学校代号 10532　　学　　号 XXX

分 类 号 TP391　　密　　级 普 通

湖南大學

HUNAN UNIVERSITY

工程硕士学位论文

基于深度学习的程序并行化方法研究

学位申请人姓名 XXX

培 养 单 位 信息科学与工程学院

导师姓名及职称 XXX 教授 XXX 高工

学 科 专 业 计算机技术

研 究 方 向 并行计算

论 文 提 交 日 期 2020 年 04 月

学 校 代 号：10532

学　　号：XXX

密　　级：普 通

湖南大学硕士学位论文

基于深度学习的程序并行化方法研究

学位申请人姓名：XXX

导师姓名及职称：XXX 教授 XXX 高工

培 养 单 位：信息科学与工程学院

专 业 名 称：计算机技术

论 文 提 交 日 期：2020 年 04 月

论 文 答 辩 日 期：2020 年 05 月

答辩委员会主席：XXX 教授

Research on Program Parallelization Method Based on Deep Learning

by

XXX

B.E.(Bohai University) 2017

A thesis submitted in partial satisfaction of the

Requirements for the degree of

Master of Engineering

in

Computer Technology

in the

Graduate School

of

Hunan University

Supervisor

Professor XXX

Senior Engineer XXX

April, 2020

湖 南 大 学

学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写的成果作品。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律后果由本人承担。

作者签名：　　日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权湖南大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于

1、保密□，在____年解密后适用本授权书。

2、不保密□。

(请在以上相应方框内打“√”)

作者签名：　　日期：　　年　　月　　日

导师签名：　　日期：　　年　　月　　日

摘　要

随着高性能计算技术的快速发展，并行编程技术在实际工程应用中占据着越来越重要的地位。尤其是在科学计算相关领域，并行化技术成为了解决程序结构复杂、计算量大、数据密集以及执行周期长等问题的关键途径。然而，现有的程序并行性分析工具在处理具备上述特征的程序时仍存在着较大的局限性，且目前的多种并行编程手段在使用时缺少统一的标准编程接口。因此，串行程序的并行化在程序的并行性识别和并行程序开发两个阶段均存在着较大的难度，研究一种在上述两个阶段均具备良好表现的串行程序并行化方法是十分必要的。

本文借助深度学习技术来研究串行程序的并行化方法，提出了包含数据集、识别方法、标记语言和辅助平台的一整套解决方案，主要工作如下：

(1)设计并构造了一个通用图数据集 GFCPD(Graphs For Code Parallelism Discovery，程序并行性识别图数据集)。本文使用程序的上下文流图(Contextual Flow Graph，XFG)作为代码语义的表示形式，构造了一个可以用于串行程序并行性识别任务的通用图数据集 GFCPD，同时设计并实现了一套生成 GFCPD 数据的自动化脚本。

(2)提出了一种串行程序并行性识别的深度学习模型。本文将串行程序的并行性识别视为二元分类问题，建立了一个基于深度图卷积神经网络(Deep Graph Convolutional Neural Network，DGCNN)的深度学习模型，并通过该模型在 GFCPD 数据集上的表现验证了深度学习方法在串行程序的并行性识别问题上具备可行性与有效性。

(3)设计并实现了一种并行标记语言 PML。本文使用 XML 技术实现了并行标记语言 PML，为不同的并行编程模型提供了格式统一的 PML 标签用于结合 C/C++ 语言进行并行程序开发，为多种并行编程手段的混合使用提供了统一的标准编程接口和更加简易便捷的途径。

(4)搭建了一个并行编程辅助平台。为了对上述工作成果进行集成和转化，本文搭建了一个并行编程辅助平台，从串行程序的并行性识别、并行程序开发、并行程序调试三个方面提供了完整的串行程序并行化流程，有效地降低了并行化技术的使用门槛。

关键词：并行编程；并行性识别；并行标记语言；深度学习；图卷积神经网络

Abstract

With the rapid development of high-performance computing technology, parallel programming technology plays an increasingly important role in practical engineering applications. Especially in the field of scientific computing, parallelization technology has become a key way to solve the problems of complex program structure, large amount of calculation, data intensive and long execution cycle. However, the existing program parallelism analysis tools still have greater limitations when processing programs with the above characteristics, and the current multiple parallel programming methods lack a unified standard programming interface when used. Therefore, the parallelization of sequential programs has great difficulty in both the parallelism discovery and parallel program development. It is necessary to do a research on parallelization method of sequential programs which has good performance in the above two stages.

The purpose of this paper is to explore the parallelization method of sequential programs with the help of deep learning technology, and proposes a set of solutions including data sets, recognition methods, markup languages and auxiliary platforms. The main work is as follows:

(1) Design and construct a general graph data set GFCPD (Graphs For Code Parallelism Discovery). In this paper, the program's Contextual Flow Graph (XFG) is used as the semantic representation of the code. A general graph data set GFCPD that can be used for discovering parallelism in sequential programs is constructed, and a set of automated scripts for generating GFCPD data was designed and implemented.

(2) A deep learning model for sequential program parallelism discovery is proposed. In this paper, the parallelism discovery of sequential programs is regarded as a binary classification problem. A deep learning model based on Deep Graph Convolutional Neural Network (DGCNN) is built, and the model is used on the GFCPD dataset. The performance verifies that the deep learning method has the feasibility and effectiveness in the sequential program parallelism discovery problem.

(3) Design and implement a parallel markup language PML. This paper uses XML technology to implement the parallel markup language PML which provides unified formatted PML tags for different parallel programming models. Providing a unified standard programming interface and a simpler and more convenient way for the mixed use of a variety of parallel programming methods.

(4) A parallel programming auxiliary platform is built. In order to integrate and transform the above work results, this paper builds a parallel programming auxiliary platform, which provides a complete sequential program parallelization process from the three aspects of serial program parallelism discovery, parallel program development, and parallel program debugging. It has lowered the threshold of parallelization technology.

Key Words: parallel programming; parallelism discovering; markup language for parallel programming; deep learning; graph convolutional neural network

目录

插图索引

附表索引

第 1 章 绪论

1.1 研究背景和意义

近十年以来，半导体的发展趋势表明，过去几十年来时钟速度的惊人提高已经结束，单核处理器对计算机应用性能提升和成本降低的贡献达到了极限[1]，但各个领域的发展仍然需要计算机性能的进一步提升。这推动了硬件和架构的设计理念向集成多处理器设计的转变[2]。近年来，多核系统越来越受欢迎，并行化已经成为了新的趋势，高性能计算领域也正在蓬勃发展。然而，由于串行的编程思想仍然深深地根植于当前的应用程序开发流程中，尽管大多数硬件设备(如多核处理器、GPU 等)均具备为并行计算而设计的设想，但目前大部分流行的应用程序仍然是串行的，未能充分利用多核技术带来的潜在性能提升[3]。换言之，在大多数情况下，现代体系结构提供的很大一部分计算资源并没有被充分利用。

针对这一现状，并行程序开发的新方法和新技术正在逐步发展起来。例如，众多程序并行性分析工具的提出以及 OpenMP、Click、MPI 以及 CUDA 等标准不一、接口各异的并行编程模型和框架的出现，使得并行程序开发的方法更加便捷和多样化[4]。从某种意义上来说，在生产科学和工业领域广泛遇到的数据密集型应用，只需通过对其代码中的并行性进行全面分析，将其中可以并行化的部分利用上述手段进行并行化改造，即可达到大幅提高应用性能和计算资源利用率的目的[5]。但需要说明的是，一种工具或手段并不能很好的完成这一目标，目前的并行程序开发过程中，往往需要在不同阶段结合应用不同的并行编程技术，以使得程序发挥更好的性能，计算资源得到更好的利用。

这些阶段主要包括：(1)在编码阶段，采用人工的方式根据实际硬件平台的特点和算法的并行性，使用上述并行编程框架在编码过程中对算法中可以并行化的步骤直接采取并行化开发，以达到使代码并行执行的目的。在这一阶段可以很好的将硬件平台优势、开发人员根据其行业经验进行的主观判断以及算法的特征结合在一起，但对于大型的应用程序，本阶段的并行程序开发任务会变得十分繁琐和复杂；(2)在编译阶段，通过编译器优化功能将程序自动并行化，这一手段的重心主要集中在程序中的循环片段上，编译器自动地识别出循环片段并通过静态分析技术分析其中的依赖关系，进而判断该循环片段能否进行并行优化，然后对可以并行化的循环片段自动生成并行化的机器代码。这一过程使开发人员摆脱了手动并行化的繁琐过程，更加准确、快速地发现了串行程序的并行潜力，但优化以后的程序不具备可读性且性能会因编译器的选择、硬件平台的架构和程序本身的特征而各有不同；(3)在代码重构阶段，使用程序并行性分析工具通过程序的静态

分析或动态分析技术，自动识别串行程序中可以并行的片段而不进行优化，仅对开发人员进行指导建议，结合人工的方式半自动的将串行代码转换为并行代码。显然在阶段(3)对程序进行并行化改造结合了(1)、(2)两个阶段的优势，且在较大程度上避免了其短板，然而该阶段在实际操作中仍具备着很大的困难。如有的程序并行性分析工具使用的分析算法对程序的部分特征，尤其是循环体中被迭代语句的数学表示存在限制；还有的则需要在程序的执行过程中对程序的动态特征进行收集和分析。这些限制对于程序复杂、计算量大、数据密集并且执行时间长的应用程序并不友好。

由此可见，虽然高性能计算技术正在逐渐发展、成熟和完善，但熟练地进行并行程序的开发仍然不是一项简单的任务。首先，它对于开发工具和硬件平台存在着很大的依赖，从编程、调试到性能分析都需要付出大量的劳动。其次，它对开发人员的要求较高，需要具备丰富的计算机知识、对并行编程理论技术的深入理解和对计算机程序特征的准确把握等。但这并没有阻挡高性能计算技术因其经济且高效的特点成为解决计算量大、数据密集型问题的主要途径，诸如医疗成像、气候模拟、药物发现、地质能源勘探等生产和科研相关领域的从业人员为了提高工作效率与质量，不断地尝试通过并行编程技术针对一些优秀的理论以及数据处理算法进行试验与改进，并且在实践过程中出现了在大型集群系统(如：超级计算机)中同时使用两种或多种并行编程模型和框架的趋势。然而由于专业知识的限制、程序并行性分析工具的局限性以及多种并行编程模型和框架在结合使用的同时缺少统一的标准编程接口，使得上述相关领域的从业人员在串行程序的并行性识别和并行程序开发方面均存在着较大的困难。这直接导致了行业经验、专业技能、数据处理需求与实际处理效率之间的不对称关系。因此，上述众多领域迫切的需要更加通用快捷的串行程序并行性识别工具和更加统一易用的并行编程标准接口，来降低并行编程的技术门槛，提高数据处理的效率，避免计算资源的浪费。

不难发现，上述两大需求中，如何更好的识别串行程序的并行性，即从串行程序中识别出可以并行化的程序片段，才是更好的进行并行程序开发的前提和基础。关于串行程序的并行性识别，传统的做法是基于程序的静态分析技术或动态分析技术，收集程序的语义信息、相关语句的数学表示以及程序的运行时特征，采用线性整数规划或逻辑回归策略，重点关注和分析程序中循环片段的并行性。但传统手段由于相关数学理论限制或收集信息的时间问题，在处理结构复杂、计算量大、数据密集并且运行时间长的应用程序时表现并不理想，而上述特点正是生产和科研领域实际应用程序的普遍特征。如何最大程度的打破传统方法的局限性，提供快捷、准确并且简单易用的串行程序的并行性识别方法，是高性能计算领域的一大重点研究方向。

近几年来，深度学习技术在自然语言处理(NLP)领域得到了广泛应用并取得

了巨大的成功[6]，这推动了 NLP 技术在程序源码分析领域的应用和推广。众所周知，自然语言拥有词、短语、句子、段落、文章的层级结构；类似的，程序源码拥有关键字、表达式、选择/分支/循环结构、函数、再到整个程序的层级结构。因此，可以将程序源码认为是计算机世界的自然语言。研究者们从代码的语义信息出发，通过对代码的抽象语法树(AST)、中间表示(IR)、编译后的二进制文件、控制流程图(CFG)、数据流程图(DFG)等各种表示形式进行词嵌入相关的研究，提供了将程序源码转换为深度学习所需的数据组织形式——向量的不同方法。并通过不同的深度学习模型完成了诸如变量名/函数名预测、代码补全、恶意代码检测、内存溢出风险检测、算法程序分类等众多源码分析任务且效果显著。可见，深度学习在程序源码分析领域同样具有巨大的潜力与活力。对于识别串行程序的并行性而言,深度学习的方法直接以程序源码为数据,无需深入分析程序的静态语义，无需收集程序的运行时特征，在很大程度上打破了传统的程序并行性识别方法的局限性。使用深度学习的方法自动识别串行程序的并行性，是一个全新的研究方向。

1.2 国内外研究现状

如 1.1 节所述，串行程序的并行性识别，是并行程序开发中的首要环节也是本文的重点研究内容之一。深度学习技术在源码分析领域的应用经验，也为将深度学习的方法应用至串行程序的并行性识别研究提供了可行性参考。本节将分别从国内和国外两个方面列举并介绍近年来外关于串行程序并行性识别的相关研究和深度学习技术在程序源码分析领域的相关研究。

1.2.1 国内研究现状

在国内，程序的并行性分析是一个老生常谈的问题。2002 年，乐晓波、汪琳等人将 Petri 网相关技术应用到循环的并行性分析中来。在数据相关性的分析过程中使用 Petri 网理论,从全局的观点出发，分析同一数据与所有语句的相关性,从而提出了将串行程序转换为并行程序的有效方法[7]。2007 年，蒋作、高毅结合相关并行技术与定义，提出了能够把串行程序中可以并行执行的片段识别出来的 PARALLELIZING 算法[8]。2009 年，梁博、安虹等人详细分析了线程级推测技术在子程序结构上的适用性并提出了其判定依据，通过动态剖析工具对该判定依据进行了详细的实验分析和论证[9]。2010 年，闫昭、刘磊对串行程序中的基本块进行了数据依赖分析，提出了一种自动并行化方法，对不存在数据依赖的基本块进行并行优化，较大程度地解决了串行程序在并行计算机系统中的运行效率问题[10]。2011 年，郭慎、李培峰等人利用支持向量机对程序的静态特征和动态特征进行特征挖掘，提出了一种基于特征的程序并行性识别方法 [11]。2013 年，王磊、曲卫

平、李敬兆研究了自动并行化中的前端分析技术,引入并改进了人工智能搜索方法,实现了在评估函数的支持下进行广度搜索和深度搜索的方法[12]。2017 年，王家龙、刘艳红、沈立面向一个高效的软件 TLS 模型 HEUSPEC,研究了代码自动生成工具 C2H 的设计与实现方法。该方法已在开源编译器 Clang 上实现，基于该方法对 Rodinia、OmpScr 等基准程序的测试结果表明,C2H 能够将带有简单标注语句的串行 C 代码转换为 HEUSPEC 并行代码,且其性能与手工编写的 HEUSPEC 并行代码的性能十分接近[13]。2018 年，王时雨、张胜兵等人基于 LLVM 编译架构，提出了一种针对图像处理类程序的可并行性分析方法，从程序中的数据依赖关系和可并行特征两个方面对程序中的循环进行分析，实现了图像处理类程序的自动并行分类[14]。将深度学习的方法应用至源码分析研究在国内则是一个较新的研究方方向，2019 年，谭丁武、张坤芳等人提出了一种程序的中间表示——ESAST 图，并基于该图提出门控注意力图神经网络模型，对给定的源码程序进行了高精度的算法分类[15]。

1.2.2 国外研究现状

国外对于程序的并行性分析更趋向于编译优化、提出软件架构和源到源转换等方法。2008 年，Uday Bondhugula、J. Ramanujam 提出了一种基于 Polyhedral 多面体模型的整数线性优化算法，并由该算法驱动实现程序源到源转换框架 Pluto，该框架可以发现串行程序中的并行性和局部可优化部分，并提供向 C/C++源码中加入 OpenMP 标记的功能[16]。这是对 Polyhedral 模型的一次优秀实践，与其他相关研究[17~20]一起证明了 Polyhedral 模型对于程序优化的实用性。2012 年，Javier Diaz 等人调查和分析了当时常见的并行编程模型和工具对高性能计算领域的适用性，发现混合并行编程可以更好地利用计算机集群功能，而异构编程也由于多核 CPU+GPU 架构的流行而越来越普遍[21]。2013 年，Daniel Fried 等人研究了一种自动分类方法，利用监督学习算法对 NAS 并行基准测试集(NPB)进行学习和分类，较为全面的比较了支持向量机、决策树等在程序并行性分类问题上的性能表现[22]。2016 年，Zhen Li 等提出了一个基于动态程序分析的通用并行性框架 DiscoPoP，框架以计算单元(CU)为基础概念，将程序表示为 CU 图，提供了与编程语言无关的高效的数据依赖分析工具[23]。David del Rio Astorga 等人提出了一种有助于发现和自动标注程序源码文件中 Pipline 并行模式片段的图形分析工具 PPAT，极大的促进了串行代码到并行代码的转换效率[24]。同年，该团队对 PPAT 进行扩展和改善，添加了对 Map、Farm 并行模式的支持，简化了源到源转换的流程[25]。2018 年，Haibo Zhao、Fei Zheng 等人提出了一种可以识别二进制文件中的可并行部分并自动将其并行化的方法，扩展了现有编译器的编译优化技术，为进一步提高程序性能做出贡献[26]。

国外基于深度学习方法进行源码分析的研究同样活跃。2015 年，Lili Mou,、Ge Li 等人提出了一种基于程序抽象语法树结构的卷积神经网络 TBCNN，并通过实验验证了相较于其他 NLP 神经网络模型，TBCNN 在程序分类问题上更加有效[27]。2017 年，Vincent J. Hellendoorn Min-je 等人对深度神经网络模型是否是源代码建模的最佳选择进行了深入探讨[28]。同年，Choi、Sehun Jeong 等人提出了一种以数据驱动的、完全端到端的检测代码缓冲区溢出的方法，该团队使用程序模板自动生成大量带有标注的数据集，通过记忆网络模型对数据集进行学习与分类，实验证明，该模型在检测程序缓冲区溢出的问题上表现优异[29]。2018 年，Carson D. Sestili 等人在文献[29]的基础上，利用记忆网络模型，针对缓冲区溢出问题检测与现有静态分析工具进行对比，并致力于为未来可能解决的问题如为深度学习提供更合适的数据等[30]。Uri Alon 等人提出使用抽象语法树中的路径表示一段程序，来代替将程序作为一个简单的序列地做法，通过训练神经网络实现了对程序中变量名称的预测[31]。同年，ALON 和他的团队提出了基于抽象语法树对源代码进行嵌入表示的 code2vec 模型，用于将代码片段表示为连续分布的向量，能够更加准确的学习代码语义[32]。此外，Miltiadis Allamanis 等人在 2018 年建议使用图来表示代码的语义结构，并提出使用基于图的深度学习方法对程序结构进行推理[33]。Tal Ben-Nun 等人在他们的论文中，提出了基于中间表示的可学习代码语义表示 NCC，将程序编译为 LLVM IR 然后再通过建立程序上下文流图(XFG)，使用图嵌入的方法对代码进行嵌入表示。相较于直接处理代码或使用抽象语法树对代码进行嵌入的方法，NCC 在算法分类等问题上具有更加表现[34]。2019 年，Zimin Chen 等人对现有的源代码嵌入方法进行了总结和介绍，详细分析了 NLP 技术在源码分析领域中的应用现状[35]。

1.3 本文的主要工作

本文基于深度学习及相关技术，研究了一种串行程序的并行化方法，从串行程序的并行性识别、并行程序开发、并行程序调试三个方面提供了完整的串行程序并行化方案。主要工作包括以下 4 个部分：

(1)构造了数据集 GFCPD，该数据集不仅覆盖了 Livermore、NPB 等常见的并行程序基准测试集，同时包含了常用数学程序库以及图的相关算法、树的相关算法等复杂算法的不同实现。是一个基于程序上下文流图(XFG)的通用数据集，是一个可以用于串行程序并行性识别任务的深度学习数据集。

(2)建立了基于深度图卷积神经网络模型架构(DGCN)的串行程序并行性识别模型，并将该模型的识别结果同传统的静态分析方法、结合动态特征的机器学习方法以及在算法分类问题上表现良好的深度学习模型进行对比，从而验证了 DGCNN 在串行程序的并行性识别任务中的可行性与有效性。

(3)基于 XML 技术实现了并行编程标记语言 PML，对 MPI、OpenMP 等现有编程模型进行设计与封装，提供统一的标准化编程接口，简化了并行编程方式，降低了并行编程门槛。

(4)实现了并行编程辅助平台，集成了串行程序的并行性识别、PML 语言的程序开发、并行程序的远程调试等功能，提供友好的用户交互环境与完整的并行程序开发流程。

1.4 本文的组织结构

第一章为绪论。介绍了本文课题的背景和意义，分析了近年来国内外对于串行程序并行化方面的研究与深度学习方法在源码分析领域的应用现状，对本文的组织结构及主要工作内容做了简要的介绍。

第二章简要介绍了本文研究中相关的理论和方法，包括程序的并行性分析、LLVM 编译器框架、程序的上下文流图(XFG)、Polyhedral 模型、pluto 编译器和图神经网络。

第三章主要介绍数据集 GFCPD(Graphs For Code Parallelism Discovery)的构造过程，对数据集的代码成分分布、构造流程及数据集规模的合理性进行了详细的介绍。

第四章是本文的重点研究内容，对深度学习方法在串行程序并行性识别的研究中是否可行进行了分析，提出了一种将 GFCPD 数据集应用于深度学习任务的数据预处理方法，建立了基于深度图卷积神经网络的串行程序并行性识别模型，并通过对比实验分析了本文建立的识别模型相较于其他方法的优劣性。

第五章介绍了如何将本文所有的工作转化为完整的程序并行化方法。主要包含了两部分的内容，首先介绍了基于 XML 的并行标记语言 PML，说明了其设计思路与实现细节；然后介绍了并行编程辅助平台，列举并展示了其基础功能的实现技术与实现效果，叙述了如何将串行程序的并行性识别特性集成到平台中，形成完整的并行程序开发流程。

结论中分析了本文所作研究的成果和局限性,对本文的研究工作进行了总结，为下一步的研究工作指明了方向。

第 2 章 相关理论与方法

2.1 程序的并行性分析

并行编程在实际工程应用中占有着举足轻重的位置。尤其在地震数据处理、物理模型计算等科学计算领域具备着巨大的应用潜力。对于计算机程序而言，并行性可以分为控制并行性和数据并行性。前者是将程序划分为多个可独立执行的部分，每个部分对相同的一份数据同时执行不同的操作；而后者则是将数据划分为多个部分，使用多个相同的程序同时处理不同的数据。若想要将串行程序改写为并行程序，使得计算机可以同时正确的处理多个任务或快速高效的处理庞大数据，从而提高程序执行的效率，以达到节约时间成本和充分利用计算资源的目的，首要任务则是关注程序中的依赖关系，进行串行程序的并行性分析。

2.1.1 数据依赖与控制依赖

数据依赖和控制依赖的存在与否往往决定了程序是否具备并行性。本小节将介绍它们并讨论它们在程序并行性分析中所扮演的角色。

一般来讲，如果一个循环中不存在控制语句(即不存在控制依赖)，且循环体中的语句无论如何变换执行顺序，循环的执行结果都不会改变，则认为这个循环是可以并行化的。因此，关于循环内的数据依赖关系分析，主要针对三种依赖形式：流相关、反相关和输出相关，如图 2.1 所示。图 2.1(a)中，语句 S1 对元素 A[i]进行写操作，语句 S2 对元素 A[i]进行读的操作，语句 S1 先于语句 S2 执行。语句 S2 中 A[i]的输入值依赖于语句 S1 中对 A[i]的操作结果，因此，必须保证语句 S1 和 S2 的先后顺序，才能保证程序执行结果的正确性，此时称语句 S1 和语句 S2 之间存在数据流相关依赖关系；图 2.1(b)中，图 2.1(b)中，语句 S1 对元素 C[i]先执行读操作，随后语句 S2 对元素 C[i]进行写操作，必须保证语句 S1 先于语句 S2 执行，才能保证语句 S1 每次都能读到正确的 C[i]值，此时我们称语句 S1 与语句 S2 存在数据反相关依赖关系；图 2.1(c)中，语句 S1 和语句 S2 均对元素 A[i]执行写操作，此时若想保证程序执行结果正确，仍需保证语句 S1 先于语句 S2 执行，此时称语句 S1 与语句 S2 存在数据输出相关依赖关系。

如果循环中存在控制语句，即后一条语句执行与否依赖于前一条语句的执行结果，则称该循环中存在控制依赖。如图 2.2(a)所示，语句 S2 与语句 S1 之间存在控制依赖，而语句 S3 与语句 S1 之间不存在控制依赖，因为语句 S3 无论如何都会被执行。值得一提的是，控制依赖不同于控制流，参考图 2.2(b)和图 2.2(c)可知，控制流图描述了一个程序的执行路径，通常编译器可以生成控制流图并从中

导出控制依赖关系。不难发现，与数据依赖类似，控制依赖也会在语句间形成执行顺序的约束，但控制依赖可以通过一种叫做预测执行的技术被消除。因此，在对串行程序进行并行性分析时，往往主要关注其中的数据依赖关系。

