communications of the ACM*, 2008, 51(1): 107-113.

[50] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster computing with working sets[C]. *2nd USENIX Workshop on Hot Topics in Cloud Computing*. 2010.

王昌晶 于 2012 年在中国科学院软件研究所计算机软件与理论专业获得博士学位。现任江西师范大学教授、博士生导师。研究领域为可信计算、软件安全。Email: wcj@jxnu.edu.cn

潘丞 于 2021 年在江西农业大学计算机科学与技术专业获得学士学位。现在江西师范大学计算机技术专业攻读硕士学位。研究领域为可信计算、并行计算。Email: panc86@jxnu.edu.cn

左正康 于 2013 年在中国科学院大学计算机软件与理论专业获得博士学位。现任江西师范大学副教授。研究领域包括软件形式化方法、安全程序设计。Email: zuo803@jxnu.edu.cn

王忠文 于 2020 年在江西师范大学软件工程专业获得学士学位。现在江西师范大学计算机技术专业攻读硕士学位。研究领域为可信计算、并行计算。Email: 202041600030@jxnu.edu.cn

黄箐 于 2018 年在武汉大学计算机软件与理论专业获得工学博士学位。现任江西师范大学计算机信息工程学院副教授。研究领域为服务软件工程的大数据安全。研究兴趣包括: 软件错误定位、软件错误分析。Email: qh@whu.edu.cn