分类号 TP399 学号 14060095

UDC 004 密级 公 开

工学硕士学位论文

大规模应用软件系统编译过程的并行算法设计与优化

硕士生姓名 邢翔

学 科 专 业 计算机科学与技术

研 究 方 向 数值预报系统及其并行计算

指 导 教 师 吴建平 研究员

国防科学技术大学研究生院

二〇一六年十一月

论文书脊

（此页只是书脊样式，学位论文不需要印刷本页。）

（硕士学位论文题目）

国防科学技术大学研究生院

Parallel Algorithm Design and Optimization for Compiling Process of Large-scale Software System

Candidate：XING Xiang

Advisor：Prof.WU Jianping

A dissertation

Submitted in partial fulfillment of the requirements

for the degree of Master of Engineering

in Computer Science and Technology

Graduate School of National University of Defense Technology

Changsha，Hunan，P.R.China

（November，2016）

独 创 性 声 明

本人声明所呈交的学位论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表和撰写过的研究成果，也不包含为获得国防科学技术大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文题目：大规模应用软件系统编译过程的并行算法设计与优化

学位论文作者签名：邓翔　　日期：2016年 10月 31日

学位论文版权使用授权书

本人完全了解国防科学技术大学有关保留、使用学位论文的规定。本人授权国防科学技术大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档，允许论文被查阅和借阅；可以将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、汇编学位论文。

（保密学位论文在解密后适用本授权书。）

学位论文题目：大规模应用软件系统编译过程的并行算法设计与优化

学位论文作者签名：邓翔　　日期：2016年 10月 31日

作者指导教师签名：　　日期：2016年 10月 31日

目 录

表 目 录

图 目 录

摘 要

大规模应用软件系统由于其规模大、程序文件多，导致软件系统的编译过程特别耗时。特别是在软件系统开发和升级过程中，由于需要频繁进行软件系统集成，从而使得系统的编译耗时严重影响软件研发效率。目前，国内外在设计软件系统的编译方法方面很少考虑到编译过程的并行实现。

本文通过分析软件系统程序文件的设计构造，提出了根据程序文件间依赖关系实现系统编译过程并行化的方法：(1)对软件系统的程序文件结构进行分析，设计出软件系统程序文件间依赖关系的分析算法，构造软件系统的依赖关系有向图；(2)通过有向图的存储方式实现图的层次构造，生成系统程序文件可以并行编译的分层结果；(3) 生成系统程序文件的分层编译脚本，并对编译脚本进行参数配置，实现系统编译过程的并行化；(4)根据程序文件在层间分配和层内动态调度上对并行效率的影响，对并行化方案进行优化调整。

文中以 WRF 模式和 YHGSM 模式为例进行并行化编译实验，这两个模式系统都自带一定程度的简单并行化编译方式。通过并行化编译实验表明，在不同进程数下，WRF 模式和 YHGSM 模式的编译时间相对于其原先编译方式所耗时间大幅度减少，编译效率得到明显提高，两模式的优化方案也取得了较好的改善效果。

基于有向图理论，依据程序文件依赖关系对程序文件进行分层，实现软件系统编译过程并行化的研究还处于初级阶段。国内外还没有比较成熟的设计方案，在实际应用上更是少见，因此对该领域的研究具有重要的前瞻性，对软件系统编译方法的设计有重要意义。

关键字：软件系统，有向图，依赖关系，编译过程，并行算法

ABSTRACT

The compilation process of large-scale software systems takes a long time in general, because of the number of program files, which affect the efficiency of software development especially in the software integration. At present, domestic and foreign researches rarely take into account the parallel implementation of the compilation process of software system.

In this paper, we proposes a method to parallelize the system compilation process according to the dependencies of software program files: (1) It analyzes the program file's structure of the software system, and proposes algorithm to get the dependencies of the program files in order to construct the dependency directed graph of the program files; (2) The hierarchical structure of the graph is realized by the storage of the directed graph, and generates a hierarchical result of a system program files which can be compiled in parallel; (3) To generate the compilation scripts of the system program file, and then configure the parameters of the scripts to realize the parallel compilation of the system; (4) The parallelization scheme is optimized by adjusting the program files in the inter-layer allocation and the intra-layer dynamic scheduling.

In this paper, WRF mode and YHGSM mode are used in the parallel compilation experiments, and both of them have a simple parallel compilation method. The experimental results show that the compiling time of WRF mode and YHGSM mode has been increased obviously comparing of their original compilation method. The compiling efficiency is significantly improved, and both of the optimization schemes have a good effect.

It is still in the preliminary stage to research on the parallelization of the compilation process of the large-scale application software system based on the program file dependency directed graph. There are few mature designs at home and abroad, especially in practice. Therefore, the research in this field has important prospective and is significant to the design of software system compiling method.