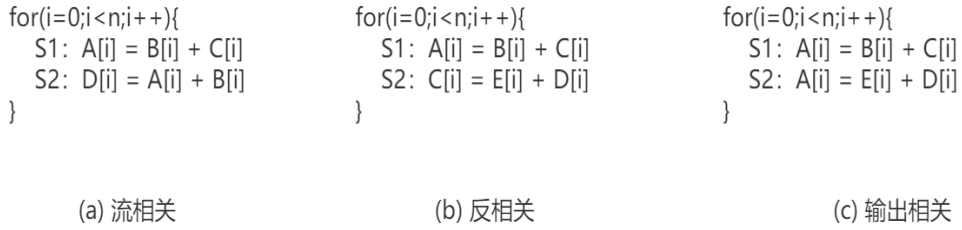


图 2.1 数据依赖关系示例图

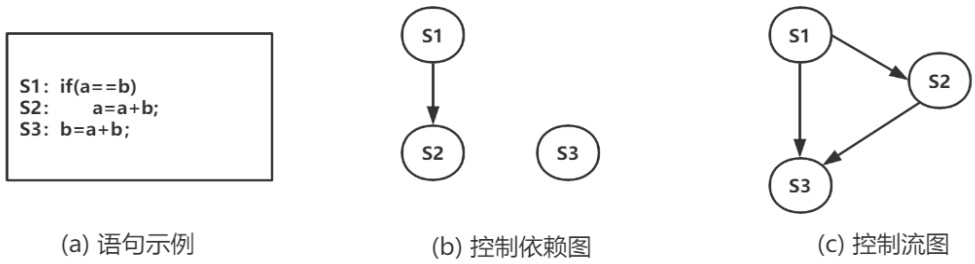


图 2.2 控制依赖关系示例图

2.1.2 静态分析与动态分析方法

串行程序的并行性分析之所以具有挑战性，主要是因为不能既准确又高效地得到程序中的依赖关系。常见的程序并行性分析方法按照是否收集程序运行时的相关信息，可以分为静态分析方法和动态分析方法。

程序的静态分析无需运行代码，是一种直接从程序代码或中间表示(IR)中提取静态特征进行分析的代码分析技术，可以在一定程度上高效地验证代码是否满足既定的各项指标。常见的程序静态分析技术有：(1)词法分析，其目的是将源程序转换为 Token 流，并生成一个相关的符号列表；(2)语法分析，通过将上述符号列表转换为抽象语法树，来判断源程序的语法结构是否正确；(3)语义分析，对语法结构正确的源程序进行语义方面的正确性检查；(4)控制流分析，生成有向的控制流图，用节点表示代码的基本块，节点间的有向边代表控制流路径和可能存在的循环。分析过程中可以生成函数调用关系图，来表示函数间的嵌套关系；(5)数据流分析，在程序中各个初始化点和引用点上，对变量的可能值信息进行收集和分析。

在很多情况下，纯静态的程序分析被证明是过于保守的，因此出现了一系列

的动态分析方法。动态分析关注的是程序运行时的真实行为，捕获程序实际运行时的依赖关系。具体的来说，通常有三种方式可以记录程序运行时的信息，追踪程序变量并记录、使用插桩技术[36]或者结合使用前两者。其中插桩技术是常用方法之一,根据实际需要在不同阶段或不同的代码级别上加入辅助代码,按照插桩、执行、分析的三部曲对程序进行动态分析。

程序的静态分析方法在生成程序的数据流图和控制流图方面存在速度与质量的双重优势，但是静态分析方法往往在评估程序的并行性时被认为是保守的。这是因为：首先，当程序规模足够庞大且分支较多时，依赖分析的搜索空间会成几何倍增长；其次，它所用到的信息有限，在分析只有在运行时才会动态分配空间的对象之间的依赖关系问题上存在短板。与之相反，动态分析方法在程序运行时跟踪检测程序中的依赖关系，它仅仅对程序运行时确实存在的对象进行分析，不受分支数量的影响，因此动态分析过程中的数据流图和控制流图是不完整的，它的分析结果也可能取决于某种程序执行的配置，且在某些时候在时间和空间上都有很高的运行开销。

2.1.3 并行性分析工具

如前文所述，发掘串行程序的并行性的实际需求是寻找程序中的数据依赖关系。当目标程序规模足够庞大，代码行数足够多时，通过手动的方式寻找程序中的数据依赖关系是不现实的，因此基于数据依赖关系理论自动预测串行程序的并行性是十分必要的。本小节中，将现有的程序并行性分析工具按照数据依赖分析器、半自动并行化工具和自动并行化工具三种类型进行了罗列(如表 2.1 所示)。

表 2.1 现有并行性分析工具信息表

类别	名称	定义
数据依赖分析器	Aristotle Analysis ystem[37] Frama-C[38] DMS® Software Reengineering Toolkit™	负责分析并揭示程序中的数据依赖关系，但不提供并行化指导，由用户通过依赖关系分析结果确定程序的并行性。
半自动并行化工具	Kremlin[39] Alchemist[40] Parwiz[41] Tareador[42] Intel® Advisor XE[43] SLX Tool Suite[44] Prism[45]	尝试在串行程序中发现潜在并行性的位置，但不显示数据依赖关系，也不对源程序进行修改，仅在程序的可并行部分对用户进行提示。

自动并行化工具	Intel® C++ Compiler Polly[46] LRPD test[47] Apollo[48] ParallWare[49] Par4All[50] PLUTO[51] Cetus[52]	自动分析并发现串行程序中的并行性，并自动将源串行程序转换为并行程序。

必须说明的是，尽管串行程序的并行性发现工具相关研究十分活跃，一些基于 Polyhedral 模型或推测执行机制对循环结构进行并行性发现和源到源转换的工作也实现了完全自动化，但由于实际应用程序具有不规则的执行模式和不同的运行规模，这些方法和工具仍然在时间、能耗、适用范围等方面存在限制性。因此，目前仍然没有适用于实际应用程序的通用且成熟的并行性发现工具。此外，静态分析仍是目前较为先进的工具中使用最广泛的方法，表 2.1 所列工具中只有 Intel Advisor XE 和 Cetus 使用了程序的运行时数据。

2.2 LLVM 编译器框架与 IR

LLVM(Low Level Virtual Machin，底层虚拟机)为任意编程语言静态和动态的编译目标提供了支持，是一个基于 SSA 的现代化编译策略。显然，其范围早已并不局限于创建一个虚拟机[53]，如今 LLVM 已经发展成为了一个模块化和可重用的编译器和工具链技术的集合，用于优化任意程序语言源码的编译时间、链接时间、运行时间以及空闲时间。

LLVM 核心库为编译器提供了相关支持，可以作为多种编程语言的编译器后台。传统的静态编译器(例如大多数 C 编译器)采用如图 2.3 所示的前端、优化组件和后端的三段式设计。前端组件解析程序源代码，检查语法错误，生成一个基于语言特性的抽象语法树(AST)来表示输入的代码。AST 将代码转换为具有新的表达语法的中间代码提供给优化器，然后优化器和后端程序将中间代码翻译成机器代码。需要说明的是，传统静态编译器对于不同的编程语言和硬件架构分别设计了不同的中间代码表示，而 LLVM 对这三个阶段采用了更加合理的设计。如图 2.4 所示，LLVM 对于不同的前端和后端均使用统一的中间代码(LLVM IR)表示。如此一来，使编译器支持一种新的语言就仅仅需要实现一个新的前端；同理，使编译器支持一种新的硬件设备，仅仅需要实现一个新的后端。优化器则是一个通用的组件，它针对的是统一的 LLVM IR，不论是支持新的编程语言，还是支持新的硬件设备，都不需要对优化器做出修改。

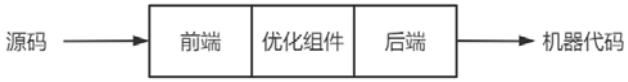


图 2.3 传统静态编译器架构示意图

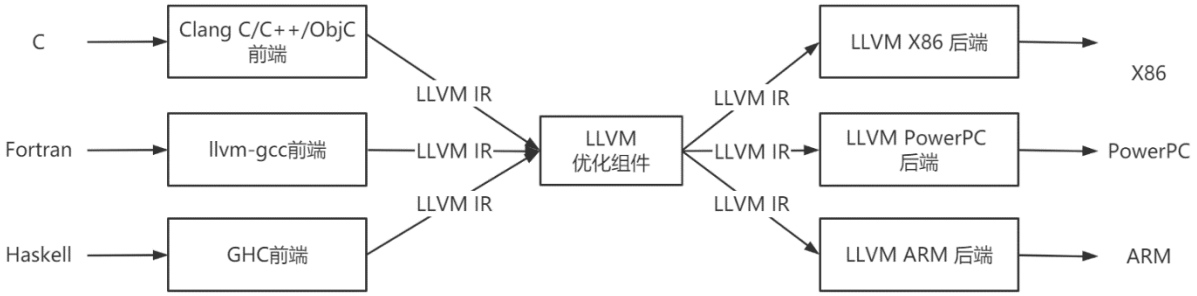


图 2.4 LLVM 架构示意图

LLVM IR 本质上是一种与源编程语言和目标机器架构无关的通用中间表示，是 LLVM 项目的核心设计和最大的优势。它使用静态单赋值(SSA)策略，即以三地址码的形式组织指令并且假设有无数个寄存器可用(为每一个变量、基本块或函数提供唯一的标识符，但标识符名称不确定)。它是经过特殊设计的，并且为了支持轻量级的运行时优化、过程函数的优化、整个程序的分析和代码完全重构以及翻译等等，它还定义了清晰的语义。假设有一段 C 语言程序如图 2.5(a)所示，那么其对应的 IR 如图 2.5(b)所示。可以看出，LLVM IR 是一种底层的类 RISC 虚拟指令集。正如真正的 RISC 指令集一样，它提供了一系列线性的简单指令，如加、减、比较和分支结构等等。和大多数 RISC 指令集不同的是，LLVM 使用了一种简单的数据类型系统来标记强类型(如使用 i32 表示 32 位整型，使用 i32**表示指向 32 位整型的指针)，而一些机器层面的细节都被抽象了出去(如函数调用使用 call 作标记，而返回使用 ret 标记)。

```
unsigned add(unsigned a,unsigned b){
        if(a==b) return b;
        return add(a-1,b+1);
}
```

(a) C语言程序

```
define i32 @add(i32 %a,i32 %b){
entry:
    %tmp1 = icmp eq i32 %a,0
    br i1 %tmp1,lable %done,lable %recurse

recurse:
    %tmp2 = sub i32 %a,1
    %tmp3 = add i32 %b,1
    %tmp4 = cail i32 add(i32 %tmp2,i32
%tmp3)
    ret i32 %tmp4

done:
    ret i32 %b
}
```

(b) LLVM IR

图 2.5 LLVM IR 示例图

2.3 程序的上下文流图(XFG)

程序的上下文流图，是由 Tal Ben-Nun 等人在文献[34]中提出的首个以通过结合程序中控制流信息与数据流信息来生成程序指令嵌入(即向量表示)的图表示形式。其中，上下文是指相互之间存在依赖关系(包括数据依赖和控制依赖)的语句，同时为了在生成嵌入过程中避免硬件特性、编程语言特性和指令组装等方面的限制和歧义，选取独立于硬件和编程语言的 LLVM IR 作为图构造过程的数据来源。

由于 LLVM 将 IR 语句划分为“基本块”，每个基本块中不存在控制流分支。如图 2.6(c)所示，不同的矩形框代表不同的基本块，在一个基本块内，程序按顺序执行并创建可追踪的数据流，同时将执行结果分配给单个标识符，因此一个基本块内的数据依赖关系清晰可见，基本块内任何标识符的上下文也很容易可以获得。但是在控制分支前后，仅靠数据流图并不足以提供足够的信息来确定给定标识符的程序上下文。XFG 恰好弥补了这一缺陷，其中既包含了数据流信息也同时包含了控制流信息。如图 2.6(d)所示，XFG 是有向多重图，其中两个节点之间可以拥有多条边。 XFG 的节点可以是变量、基本块或函数名称等标识符，在图中分别显示为椭圆形或矩形。相应地，一条边代表一种数据依赖(黑色)或控制依赖(浅蓝色)。

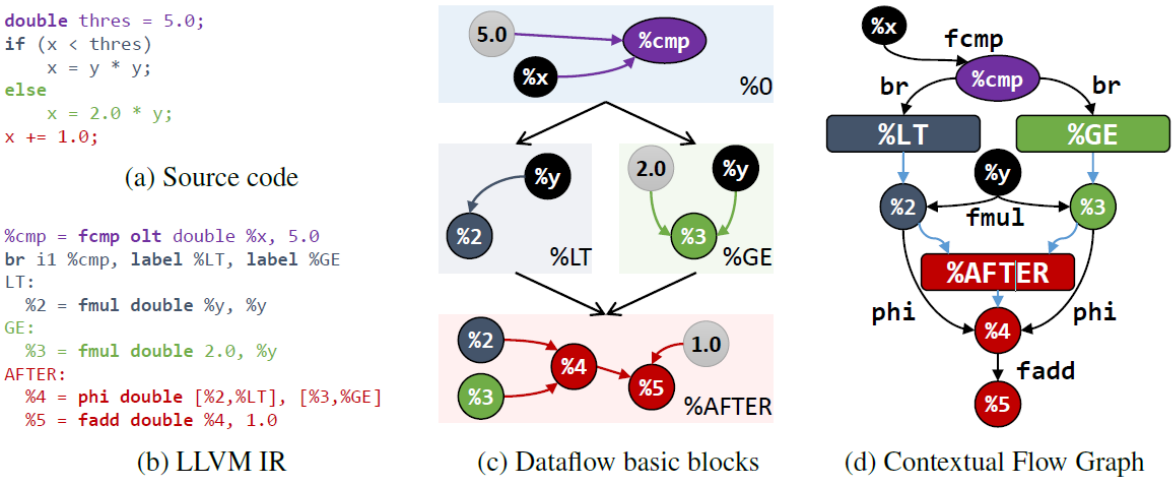


图 2.6 XFG 示例图

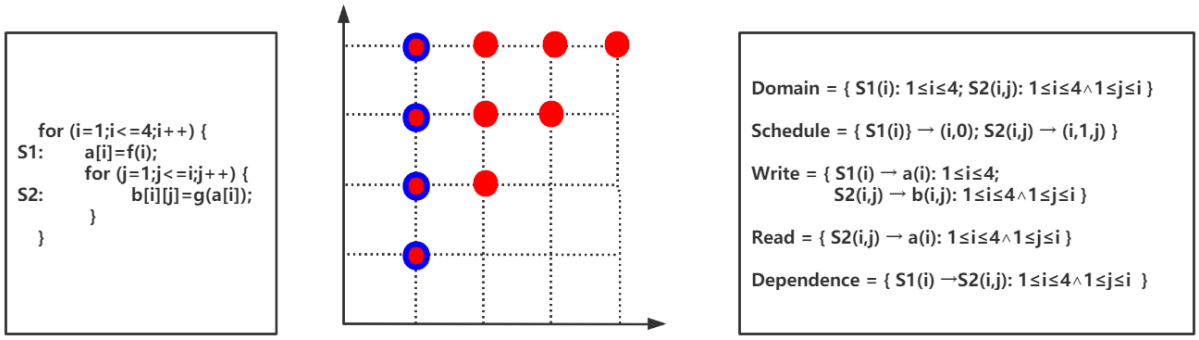
由于 LLVM IR 中对标识符名称的分配是不确定的，因此上述使用标识符作为 XFG 节点的表示方法使得 XFG 节点的表示空间变得无限大。为了更加贴合上下文的概念和更好的学习代码的嵌入表示，文献[34]的实验过程中，在假设相同上下文中出现的语句大概率具有相似的语义的前提下，对 LLVM IR 语句的类型进行了总结并将 XFG 中的节点向 IR 语句级别进行了二次整合生成了 Dual-XFG，使其所有的节点均为 LLVM IR 语句，即 LLVM IR 文件中的一行。除非特殊说明，本文接下来所提到的 XFG 均指 Dual-XFG。

XFG 的构造遵循以下规则：(1)一个基本块内的数据依赖关系必须全部用边进行连接；(2)基本块与基本块之间的依赖关系必须使用边将基本块标识符相连接；(3)没有父级数据流入的基本块，与图的根节点相连；(4)对于外部代码调用，若为静态调用，在编译过程中可以访问代码，则将该部分代码语句包含在 XFG 中，若为动态链接，在编译过程中不可访问代码，则使用调用语句代替。值得一提的是，对于一个有 n 个 SSA 语句的 LLVM IR 文件，构造其 XFG 的时间复杂度为 O(n)，这说明了使用 XFG 进行程序语义的表示对于大规模代码(如 Tensorflow 源码)的分析具有巨大价值。

2.4 Polyhedral 模型和 Pluto 编译器

在国内关于 Polyhedral 模型的研究中，多将其译为多面体模型。由于多面体模型具有应用范围广、表示能力强和优化空间大的优点，目前基于多面体模型的编译优化技术是串行程序自动并行化研究领域的一个热点，是解决串行程序自动转换为并行程序的一种有效手段[54]。

基于多面体模型的编译技术在满足循环的上下界约束条件的前提下，以空间内的多面体形式表示程序语句，通过在该多面体上进行的几何操作对程序特征进行分析并对程序性能进行优化。多面体模型将程序及其语义表示为迭代空间、访存映射、依赖关系以及调度。其中，迭代空间是指在循环的一次迭代内，随着迭代变量(i 或 j 等等)变化的某一条语句(如：A[i]=B[i]+C[i])的所有实例的集合(如{A[i]=B[i]+C[i],(i=0,1,2…n)})；访存映射用于表示内存访问操作与程序语句之间的对应关系；依赖关系是指访问相同内存地址的两个程序语句实例之间的偏序关系(其中至少有 1 个操作类型为写操作)；调度则是在依赖关系的约束下对语句实例之间执行顺序的表示。如图 2.7(a)所示循环片段，其多面体表示如图 2.7 (b)所示，蓝色空心圆形表示语句 S1 的实例，红色实心圆形表示语句 S2 的实例；图 2.7(c)中列出了该循环片段对应的迭代空间(Domain)、调度(Schedule)、访存映射(Write and Read)和依赖关系(Dependence)。



(a) 示例循环嵌套　(b) 图 (a) 所示循环嵌套的多面体表示　(c) 图 (a) 所示循环嵌套的集合与映射

图 2.7 Polyhedral 模型示例图

20 世纪 90 年代以来，基于 Polyhedral 模型的编译技术在程序自动并行化研究领域取得了诸多成果，甚至在许多方面可以代表这一领域目前最先进的水平。基于 Polyhedral 模型的编译工具的工作流程如图 2.8 所示，其中红色虚线框内的部分是其主要构成组件。首先,编译工具的前端由抽象分析组件构成，其作用是根据多面体模型的静态仿射约束从输入的程序源码中识别并提取满足条件的程序片段，同时将其表示成多面体的形式；然后，以迭代空间和访存映射为依据,调用整数线性规划组件计算其中的依赖关系。这里的静态约束是指某一个程序段中的控制流和依赖关系在编译阶段可以被判定并且能够用有限的集合进行表示。编译工具的中间优化部分由调度变换组件构成，其作用是通过调用整数线性规划组件，计算得到一个既满足依赖关系又可以充分发挥目标程序性能和利用硬件体系结构特点的调度结果。最后，编译工具的后端由代码生成组件构成，该部分的功能又可以分为两个阶段：第一阶段借助整数线性规划组件，将迭代空间和调度结果转换生成 AST; 第二阶段将 AST 转换为符合目标程序语言语法规范的最终代码。

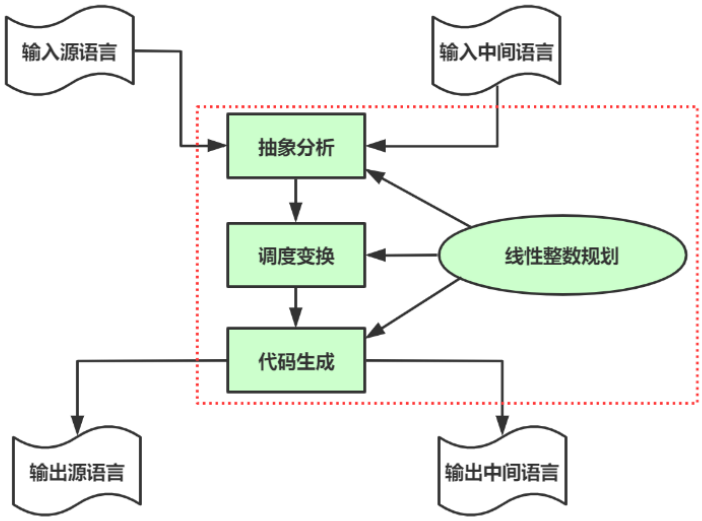


图 2.8 基于多面体模型的编译工具工作流程示意图

Pluto 编译器是一个很好的开发程序并行性和数据局部性的优化工具，是众多基于 Polyhedral 模型的编译器中应用范围最广、最成功的编译器之一，以该编译器为平台实现的 Pluto 调度算法和 Pluto+算法代表了 Polyhedral 模型调度最先进的研究水平。文献[16]也因此于 2018 年被 PLDI (ACM SIGPLAN Conference on Programming Language Design and Implementation) 会议以“Pluto 调度算法至今在众多领域包括将机器学习算法部署在特定加速部件等方面都发挥着重要作用”的评语评为近十年最有影响力的论文。

从编程语言的层面来讲，Pluto 编译器是一个从 C 语言程序到 OpenMP 程序的源到源编译器，并且自开源以来不断地尝试兼容包括 PIP、ISL、PET、Clan 等在内的各种开源库。从优化角度来看，Pluto 算法的实现思路决定了 Pluto 编译器

是一个试图使用调度来尽可能多的解决程序并行化中遇到的优化问题的工具。在程序的分析方法层面，Pluto 编译器是一个仅支持 C 语言文件作为输入的静态分析工具。此外，Pluto 编译器还可以被用来开发程序多粒度的并行性，比如对于存在嵌套循环的程序段，可以将外层的循环做 OpenMP 并行标识，而将最内层可以进行向量化的循环做 ivdep 和 vector always 标识，然后再交给基础编译器做进一步的向量化开发。遗憾的是，由于前文提到的 Polyhedral 模型本身对于程序静态约束的要求，Pluto 编译器目前还没有办法直接用于大型应用测试集的优化。

2.5 图神经网络与图卷积神经网络

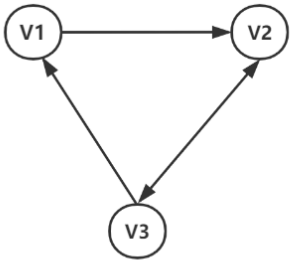
虽然深度学习已经在欧几里得空间数据中取得了很大的成功[55]，但越来越多的应用程序数据开始以图的形式进行表示，大量的学习任务要求处理元素间含有丰富关系信息的图数据。如物理系统的建模、分子指纹的学习、蛋白质结构的预测和疾病的分类等都需要模型从图数据中去学习，但传统的神经网络模型并不能很好的处理图类型的数据，图神经网络(GNN)应运而生。GNN 的概念是由 Gori 等人首次提出的[56]，后来的研究人员在设计 GNN 的架构时又借鉴了卷积网络(CNN)的思想，出现了图卷积神经网络(GCN)。

2.5.1 图的定义

近年来，由于图的强大表达能力，用机器学习方法分析图的研究受到了越来越多的关注。图是一种对一组对象(图的节点)及其之间的关系(图的边)的集合进行建模的数据结构。在计算机科学中，一个图 G 可以由它所包含的顶点和边的集合来进行描述。而在机器学习领域，由于图的数据复杂且规模并不确定，往往需要将图的邻接矩阵加入到图的定义中来，具体定义如下：定义图 G=(V,E,A)，其中 V 代表图的节点集合，E 代表图的边的集合，A 代表图的邻接矩阵。在一个图中，vi∈V 表示一个节点，eij=(vi,vj)表示一条边。对于一个具有 N 个节点的图 G 来说，它的邻接矩阵 A 是一个 N*N 的矩阵，Aij 表示图的第 i 个节点 vi 和第 j 个节点 vj 之间的关系，如果 Aij>0 则两个节点间存在一条边 eij，Aij=0 则不存在。根据顶点之间的边是否存在方向依赖关系，图可以分为有向图和无向图。如图 2.9(a)所示，有向图 G=({v1，v2，v3}，{e12，e23，e31，e32}，A)，其中 A 如图 2.9(b)所示。

图数据的复杂性对现有机器学习算法提出了重大挑战，因为图数据是不规则的。每张图大小不同、节点无序，一张图中的每个节点都有不同数目的邻近节点，使得一些在图像中容易计算的重要运算(如卷积)不能再直接应用于图。此外，现有的机器学习算法默认其数据实例是彼此独立的。然而，图数据中的每个实例都与周围的其它实例相关，含有一些复杂的连接信息，用于表示数据之间的依赖关

系。



(a) 有向图G

	V1	V2	V3
V1	0	1	0
V2	0	0	1
V3	1	1	0

(b) G的邻接矩阵

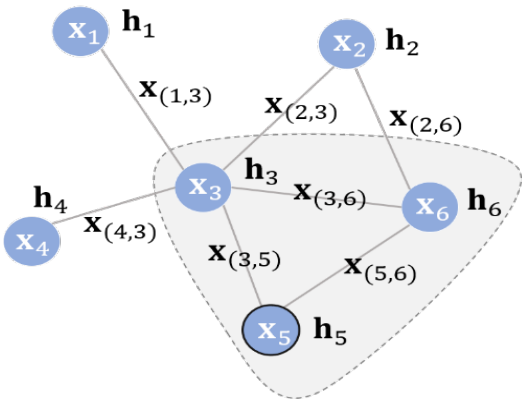
图 2.9 有向图及其邻接矩阵示例图

2.5.2GNN 简介

图神经网络(Graph Neural Networks，GNN)是在图域上运行的深度学习方法。近年来，GNN 较好的可解释性和其在图数据分析中表现出的超高性能使其成为了一种广泛应用的图分析方法。GNN 是一种连接模型，它通过图的节点之间的消息传递来捕捉图的依赖关系。与标准的神经网络不同的是，图神经网络保留了一种状态，可以表示来自其邻域的具有任意深度的信息。虽然原始的 GNN 很难为一个固定点进行训练，但是网络结构、优化技术和并行计算的最发展已经使它们具备了学习的能力。

GNN 的学习目标是通过迭代获得每个节点的隐藏状态 h_v，其中包含了来自相邻节点的状态信息。为了更好的理解图神经网络如何使每个节点都能感知到图上的其他节点，接下来就某一时刻图神经网络节点状态的更新与输出做如下简要说明：给定一张图 G，每个节点 v 都有其自己的特征x_v和隐藏状态h_v，节点 v 与节点 u 之间的边也具备特征$x_{(v,u)}$。那么在 t+1 时刻，节点 v 的隐藏状态将按照式 2.1 的方式更新。其中，f是隐藏状态的状态更新函数，$x_{co[v]}$指的是与节点 v 相连的所有边的特征，$x_{ne[v]}$指的是节点 v 的所有邻节点的特征，$h^t_{ne[v]}$则指节点 v 的所有邻节点在 t 时刻的隐藏状态。需要注意的是，f是一个全局共享的函数，是对所有节点都成立的。以图 2.10 为例，假设某一时刻节点 5 为中心节点，其隐藏状态h_5的更新函数如图所示。其中x_3，x_5，x_6分别是节点 3、5、6 的特征，对应式 2.1 中的$x_{ne[v]}$；$x_{(3,5)}$、$x_{(5,6)}$是与节点 5 相连的两条边的特征，对应式 2.1 中的$x_{co[v]}$；而h_3、h_6则是节点 3 和节点 6 的隐藏状态，对应式 2.1 中的$h^t_{ne[v]}$。如此不断地利用当前时刻中心节点的邻节点的隐藏状态作为部分输入来生成下一时刻中心节点的隐藏状态，直到每个结点的隐藏状态变化幅度很小，整个图的信息流动趋于平稳，每个结点也就都感知到了其邻节点的信息。

$$h_v^{t+1} = f\left(x_v, x_{co[v]}, h^t_{ne[v]}, x_{ne[v]}\right) \quad \text{(式 2.1)}$$



$$\mathbf{h}_5 = f(\mathbf{x}_5, \mathbf{x}_{(3,5)}, \mathbf{x}_{(5,6)}, \mathbf{h}_3, \mathbf{h}_6, \mathbf{x}_3, \mathbf{x}_6)$$

图 2.10 GNN 状态更新示例图

状态更新公式仅仅描述了如何获取每个节点的隐藏状态，除此以外，往往还需要另外一个函数g来描述如何适应下游任务。如在节点分类问题中，需要根据节点 v 最终时刻的隐藏状态h_v按式 2.2 所示的方式得到其输出o_v用于分类任务。其中，g又被称为局部输出函数，与f类似，g也是一个全局共享的函数。

$$o_v = g(h_v, x_v) \qquad \text{(式 2.2)}$$

综上，GNN 的节点状态更新和输出整体流程如图 2.11 所示。可以很清楚的看到，每一时刻的节点状态都与前一时刻其邻节点的状态密切相关。如 T_1 时刻，因为结点 1 与结点 3 相邻，所以节点 1 在 T_1 时刻的状态更新过程接受节点 3 上一时刻的隐藏状态作为输入。直到 T_n 时刻，各个节点隐藏状态收敛，每个结点后面接一个输出函数g即可得到该结点的输出o_v。需要说明的是，对于不同的图来说，其节点隐藏状态收敛的时刻可能不同。

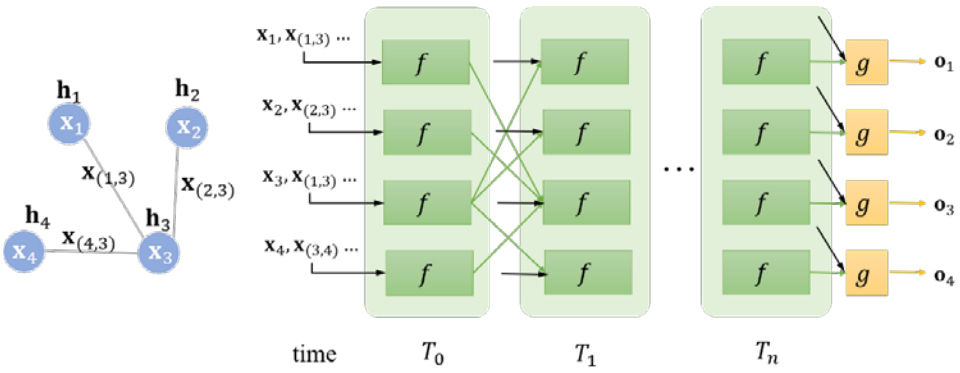


图 2.11 GNN 状态更新与输出流程示例图

2.5.3GCN 简介

作为 GNN 的一种特殊架构，图卷积神经网络(GCN)将常用于图像处理的卷积神经网络(CNN)应用到了图数据相关的深度学习研究中，它为图数据的处理提供了一个崭新的思路，由 Thomas Kpif 于 2016 年首次提出[57]。

与 CNN 类似，可以将 GCN 中的卷积理解为在局部范围内的图节点特征抽取方法，这里的局部指的是中心节点及其邻节点的集合。如图 2.12(a)所示，如果把图像中的每个像素点视作图的一个节点，一张图片也可以看作是一个非常稠密的图。图 2.12(b)则是一个普遍意义上的图结构，阴影部分代表卷积核。不难发现，在以图像为代表的欧几里得空间中，每个节点的邻节点数量都是固定的。如图 2.12(a)中任何节点的邻节点始终是 8 个(边缘上的节点可以做填充)。但在图这种非欧几里得空间数据中，节点的邻节点数量并不固定。如图 2.12(b)中绿色节点的邻节点有 2 个，但同时存在某个节点有 5 个邻节点的情况。欧式空间中的卷积操作实际上是用固定大小可学习的卷积核来抽取像素的特征，但由于图中节点的邻节点数量不固定，所以传统的卷积核并不能满足图数据处理的需求。

因此，图卷积神经网络的本质是提供了适用于图的可学习的卷积核用于抽取图上节点的特征。与上文提到的 GNN 相比，同样是结合邻节点的状态信息来更新中心节点的状态，且同样具备局部输出函数g来完成更多的下游任务(如节点分类等)；不同的是，GNN 通过多次循环迭代以达到节点的隐藏状态收敛的目的，每次迭代的参数(即状态更新函数f)是全局共享的，而 GCN 则是以卷积的方式通过卷积层堆叠，不同的卷积层根据任务需要可以采用不同的参数，完成节点状态的更新。

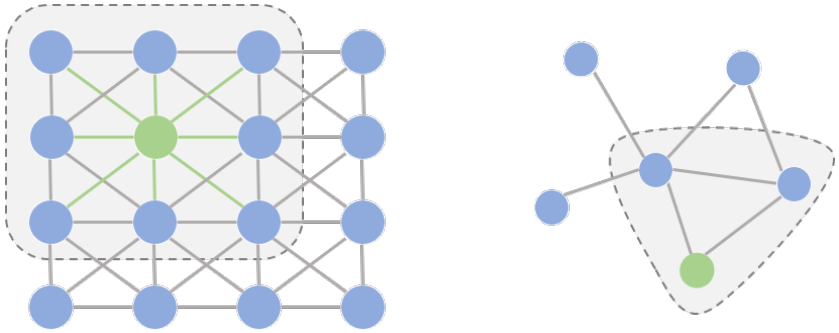


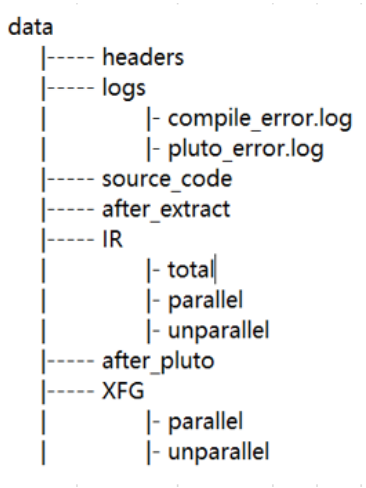
图 2.12 卷积核示例图

2.6 本章小结

本文的研究内容将深度学习与串行程序的并行性识别相结合，为了使后续章节中的内容更加清晰易懂，本章除了对研究过程中使用到的编译工具、程序语义表示方法给出具体的说明外，在串行程序的并行性分析方面，简要解释了串行程序并行性分析的概念与意义，对串行程序并行性分析过程中重点关注的数据依赖与控制依赖的概念做了详细阐述，对程序的静态分析方法和动态分析方法进行了简要说明，并对现有的程序并行性分析工具做了分类罗列；在深度学习方面，重点介绍了后续实验过程中使用到的图神经网络及图卷积神经网络的相关概念与理论。

第 3 章 GFCPD 的构造

所谓兵马未动，粮草先行。众所周知，在深度学习相关的研究中，输入数据的质量在很大程度上决定了实验的结果好坏。可见数据在深度学习相关研究中的地位无异于粮草之于兵马。由于目前并没有合适的深度学习数据集可以直接用于串行程序并行性识别的研究，为了更好的将深度学习方法应用到本文的研究中来，本章构造了一个适用于串行程序并行性识别任务的深度学习图数据集 GFCPD。GFCPD 是一个基于 XFG 的通用图数据集，本章详细介绍了其构造过程中数据搜集、数据提取、数据标注以及 XFG 的生成四大阶段，并对其规模进行了论证和说明。在开始接下来的几个小节之前，为了避免下文叙述中的逻辑混乱，需要提前对本章工作的文件夹目录结构做如图 3.1 说明。



data	数据文件夹根目录
headers	头文件存放根目录
logs	脚本日志文件根目录
logs/compile_err	编译错误日志文件
logs/pluto_error	Pluto编译器转换错误日志文件
source_code	搜集的源码文件(.c/.cpp)按照分类分别存储
after_extract	循环提取重构后的文件根目录(.c/.cpp)
IR	LLVM IR文件根目录
IR/tottal	编译验证阶段所有IR文件存放目录(.ll)
IR/parallel	数据标注阶段可并行文件对应的IR文件目录(.ll)
IR/unparallel	数据标注阶段不可并行文件对应的IR文件目录(.ll)
after_pluto	pluto编译器转换后的文件(.c/.cpp)
XFG	程序的上下文流图文件根目录
XFG/parallel	可并行文件IR(IR/parallel)对应的XFG目录(.p/.txt)
XFG/unparallel	不可并行文件IR(IR/unparallel)对应的XFG目录(.p/.txt)

图 3.1 第 3 章工作目录结构说明图

3.1 数据搜集

数据搜集是 GFCPD 数据集构造的先导工作，为了使本数据集以及接下来基于本数据集所做的相关研究具备普适性和可靠性，数据搜集工作主要分为基准测试程序、现有数学运算程序库和一般程序三个方面进行。本小节接下来的内容将对上述过程搜集到的数据做详细介绍。在此之前需要说明的是，上述过程中搜集的代码均为 C/C++语言编写的串行版本代码，按照类别分别存储在 source_code 文件夹中。

(1)基准测试程序

基准测试程序部分主要考虑了在大多数程序并行性相关研究中均有使用的利弗莫尔循环(Livermore loops)和 NAS 并行基准测试程序(NPB)。其中，Livermore loops 是由 Francis H. McMahon 从劳伦斯利弗莫尔国家实验室的计算机上运行的

科学源代码中创建的。它由 24 个循环组成(详见表 3.1)，每个循环运行一个数学核。该基准于 1986 年发布了 Fortran 版本，但后来被移植到许多编程语言中。

表 3.1 Livermore 循环详情表

序号	数学核	序号	数学核
1	流体力学	13	2-D 单元格粒子
2	不完全的楚列斯基共轭梯度法	14	1-D 单元格粒子
3	内积	15	任意 Fortran
4	带状线性系统法	16	蒙地卡罗搜索
5	三对角线性系统法	17	隐式条件计算
6	一般线性递归方程	18	显式流体力学计算
7	状态方程	19	一般线性微分方程
8	交替方向隐式集成	20	离散坐标转置
9	集成预测因子	21	矩阵乘积
10	差异预测因子	22	普朗克定律
11	一阶和	23	2-D 隐式流体力学计算
12	一阶差分	24	数组中最小值的第一个位置

NPB 是由美国航空航天局发布的一套并行基准，该基准是一个流体动力学计算的程序集合。目前已经成为公认的用于测评大规模并行机和超级计算机性能的标准测试程序。如表 3.2 所示，本文所使用的 NPB 版本由 8 个程序模块组成，包括 5 个核心程序模块和 3 个模拟程序模块。

表 3.2 NPB 详情表

模块名称	简介
IS	整数排序
EP	计算 Gauss 伪随机数
CG	大型稀疏对称正定矩阵的最小特征值的近似值的求解模块
MG	一个简化的多栅格核心基准测试
FT	利用快速傅里叶变换来解决 3 维的偏微分方程
BT	求解 3 对角线方程组
SP	求解 5 对角线方程组
LU	利用对称超松弛法求解块稀疏方程组

(2)数学运算库

数学运算程序库部分采用了 GNU 高精度算术运算库 GMP、基础线性代数子程序库 BLAS 及有效支持线性代数、矩阵和矢量运算、数值分析及其相关的算法的 Eigen C++程序库。

(3)一般程序

一般程序部分综合考虑经典并行例程、矩阵操作、图和树的相关算法、动态规划相关算法等方面，主要搜集了如表 3.3 所示题目的相关实现代码。为了保证数据的正确性和严谨性，对每一份程序源码均进行了代码风格检查及编译测试，对编码格式良好且无编译错误的源码文件予以采用。

表 3.3 一般程序详情表

序号	题目	序号	题目
1	函数积分	17	图像平移
2	数字累加	18	图像旋转
3	求解圆周率	19	背包问题
4	埃氏筛法生成质数	20	图的最短路径问题
5	判断质数	21	图的广度优先搜索
6	第 K 小的数	22	图的深度优先搜索
7	数组求和	23	图的极大独立集
8	N 的阶乘	24	N 皇后问题
9	矩阵乘法	25	高斯消元法求解线性方程
10	数组前缀求和	26	多重网格法解微分方程
11	二分法查找函数零点	27	共轭梯度法的实现
12	矩阵转置	28	霍夫变换
13	整数集合的缩灭	29	离散傅里叶变换
14	快速傅里叶变换	30	K 值算法
15	遗传算法	31	曼德勃罗特集的计算
16	平面凸包问题	32	PSO 算法

3.2 数据提取

正如 2.1 所述，循环片段往往是串行程序最具备并行潜力的部分。因此，在数据搜集工作结束后，为了对每一个循环片段分别进行并行潜力识别，本文按照如下步骤对搜集到的源代码文件进行了进一步的提取和处理：首先，将源代码文件中所有的循环片段提取出来；然后，考虑到构造 XFG 需要将代码文件编译为 LLVM IR，又对所有的循环片段进行重构，使每一个循环片段均独立成为一份完整的 C/C++代码，以满足编译需求；最后，编译检验重构后的新代码文件，在尽可能多的保留原始数据的前提下同时保证其正确性。本小节接下来的内容将对上述三个步骤进行展开描述。

3.2.1 循环提取

由于原始数据的数量较多，为了简化提取工作，缩小时间成本，本文并未借助编译工具的静态分析功能，而是采用面向对象的思想，将程序抽象为图 3.2 所示结构，利用正则表达式匹配技术，直接从程序源码中提取循环片段。如图 3.2(a)所示，从本文所研究内容的角度出发，程序可以看作是函数的集合，而函数又可以看作是循环片段及其他代码的集合。于是将函数抽象为图 3.2(b)所示的对象结构，同时将循环片段抽象为图 3.2(c)所示的对象结构，并通过 fun_name 属性将二者相互关联。

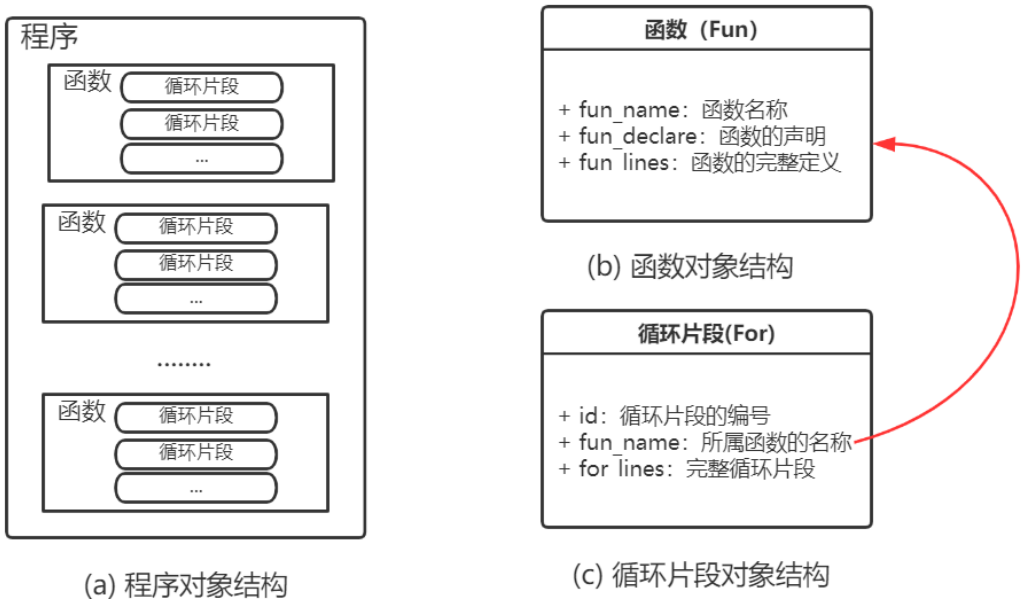


图 3.2 程序的抽象结构示意图

很显然，提取工作的目标数据是循环片段，但为了方便后面对循环片段的重构工作，本文在循环片段提取过程中第一阶段先将函数从源代码中分离了出来，然后第二阶段将循环片段从函数中再分离出来。同时提取了程序中的函数列表和循环片段以及二者之间的对应关系。提取流程如图 3.3 所示(图中虚线标号对应下述步骤编号)：

(1)对于每一份源码文件 file，首先从中读取所有非空行，存储在列表 lines_list 中；然后将 lines_list 中的所有元素使用换行符“\n”拼接成为字符串对象 lines_str；

(2)遍历 lines_list 列表，利用正则表达式匹配该文件中所有的函数定义，按照图 3.2(b)所示结构存储所有函数的名称、声明(仅有返回值类型、函数名称、参数列表，不包含函数的具体实现)与其完整的定义(声明+具体实现)，得到函数列表 fun_list；

(3)遍历列表 fun_list，每次迭代处理一个函数对象，在 lines_str 中将该函数的完整定义使用其声明进行替换；在其完整定义中按照正则表达式规则匹配循环片段并编号，按照图 3.2(c)所示结构存储循环片段得到循环列表 for_list；同时将函数完整定义中的循环片段以单行注释的形式使用编号进行替换。

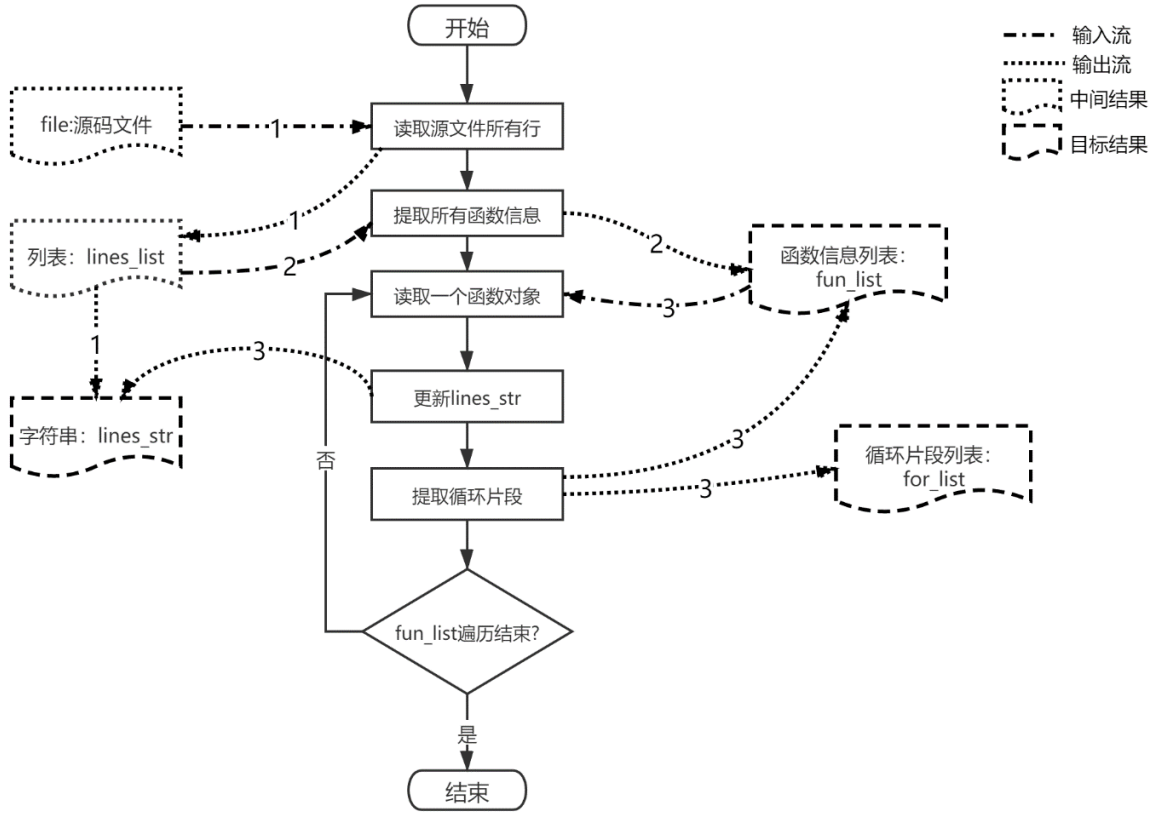


图 3.3 循环提取算法流程图

至此，循环片段的提取工作宣告完成，提取结果的具体形式如图 3.4 所示。可以发现，如果一份源码文件中不包含空白行，头文件引用及宏定义共有 x1 行，定义了 x2 个函数(包括 main 函数)，那么提取结束后，源程序文件共剩余 x1+x2 行；如果一个函数共有 y1 行，其中含有 2 个循环分别为 z1 行和 z2 行，那么提取结束后该函数的完整实现共有 y1-z1-z2+2 行。在此过程中我们更新了字符串对象 lines_str，得到并更新了函数列表 fun_list，同时得到了循环片段列表 for_list，这三者将是下一步代码重构的重要依据。

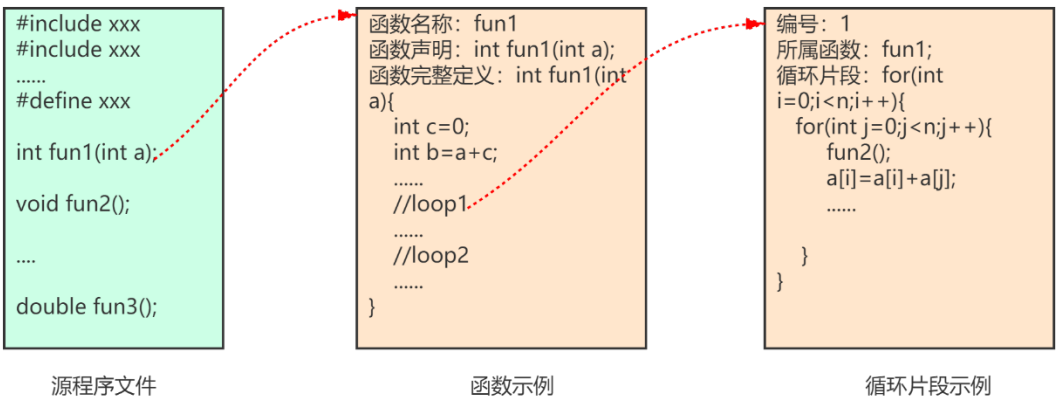


图 3.4 提取结果示例图

3.2.2 代码重构

考虑到构造 XFG 需要将源码文件编译为 LLVM IR，在此部分我们对上述提取过程得到的循环片段按照如下规则进行了重构：(1)保留原程序中的头文件引用、宏定义和变量的声明等内容；(2)每段重构后的新程序中有且仅有一个循环片段(若一个源程序文件提取后包含多个循环片段，那么则将其重构为多个新代码文件)；(3)保留原程序中的函数调用逻辑；(4)最大程度保留原程序语义。不难发现，规则(1)、(3)共同解释了本文在提取循环片段过程中，为何需要不断更新原文件内容 lines_str 与函数列表 fun_list，使用函数的声明对其完整定义进行替换；规则(2)正是本文在提取循环片段过程中，对函数的完整定义中的循环片段使用其编号进行替换的原因所在；而规则(4)则是规则(1)、(2)、(3)存在的理由。具体的重构流程如图 3.5 所示，其中 lines_str、fun_list、for_list 为循环提取过程的输出结果。