Key Words：software system，directed graph，dependencies，compilation，parallel algorithm

第一章　绪论

1.1 研究背景

随着现代科学技术的进步，科学实验研究已经触及到社会生活的各个方面，实验难度也逐渐上升。由于在现代研究实验中，经常遇到在现阶段科学实验理论仍不完备以至于无法建立实体模型进行研究或者实验破坏性大而不能频繁实验以及实验条件要求苛刻而难以进行重复性实验等问题，致使许多科学问题无法继续开展研究。正是由于传统的理论科学与实验科学已经渐渐不能满足日益进步的现代科学研究的需求，从而促使基于计算机技术的数值模拟和科学计算得到迅速发展，使得许多传统科学方法无法进行研究的问题有了新的解决思路和方法。同时，由于科学研究的日益精细化，科学计算向并行计算提出了更高的性能需求，使得高性能并行计算技术得到了迅速的发展，而如何在科学研究中更好地发挥并行计算的性能优势逐渐成为全世界科研工作者的工作焦点[1-10]。随着 CPU 制造技术的不断发展和单处理器所碰到的物理极限以及功耗等问题造成的体系结构技术上的变革[11-12]，使得国际上对并行计算性能的关注逐渐转移到了多核处理器上，即实现在单个处理器上执行多个核进行工作。

多核处理器在应用设计方面取得了很多实质性的突破。比如，多核处理器支持真正意义上的任务并行操作，可以轻易地实现数倍单处理器任务同时执行，极大地提升了处理器性能；而且，多核处理器突破了单核处理器功耗暴涨的限制，处理器性能提升的方式从单核时的提升时钟频率转换为直接增加处理器核的数目，实现了当处理器芯片的功耗维持在相对较低的水平时，可以通过简单地增加处理器核的数目来提升计算机性能。正是由于多核处理器的这些突破使得在多核环境下进行的并行计算获得了科学界的广泛关注[13-22]。S. Borkar 预测多核处理器将是高性能的主要发展趋势。W. Hwu[23]预测未来科学计算领域将广泛应用多核处理器来进行数值模拟的科学计算任务。

1.2 编译技术简介

自从 20 世纪 50 年代出现了以 Fortran 为代表的计算机高级编程语言以后，由于高级编程语言在抽象层次和使用方便等方面相对于机器代码和汇编程序有着明显的优势，使得高级编程语言获得了迅速发展[24]。

目前，常用的运行高级语言程序方式有两种，一种是编译执行方式，即先把源程序文件用翻译成可以执行的目标程序文件，然后再运行，如图 1.1 所示[24]；另

外一种是解释执行，如图 1.1 所示[24]，这种方式不产生源程序的目标文件，而是对源程序文件内容进行分析，然后执行直接产生结果。

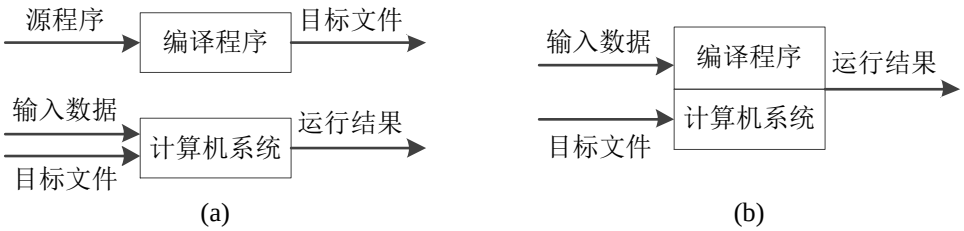

图 1.1　高级语言程序的编译方式

解释执行的方法一般用于测试，由于系统的软件规模、应用方向，软件系统程序一般都选用编译执行。高级语言程序的编译执行过程分为以下几个阶段：

1.词法分析：词法分析的主要工作是识别出程序中的单词符号、在编译程序符号表中查找并登记单词符号及其信息，譬如单词符号的类型、内部表示数值等。

2.语法分析：语法分析主要工作是在上一步工作的基础上得到各类的语法单位。

3.语义分析和中间代码生成：语义分析主要是检查程序文件内容中语义的正确性，完成语义分析后便生成中间代码。引入中间代码的目的主要是使编程语言及其程序不依赖特殊的计算机类型，方便程序文件的移植。