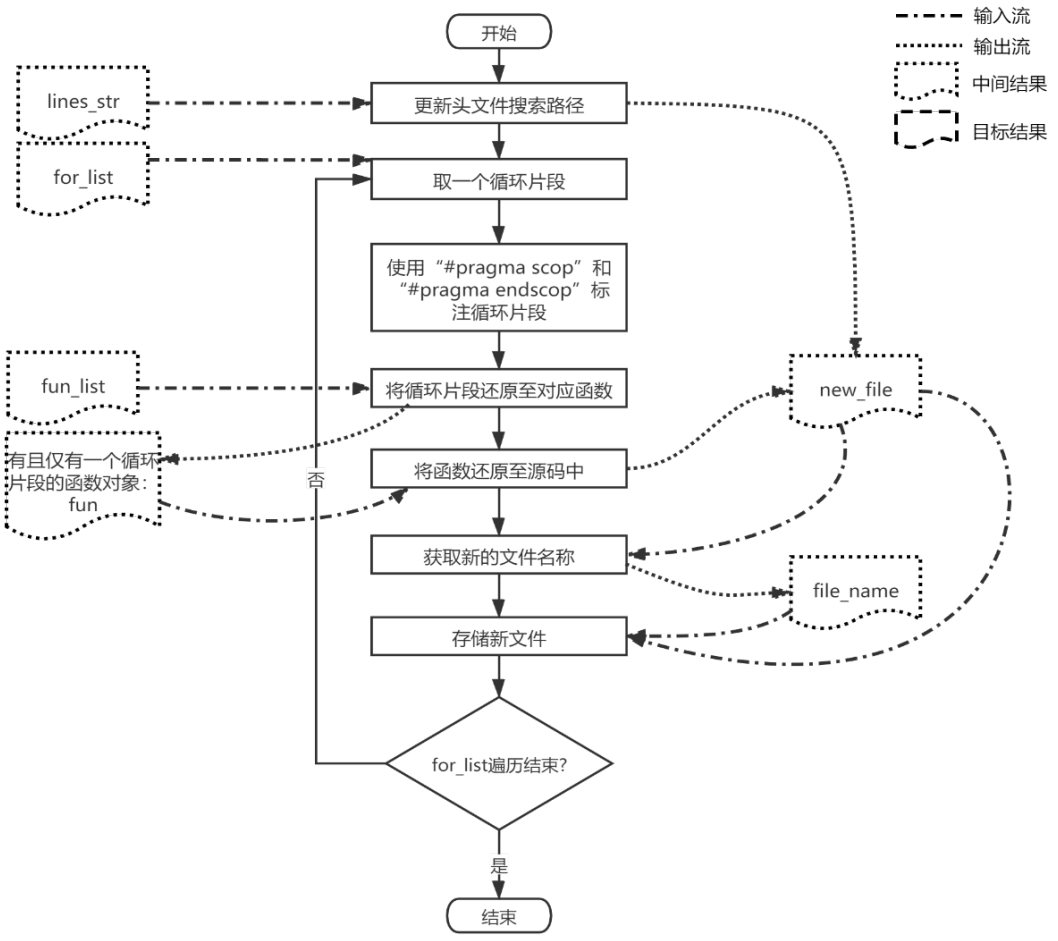


图 3.5 代码重构算法流程图

首先，将搜集到的程序中所涉及到的头文件(.h 文件)均放在同一目录 headers 下并使用正则表达式匹配 lines_str 中的头文件引用部分，保持其他内容不变的前

提下将头文件搜索路径统一修改为“headers/”，把更新后的 lines_str 内容存为 new_file 用来构造新的代码文件；然后，遍历循环片段列表 for_list，对于每一个循环片段，使用“#pragma scop”和“#pragma endscop”进行包裹标注(由于 3.3 节使用 Pluto 编译器进行数据标注的需要)，在 fun_list 中找到其对应的函数对象 fun，在 fun 的完整定义中使用本循环片段替换其对应的注释；接下来，在 new_file 中使用上述仅包含一个循环片段的 fun 函数定义替换其声明以构造新的完整程序；最后，为了避免程序命名出现重复和歧义，将重构后的 new_file 内容作为字符串，采用其长度为 5 个字节的哈希码作为文件名称并存储在 after_extract 文件目录中。仍以图 3.4 所示内容为例，若针对循环片段 1 进行重构，其重构结果如图 3.6 所示。

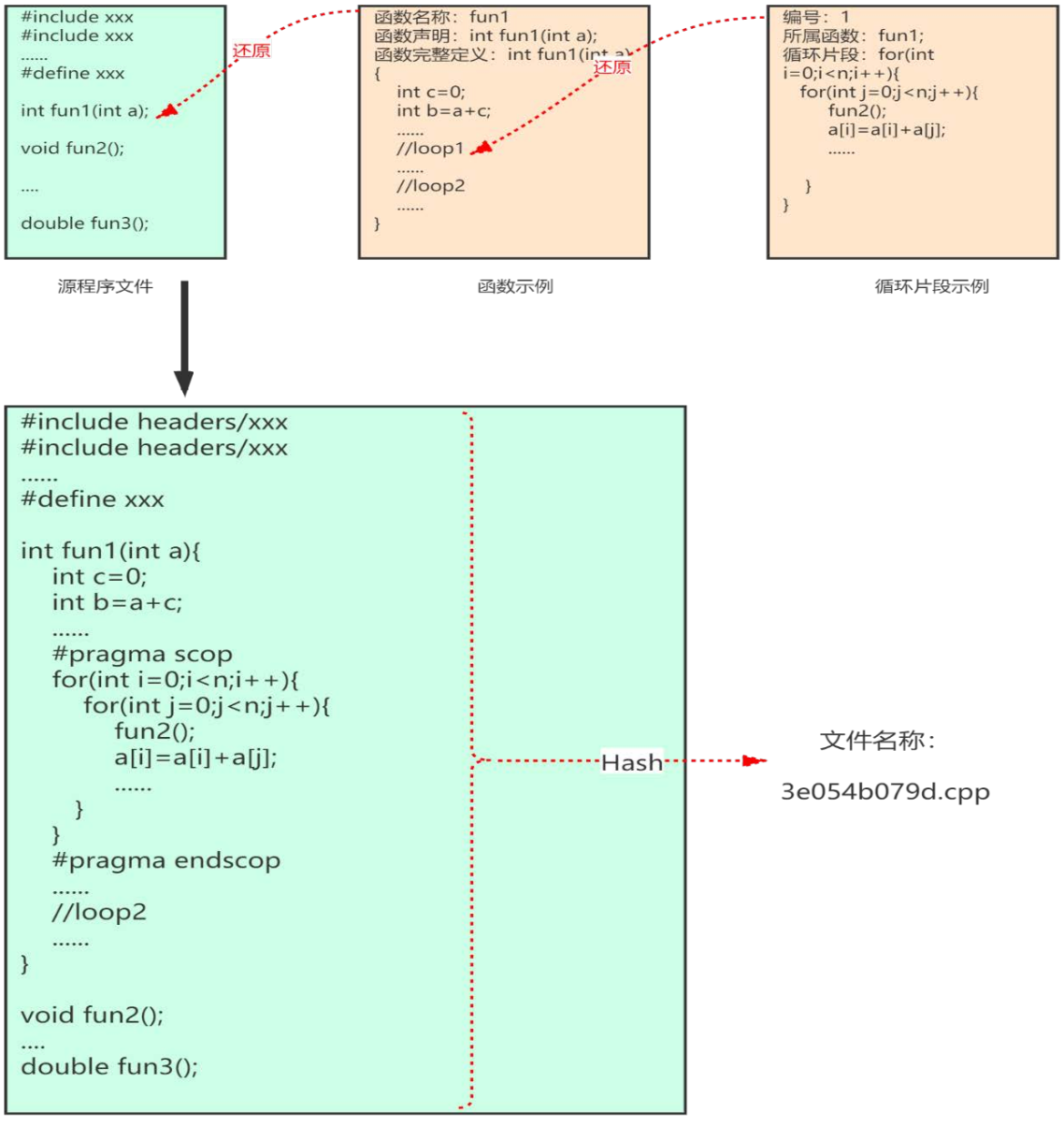


图 3.6 重构结果示例图

3.2.3 编译检验

为了保证重构后的代码可用，本文在代码重构后对新生成的代码文件使用Clang 编译器进行了逐一编译检验。在此过程中，本文使用如图 3.7 所示的编译命令直接将 C/C++源码编译为了 LLVM IR 文件以便后期使用。为了避免编译器优化对串行程序中关于可并行代码段的特征产生影响，编译选项中的优化等级采用-O0 且不附加任何编译优化选项。由于数量较多，本文为编译过程开发了对应的自动化脚本。脚本执行过程中，对于编译通过的代码文件，将其生成的 IR 文件统一存储在 IR/total 目录中；对于编译未通过的代码文件，将其编译日志进行记录(logs/compile_error.log)并维护一个产生编译错误的文件列表 error_list。在脚本结束后对产生编译错误的文件进行人工核验，根据 error_list 和编译日志将其中产生编译错误的地方进行人工修正并重新编译；若存在无法人工修正的代码文件，则将其对应的 C/C++文件从 after_extract 文件夹中删除。

```
clang++ -S -emit-llvm -std=c++11 -O0 -march=native input_file output_file
```

图 3.7 编译命令

3.3 数据标注

数据的标注工作，是指为搜集到的每一份数据都提供一个类别标签，使得该标签作为数据的一部分存在于数据集中，即标签也属于数据。如果说数据是深度学习模型的输入，那么深度学习模型的任务则是学习数据的某些特征，建立这些特征与标签之间的对应关系，训练好的深度学习模型的作用则是给没有标签的数据贴上正确的标签。显然，数据标注的首要任务是确定标签的种类和内容。根据本文的研究内容，需要将 3.2 节中处理好的数据按照建议并行化和不建议并行化的标准分为两类。

由于串行程序能否被并行化并没有定性指标，为了统一数据标注工作的标准本文对于所有重构后的代码文件均首选使用 Pluto 编译器，基于 Polyhedral 模型进行从 C/C++到 OpenMp 的源到源转换。对于每一份代码文件，执行如图 3.8 所示命令，命令相关参数含义如表 3.4 所示。

```
./polycc ../data/pre_data/after_extract/%s  --noprevector --tile --parallel --innerpar -o output_file
```

图 3.8 Pluto 转换命令

表 3.4 Pluto 命令参数表

参数	含义
--noprevector	不进行向量化标注
--tile	进行循环分解
--parallel	进行并行性挖掘并转换为 OpenMP
--innerpar	发掘内层循环并行性

为了更好的发现串行程序的并行性，我们允许 Pluto 编译器进行循环分解并且同时挖掘外层循环和内层循环的潜在并行性，此步骤的转换结果均存储在 after_pluto 文件目录下，具体的转换结果如图 3.9 所示。不难发现，Pluto 编译器对于可以并行的循环片段会在合适的位置添加“#pragma omp parallel ...”编译制导语句，而对于不可并行的循环片段则保持原文件不变。由于我们重构后的每段代码有且仅有一个循环片段，因此仅需对转换结果进行遍历，对于在其内容中发现“#pragma omp parallel”字样的文件，将其对应的 IR 文件从 IR/total 目录移动至 IR/parallel，而对于不可并行的文件，则将其将其对应的 IR 文件从 IR/total 目录移动至 IR/unparallel。如此一来，就较为方便地对大多数重构后的代码进行了快速标注。

```
#include <omp.h>
#include <cmath>
#include <iostream>
using namespace std;
void init_array(int m, int n){
  int t1, t2, t3, t4;
  int lb, ub, lbp, ubp, lb2, ub2;
  register int lbv, ubv;
  /* Start of CLooG code */
  if ((m >= 1) && (n >= 1)) {
    lbp=0;
    ubp=floord(m-1,32);
    #pragma omp parallel for private(lbv,ubv,t2,t3,t4)
    for (t1=lbp;t1<=ubp;t1++) {
      for (t2=0;t2<=floord(n-1,32);t2++) {
        for (t3=32*t1;t3<=min(m-1,32*t1+31);t3++) {
          for (t4=32*t2;t4<=min(n-1,32*t2+31);t4++) {
            C[t3][t4] = ((t3+t4) % 100) / m;
            B[t3][t4] = ((n+t3-t4) % 100) / m;
          }
        }
      }
    }
  }
/* End of CLooG code */
//loop1001
}
```

可以并行

```
#include <iostream>
using namespace std;
int fun(int fibo[100],int temp){
  int i=0,n=10;
  int t1, t2;
  int lb, ub, lbp, ubp, lb2, ub2;
  register int lbv, ubv;
  /* Start of CLooG code */
  if (n >= 3) {
    for (t1=2;t1<=n-1;t1++) {
      temp=fibo[t1-1]+fibo[t1-2];
      fibo[t1]=temp;
    }
  }
  /* End of CLooG code */
}
```

不可并行

图 3.9 Pluto 转换结果示例图

与编译类似，这一过程仍然通过自动化脚本完成。由于 Polyhedral 模型本身

存在如 2.4 所述限制，Pluto 编译器并不能转换所有重构后的代码文件。因此在脚本执行过程中同样记录了发生转换错误的转换日志(logs/pluto_error.log)并维护一个转换错误列表 error_list。脚本执行结束后，本文采用人工与其他代码分析工具结合的方式，对于转换失败的文件进行了重新标注。首先，在此类文件的循环片段上无差别的加上“#pragma omp parallel for”标记，编译执行该文件并记录其修改前后的执行时间，通过二者的差异来判断其是否具备并行潜力。然后，使用 Intel® Advisor XE 对上述文件进行再次分析与判断。将人工与工具两次判断的结果进行比对，对于结果相同的文件按照其具体分类进行吸收，对于结果不同的文件则予以舍弃。

经过上述两个层次的标注后，重构后的代码文件及其对应的 IR 文件被按照是否建议使用 OpenMP 进行并行优化的标准分为了两个部分并分别存储在不同的文件目录中，文件目录名称即为它们所对应的标签名称，至此数据标注工作完成。

3.4XFG 的生成

如 2.3 节所述，XFG 是一种同时包含程序数据流信息和控制流信息的一种图结构表示，其每个节点均代表一条 LLVM IR 语句。在开始描述 XFG 的构造过程之前，需要对其中涉及到的 LLVM IR 语句基本结构进行说明。如图 3.10 所示，一条 LLVM IR 语句一般主要由输出变量标识符、IR 指令、操作数据类型、输入变量标识符、其他参数、元数据、注释等 7 部分组成。其中，前 5 部分是 LLVM IR 的语义载体，是我们构造 XFG 的主要依据。

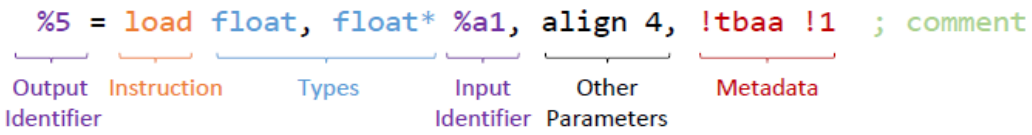


图 3.10 LLVM IR 语句基本结构示例图

因此，为了简化构造过程，需要先对 LLVM IR 文件进行如下预处理操作：(1)构造函数列表：对于每一个文件，构造一个用于存储文件中声明的所有函数名称的列表，用于 XFG 构造过程中标识本文件中声明的函数；(2)删除不属于 IR 语句的内容：LLVM 编译的 IR 文件中，还记录了如图 3.11 所示 IR 文件的源文件名称、编译环境和编译选项等信息，此类信息与程序语义无关，不应被用来构造 XFG，因此将其删除；(3)取消缩进与空行：对文件的每一行采取内容前后删除空格操作，同时删除文件中的空白行；(4)删除行尾的元数据：将每一行末尾以叹号开头的元数据移除；(5)将占用多行的单条语句合并为一行。接下来将按照如图 3.12 所示流程，对于 IR/parallel 及 IR/unparallel 目录下的每一个 IR 文件分两个阶段构造 XFG。

```
; ModuleID = '../data/handled/parallel/3e054b079d.cpp'
source_filename = "../data/handled/parallel/3e054b079d.cpp"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 6.0.1-svn334776-1~exp1~20190309042730.123 (branches/release 60)"}
```

图 3.11 非 IR 指令内容示例图

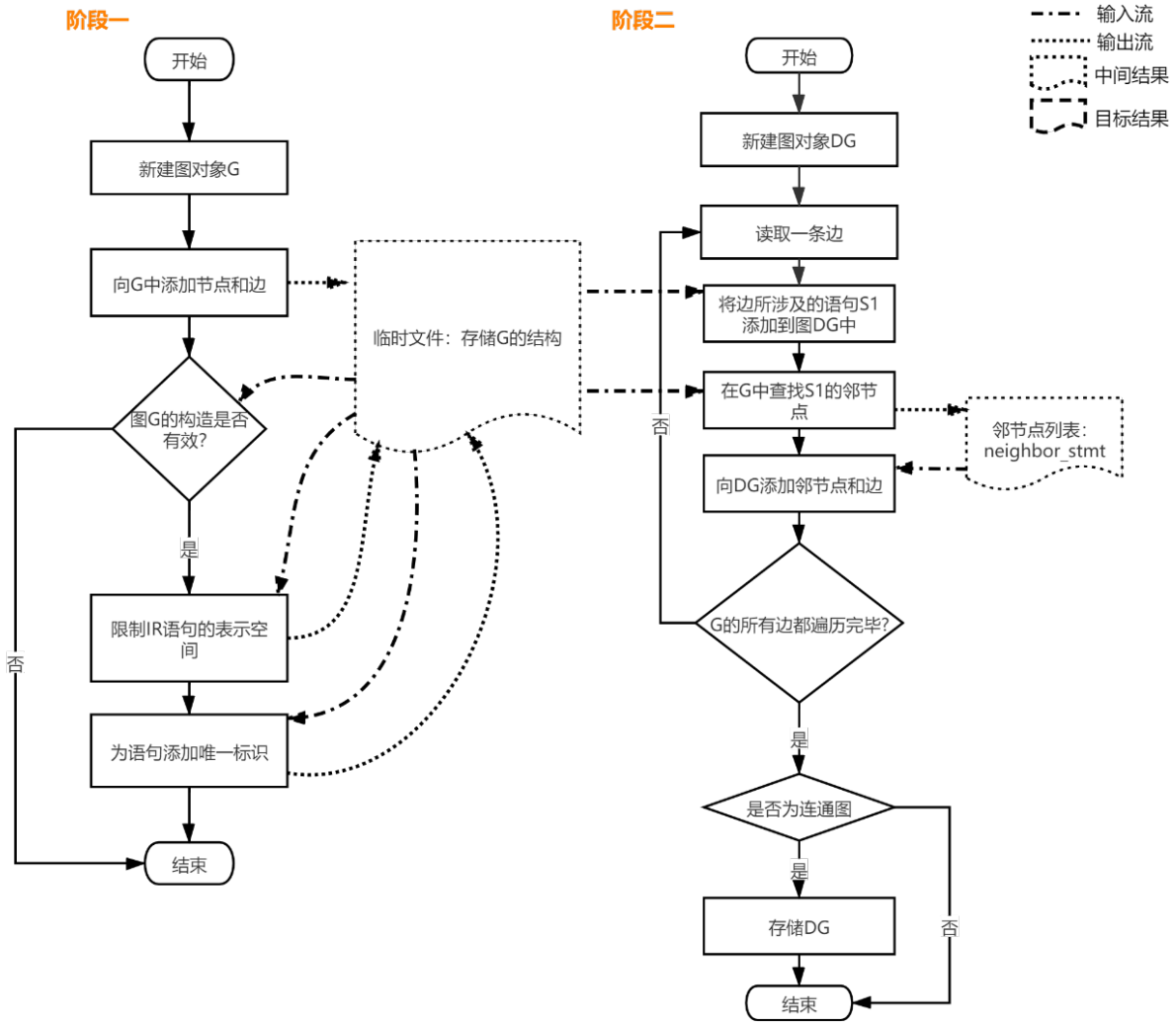


图 3.12 构造 XFG 的算法流程图

第一阶段，遍历 IR 文件中的每一行语句，按照如下步骤构造基于标识符的 XFG：

(1)添加节点和边：严格按照全局变量声明、方法定义、基本块、变量赋值、存储指令、分支指令、转换指令、方法调用、返回语句、不可达代码的顺序，向图中添加节点和边。这一步骤中，会默认首先向图中添加唯一的全局根节点，基

本块、变量声明等与其他标识符没有相关性的节点均为其添加一条直接与根节点相连的边。如果一行语句中出现了两个标识符，那么可以简单的认为这两个标识符节点之间存在一条边。

(2)存储图结构：将步骤(1)中构建的图结构存储在临时文档中，如图 3.13 所示。其中，节点的信息使用其标识符进行表示，边的信息则采用形如[(node1,node2),statement]的格式分别存储其相关节点以及确定这条边的 IR 语句。

(3)使用以下标准评估此图构造的有效性：(a)每个节点都有 ID；(b)所有语句都添加到了图中；(c)没有孤立的节点；(d)节点标识符没有重复；(e)列出多重边；(f)确保为连通图。

```
Nodes (6)
--------------------------------------------------------------------------------
caxpy__0_%38                                ,
caxpy__0_%39                                ,
caxpy__0_%40                                ,
caxpy__0_%41                                ,
caxpy__0_%42                                ,
caxpy__0_%43                                ,

Edges (6)
--------------------------------------------------------------------------------
(caxpy__0_%38          , caxpy__0_%42          ) %42 = fsub <4 x float> %38, %41
(caxpy__0_%39          , caxpy__0_%43          ) %43 = fadd <4 x float> %40, %39
(caxpy__0_%40          , caxpy__0_%43          ) %43 = fadd <4 x float> %40, %39
(caxpy__0_%41          , caxpy__0_%42          ) %42 = fsub <4 x float> %38, %41
(caxpy__0_%42          , caxpy__0_%48          ) %48 = fadd <4 x float> %strided.vec59, %42
(caxpy__0_%43          , caxpy__0_%49          ) %49 = fadd <4 x float> %43, %strided.vec60
```

图 3.13 基于标识符的 XFG 存储结构示例图

(4)遍历图的边，在其语句中用“<%ID>”替换所有“%### xxx”形式的局部标识符，用“<@ID>”换所有“@### xxx”形式的全局标识符，用“<LABEL>”替换所有的基本块标识符。将语句中无特殊标识意义的整数、浮点数、字符串分别使用“<INT>”、“<FLOAT>”、“<STRING>”替换，将语句末尾的强类型标记(i32、i64 等)使用<TYP>替换。本步骤前后的对比如图 3.14 所示，可以发现语句中的个性化信息被抹除，仅剩下了指令内容与操作逻辑(如对浮点型局部变量数据的加操作)，对 IR 的表示空间进行更进一步的限制。

```
%43 = fadd <4 x float> %40, %39
%41 = fmul <4 x float> %broadcast.splat57, %strided.vec53
%48 = fadd <4 x float> %strided.vec59, %42
%49 = fadd <4 x float> %43, %strided.vec60
%41 = fmul <4 x float> %broadcast.splat57, %strided.vec53
```

步骤 (4) 之前

```
<%ID> = fadd <4 x float> <%ID>, <%ID>
<%ID> = fmul <4 x float> <%ID>, <%ID>
<%ID> = fadd <4 x float> <%ID>, <%ID>
<%ID> = fadd <4 x float> <%ID>, <%ID>
<%ID> = fmul <4 x float> <%ID>, <%ID>
```

步骤 (4) 之后

图 3.14 步骤(4)前后对比效果图

(5)为边信息中的语句添加唯一标识符：在 XFG 中，每一条 LLVM IR 语句均代表一个不同的节点。但是由图 3.14 可知在步骤(4)之后，不同的语句间若指令内

容与操作逻辑相同，则变得没有差异，因此需要在其末尾根据其出现的位置加上唯一标识以示区别(如图 3.15 所示)。

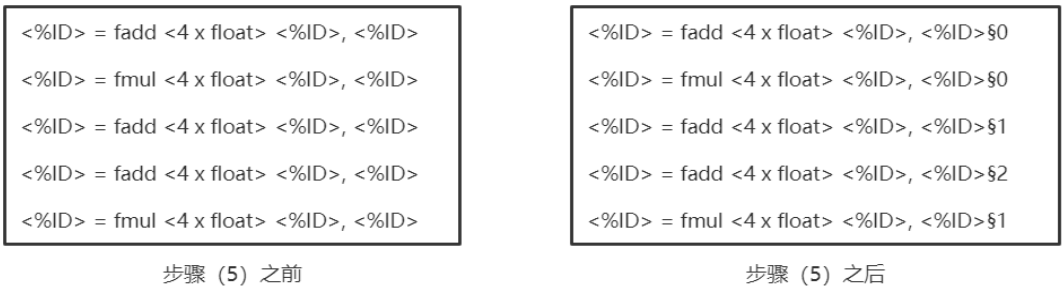


图 3.15 步骤(5)前后对比效果图

第二阶段，读取第一阶段的构造成果 G，对于其每一条边 E，按照以下步骤继续构造基于 LLVM IR 语句的 Dual-XFG：

(1)将 E 涉及到的语句 S1 作为节点添加到 Dual-XFG 中。

(2)在图 G 的所有边中找到包含边 E 的两个顶点其中任意一个的 IR 语句集合 neighbor_stmt 作为 S1 的上下文，即 S1 的邻节点。

(3)遍历 neighbor_stmt，将其中的语句添加到 Dual-XFG 中，并添加 S1 与其之间的边。

(4)检查 Dual-XFG 是否为连通图。

(5)存储 Dual-XFG 到相应的文件夹中(XFG/parallel 或 XFG/unparallel)，存储格式如图 3.16 所示。

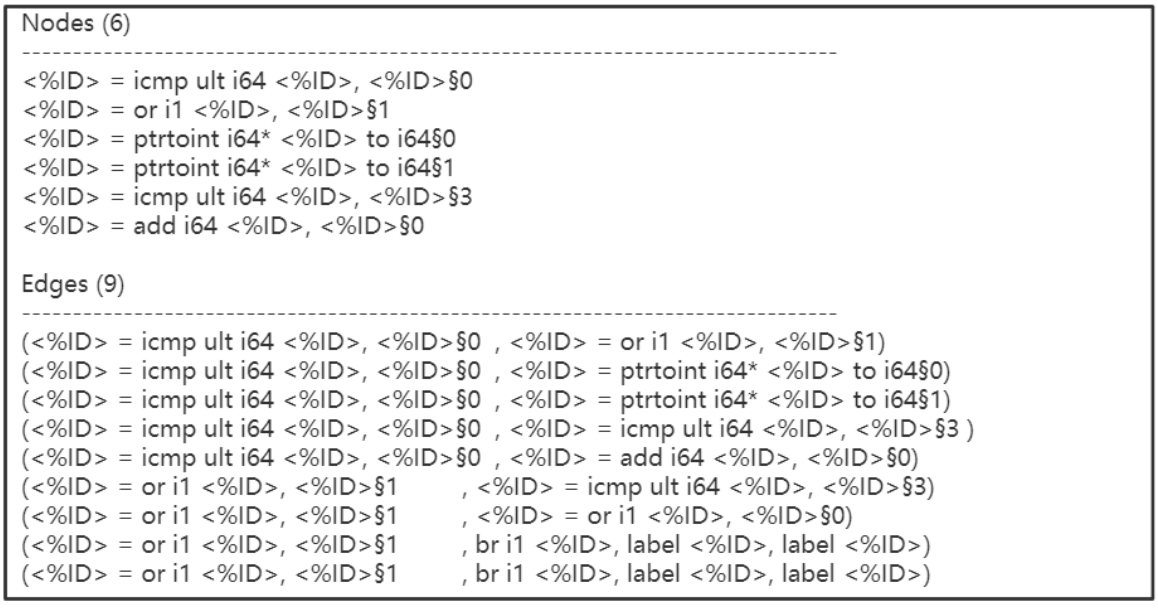


图 3.16 基于 LLVM IR 语句的 Dual-XFG 存储结构示例图

至此，程序的上下文流图的构造工作全部完成，数据集 GFCPD 的构造也随之结束，最终数据格式如图 3.17 所示。其中，Dual-XFG 是一个图结构的数据对象，lable 是其所对应的类别标签(也即数据文件所在的文件夹名称：parallel 或

unparallel)。

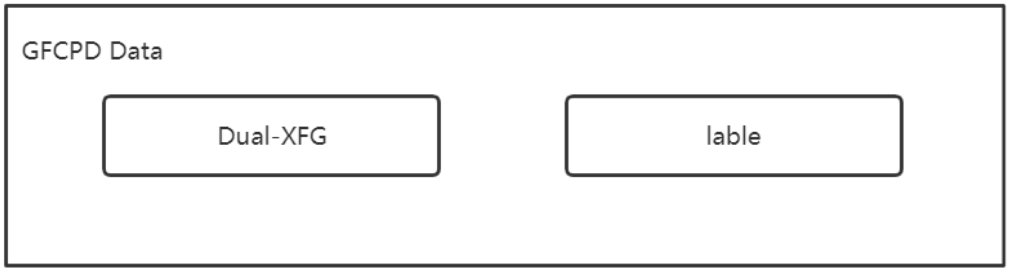


图 3.17 GFCPD 数据格式示意图

3.5 数据集规模

由于目前的研究中，并没有专门用于挖掘串行程序并行性的图结构数据集。因此为了使深度学习模型具备更佳的表现，在 GFCPD 数据集的规模问题上本文一方面考虑了接下来的研究中使用的深度图卷积神经网络模型 DGCNN[58]在训练过程中所使用的图数据集，如表 3.5(1~8)所示；另一方面由于本研究也可以认为是一个二分类问题，考虑了一些常用的二分类问题图数据集，如表 3.5(9~14)所示。

表 3.5 DGCNN 使用的数据集信息表

序号	名称	图的总数	类别	节点数均值	边的数量均值
1	MUTAG	188	2	17.93	19.79
2	PTC_MR	344	2	14.29	14.69
3	NCI1	4110	2	29.87	32.3
4	PROTEINS	1113	2	39.06	72.82
5	D&D	1178	2	284.32	715.66
6	IMDB-B	1000	2	19.77	96.53
7	IMDB-M	1500	3	13	65.94
8	COLLAB	5000	3	74.49	2457.78
9	AIDS	2000	2	15.69	16.2
10	BZR	405	2	35.75	38.36
11	BZR_MD	306	2	21.3	225.06
12	COX2	467	2	41.22	44.54
13	DHFR	467	2	26.96	48.42
14	KKI	83	2	77.52	198.32
均值	--	1297.21	--	48.29	277.97
最大值	--	5000	--	284.32	2457.78

根据对表 3.5 中所列的 14 个常见图数据集的分析结果，为了使 DGCNN 模型可以更好地发挥其性能，本文对 GFCPD 数据集的规模做了如下规划：(1)图的总数不做限制；(2)节点数目平均值不大于 284.32 且不小于 48.29；(3)边的数量均值不大于 2457.78 且不小于 277.97。由于通过 3.1~3.4 节构造的原始数据集相关统计信息与上述规划相差甚远，本文对 GFCPD 数据集进行了进一步的裁剪，裁剪后的统计信息如表 3.6 所示。

表 3.6 GFCPD 数据集信息表

数据集状态	图的总数	类别数目	节点数均值	边的数量均值
裁剪前	1074	2	493.72	4583.05
裁剪后	850	2	130.24	539.32

3.6 本章小结

本章根据串行程序并行性识别任务的实际需要，从图的角度出发，实现了从 C/C++串行源码自动生成 GFCPD 数据的方法，并详细介绍了数据集 GFCPD 的构造过程。列举了其数据来源，描述了其数据处理方法、数据构造原则及相关算法流程，并就数据集规模的合理性问题做了一些论证。

第 4 章 基于 DGCNN 的串行程序并行性识别方法

由 1.2 节可知，在程序的并行性分析问题上，出现过使用决策树、支持向量机等传统的机器学习方法进行串行程序可并行性识别的相关研究，这为使用深度学习方法实现这一目的提供了可行性参考。虽然目前还并未出现将深度学习方法应用于串行程序可并行性识别的研究成果，但代码的语义表示、不同算法的代码分类、代码缺陷检测等问题的深度学习解决方案为此提供了理论与方法的支持。综上所述，深度学习的方法在串行程序可并行性识别问题上具备着良好的可行性。本章以自动识别串行程序中可以并行的代码片段为研究目的，提出了一种基于深度图卷积神经网络(DGCNN)架构的识别方法，将深度学习方法应用于串行程序的并行性识别研究。接下来的几个小节将从数据预处理、模型建立、实验结果与分析三个方面对该方法进行详细介绍。

4.1 数据预处理

在第 3 章中，考虑到数据集的通用性，为了使更多的研究可以直接使用 GFCPD 数据集，在其构造过程中并未对其做任何专适用于本研究的预处理工作。因此，为了更好地进行接下来的研究，需要结合本文所使用的深度图卷积神经网络模型 DGCNN 对实验数据进行进一步的预处理操作。在此之前，首先对 DGCNN 的输入数据格式进行如图 4.1 所示说明。对于每一个输入数据对象，graph 为一个图结构的数据对象，包含了节点和边的集合，其中节点使用其编号表示，边使用节点编号的有序对表示；lable 是图 graph 所对应的分类标签；node_tags 是一个列表对象，包含了图 graph 所有节点对应的节点类型标签；node_features 是一个列表对象，包含了图 graph 所有节点的特征向量。对比图 3.17 不难发现对于 GFCPD 而言，节点的类型标签和节点的特征向量是两个新的属性。显然，接下来的数据预处理工作就是为 GFCPD 数据集中的数据，确定并添加上述两个属性，用于做为 DGCNN 模型的训练数据。

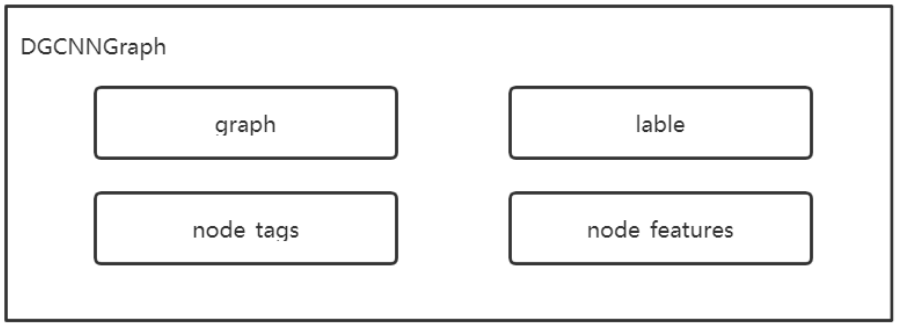


图 4.1 DGCNN 输入数据格式示意图

(1)确定节点的类型标签

如 3.4 节所述，GFCPD 数据集中，XFG 中的节点是一条 LLVM IR 语句，且在其构造过程中对 IR 语句进行了处理，使其仅保留了指令内容与操作逻辑。这决定了在本步骤中节点的类型标签只可以根据其指令内容或操作逻辑进行确定。

考虑到仅通过人工总结并不一定能够完全覆盖所有的数据操作逻辑，同时为了限制标签数量，本文对 LLVM IR 的指令类型(如加、减、逻辑与、逻辑或等)进行了如表 4.1 所示总结，并以此作为 XFG 中节点的类型标签。其中 0 号标签表示未知指令，用于兼容特殊情况。对于 0 号标签以前外的其他标签，均设计了一个或多个正则匹配规则(一个指令类型可能对应多个指令，因此需要设计多个正则匹配规则，以列表的形式保存)，可以通过遍历 XFG 的每一个节点，使用正则表达式为其匹配正确的类型标签。

表 4.1 节点类型标签信息表

类型编号	节点类型	正则表达式
0	!UNK	--
1	GEP	<%ID> = getelementptr .*
2	addition	<%ID> = add .* <%ID> = fadd .*
3	allocate on stack	<%ID> = alloca .*
…	…	…
45	xor	<%ID> = xor .*

(2)确定节点的特征向量

文献[34]中将 XFG 定义为基于 LLVM IR 的可学习的代码语义表示方法，并通过 DeepWalk 的方法对其数据集(如表 4.2 所示)所生成的 XFG 进行图嵌入训练，为 XFG 中的每一个节点所代表的 LLVM IR 语句生成了向量表示。该文通过聚类，类比，语义测试等方法验证了上述向量能够较好地表示代码语义，同时通过算法分类、异构计算平台预测等实验证明了上述向量的可学习性。

表 4.2 图嵌入训练所使用的数据集信息表

涉及学科	源码程序
机器学习	Tensorflow
操作系统	Linux kernel
计算机视觉	OpenCV NVIDIA samples
高性能计算	AMD APP SDK BLAS

基准测试	NAS(NPB) Parboil PolybenchGPU Rodinia SHOC
科学计算	COSMO

由于 GFCPD 是一个基于 XFG 的图数据集，本文所使用的节点特征向量同样需要对 XFG 进行图嵌入训练。考虑到文献[34]的图嵌入过程所使用的数据集进行了较为全面地学科覆盖，基本包含了 GFCPD 原始数据中的所有程序或其类似算法，因此使用该嵌入结果作为 GCFCPD 数据集的节点特征向量是一个合理且不错的选择。该嵌入结果的相关信息如表 4.3 所示，其中共包含两个文件，一个存储的是 LLVM IR 语句及其编号，另一个存储的是语句编号及其对应的嵌入向量。可以通过字符串完全匹配的方法确定 XFG 中节点所对应的编号，从而确定其节点的特征向量。

表 4.3 图嵌入结果详细信息

词典大小(IR 数量)	向量维度
8565	(1, 200)

(3)总体流程

确定了节点的类型标签及其特征向量的表示方法后，就可以开始对 GFCPD 数据集中的数据进行进一步的改造，以适配 DGCNN 模型。总体流程如图 4.2 所示，对于每一个 GFCPD 数据做如下操作：首先从 GFCPD 数据集中读取一个 Dual-XFG，获得其节点列表，即 LLVM IR 语句的列表；然后遍历该列表，对于每一个 IR 语句，通过正则匹配确定其类型标签，并将其标签所对应的编号存储在 node_tag 列表中，通过字符串完全匹配的方法从嵌入结果中查找 IR 语句对应的特征向量，存储在 node_feature 中；接下来对 XFG 中的节点编号，将节点内容使用其编号表示，将边使用节点编号的有序对表示，生成新的图 graph；最后将原数据 lable、graph、node_tag、node_features 按照图 4.1 所示格式存储。

4.2 基于 DGCNN 的串行程序并行性识别模型

4.2.1 DGCNN 的构建

DGCNN 是在文献[58]中提出的一种端到端的用于图分类的深度学习架构。与现有的图神经网络相比，它主要具有如下优势：首先它直接接受图结构的数据作为输入，而无需将图数据先转换为张量；其次，它可以通过对顶点的特征进行排序来更好的学习图的拓扑特征，而并非对顶点特征进行汇总；最后，实验证明它

在许多图类型的基准数据集上相较于现有的图神经网络具有更好的性能。

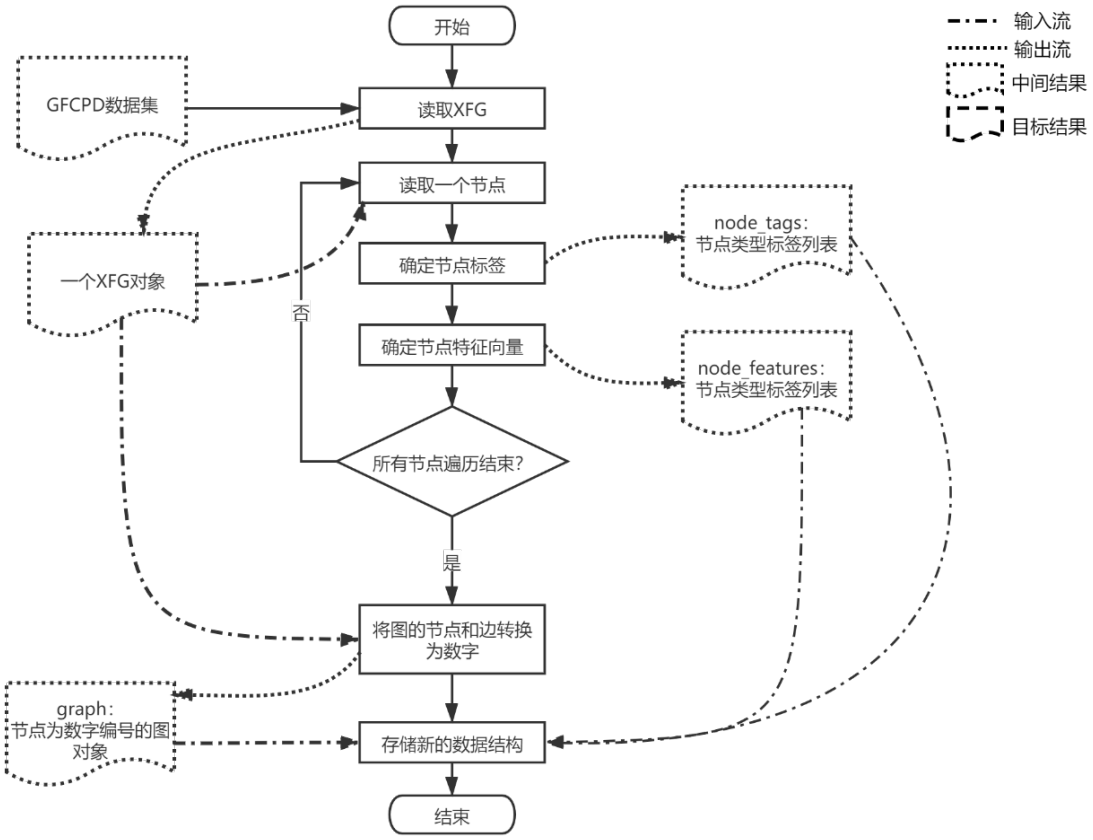


图 4.2 数据预处理总体流程图

DGCNN 的总体架构如图 4.3 所示，其主要由三个部分组成：(1)图卷积层，输入的图数据首先通过多个图卷积层,其中每个节点的特征在其邻节点之间传播，以此来提取图中节点的局部结构特征；(2)SortPooling 层，根据图卷积层的输出，按照每个节点在图结构中的不同角色及其重要程度进行排序，并统一输出为 k*n 的矩阵，其中 k 表示将使用排序后前 k 个节点的特征作为图的表示，n 表示卷积后节点特征向量的维度；(3)传统的卷积层和全连接层，读取图的表示并进行类别预测。为了更好地理解，接下来将以本文所使用的数据为例，对其中的一些细节做出说明。

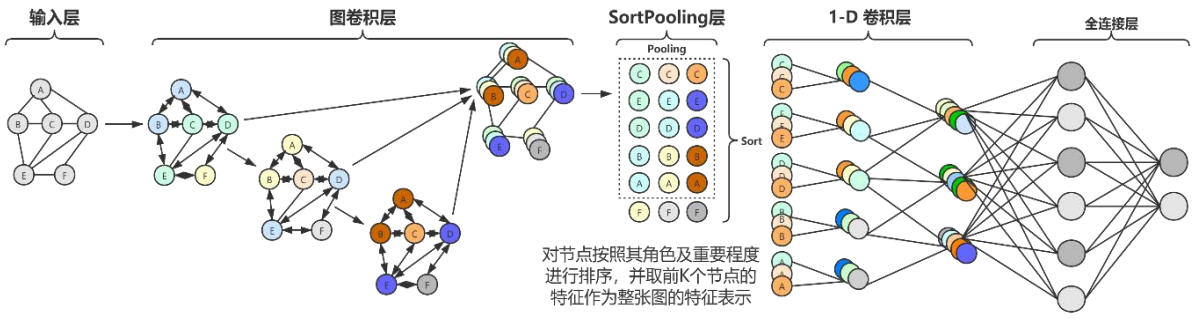


图 4.3 DGCNN 的总体结构示意图

如 4.1 所述，本文所使用的图数据中，节点嵌入向量的维度为 1*200，节点的类型标签为 46 个，我们对节点的类型标签进行如图 4.4 所示过程的 One-hot 编码，并将其连接在节点的嵌入向量后，将新生成的维度为 1*246 的向量作为节点的特征向量进行输入。实验中共使用了四层图卷积对每一个图节点进行卷积操作，将每次卷积的结果连接起来作为其卷积后的节点特征向量，其中图卷积层的输出维度分别为 32、32、32、1，那么经过卷积后每个节点的特征向量的维度均为 1*97。假设一张图具有 m 个节点，那么经过卷积后该图的表示为一个 m*97 的特征矩阵，每一行代表一个节点的特征向量。

SortPooling 层读入上述维度为 m*97 的图特征矩阵，由于该矩阵每一行的最后一列是最后一层图卷积的输出，可以被认为是融合了某一个节点本身特征和类型标签的最佳表示，因此我们使用每一行的最后一列作为标准，对该矩阵的所有行进行重新排列，以此来表示图中的节点在图结构中的不同角色以及其功能。如此可以推出，如果不同图中的两个节点具备相似的角色和功能，那么在两张图的特征矩阵中，它们的相对位置相同。因此，可以认为上述排序结果决定了每个节点及其特征在图分类中的重要程度。

在训练过程中，可以设定一个 k 值，表示使用每张图排序后的前 k 个节点，即特征矩阵的前 k 行作为图的整体表示(对于节点不足 k 个的图数据，将在其特征矩阵后使用 0 进行补齐)，以进一步提取更重要的特征进行下一步的分类。

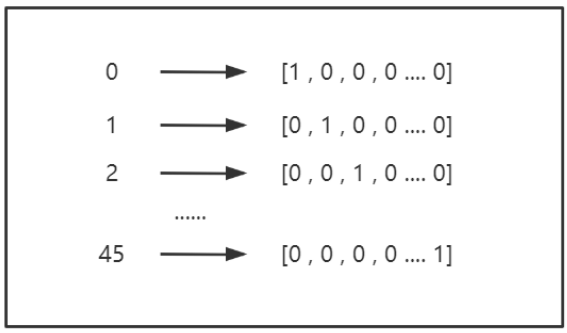


图 4.4 One-hot 编码示例图

4.2.2 损失函数的确定

原始 DGCNN 架构中，在最后阶段进行图的分类预测时采用的是 softmax 交叉熵损失函数(在 Tensorflow 中对应为 softmax_cross_entropy_with_logits_v2 函数)，该函数是在分类问题中较为常用的的逻辑损失函数。然而该函数在处理噪声数据时(即数据标签存在错误的情况)，往往会暴露出以下两大缺陷：(a)由于逻辑损失函数没有上界，错误的数据标签在损失值计算过程中导致出现了较大的异常值，对总体损失值起到了决定性作用，可能会牺牲部分正确样本，对训练结果产生不良影响；(b)由于逻辑损失函数在传递过程尾部以指数方式快速衰减，因此为了使

错误标签的函数值更加接近于 0，会将决策边界更接近于错误的数据标签，降低训练的准确率。针对上述两个问题，Ehsan Amid 等人于 2019 年提出了一种双参数可调的双稳态逻辑损失函数(Bi-Tempered Logistic Loss)[59]，通过引入两个参数 t1，t2 分别限制损失函数的上界和其在传递过程尾部的衰减速度，将噪声数据对分类问题训练结果的影响降到了最低。其中，0<t1≦1 限制了损失函数的上界，t1 越小，它对损失函数上界的限制就越强；t2≧1 限制了损失函数在传递过程尾部的衰减速度，t2 越大，其衰减速度也就越小。

考虑到串行程序中的循环片段能否并行，并没有定性的指标进行区分，虽然本文在 3.3 节中采用 Pluto 编译器对数据的分类标签进行了标准上的统一，但对于不能被 Pluto 编译器标注的数据仍采用了人工的方式进行标注。出于严谨考虑，本文在此假设 GFCPD 数据集中的数据标签存在某些随机产生的错误，为数据集引入了随机噪声，并通过分别使用两种损失函数进行实验对比，对上述假设进行了验证并确定了串行程序并行性识别模型中的损失函数，实验结果如图 4.5 所示。

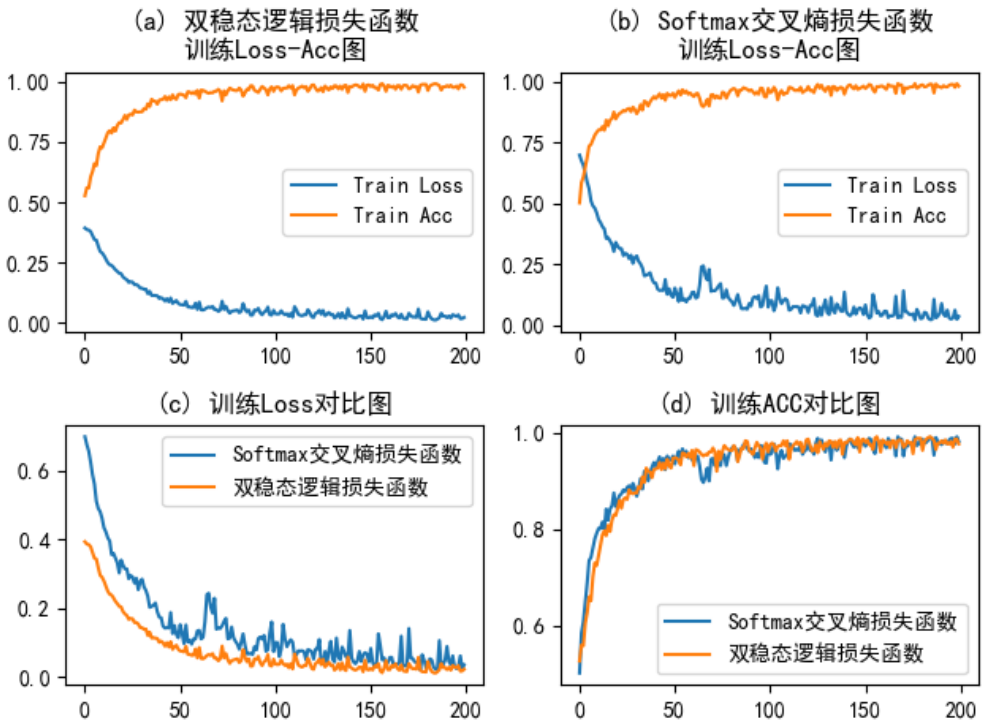


图 4.5 损失函数对比实验效果图

综合图 4.5(a)和图 4.5(b)可以发现，双稳态逻辑损失函数在训练过程中的整体表现较 softmax 交叉熵损失函数更加稳定。从图 4.5(c)与图 4.5(d)中可以看出来，双稳态逻辑损失函数在损失值的计算过程中，损失值的下降更加稳定，收敛速度较快；在训练准确率方面，双稳态逻辑损失函数也可以更好的收敛。本次实验的相关参数及所训练的模型在测试集上的准确率数据如表 4.4 所示，其中 Sortpooling-k 被选取为 GFCPD 数据集中所有图的节点数量均值，即每张图均使用其排序后的前 131 个节点的特征向量作为图的特征表示；双稳态逻辑损失函数