4.代码优化：代码优化的主要工作是对上一步产生的中间代码进行变换，为生成目标代码文件做好准备。

5.目标代码生成：这是编译过程的最后步骤，主要是根据中间代码生成可执行文件。

上述程序文件编译步骤的五个阶段分别对应着完成其程序文件编译过程的五个相应部分，如图 1.2 所示[24]。

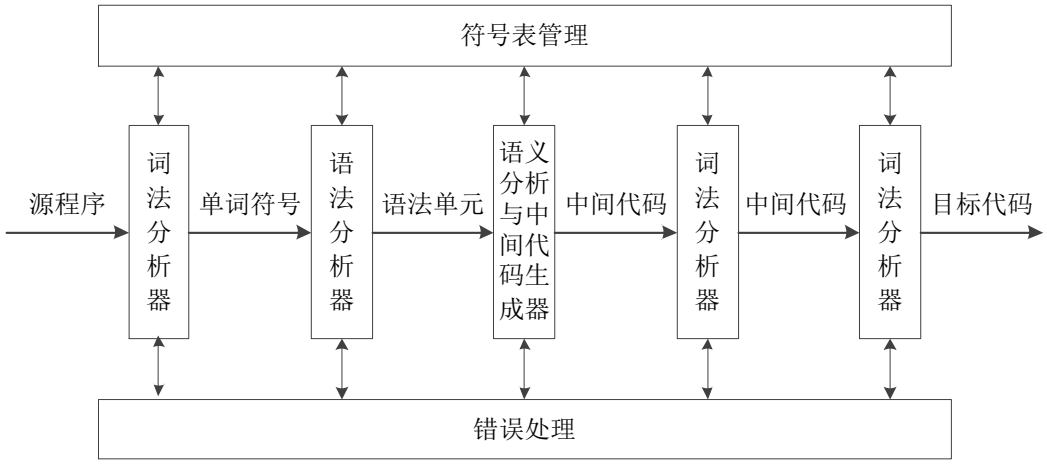

图 1.2　编译程序结构图

一个源程序文件到生成最终的可执行代码要经过的工作流程[25-26]，如图 1.3 所示，其中预处理器负责将源程序文件中的头文件和宏展开；编译器则是负责将生

成的中间代码转换成 ASCII 汇编语言文件；汇编器是将上一步的汇编文件生成目标文件；链接器将所有程序的目标文件链接生成最终的可执行文件，当用户运行此该文件时，操作系统会拷贝可执行文件的代码和数据，执行程序产生结构[27]。链接器是实现软件系统并行编译的硬件基础，它使得软件系统程序文件的分离编译得以实现。因为对于大规模应用软件系统来说，源程序文件是非常庞大的，而正是由于链接器的存在，使得在开发维护软件系统时可以将软件系统程序分解成易于修改和编译的模块文件，而不必将整个系统程序内容编写在一个程序文件中，实现对软件系统进行模块化设计，提高工作效率。

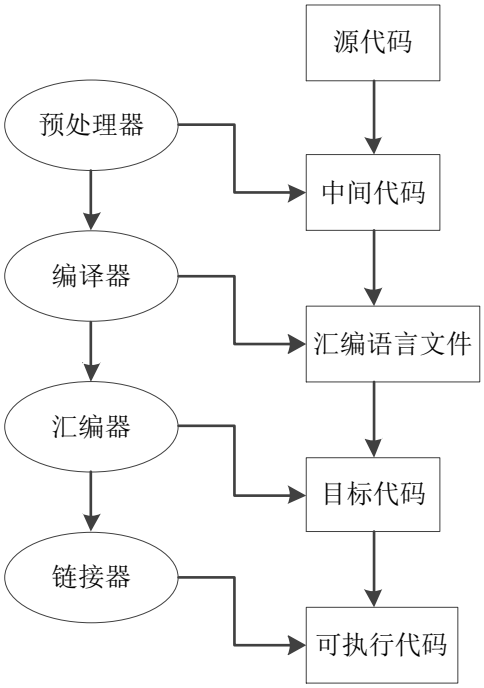

图 1.3　现代编译器工作流程

随着并行技术的发展，并行编译技术[28]成为高级语言程序编译方式中重要的一部分[29-31]。并行编译有两层含义：一种是串行程序的并行化，另一种是并行程序的编译程序。由于编写并行程序难度大，所以大部分并行编译都是以实现串行程序的并行化为主，将串行程序转换为并行代码，因此实现串行程序并行化的自动并行编译技术[32]成为了重要的研究方向。

串行程序的并行编译分为与机器无关的分析和与机器有关的处理。与机器无关的分析是指预处理阶段，依据编译指令对程序文件扩展相应的运行库子程序文件；与机器有关的处理主要是对程序进行依赖关系分析及向并行化程序的转换两部分，包括前端处理、主处理和后端处理，其中前端处理是将源程序转化为中间形式， 主处理器主要是完成对中间语言的处理和优化，后端处理是生成目标语言和编译指令的组合[33-34]。

并行编译过程如图 1.4 所示[35-36]，一般在实际应用中串行的源程序转换成并行

化的程序时采用并行化工具独立于并行编译器，因为该方法可以利用现有的串行编译器来完成，生成的并行化源代码有利于接下来的改进和优化，而且具有很好的可移植性[37]。

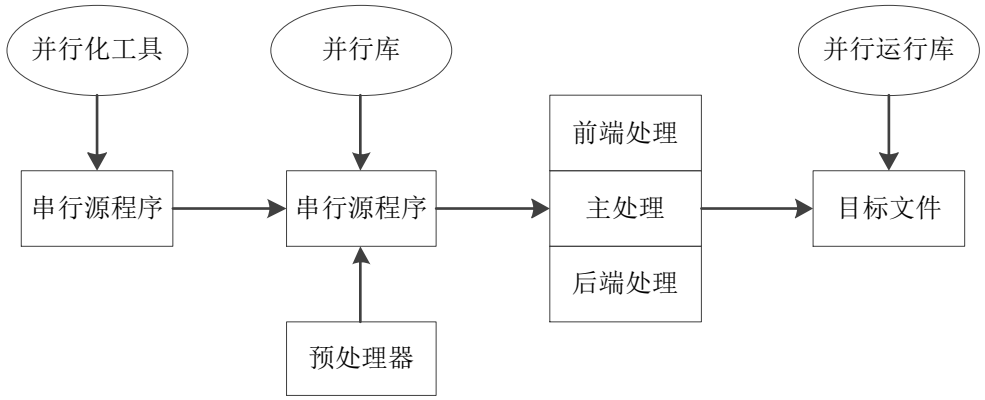

图 1.4　并行编译工作流程

1.3 并行算法的性能评价

并行算法是指适合于特定并行计算模型上对问题进行求解的算法，通过诸多进程协同完成若干可同时执行任务的过程，从而实现对给定问题的求解。

并行算法的复杂性在于它所包含的工作量，常用的度量标准包括算法的运行时间和处理器数与问题规模 w 的关系以及同步和通信[38]。运行时间 $t(w)$是指算法从开始求解问题到求解结束所消耗的时间；处理器数 $p(w)$是指求解给定问题所用处理器数目，为 $p(w)= w^{1-\varepsilon}$，$0<\varepsilon<1$。同步是并行计算在为了保证各进程正常工作而强迫各进程必须在某一时间点上相互等待的过程；通信是指并行计算的各进程进行相互间数据交换的过程。

设计并行算法时主要从并行算法成本、加速比、并行处理效率以及可扩展性等几个方面考虑算法的性能。并行算法成本 $c(w)$是指算法求解问题的运行时间与使用的处理器数的乘积；加速比为：

$$S(w)=\frac{t_s(w)}{t_p(w)}$$

加速比是用来评估并行算法对运行时间改进的效用[39]，其中$t_s(w)$表示并行算法在一个核上串行运行所需的时间，$t_p(w)$表示并行算法并行运行所需的时间，因为$t_s(w) \le p(w) \times t_p(w)$，所以加速比存在界限$1 \le S(w) < p(w)$；并行处理的效率为：

$$E(w) = \frac{\text{加速比}}{CPU\text{数量}} = \frac{S(w)}{p(w)}$$

它通过加速比与处理器数的比值，反映了并行计算时计算机资源的使用情况。虽然加速比可以用来评估算法对优化运行时间的作用好坏，但是却不能反应对计算

资源利用的效率，因为有些算法加速比很好但是资源利用率却不高。因此，在这种情况下，加速比$S(w)$就不能很准确地评价一个算法的好坏，而考虑了加速比与处理器数关系的并行效率$E(w)$则是比较好的选择。可扩展性是指对于求解给定问题，并行算法的性能随着处理器核数的增加能否按比例地提高[40-42]。

1.4 国内外研究现状

并行编译指的是将程序文件编译为可并行执行的代码，本文所做的并行化编译是指实现大规模应用软件系统编译过程的并行化，与传统的并行编译概念有所不同。

在传统的软件系统开发过程中，对于中小型软件系统来说，编译过程一般是比较快速的，所占用的时间几乎可以忽略。但是，在大规模应用软件系统中，由于这些软件系统包含成百上千个源程序文件，规模大、程序文件数量多，百万行源代码几乎比比皆是，从而使得编译过程极为耗时，大比例占据开发周期时间。目前，软件系统程序文件的编译方法类似于按照一个串行路线进行一系列程序文件的编译过程，这种方法极为耗时，易造成计算资源浪费。例如，数值天气预报系统 WRF 模式编译依次耗时 50 分钟左右，国内的 YHGSM 预报系统的编译时间约为 40 分钟。而且对于大规模应用软件系统来说，为了保证软件的质量，开发过程中需要频繁进行持续集成[43]，而每次集成都需要完成一次完整的编译。一般对于大规模应用软件系统，集成活动会占据整个软件系统开发周期的很多时间[44]，其中占据时间比例最大的就是编译过程。比如，在 Chromium 开源项目中，每次集成就会触发一次完整的编译和测试，这个过程非常耗时，通常需要两到三个小时才能得到结果。可见，编译过程的耗时直接影响着大型软件系统的开发维护周期，因此对大规模应用软件系统编译过程的并行化设计是十分必要的。