中的参数 t1,t2 的确定参考了文献[59]中针对随机噪声所进行的实验参数设置，并通过多次实验进行了调整。从表 4.4 中的数据可知，在测试集的准确率方面双稳态逻辑损失函数仍旧占据上风。

表 4.4 损失函数对比试验参数及测试结果表

损失函数	数据集	Sortpooling-k	Num_epoch	Batch_size	Test-Acc(%)
Softmax 交叉熵损失函数	GFCPD	131	200	1	84.20
双稳态逻辑损失函数(0.5,1.5)	GFCPD	131	200	1	87.59

实验表明,本文针对 GFCPD 数据集的数据标签中存在随机噪声的假设成立，且双稳态逻辑损失函数可以有效并显著地将噪声对分类结果的影响降低。因此，本文在串行程序并行性识别模型中，采用了双稳态逻辑损失函数并进行了一系列的对比试验。

4.3 实验结果与分析

为了验证本文所建立的基于 DGCNN 的串行程序并行性识别模型在串行程序并行性识别问题上的有效性与优越性，本文将该模型与基于程序静态分析的识别方法、结合动态特征的机器学习识别方法和基于深度学习的算法分类模型分别进行了对比。在所有对比实验中，DGCNN 架构的相关参数设置均如表 4.5 所示。

表 4.5 对比实验中 DGCNN 模型参数表

参数名称	参数值
图卷积层	4
图卷积层输出维度	32，32，32，1
学习率	0.0001
Sotpooling-k	131
Num_epoch	100
Batch_size	1
(t1，t2)	(0.5,1.5)

(1)对比实验一：与基于程序静态分析的识别方法对比

本文在数据的标注过程中，所使用的 Pluto 编译器对于串行程序中的循环片段是否可以并行，采用的就是经典的静态分析策略，通过将循环片段转换为 Polyheadral 模型，在有限的依赖关系集合内，通过线性整数规划策略对其并行方案进行求解。

在此次对比实验过程中，Pluto 编译器对于 GFCPD 数据集的准确率数据来源

于数据标注过程的统计信息，实验结果如表 4.5 所示。显然，在裁剪前的 GFCPD 数据集上，基于 DGCNN 的串行程序并行性识别模型的准确率明显高于基于静态分析的源到源编译器 Pluto。

表 4.6 对比实验一实验结果表

方法	数据集	准确率(%)
Pluto	裁剪前的 GFCPD	71.60
DGCNN	裁剪前的 GFCPD	86.79

(2)对比实验二：与结合动态特征的机器学习识别方法对比

所谓程序的动态特征主要是指在程序的运行过程中所收集到的信息，常见的程序动态特征有循环中的指令数量、循环的执行时间、关键路径长度、循环内指令间的依赖计数、外部指令对循环内指令的依赖计数以及循环内指令对外部指令的依赖计数等信息。

文献[23]中使用了包括支持向量机(SVM)、决策树(Decision Tree)、增强学习算法(AdaBoost)等传统的机器学习方法对上述程序的动态特征进行了学习并实现了对串行程序并行性的预测。本文将基于 DGCNN 的串行程序可并行性识别模型在 NPB 数据集上与其实验结果进行了对比，对比结果如表 4.7 所示。

实验结果表明，基于 DGCNN 架构的串行程序可并行性识别方法的准确率优于 SVM 和决策树。虽然相较于 AdaBoost 算法还有一定的差距，但是考虑到收集程序的动态信息需要运行程序，容易产生较大的时间成本和大量的数据预处理工作，本文所建立的模型在此方面仍具备一定的优越性。

表 4.7 对比试验二实验结果表

方法	数据集	准确率(%)
SVM	NPB	85
Decision Tree	NPB	85
AdaBoost	NPB	92
DGCNN	NPB	87.20

(3)对比实验三：与基于深度学习的算法分类模型对比

文献[34]在验证其对于 XFG 节点的嵌入表示是否能够有效地代表程序语义的实验中，提出了一种基于 LSTM 的神经网络模型(本文称之为 NCC-Model)，并在对 POJ-104 数据集进行算法分类的实验中在达到了 94.83%的准确率，略高于文献[27]中 TBCNN 模型在该数据集上的准确率。考虑到本文在 XFG 的节点特征向量的选择上，与上述实验中的代码嵌入向量相同，所以本文此次实验的对比对象为 NCC-Model，其主体部分由两层 LSTM 和一层 Batch normalization 构成，详细结构如图 4.6 所示。

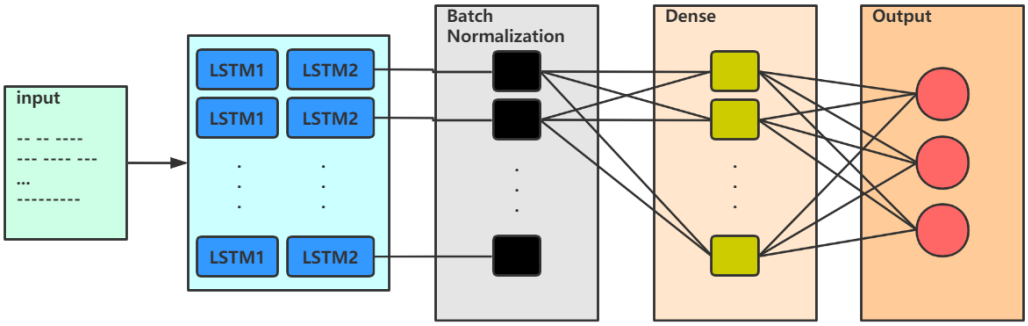


图 4.6 NCC-Model 模型结构示意图[34]

在实验过程中，将 GFCPD 数据集中每一个 XFG 所有节点的特征向量按照其节点编号的顺序逐行添加至图的特征矩阵中，并以此矩阵作为 NCC-Model 的输入数据，针对该模型在 GFCPD 数据集上的表现与 DGCNN 进行了对比。除此之外，本文还在上述模型的输入层之后加入了一层 1-D 卷积层，以此来观察通过卷积层提取特征后，该模型在 GFCPD 数据集上的表现。具体实验结果如表 4.8 所示，实验表明现有的算法分类模型并不适用于串行程序的可并行性识别任务，而基于 DGCNN 的识别模型可以很好的识别串行程序中的循环的并行性。

表 4.8 对比实验三实验结果表

模型	数据集	准确率(%)
NCC-Model	GFCPD	67.44
CNN+NCC-Model	GFCPD	67.97
DGCNN	GFCPD	87.59

综上所述，本文所提出的基于 DGCNN 架构的串行程序可并行性识别方法，在程序可并行性分析任务中的具备良好的可行性与有效性，相较于目前传统的静态分析方法、结合动态特征的机器学习方法以及在算法分类问题上表现良好的深度学习模型均表现出强大的竞争力。

4.4 本章小结

本章基于深度图卷积神经网络(DGCNN)架构，提出了一种用于串行程序并行性识别的方法，并从数据预处理、模型的介绍与改进以及对比实验三个方面详细介绍了如何通过 DGCNN 架构将深度学习的方法应用到程序可并行性分析的研究上来。该方法直接以图数据作为输入，避免了传统程序分析过程中对于程序信息提取或运行程序所造成的时间开销与额外工作量，从另一个角度实现了串行程序的并行性识别工作。实验结果表明，该方法在多数情况下具备良好的稳定性与有效性。

第 5 章 PML 与并行编程平台

如 1.1 所述，即使高性能计算技术正在高速发展，并行程序的开发也仍然还是一个较为复杂的任务。尤其是在医疗成像、气候模拟、药物发现、能源勘探等众多学科交叉领域，近年来出现了在大型集群系统(如：超级计算机)中同时使用两种或多种并行编程模型和框架技术来提高数据处理效率的趋势，但是相关行业软件在并行编程方面对于从业人员并不友好。下面以能源勘探行业为例，对上述领域中的行业软件存在的一些共性问题进行讨论和说明。

虽然能源勘探领域的相关理论发展已经相对比较成熟，且相关研究也比较完善和透彻，但是由于计算机硬件技术和计算能力的限制，相当多的优秀理论仍尚未投入实际的工程应用。近几年来，随着计算机硬件及相关技术的飞速发展，计算能力得到了显著提高，越来越多的能源勘探领域从业人员开始使用 MPI、OpenMP、CUDA 等并行编程技术来提高工作效率与质量。综合对比该行业内较为流行的三种现有的能源勘探软件系统，其中 Seismic Unix 和 Madagascar 主要用于科学研究的开源软件系统，而 GeoEast 则是由由中国石油集团自主研发的商业能源勘探软件。从表 5.1 所示内容可以发现，在功能上三者尽可能多的包含了能源勘探整体流程所需要的算法以及功能；在技术上都支持系统的分布式部署和数据的并行管理；而且 GeoEast 还具备任务的并行调度管理和一些其他特性。但是三者无一例外地都忽略了在用户自主扩展编程过程中的并行编程问题以及单个处理流程内部的线程级并行优化问题。用户需要使用普通的 IDE 在没有任何并行编程指导的情况下去手工编写实现自己的数据处理功能。显然，由于各种并行编程模型和框架之间缺少一个简单易懂，方便易用的统一编程规范接口，彼此在使用方法、系统环境要求等诸多方面的标准并不统一，对上述领域的从业人员在并行编程方面造成了较大阻碍，参差不齐的贡献者水平也使得相关开源软件可扩展性受损、发展受阻。

表 5.1 能源勘探行业软件对比信息表

对比项	Seismic Unix	Madagascar	GeoEast
算法覆盖	√	√	√
数据处理	√	√	√
分布式部署	√	√	√
任务并行调度	×	×	√
流程内并行策略	×	×	×
扩展编程方式	本地 IDE	本地 IDE	本地 IDE

并行编程指导	×	×	×
应用形式	命令行工具	命令行工具	Web 应用

为了解决这一问题，国家重点研发计划“面向 E 级计算的能源勘探高性能应用软件系统与示范”在其课题一(任务 3：“面向 E 级计算的能源勘探应用软件并行化策略”)中，明确提出了要研究一种适合能源勘探领域通用的并行化标记语言和代码生成工具，建立标记语言与并行混合编程模型之间的映射，实现能源勘探程序到 E 级计算平台上并行程序的自动生成。

5.1 基于 XML 的并行标记语言

在计算机文本处理过程中，标记语言是一种使用有限且通用的标记将文本内容和其他相关信息结合起来的技术手段。HTML 和 XML 是当前常见的描述性标记语言，二者均属于标准通用标记语言(Standard GeneraiizesMarkup Language，SGML)。其中，HTML 具有提前设定的语义和简单精炼的语法，这意味着它在保证通用性和易用性的前提下同时规定了如何在特定媒体上呈现结构化数据，因此 HTML 被广泛的应用在网页的创建和互联网信息的描述中。XML 借鉴了 HTML 成功的经验，将 HTML 的高效易用与 SGML 的丰富功能相结合，多被用于定义数据本身的结构、数据类型及相关数据操作。XML 因其显著的技术优势(将在 5.1.1 小节进行详细介绍)具有着更加广泛的应用和更加广阔的前景。本节内容中，并行标记语言 PML 被设计用来统一并行编程接口，对串行程序可以并行的部分及具体并行优化操作进行标识和描述，接下来的几个小节将详细介绍并行标记语言 PML 从构思到实现的具体细节。

5.1.1 可行性分析

(1)并行编程模型是一种类 XML 的标记语言

一般地，目前较为常用的程序并行化途径大致可以分为以下 4 种：(a)采用消息传递模型 MPI，通过在串行程序中调用 MPI 库函数(多为 MPI_XXX()的形式)，完成数据在线程间、节点间的交互通信，提高执行效率。(b)使用共享内存模型 OpenMP，通过在串行程序中插入编译制导语句(多为#omp parallel XXX 的形式)，以注释的形式告知计算机如何将代码编译为可并行执行的程序。(c)采用 CUDA 编程，同时利用 CPU 和图形处理器(GPU)，调用 CUDA 核函数，将原串行程序改造为 CUDA 程序，加快应用程序的运行速度。(d)混合编程模型，即同时采用上述三种途径中的两种或多种。本质上，上述途径(a)、(b)、(c)均可以看作是对现有编程语言(如：C、Fortran 等)，通过添加库函数、插入编译制导语句等手段进行的扩展。而向串行程序中添加的库函数调用及编译制导语句，完全可以看作是对串行程序中可并行执行部分的一种标识，是对如何进行并行优化的一种描述。这与可扩展

标记语言 XML 的用途类似，因此从本质上我们也可以认为 MPI、OpenMP 等并行编程模型是一种并行标记语言，只不过他们对标记的形式、使用方法的定义各有不同。

(2)XML 技术优势显著

在设计并行标记语言方面，XML 主要存在如下几个方面的技术优势：(a)它具备良好的可读性。人们可以在 XML 文档中对相关语义定义唯一且固定的标记。(b)它具备强大的可扩展性。XML 允许用户自定义标记集合以满足他们的具体需要，并无障碍地将这些标记集合投入到实际使用中。(c)它便于信息的检索。XML 通过在文档中插入特定的标记表示相应的含义，所以可以简单而高效地根据标记对整个文档进行搜索。这样一来，如果用户想要在超大规模的代码文件中搜索所有的并行标记便不再困难。(d)它支持不同文字不同符号间的信息交互。由于 XML 使用 Unicode 标准，可以有效地支持所有编程语言中所使用的各种符号以及世界各地的语言文字，甚至可以将并行标记语言的标签设计为中文。(e)具备与代码类似地层级结构。在编程语言中，一份代码包含着若干个函数，一个函数中包含了若干个结构(分支、循环等)，一个结构中又包含了若干语句。类似地，XML 文档在逻辑上是一种树状结构，文档中的每一个元素均以节点的形式按照层次关系进行了合理地组织。

(3)存在基于 XML 实现标记语言的相关研究

实际上，现在许多领域都在使用 XML 设计和实现其业内通用的标记语言。比较典型的有：化学标记语言 CML、矢量图形标记语言 VML 以及无线通信标记语言 WML 等。此外，表 5.2 还列出了一些近年来 XML 被用于设计其他专用标记语言的相关研究。

表 5.2 基于 XML 的标记语言相关研究信息表

序号	研究内容
1	基于 XML 的 Agent 通信语言[60]
2	基于 XML 的 STEP-NC 程序解释器[61]
3	基于 XML 的仿真想定标记语言 SSML[62]
4	基于 XML 的软 PLC 语言编辑系统[63]
5	基于 XML 的消息队列标记语言[64]
6	基于 XML 的虚拟仪器标记语言—VIML[65]
7	基于 XML 的机械图形标记语言的研究与开发[66]

综上所述，XML 显然是定义新的标记语言的一个不错的选择，基于 XML 技术来实现并行标记语言 PML 是完全可行的。

5.1.2 PML 的设计与实现

(1)技术路线

PML 旨在提出一种适合科学计算相关领域使用的并行编程标记语言和代码生成工具，建立 PML 标签与经典并行混合编程模型(MPI+OpenMP)之间统一的映射接口，实现串行程序到并行程序的源到源转换。

本文将 XML 作为实现并行标记语言 PML 的基础技术路线(如图 5.1 所示)，首先通过大量参考编写良好的 MPI+OpenMP 混合编程模型代码，总结归纳其常用函数、指令以及格式并进行抽象概括，以达到设计尽量少的 PML 标签进行尽量多的功能覆盖的目的。然后定义 XSLT 文件，描述所设计的 PML 标签与并行编程模型之间的对应关系及具体的转换规则，为从串行代码到并行代码的转换提供转换模板。最后开发转换程序，按照 XSLT 文件实现从加入了 PML 标签的串行代码自动转换为并行代码的功能。

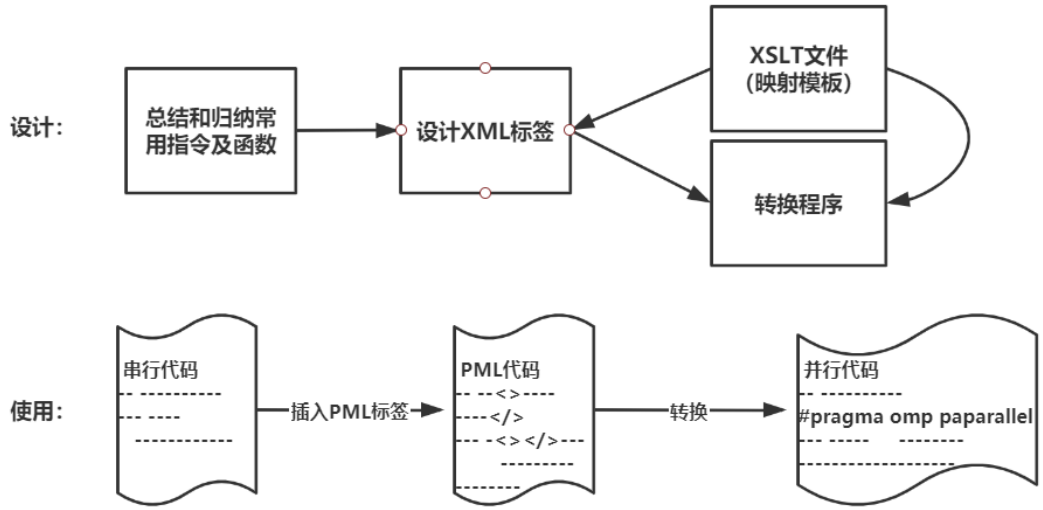


图 5.1 并行标记语言技术路线图

(2)标签详情

为了方便使用和理解，统一不同并行编程模型和框架的编程接口，本文将 PML 语言的标签全部设计为格式统一的中文标签。PML 标签共计 70 个，其中 PML 根节点标签 1 个，MPI 相关标签 36 个，包括 5 个初等例程、8 个点对点消息传递例程、6 个组通信例程以及 17 个常用的参数标签；OpenMP 相关标签 33 个，包括 13 个常用指令、8 个常用 API 库函数以及 12 个常用子句。在 PML 的所有标签中，参数标签统一设置，不存在各个例程中含义相同的参数标签不统一的情况，且<返回值>标签在不需要使用其返回值时均可以缺省，当需要返回值时，仅需将返回值接收变量写在该标签内即可。以 MPI 为例，其初始化与终止例程的简单 C 语言代码示例与对应的 PML 程序示例如图 5.2 所示，可以发现，PML 语言所写的程序源码文件为 XML 文档(后缀名为.xml)，且 PML 并不改变原有的 C 语言编程语法规范与编程逻辑，仅仅只是在本应调用并行编程模型的位置，将不

同的调用语句替换为了标准统一的 PML 标签。

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main( int argc, char **argv )
{
  int    err = 0;
   double t1=1, t2=0;
   double tick;
   int    i;
   MPI_Init( &argc, &argv );
   for(i=0;i<3;i++){
        err=t1+t2;
        t1=err+t1;
    }
    MPI_Finalize();
}
```

C代码

```
<PML>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main( int argc, char **argv )
{
  int    err = 0;
   double t1=1, t2=0;
   double tick;
   int    i;
   <MPI初始化>
        <argc>argc</argc>
        <argv>argv</argv>
   </MPI初始化>
   for(i=0;i<3;i++){
        err=t1+t2;
        t1=err+t1;
    }
    <MPI终止 />
}
</PML>
```

PML代码

图 5.2 C 语言与 PML 程序示例图

(3)转换机制

由上文所述技术路线可知，PML 标签向普通 C/C++语言代码转换的功能主要由 XSLT 文件负责，XSLT 文件本身也是一种 XML 格式的文档，可以看作是普通 XML 文档的样式表(类似 CSS 之于 HTML)，用于定义 XML 标签的转换逻辑，这些逻辑会应用于 XML 文档树状结构的节点集上，然后生成 HTML、文本或其他形式的输出文件。

在 PML 语言的 XSLT 文件中，将输出类型定义为文本类型，并且为每一个设计好的 PML 标签都定义了转换规则。每个转换规则一般都与 XPATH 关联，XPATH 代表着标签在 XML 文档的树状结构中的层次信息，表明了这个规则适用于哪一个或哪一类节点，用于定位标签在 XML 文档中的位置从而查找匹配正确的标签进行转换。当然，这种规则也被称做模板(Template)，在 XSLT 中使用标签<xsl:template>表示，并使用该标签的 match 属性来关联 XPATH 表达式。仍以图 5.2 所示 PML 程序为例，其中所使用到的 MPI 环境终止标签的 XSLT 转换规则如图 5.3 所示，此规则仅作用于<MPI 终止>标签，并且 XPATH 规定这个<MPI 终止>标签必须是根节点的直接子节点，才能使用这个规则。值得注意的是，XSLT 文档还提供了条件判断、选择分支等功能。在图 5.3 中，<xsl:choose>标签用于结合<xsl:when> 和 <xsl:otherwise> 来表达条件分支。在该规则中，如果<返回值>标签存在且不为空(假设为 ret)，则对应的将标签转换为“ret=MPI_Finalize();”，否则直接转换为“MPI_Finalize();”。

```
- <xsl:template match="MPI终止">
  - <xsl:choose>
    - <xsl:when test="返回值!=''">
        <xsl:value-of select="返回值"/>
        =MPI_Finalize();
      </xsl:when>
      <xsl:otherwise> MPI_Finalize(); </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
```

图 5.3 XSLT 示例图

图 5.2 中的 PML 程序向并行化的 C 语言程序转换的完整流程如图 5.4 所示。对于根节点标签<PML>下的文本内容，直接将其保留；而对于其他 PML 标签包裹的内容则通过 match 属性匹配 XSLT 模板中定义的转换规则进行转换。为了实现这一流程的自动化，本文开发了一个专门用于 PML 代码向 C 语言代码转换的转换程序，算法描述如图 5.5 所示。其中为了适应接下来的章节中所提及到的部分功能，算法中所使用的 PML 代码由浏览器端 http 请求传回，对 PML 代码中的部分内容进行了一些预处理工作以避免一些字符编码问题。

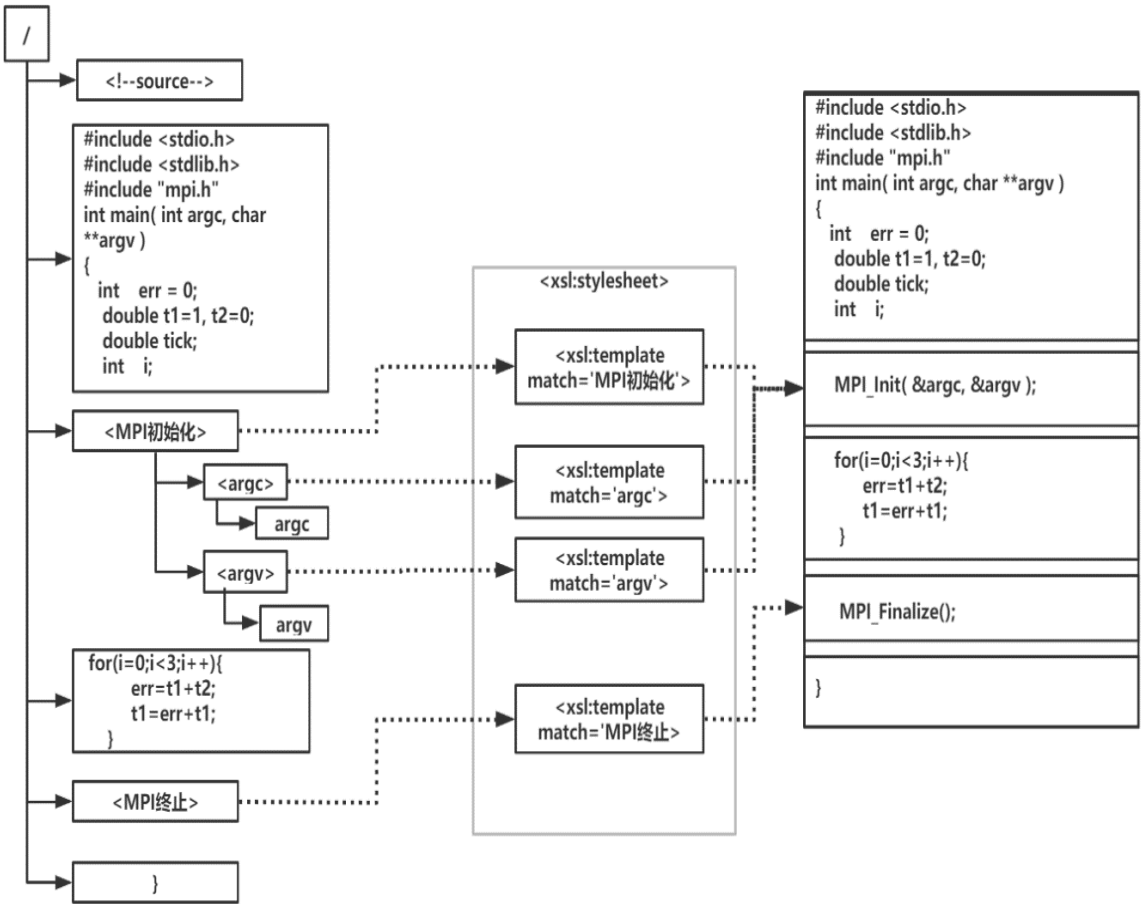


图 5.4 PML 向 C/C++语言转换示例图

转换算法