大规模软件系统多采用模块化程序设计思路，系统由许多程序功能模块组成的，不同模块负责实现不同的功能。大规模应用软件系统的这种程序设计结构具备了实现软件系统编译过程并行化的基础，因为软件系统的每个程序文件并不一定用到所有定义的模块，因此很多程序文件之间是相互独立的，这些程序文件的编译过程是可以轻易实现并行化的，而且对于有依赖关系的程序文件，也可以通过分析依赖关系，实现编译过程的并行化。比如，如果一个程序文件 A 对另一个程序文件有依赖关系，即程序文件 A 依赖于程序文件 B，那么只要在程序文件 B 先编译完就可以对程序文件 A 进行编译，因此只要将程序文件 A、B 放在不同层次的 makefile 里，就可以实现编译过程的并行化。在这里，将每个程序文件看成一个节点，程序文件编译时两两之间的这种依赖关系用有向边来表示，则可以将整个模式系统中程序文件之间的依赖关系用一个有向无回路图来表示。这样，就

可以根据程序编译过程中程序文件之间的依赖关系，通过对有向无回路图进行宽度搜索形成的层次关系可知，每层内的程序文件不存在相互之间的依赖关系，是可以并行编译的，从而可以将第一层中的所有程序文件同时进行编译，等前一层编译结束则开始下一层中所有程序文件的编译，依次进行至最后一层，就可以完成整个模式系统的编译。

目前，国内外在大规模应用软件系统编译过程方面很少有考虑实现编译过程并行的设计方案，虽然有些软件系统考虑了编译过程的并行化，但是仅仅针对于完全没有依赖关系的纯子程序文件，例如数值预报系统 WRF 模式和国内的 YHGSM 模式，而不是利用图论知识对所有程序文件依赖关系进行分析实现整个软件系统编译过程的并行化。虽然考虑完全没有依赖关系的程序文件编译过程的并行可以提高编译效率、缩短编译时间，但是由于这些纯子程序文件数目少，在整个软件系统程序文件中只占到很小一部分比例，所以随着并行进程的增加，这种方法的并行效率会逐渐降低，不能充分发挥并行编译的优势。

1.5 主要研究内容

本论文主要研究和改进大规模应用软件系统的编译方式。为了能够充分利用多核计算机资源，加快大规模应用软件系统编译的速度，提高开发人员的工作效率，文中以数值预报模式系统 WRF 和 YHGSM 模式为研究对象，在 Linux 平台上完成这两种应用系统编译过程的并行化实现。所有实践过程在一个服务器上进行，该服务器含 16 个 Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz(cache 20480 KB)处理器核，其操作系统为 Linux version 2.6.32-220.el6.x86_64。本文提出的并行化编译方法主要应用于大型软件项目的编译过程，可以大幅度减少软件系统编译时间，提高工作效率。

本文主要工作有以下几个方面：

1、阅读分析软件系统的程序文件内容，分析程序文件的基本构造及模块设计，结合原有编译方式的日志，深入了解软件系统的编译流程。

2、根据应用软件系统程序文件结构特点进行依赖关系算法分析，获得整个系统程序文件编译所需要的依赖关系，生成程序文件的依赖关系有向图，并根据有向图存储和层次构建将系统所有程序文件进行分层。

3、结合上一部分的软件系统程序文件的分层结果，生成实现分层并行编译的编译脚本，实现大规模应用软件系统在多个进程下的并行编译。同时根据可能影响并行编译效率的因素，对软件系统程序文件分层的情况进行层间分配和层内动态调度上的优化，进一步缩短编译时间，优化编译并行化方案。

1.6 本文组织结构

全文总共分为 6 章，内容安排如下：

第一章：绪论。介绍了课题的研究背景，包括编译技术简介、并行算法性能评价以及软件系统编译研究领域的国内外发展现状，然后通过对比现有大规模应用软件系统的编译方式，详细介绍大规模应用软件系统并行编译的设计思想，最后概括介绍了本文的主要研究内容和论文的结构布局。

第二章：图论。主要介绍了在本文中实现程序文件依赖关系分层分析需要使用到的图论基本知识，内容包括有：图论的基本简介、有向图和有向无回路图概念、有向图的存储以及图的层次结构与构建。

第三章：软件系统的并行化编译算法设计。本章首先介绍了软件系统程序文件依赖关系的基本概念和分析原理，从科学计算的角度分析大规模应用软件系统的程序语言结构和程序文件特点，并以此为基点出发，详细阐述获取软件系统程序文件依赖关系的算法设计与分层算法设计，最后介绍了实现分层的程序文件编译过程的并行化的方法。