```
1  获取PML代码
2  对C/C++语言所定义的操作符进行转义
3  为PML代码加上根节点<PML> </PML>
4  将PML代码转换为输入流Ins
5  读取XSLT文档style
6  将Ins转换为树状结构tree_ins
7  for node in tree_ins:
8          获取XPATH
9          根据XPATH匹配style
10         根据style进行转换
11 返回转换结果
```

图 5.5 PML 转换算法描述图

5.2 并行编程辅助平台

考虑到 PML 的设计是为了更好的进行并行程序编程开发，目前主流的 XML 文档编辑器虽然功能完善且使用方便，但是很难为 C/C++语言提供诸如着色、格式化等特性功能，同时为了使本文的所有研究成果更好的统一与融合，本着部署方便、使用简单和易于扩展的原则，本文结合 Web 开发相关技术与部分开源项目搭建了一个基于 MVC(Model-View-Controller)三层架构模式的并行编程辅助平台。为用户提供串行程序开发功能、串行程序的并行性自动识别功能、PML 并行标记语言开发功能、并行程序自动转换功能和远程集群系统调试功能。

平台的详细架构如图 5.6 所示，模型层负责数据的建模与访问，将 PML 标签存储在 MySQL 数据库中；视图层负责和用户交互，为用户提供串行程序的并行性识别、PML 使用指导、程序编辑、程序调试等具体的功能体验；控制层则负责视图层各种功能体验的逻辑实现，处理用户发来的请求并作出正确响应。在使用过程中，用户通过浏览器访问本平台，在编辑器中进行 C/C++串行程序的开发，通过使用串行程序的并行性识别功能自动识别出串行程序中可以并行的部分，然后使用 PML 标签对可以并行的部分进行并行化开发，将开发好的 PML 程序提交到服务器自动转换为并行程序，最后通过 SSH 终端在浏览器中实现程序在远程集群环境上的调试。

为了更好地说明本文所有研究成果的融合过程，本章接下来的内容将首先按照 MVC 三层架构的顺序，依次介绍并行编程辅助平台各个基础功能的实现技术，然后再单独介绍如何将串行程序的并行性识别功能集成到平台上来。

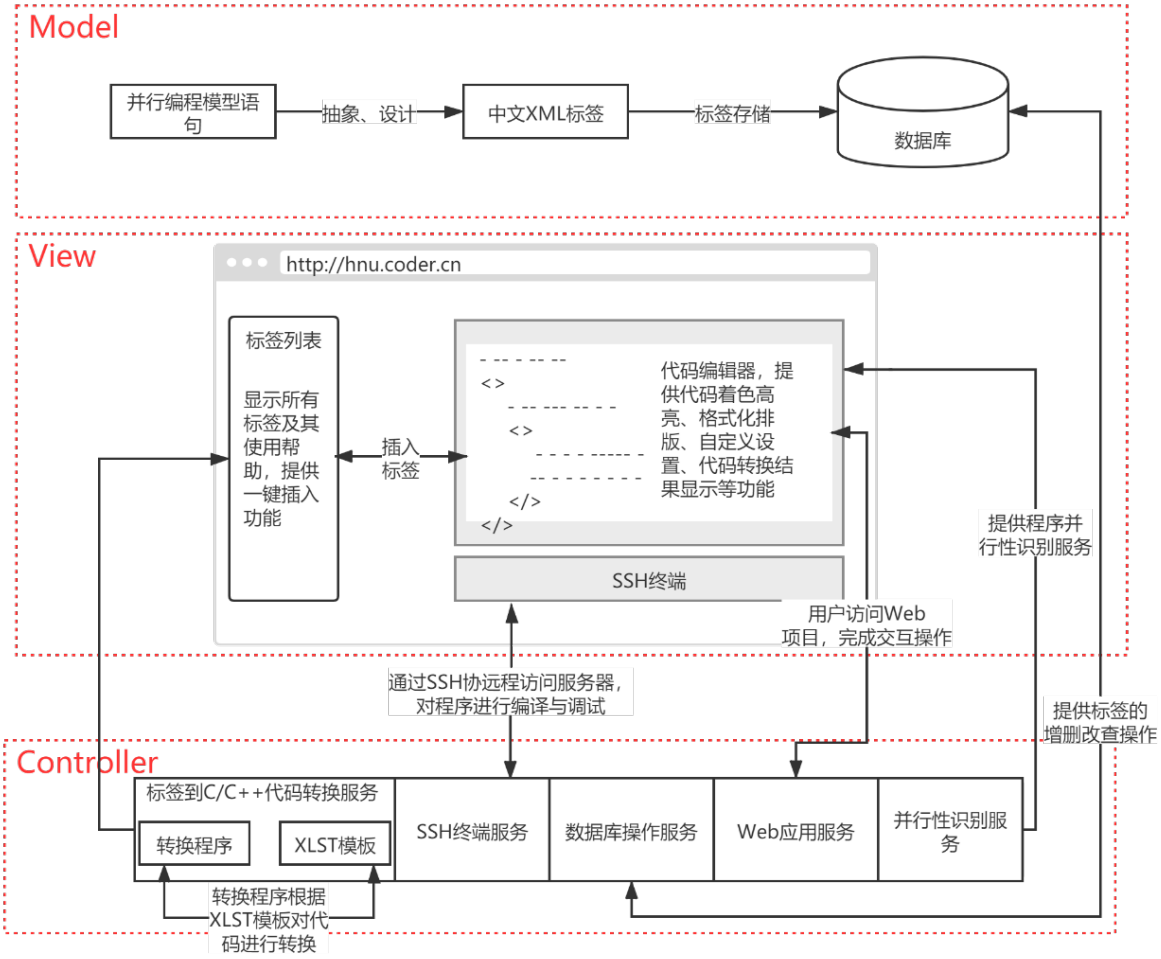


图 5.6 并行编程辅助平台架构图

5.2.1 基础功能的实现

(1)模型层

模型层的主要功能是为数据相关的操作提供支持服务，保证数据的一致性与稳定性。在本平台的生命周期中，数据主要指的是 PML 语言的编程标签。由于在本研究中 PML 标签被定义为统一的标准编程接口，因此数据相关的操作主要为读取操作。

本平台在模型层采用了较为流行的关系型数据库 MySQL。在数据库存储结构的设计中，考虑到需要在视图层为用户提供 PML 使用指导及 PML 标签的插入等交互功能，对编程模型的分类、标签的功能分类、标签的基础信息以及标签的使用引导等方面均设计了相应的表或字段。详细的设计信息如表 5.3 所示。同时，为了保证平台运行过程中数据的一致性与稳定性，本平台严格按照数据库表的结构封装了相应的 Java 实体类，将表中的字段设计为私有的类属性，并通过为每一个属性提供 get 和 set 方法来控制对私有属性的访问和修改行为，从根本上避免数据的一致性问题和安全性问题。

表 5.3 数据库表结构信息表

序号	表名	字段	描述
1	pml_tag	Id	标签编号
		Model_type	并行编程模型类别，外键
		Tag	标签内容
		Code	C 语言代码
		Desc	标签功能描述
		Param	参数详解
		Tag_type	标签功能类型，外键
2	tag_type	Type_name	PML 标签类别名称
		Type_id	类别编号，用于外键
3	model_type	Type_name	模型名称
		Type_id	类别编号，用于外键

(2)视图层

视图层的主要作用，是为用户提供全面的功能服务和良好的交互体验，考虑到用户访问本平台的本地环境和浏览器不尽相同，为了平台更好的兼容性和可扩展性，视图层主要基于开源软件技术进行构建。在网页布局上采用 BootStrap 框架，使用其自带的全局 CSS 设置和网格系统，再结合 HTML5+CSS3+JavaScript 技术实现页面美化。在 PML 编辑功能部分，参考了当前较为流行的在线编辑器(如 Cloud9 等)的技术路线，采用基于 JavaScript 的富文本编辑器 Ace Editor 实现了 PML 代码编辑，通过修改或重写其 Js 脚本实现了 PML+C/C++混合编程代码的着色及高亮显示、代码格式化、用户个性化设置等基础功能。

视图层的实现效果如图 5.7 所示，其主要由两部分构成。红色框内的部分为标签列表，该部分将数据库中的 PML 标签按照并行编程模型的类型进行分类，然后再按照标签的功能类型进行罗列。对于每一个 PML 标签，均为其提供详情和插入两个选项，分别通过列表上的两个按钮实现。用户可以点击详情按钮，通过如图 5.8 所示页面查看该 PML 标签的详细信息，其中包括该标签的 PML 代码、C 语言代码、功能简介以及参数含义；用户还可以点击插入按钮，直接将该标签对应的 PML 代码插入至编辑器中光标所在位置。编辑器指的是图中由蓝色框圈出的部分，该部分为用户提供了普通的 C/C++代码编辑、PML 语言编辑以及对应的代码高亮着色、格式化和个性化设置等功能。此外，编辑器部分在后续的介绍中，还将通过绿色框内的选项按钮被赋予更多的功能角色。

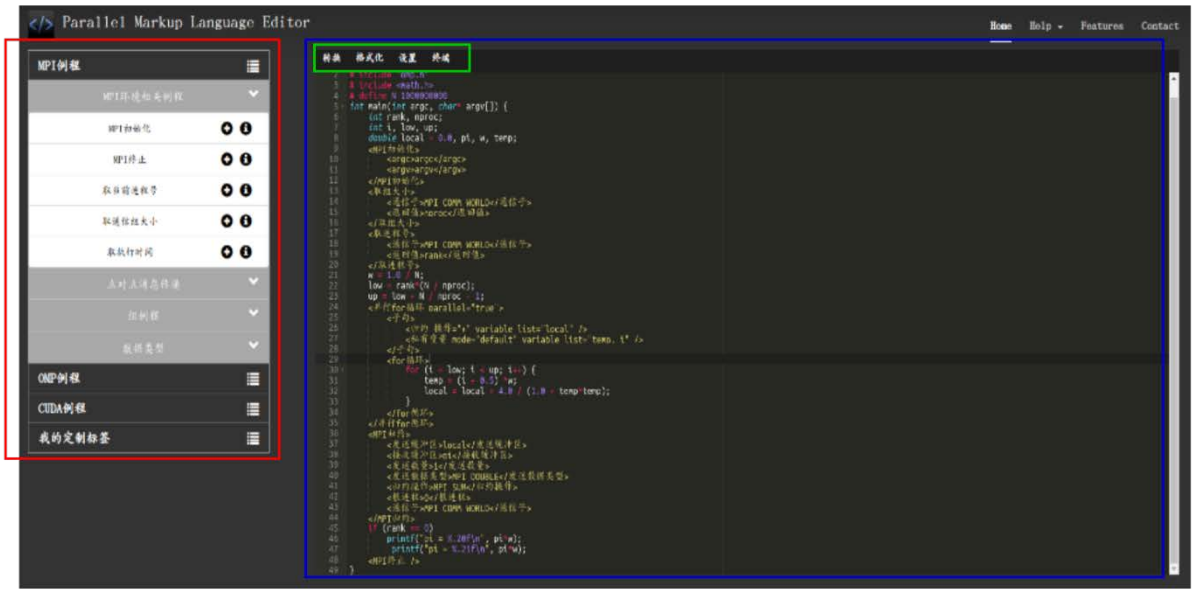


图 5.7 平台视图层效果图

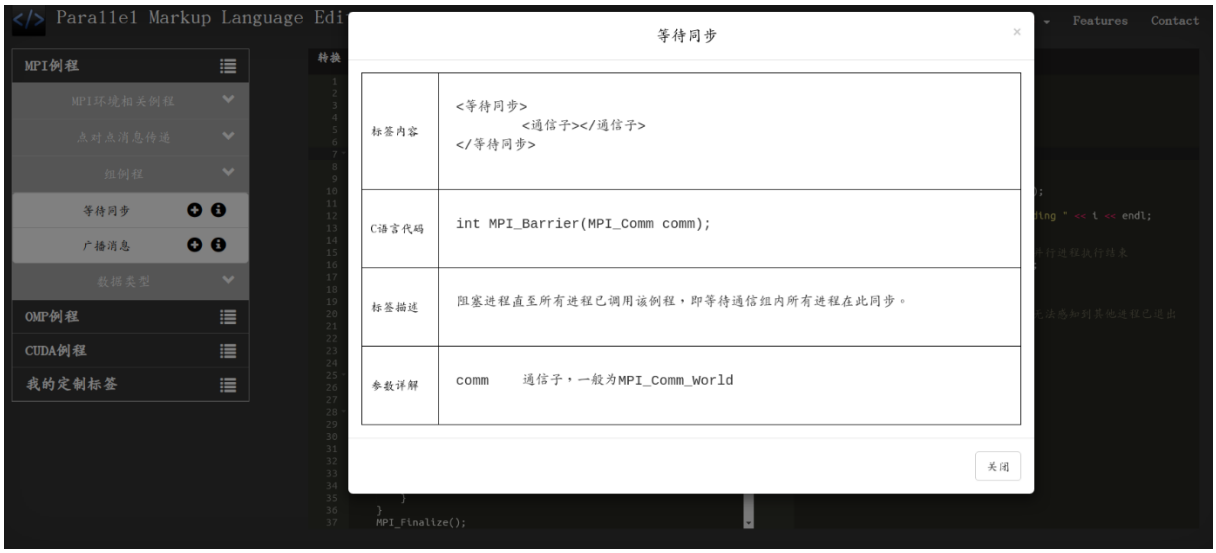


图 5.8 标签详情页面效果图

(3)控制层

控制层的主要功能可以分为 web 服务器功能、代码转换功能、SSH 跳转机功能、数据库操作功能和串行程序的并行性识别功能，是整个平台的逻辑核心。它可以在保证平台被用户正常访问的前提下，稳定持续地帮助用户识别串行代码中可以并行的部分，并为用户提供 PML 代码向 C/C++代码的转换服务，方便用户通过 SSH 协议连接远程集群系统进行代码调试。其中，各项功能的相关实现技术如表 5.6 所示。需要注意的是，GateOne 是一个开源的基于 HTML 的 Web Terminal SSH 客户端，除了使用正常的 HTML 标签将其嵌入在我们的 Web 应用中之外，GateOne 终端的使用还需要在本地服务器的应用脚本和远程服务器的配置文件上进行双重个性化配置才可以正常使用，相关配置文件如图 5.9 所示。

表 5.4 控制层相关实现技术信息表

功能	技术
web 服务器功能	Apache Tomcat7.0
SSH 跳转机功能	GateOne
数据库操作功能	JDBC
代码转换功能	XSLT+转换程序
串行代码的并行性识别功能	DGCNN 识别模型

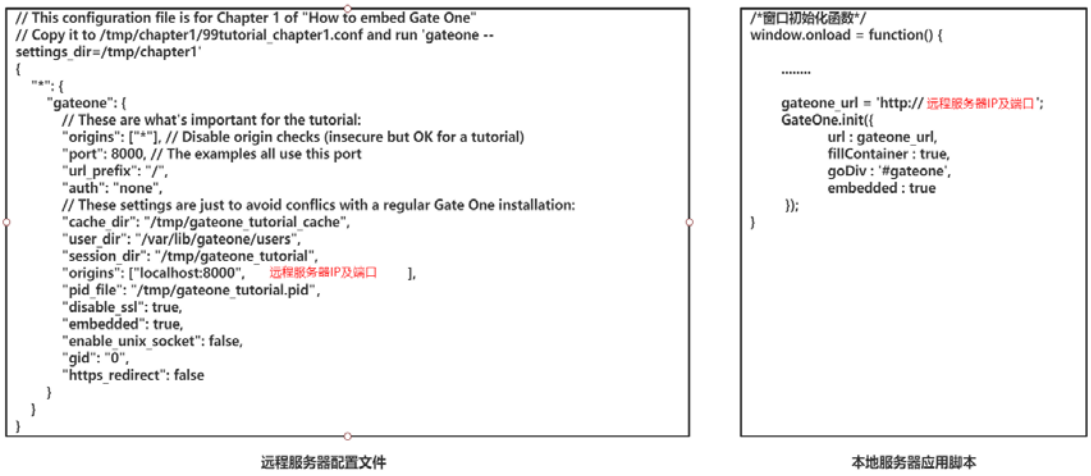


图 5.9 GateOne 配置示例图

以计算 PI 的程序为例，用户通过浏览器访问本平台，在编辑器中开发串行版本的 C 语言程序，通过并行性识别功能对代码中可以并行的部分进行识别，然后通过 PML 标签对程序进行并行化开发，点击转换选项得到如图 5.10 所示转换结果。为了验证转换后的并行程序是否正确，用户可以点击终端选项，在网页中开启 SSH 终端，将该代码上传至远程服务器进行调试，调试结果如图 5.11 所示。

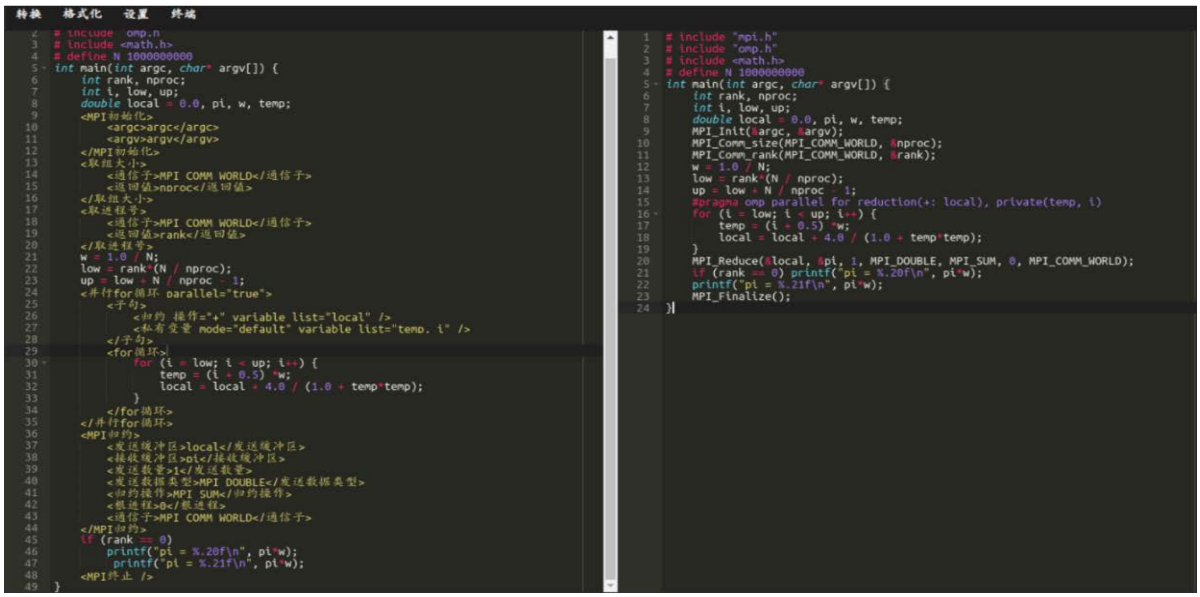


图 5.10 代码转换效果图

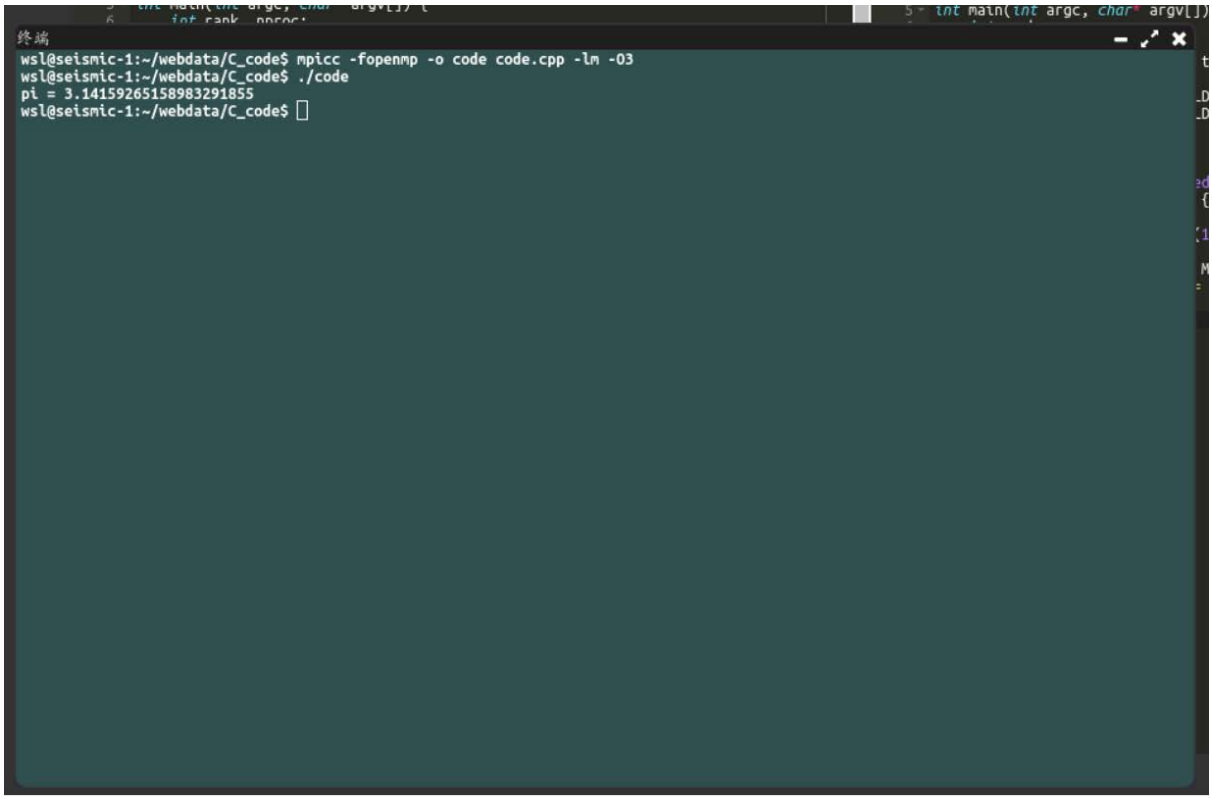


图 5.11 SSH 终端远程调试效果图

5.2.2 串行代码的并行性识别功能的集成

为了使本文的研究成果更好的融合与统一，使并行编程辅助平台的使用流程更加完善，本小节对本文第 4 章中串行程序的并行性识别功能进行了集成，整体流程如图 5.12 所示。接下来将分为数据提取、数据预处理和并行性识别三个阶段进行详细介绍。

在数据提取阶段，对于用户提交的需要进行并行性识别的串行代码，仍沿用 3.2 节所介绍的 GFCPD 数据构造方式，首先按顺序提取程序中的所有循环片段并进行编号，在源码文件中以注释的形式使用循环片段的编号替换其本身；然后对于每一个循环片段均进行代码重构，同时使用该循环片段的编号对重构后的代码文件进行命名，重构结果如图 5.13 所示；最后对重构后的代码文件逐一构造 XFG。需要注意的是，图 5.13 可以看出，本过程中虽然在提取循环片段时需要提取所有函数的相关信息，但是并没有将函数的完整定义使用其声明在源码文件中进行替换，这是为了便于在最后阶段将并行性识别的结果反馈在视图层中的同时，不改变代码的本身结构和内容。

在数据预处理阶段，对于上述阶段的中间结果 XFG_list 采用如 4.1 节所述的预处理方式，按顺序对 XFG 进行嵌入表示生成 data_list 以作为下一阶段的输入。必须说明的是，此处的顺序代表了循环片段在源码文件中的相对顺序，为了使并行性识别的结果正确的反馈给视图层，该顺序不可进行改变。

在并行性识别阶段，平台通过后台程序，调用使用 GFCPD 数据集训练好的 DGCNN 模型，对数据预处理阶段生成的所有数据进行预测。预测结果是一个与循环提取过程中循环片段的编号顺序相同的结果列表 result_list。仍以图 5.13 所示的源码文件为例，假设其预测结果如图 5.14(a)所示，此结果说明源码文件中存在着两个循环片段，且第一个循环片段可以并行化，而第二个循环片段不可以并行化。此时，平台后台程序会将第一个循环片段按照如图 5.14(b)所示形式进行标注，然后在源码文件中使用该形式替换循环片段所对应的注释内容“//loop1”；而对于第二个循环片段则不做任何标注，直接还原至源码文件中。最后后台程序将如图 5.14(c)所示结果反馈给视图层展示在编辑器中，用户可根据识别结果做进一步并行化改造。

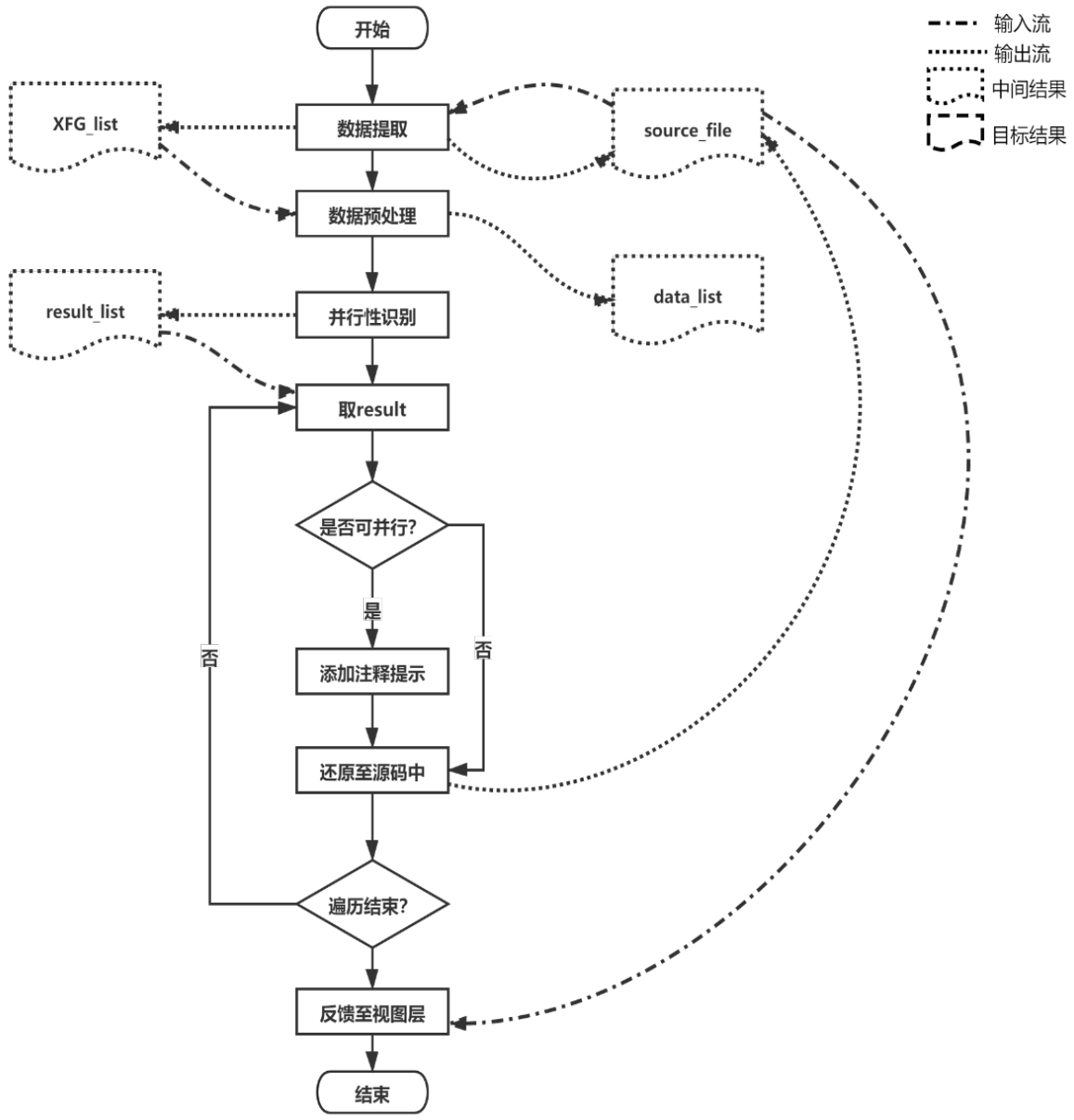


图 5.12 串行代码的并行性识别功能集成流程图

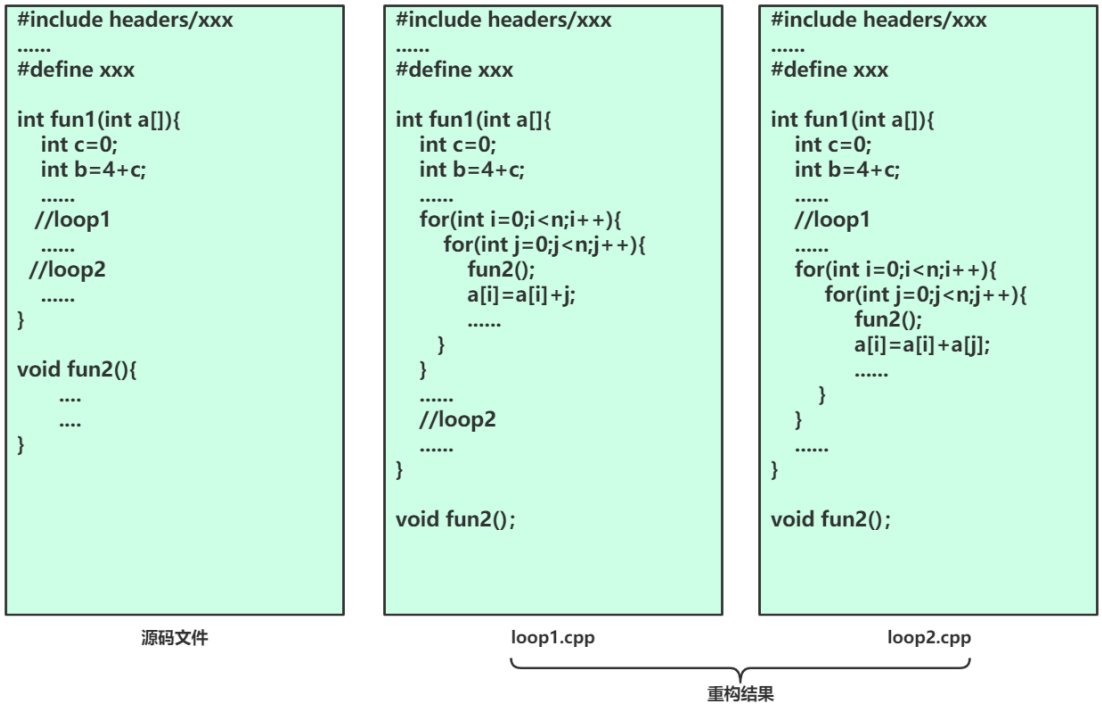


图 5.13 数据提取阶段提取结果示例图

(a) 预测结果