第四章：大规模软件系统的并行化编译实验。本章是本文的核心工作，分别以数值预报模式系统 WRF 和 YHGSM 模式为对象，详细介绍了大规模软件系统的并行化编译方法在这两种系统中的实现。首先简述这两种大规模应用软件系统的程序文件构造和编译方式，然后利用第三章介绍的算法找出两种软件系统的程序文件的依赖关系，并得到程序文件的分层结果，同时生成程序文件并行化编译的 makefile 文件，并实现 WRF 模式和 YHGSM 模式在多个进程下进行并行编译。

第五章：软件系统程序文件并行化编译的优化设计。本章针对前一章介绍的并行化编译设计，提出可能影响并行效率的因素，并介绍了软件系统程序文件并行化编译在层间分配和层内动态调度上的优化设计，得到较好的效果。

第六章：结束语。对本文的主要工作内容和重要实验结果进行了总结分析，并根据并行化编译的效果展望了未来需要改进的方向。

第二章　图论

2.1 图论基本简介

图论是一种研究具有独特性质的物体集合及其之间关系的数学方法。它作为一门新兴学科，在近几年中随着科学计算的快速发展以惊人的速度广泛应用于各个领域，受到越来越多科学界的重视。

2.1.1 图的基本概念

图通常包括以下几个部分：

顶点：顶点集合表示为 V(G)=$\{v_1, v_2, v_3, \dots, v_n\}$，其中 n 表示顶点的个数。

边：边集合表示为 E(G) =$\{e_1, e_2, e_3, \dots, e_n\}$，其中 n 表示边的个数。

顶点的度：图中与顶点V_f相连接的边的总数称为V_f的度，记为d_i。

权值：边相连接的两顶点间的紧密程度称为边的权值。

图在数学表达中可以定义成一个偶对(V，E)，记作 G=(V，E)，其中 V 是图的顶点的集合，E 是图的边的集合。一个图的最基本的性质就是该图是否连通，若图 G 中的两个顶点 u、v 间存在一条(u，v)道路，使得顶点 u、v 相连接，那么称在图 G 中顶点 u 和 v 是连通的；若图 G 中所有顶点都是连通的，那么称图 G 是连通的[45]。

连通是顶点集 V 上的一个等价关系，设顶点 u 和 v 是连通的，并用 u≡v 来表示，那么这种顶点间的连通关系存在以下性质：(1)反身性：u≡u；(2)对称性：若 u≡v，则 v≡u；(3)传递性：u≡v，v≡w，则 u≡w。

2.1.2 图的矩阵表示

在计算机科学中，很多算法都涉及到图，因此需要一种方便的方法利用计算机存储和操作图。最常用的图存储方法是矩阵存储，通过将图用矩阵存储在计算机中，那么就可以利用矩阵运算来了解它的有关性质，下面将介绍本文需要用到的关联矩阵和邻接矩阵。

定义 2.1：设图 G=(V，E)的顶点集合和边集分别为 V(G)=$\{v_1, v_2, v_3, \dots, v_p\}$；E(G) =$\{e_1, e_2, e_3, \dots, e_p\}$，图 G 的关联矩阵定义为 **B**(G)=$(b_{ij})_{p*q}$，其中$b_{ij}$代表顶点$v_i$与边$e_j$关联的次数。

从图的关联矩阵 **B**(G)的定义，可以得出下列直观的结果：

1、因为每条边都关联这两个顶点，所以 **B**(G)的每列元素的和都是 2；

2、**B**(G)的每行元素之和对应顶点的度数；

3、如果一个顶点对应行的所有元素都为零，那么该顶点称为孤立点；

4、重边表示在矩阵中有相同的列。

定义 2.2：设图 G=(V，E)的顶点集为 V(G)={ $v_1, v_2, v_3, \dots, v_p$}，G 的邻接矩阵[46]定义为 n 阶方阵 $\boldsymbol{M}(G) = (a_{ij})_{p*p}$，其中$a_{ij}$表示 G 中顶点$v_i$与$v_j$之间的边数。

图的邻接矩阵有以下性质：(1)无环图的邻接矩阵$\boldsymbol{M}(G)$中某行(列)的元素之和与该顶点v_i的度数相等；(2) 邻接矩阵$\boldsymbol{M}(G)$是一个对称矩阵；(3) 两个图 G 和 H 同构的充要条件是存在一个置换矩阵 $\boldsymbol{P}$，使得$\boldsymbol{M}(G) = \boldsymbol{P}^{\mathrm{T}}\boldsymbol{M}(H)\boldsymbol{P}$。

2.2 有向图基本概念

在图论的应用中，经常会遇到不仅需要描述问题的图的图形，而且还需要指出图中每一条边的方向。这是因为在有些问题中，一对顶点之间的关系不是对称的。比如，在时序电路中，从一个状态到另外一个状态的转换，往往都具有方向性。这种方向性表示所描述的物理系统的某种次序或单向性质，同时为了描述某种参考系统而赋予边以一定的方向，那么这样的关系就不能简单地用前面所介绍的图来表示。为此，引入有向图的概念。

当图的边是有方向的时候，便称该图为有向图。有向图通过把具体系统的每个物体对象抽象成图中的节点，把物体之间的关系抽象成联系两个节点之间的有向边，直观形象地表现出该系统的各个对象及其相互关系。

有向图最重要的性质在于它的有向边，有向图中的边是由两个顶点组成的有序对，与一条边相连的两个顶点具有一定的次序关系。例如，e_{ij}表示顶点(V_i， V_j)之间的边，在无向图中e_{ij}可以表示顶点(V_i， V_j)或(V_j， V_i)之间的无向边，和e_{ji}表示的是同一条边，但在有向图中，e_{ij}表示以V_i为起点和V_j为终点的有向边，而e_{ij}表示的是以V_j为起点和V_i为终点的有向边，二者是连接同一对顶点的两个不同边。同时，对于有向图来说，顶点的度分为入度和出度，其中入度是指以某顶点为终点的有向边数目，出度是指以某顶点为起点的有向边数目。如果某顶点的出度为正数而入度为零，说明该顶点有出无进，则称该顶点为源；如果某顶点的入度为正数而出度为零，说明该顶点有进无出，则称该顶点为汇。

通路与回路是图论中两个重要的概念，下面先给出本文中所用到的通路与回路的定义。

定义 2.3：设 G 为有向图，顶点v_{i_0}到v_{i_l}的通路是指在有向图 G 中顶点与边的交替序列Γ=$v_{i_0}e_{j_1}v_{i_1}e_{j_2}\dots e_{j_l}v_{i_l}$，其中$v_{i_{r-1}}$为$e_{j_r}$的始点，$v_{i_r}$为$e_{j_r}$的终点，$r = 1,2,\dots,l$，Γ中边数$l$称为Γ的长度。若$v_{i_0}$=$v_{i_l}$，则称通路Γ为回路。

如果Γ中的所有边各异，那么称Γ为简单通路；如果当Γ为简单通路时并且存在v_{i_0}=v_{i_l}，那么称Γ为简单回路；如果Γ中存在多个边重复出现，那么称Γ为复杂通路，同时如果存在v_{i_0}=v_{i_l}，那么称Γ为复杂回路[47]。

如果某有向图中既不包含回路，也不包含循环边，那么称该有向图为有向无回路图。由于有向无回路图中的节点可以依次计算，所以如果有向图中包含回路，那么必须要去掉这些回路，否则有向图的计算就会存在死循环。文[48]提出如何在回路中消除相应的有向边，避免在有向图计算过程中出现死循环。在本文中提到的所有有向图均假设为无回路的。

2.3 层次结构与构建

为了方便建立有向图的层次结构，常采用邻接矩阵和可达矩阵来存储有向图，并根据可达矩阵的性质来进行图的层次构建。

定义2.4：设有向图表示为G =(V，E)，那么该有向图的邻接矩阵为A=$(A_{ij})_{n*n}$，其中 n 为节点个数，A_{ij}为：

$$A_{ij}=\begin{cases}1，\text{两节点间存在直接有向边}\\0，\text{两节点间不存在直接有向边}\end{cases}$$

同时该有向图的可达矩阵为 P=$(P_{ij})_{n*n}$，其中 n 为节点个数，P_{ij}为：

$$P_{ij}=\begin{cases}1，\text{节点}v_i\text{到}v_j\text{至少存在一条有向路径}\\0，\text{节点}v_i\text{到}v_j\text{间不存在有向路径}\end{cases}$$

为了方便理解有向图矩阵存储概念，下面将以一个具有 5 个节点的有向图为例来介绍如何构造邻接矩阵和可达矩阵。

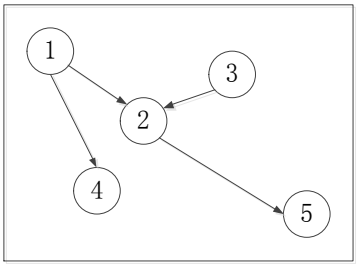

图 2.1　有向图示意图

在图 2.1 的有向图中，如果节点之间存在直接的有向边，那么节点之间的关系就用数值“1”表示；如果节点之间不存在直接的有向边，那么节点之间的关系就用数值“0”表示，以此类推，遍历各个节点便可以得到该有向图所有节点间的相邻关系。在图 2.1 中可以看到，节点 1 与节点 2、4 间有直接的有向边，那么节点 1 与节点 2、4 间关系用“1”来表示，同时节点 1 与节点 3、5 没有直接的有向边，则节点 1 与节点 3、5 间关系用数值“0”来表示。通过分析图中所有节点间关系