```
# ===可以并行的循环===

for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
        fun2();
        a[i]=a[i]+j;
        .....
    }
}

# ===结束可以并行的循环===
```

(b) 标注形式

```
#include headers/xxx
.....
#define xxx

int fun1(int a){
   int c=0;
   int b=a+c;
   .....
  # ===可以并行的循环===

   for(int i=0;i<n;i++){
      for(int j=0;j<n;j++){
         fun2();
         a[i]=a[i]+j;
         .....
      }
   }

   # ===结束可以并行的循环===
   .....
   for(int i=0;i<n;i++){
      for(int j=0;j<n;j++){
         fun2();
         a[i]=a[i]+a[j];
         .....
      }
   }
   .....
}

void fun2(){
     ....
     ....
}
```

(c) 最终反馈结果

图 5.14 串行程序并行性识别结果示例图

5.3 本章小结

本章中提出并实现了一种基于 XML 的并行标记语言 PML，消除了各个编程模型和框架之间的标准壁垒，为用户提供了统一的标准编程接口，大大降低了并行程序开发的技术门槛，同时基于 Web 开发相关技术，提供了轻量级的在线编程平台，在满足正常代码编辑需求的前提下，集成了基于深度学习的串行代码并行性识别功能，为用户的并行编程提供技术指导。

结论

随着社会和科技的发展，计算机硬件和架构设计理念的转变使得多核系统越来越受欢迎，多核技术为计算机的性能带来了巨大的潜在提升。为了充分开发和利用这一潜力，高性能计算领域进入了新的发展时期，并行编程技术的相关研究也活跃了起来，并行化成为了新的发展趋势。并行编程在诸如医疗成像、气候模拟、药物发现、地质能源勘探等众多领域的实际工程应用中占据着越来越重要的位置。然而，并行程序开发并不是一项简单的工作，虽然目前并行编程方法和程序并行性分析工具的发展势头良好，但对于上述领域的从业人员来讲熟练且高效地进行并行程序开发仍存在着较大的难度。主要体现在两个方面：一方面，由于需要对程序的数学表示进行限制或收集程序的运行时特征，目前的程序并行性分析与优化工具存在着较大的局限性，这些局限性使得其对于上述领域中普遍存在的程序复杂、数据量大且执行时间长的应用程序并不友好；另一方面，目前广泛使用的并行编程模型(如 MPI、OpenMP 等)所采用的编程方法和手段并不统一，但实际工程中往往需要结合其中的两种或多种，多个不同标准的学习和交叉使用为此增加了一定难度。

针对上述两个方面，本文研究了一种基于深度学习的串行程序并行化方法，为开发人员从程序的并行性识别，到统一并行编程模型接口，再到并行程序的开发与调试提供了整套的解决方案。本文的主要工作如下：

(1)以程序的上下文流图(XFG)为数据格式，构造了一个可以用于串行程序的并行性识别的深度学习图数据集 GFCPD，设计并开发了包括数据提取、代码重构、编译检验、数据标注以及 XFG 生成等流程在内的用于生成 GFCPD 数据集数据的自动化脚本。

(2)构建了基于图卷积神经网络的串行程序并行性识别模型。通过该模型针对 GFCPD 数据集进行学习，使用深度学习的方法实现了串行程序的并行性识别。实验表明，本文所建立的模型在串行程序的并行性识别任务中，存在着良好的可行性与有效性。相较于目前传统的静态分析方法、结合动态特征的机器学习方法以及在算法分类问题上表现良好的深度学习模型均表现出强大的竞争力。

(3)基于 XML 技术设计并实现了并行标记语言 PML。为目前较为流行的两种并行编程模型(MPI 与 OpenMP)设计了格式统一、使用方便的中文标签，并实现了从 XML 标签到 C/C++代码的转换程序，提供了更加简易快捷的并行编程途径。

(4)搭建了并行编程辅助平台，将串行程序的并行性预测、并行标记语言 PML 两大功能集成在该平台上，同时通过 GateOne SSH 终端技术将远程调试变为了可能。用户可以在辅助平台中按照正常的 C/C++编程语法规范在编辑器内编写串行

程序，平台通过并行性识别模型对用户的程序进行并行性识别并在可以并行的循环片段前后加以注释提醒；然后用户可以通过列表查看 PML 标签说明，通过手动编写或点击按钮的方式直接向编辑器内的指定位置插入 PML 标签；最后用户将 PML 程序提交至后台进行转换，同时使用 SSH 终端连接其远程服务器，将转换后的 C/C++代码上传至远程服务器进行调试。

虽然本文在完成研究的整个过程中，取得了一定的阶段性成果，完成了最初设定的目标，但是本研究仍然存在着一些问题以及一些新的研究方向，需要进一步的解决和探索，主要有以下几方面：

(1)本文在构造 GFCPD 数据集的过程中，由于搜集的原始数据有限，且部分源码文件因其不可编译或无法确定并行性而被舍弃，导致了 GFCPD 数据集规模的小幅度缩减，如果对 GFCPD 数据集进行进一步的扩充与完善可能会使本研究的研究成果更加出色。

(2)程序可以使用数据流图、控制流图、函数调用图等多种图表示形式分别表示不同方面的语义信息。因本文在研究中所使用的深度图卷积神经网络模型仅接收一个图对象作为输入进行训练和预测，所以本文选取了在设计上结合了数据流图和控制流图的图表示形式 XFG，但 XFG 并不能被认为全面完整地表示了程序的语义信息。如何使用接收多图作为输入的图神经网络模型，更完整的利用程序的语义信息进行并行性分析，是下一步研究的重要内容。

(3)PML 编程语言目前仅支持 MPI 和 OpenMP 两种并行编程模型，下一步的研究可以向其中添加 CUDA 等其他编程模型的支持，并在辅助平台上为用户提供定制功能，允许用户将自己常用的编程结构或标签组合定制为新的自定义标签。

参考文献

[1] Kirk D B, Wen-Mei W H. Programming massively parallel processors: a hands-on approach[M]. Morgan kaufmann, 2016.

[2] Hwu W, Keutzer K, Mattson T G. The concurrency challenge[J]. IEEE Design & Test of Computers, 2008, 25(4): 312-320.

[3] H. Sutter. Welcome to the Jungle.http://herbsutter.com/welcome-to-the-jungle/, 2012. [Last access 4th April 2020].

[4] Sanchez L M, Fernandez J, Sotomayor R, et al. A comparative study and evaluation of parallel programming models for shared-memory parallel architectures[J]. New Generation Computing, 2013, 31(3): 139-161.

[5] McCool M, Reinders J, Robison A. Structured parallel programming: patterns for efficient computation[M]. Elsevier, 2012.

[6] Sun X, Liu X, Hu J, et al. Empirical studies on the nlp techniques for source code data preprocessing[C]//Proceedings of the 2014 3rd International Workshop on Evidential Assessment of Software Technologies. 2014:32-39.

[7] 乐晓波, 汪琳, 黄敏. 用 Petri 网分析循环程序的并行性[D]., 2002.

[8] 蒋作, 高毅. 关于串行程序并行化[J]. 云南民族大学学报 (自然科学版), 2007, 16(3): 274-276.

[9] 梁博, 安虹 [1, 王莉, 等. 针对子程序结构的线程级推测并行性分析[D]. , 2009.

[10] 闫昭. 程序并行识别方法及应用研究[D]. 长春: 吉林大学, 2009.

[11] 郭慎, 李培峰, 朱巧明. 一种基于特征的程序可并行点发现方法[J]. 计算机应用与软件, 2011, 28(4): 24-26.

[12] 王磊, 曲卫平, 李敬兆. 基于人工智能搜索和数据依赖分析的程序并行化[J] 现代电子技术, 2013, 36(6): 1-3.

[13] 王家龙, 刘艳红, 沈立. 线程级猜测并行系统代码自动生成工具的设计与实现[J]. 计算机科学, 2017, 44(11): 114-119.

[14] 王时雨, 张盛兵, 安建峰, 等. 基于 LLVM 架构的图像处理程序的并行分类[J]. 微电子学与计算机, 2018, 35(1): 66-71.

[15] 谭丁武,张坤芳,刘燕,郑一基,鲁鸣鸣.基于门控图注意力神经网络的程序分类[J].计算机工程与应用,2020,56(07):176-183.

[16] Bondhugula U, Hartono A, Ramanujam J, et al. A practical automatic polyhedral parallelizer and locality optimizer[C]//Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementati

on. 2008: 101-113.

[17] Baghdadi R, Ray J, Romdhane M B, et al. Tiramisu: A polyhedral compiler for expressing fast and portable code[C]//2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2019: 193-205.

[18] Palkowski M, Bielecki W. Tuning iteration space slicing based tiled multi-core code implementing Nussinov's RNA folding[J]. BMC bioinformatics, 2018, 19(1): 12.

[19] Palkowski M, Bielecki W. Parallel Tiled Codes Implementing the Smith–Waterman Alignment Algorithm for Two and Three Sequences[J]. Journal of Computational Biology, 2018, 25(10): 1106-1119.

[20] Palkowski M, Bielecki W. Tiling Nussinov's RNA folding loop nest with a space-time approach[J]. BMC bioinformatics, 2019, 20(1): 208.

[21] Diaz J, Munoz-Caro C, Nino A. A survey of parallel programming models and tools in the multi and many-core era[J]. IEEE Transactions on parallel and distributed systems, 2012, 23(8): 1369-1386.

[22] Fried D, Li Z, Jannesari A, et al. Predicting parallelization of sequential programs using supervised learning[C]//2013 12th International Conference on Machine Learning and Applications. IEEE, 2013, 2: 72-77.

[23] Li Z. Discovery of potential parallelism in sequential programs[D]. Technische Universität Darmstadt, 2016.

[24] del Rio Astorga D, Dolz M F, Sanchez L M, et al. Discovering pipeline parallel patterns in sequential legacy C++ codes[C]//Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores. 2016: 11-19.

[25] del Rio Astorga D, Dolz M F, Sánchez L M, et al. Finding parallel patterns through static analysis in C++ applications[J]. The International Journal of High Performance Computing Applications, 2018, 32(6): 779-788.

[26] Zhao H, Zheng F, Wu J, et al. Automatic Parallelization for Binary on Multi-core Platforms[C]//Proceedings of the 2nd International Conference on Computer Science and Application Engineering. 2018: 1-6.

[27] Mou L, Li G, Zhang L, et al. Convolutional neural networks over tree structures for programming language processing[C]//Thirtieth AAAI Conference on Artificial Intelligence. 2016.

[28] Hellendoorn V J, Devanbu P. Are deep neural networks the best choice for modeling source code?[C]//Proceedings of the 2017 11th Joint Meeting o

n Foundations of Software Engineering. 2017: 763-773.

[29] Choi M, Jeong S, Oh H, et al. End-to-end prediction of buffer overruns fr om raw source code via neural memory networks[J]. arXiv preprint arXiv: 1703.02458, 2017.

[30] Sestili C D, Snavely W S, VanHoudnos N M. Towards security defect pre diction with AI[J]. arXiv preprint arXiv:1808.09897, 2018.

[31] Alon U, Zilberstein M, Levy O, et al. A general path-based representation for predicting program properties[J]. ACM SIGPLAN Notices, 2018, 53(4): 404-419.

[32] Alon U, Zilberstein M, Levy O, et al. code2vec: Learning distributed repre sentations of code[J]. Proceedings of the ACM on Programming Language s, 2019, 3(POPL): 1-29.

[33] Allamanis M, Brockschmidt M, Khademi M. Learning to represent program s with graphs[J]. arXiv preprint arXiv:1711.00740, 2017.

[34] Ben-Nun T, Jakobovits A S, Hoefler T. Neural code comprehension: A lear nable representation of code semantics[C]//Advances in Neural Information Processing Systems. 2018: 3585-3597.

[35] Chen Z, Monperrus M. A literature study of embeddings on source code[J]. arXiv preprint arXiv:1904.03061, 2019.

[36] 王克朝, 成坚, 王甜甜, 等. 面向程序分析的插桩技术研究[J]. 计算机应用研究, 2015, 32(2): 479-484.

[37] Harrold M J, Larsen L, Lloyd J, et al. Aristotle: A system for developmen t of program analysis based tools[C]//Proceedings of the 33rd annual on S outheast regional conference. 1995: 110-119.

[38] Kirchner F, Kosmatov N, Prevosto V, et al. Frama-C: A software analysis perspective[J]. Formal Aspects of Computing, 2015, 27(3): 573-609.

[39] Garcia S, Jeon D, Louie C M, et al. Kremlin: rethinking and rebooting gp rof for the multicore age[J]. ACM SIGPLAN Notices, 2011, 46(6): 458-46 9.

[40] Zhang X, Navabi A, Jagannathan S. Alchemist: A transparent dependence distance profiling infrastructure[C]//2009 International Symposium on Code Generation and Optimization. IEEE, 2009: 47-58.

[41] Ketterlin A, Clauss P. Profiling data-dependence to assist parallelization: Fr amework, scope, and optimization[C]//2012 45th Annual IEEE/ACM Intern ational Symposium on Microarchitecture. IEEE, 2012: 437-448.

[42] Subotic V, Ayguadé E, Labarta J, et al. Automatic exploration of potential

parallelism in sequential applications[C]//International Supercomputing Conference. Springer, Cham, 2014: 156-171.

[43] Blair-Chappell S, Stokes A. Parallel programming with intel parallel studio XE[M]. John Wiley & Sons, 2012.

[44] Silexica GmbH. https://www.silexica.com. [Last access 4th April 2020].

[45] Bischof C, Huss-Lederman S, Sun X, et al. The PRISM project: Infrastructure and algorithms for parallel eigensolvers[C]//Proceedings of Scalable Parallel Libraries Conference. IEEE, 1993: 123-131.

[46] Grosser T, Groesslinger A, Lengauer C. Polly—performing polyhedral optimizations on a low-level intermediate representation[J]. Parallel Processing Letters, 2012, 22(04): 1250010.

[47] Rauchwerger L, Padua D A. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization[J]. IEEE Transactions on Parallel and Distributed Systems, 1999, 10(2): 160-180.

[48] Jimborean A, Clauss P, Dollinger J F, et al. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons[J]. International Journal of Parallel Programming, 2014, 42(4): 529-545.

[49] Gómez-Sousa H, Arenaz M, Rubiños-López Ó, et al. Novel source-to-source compiler approach for the automatic parallelization of codes based on the method of moments[C]//2015 9th European Conference on Antennas and Propagation (EuCAP). IEEE, 2015: 1-6.

[50] Amini M, Creusillet B, Even S, et al. Par4all: From convex array regions to heterogeneous computing[C]. 2012.

[51] Bondhugula U, Baskaran M, Krishnamoorthy S, et al. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model[C]//International Conference on Compiler Construction. Springer, Berlin, Heidelberg, 2008: 132-146.

[52] Lee S I, Johnson T A, Eigenmann R. Cetus–an extensible compiler infrastructure for source-to-source transformation[C]//International Workshop on Languages and Compilers for Parallel Computing. Springer, Berlin, Heidelberg, 2003: 539-553.

[53] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE, 2004: 75-86.

[54] 赵捷, 李颖颖, 赵荣彩. 基于多面体模型的编译“黑魔法”[J]. 软件学报, 2018 (8): 15.

[55] Wu Z, Pan S, Chen F, et al. A comprehensive survey on graph neural networks[J]. IEEE Transactions on Neural Networks and Learning Systems, 2020.

[56] Gori M, Monfardini G, Scarselli F. A new model for learning in graph domains[C]//Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005. IEEE, 2005, 2: 729-734.

[57] Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks[J]. arXiv preprint arXiv:1609.02907, 2016.

[58] Zhang M, Cui Z, Neumann M, et al. An end-to-end deep learning architecture for graph classification[C]//Thirty-Second AAAI Conference on Artificial Intelligence. 2018.

[59] Amid E, Warmuth M K K, Anil R, et al. Robust Bi-Tempered Logistic Loss Based on Bregman Divergences[C]//Advances in Neural Information Processing Systems. 2019: 14987-14996.

[60] 凌兴宏. 基于 XML 的 Agent 通信语言[J]. 计算机应用研究, 2003, 20(7):152-154.

[61] 于东, 李筱飚, 黄艳, et al. 基于 XML 的 STEP-NC 程序解释器的设计与实现[J]. 小型微型计算机系统, 2009, 30(10):1956-1959.

[62] 陈欣, 胡晓惠, 付勇, et al. 基于 XML 的仿真想定标记语言 SSML[J]. 系统仿真学报, 2004, 16(9):1928-1930.

[63] 吴倩, 陶亦亦, 陆春元. 基于 XML 的软 PLC 语言编辑系统的设计与实现[J]. 机械制造与自动化, 2007, 36(2):100-102.

[64] 陈林, 黄晔. 基于 XML 的消息队列标记语言[J]. 计算机工程, 2007, 33(19):85-87.

[65] 钟莹, 陈祥献. 基于 XML 的虚拟仪器标记语言——VIML[J]. 仪器仪表学报, 2002, 3.

[66] 姚屏. 基于 XML 的机械图形标记语言的研究与开发[D]. 长沙: 中南大学, 2005.

附录 A 攻读学位期间发表的学术论文与获得的成果

[1] 湖南大学：并行编程助手软件 V1.0.软件著作权.证书号：软著登字第 4915889 号.登记号：2020SR0037193.

附录 B 攻读学位期间参加的科研项目

[1] 面向 E 级计算的能源勘探高性能应用软件系统与示范，项目号：2017YFB0202900.时间：2017.07-2021.06.

致　谢

时光荏苒，随着毕业论文接近尾声，我三年的硕士生涯也将告一段落。在此，衷心地对我的母校，对给予我鼓励和帮助的老师、同学以及亲友表达诚挚的谢意。感谢你们陪伴着我一起度过的美好时光；感谢你们对我学习和生活的默默关怀；感谢你们一直激励着我奋力前行。

感谢我的导师 XXX 教授。彭老师品德高尚、治学严谨。在我三年的学习期间，为我提供了力所能及的帮助与支持。在本文研究课题的进行过程中，老师耐心地听取了我每次的进展报告，及时地提出了技术修改方案，传授了做学问的技巧，保证了本文的撰写工作顺利进行。感谢彭老师一直以来地信任和培养，学生当以您为榜样在学术和专业上不断奋斗。

感谢朝夕相处的老师和同学们，感谢你们这些年来的关心，帮助和信任。在实验室的日子，是我学习生涯中最宝贵的财富。感谢读研期间你们同我一起坚持不懈的奋斗，无论炎夏还是寒冬。感谢 XX 老师、XXX 博士、XXX 同学和 XX 同学，在团队合作中营造了良好的学术氛围，让我感受到了钻研学术的满足和快乐。感谢我的室友 XXX 同学和 XXX 同学，感谢你们的包容和理解，让我在三年的校园生活中收获了难忘的室友情谊。

感谢我的父母，二老多年来对我的支持与付出让我更有力量去面对一切困难。同时，我还要感谢女朋友 XXX 同学，纵使身在异地，也总是给与我及时的陪伴与鼓励，让我有源源不断的动力勇往直前。

感谢我的母校湖南大学，给予了我丰富多彩的校园生活和宝贵的学习机会。有欢乐也有磨难，这段校园生活将成为我人生中一段难忘的过往。感谢母校以及信息科学与工程学院在我学习生涯中提供的宝贵资源，我将一直谨记校训，不忘初心，在今后的道路上砥砺前行。

最后，衷心感谢各位老师和评审专家，感谢你们百忙之中抽出时间审阅本文并提出了宝贵意见，谢谢你们的辛勤工作和无私付出。