后，可以得到图 2.1 种有向图的邻接矩阵 A，如表 2.1 所示：

表 2.1　有向图的邻接矩阵

	节点 1	节点 2	节点 3	节点 4	节点 5
节点 1	0	1	0	1	0
节点 2	0	0	0	0	1
节点 3	0	1	0	0	0
节点 4	0	0	0	0	0
节点 5	0	0	0	0	0

对于有向图的可达矩阵，如果节点间存在直接的有向边或者通过直接的有向边可以到达，那么节点间关系就用数值“1”来表示。在图 2.1 的有向图中，由于节点 1 与节点 2 有直接的有向边，同时节点 2 与节点 5 有直接的有向边，所以节点 1 与节点 5 之间要用数值“1”表示，以此类推得到图 2.1 的可达矩阵 P，如表 2.2 所示：

表 2.2　有向图的可达矩阵

	节点 1	节点 2	节点 3	节点 4	节点 5
节点 1	0	1	0	1	1
节点 2	0	0	0	0	1
节点 3	0	1	0	0	1
节点 4	0	0	0	0	0
节点 5	0	0	0	0	0

如果邻接矩阵 A 能通过 E^TAE 运算转换成为分块三角阵，那么称邻接矩阵 A 是可以分层的；否则是不可以分层的。基于可达性的分层方法如下：

(1)根据有向图得到图的邻接矩阵A_{ij}；

(2)判断是否可以分层，若可以分层，则进行下一步；若不可以分层，则退出；

(3)将邻接矩阵A_{ij}转化成为可达矩阵P_{ij}，获取矩阵P_{ij}的各个节点第 i 列的数值，存储数值为 0 值对应的节点数，并将这些节点组成集合，这就是第 1 层的节点集合；

(4)移除矩阵P_{ij}中已经分配的第 1 层的各个节点，判断是否还有节点，如果还有节点，则根据剩下的节点便可以得到一个新的矩阵，若剩余节点数为 0，则退出；

(5)回到步骤 3，对新的矩阵进行操作得到第 2 层的节点集合，依次处理，判断图的剩余节点是否为 0，直到处理完所有的节点，便可以得到该有向图的完整分层结果。

本文将在论文的第 3 章中以数值预报系统 WRF 模式和 YHGSM 模式为例，详细描述如何借助有向图的矩阵存储方式实现软件系统程序文件分层的算法设计。

2.4 本章小结

本章介绍了图论的一些基本理论，对论文中需要用到的基本图论知识进行了定义，规范了论文中有关图论方面的表述规则，并且针对论文中处理软件系统程序文件的依赖关系等问题引入了有向图的概念，并通过实际应用进一步提出有向无回路图的定义以及关于有向图矩阵存储的方法。同时，详细阐述了根据有向图矩阵存储的形式进行图层次构建的原理，并给出有向图分层的算法步骤。

第三章 软件系统的并行化编译算法设计

在前面的章节中，已经介绍了实现软件系统编译过程并行化所需要的基本知识和技术，同时也明确了实现并行化编译的基本需求。本章给出并行化编译的算法设计，主要包括对大规模应用软件系统程序文件的结构特征分析、详细阐述程序文件依赖关系算法的设计和分层编译的并行化设计。

3.1 程序文件依赖关系分析

3.1.1 程序文件依赖关系基本概念

随着计算机技术和社会工程需求的发展，大型应用软件系统的规模越来越大、程序文件越来越多，导致了对软件系统程序文件的依赖关系分析的难度日益上升。目前，程序分析主要有两种方法：一种是查看软件系统的概要或者详细设计，但是由于系统的设计概要主要是概括描述软件系统的设计，和实际软件系统程序内容有较大区别，很难全面地了解软件系统程序文件之间的关系和功能；另一种是直接分析软件系统的源代码文件，虽然这样可以了解到系统程序的细节，但是由于程序机器语言与人类语言之间存在着语义间隙，源程序文件理解起来晦涩难懂，不利构建对整个系统程序文件间的依赖关系的认识，而且软件系统的程序文件规模大、数量多，导致这种分析方法费时且效率低。本文提出的程序文件依赖关系分析方法是一种处于文档和代码文件之间的分析方法，既可以比概要更详细地分析软件系统程序文件，又可以避免分析源程序代码的繁琐，很好地弥补了目前两种程序文件分析方法的不足。

程序文件依赖关系分析是一种重要的程序分析和理解的方法。程序文件间的依赖关系是指在多个程序文件中，如果程序文件 A 编译前，程序文件 B 必须已经编译完毕，否则程序文件 A 不能进行编译，那么称程序文件 A 依赖于程序文件 B，即程序文件 A 与 B 存在依赖关系。从 70 年代起，依赖关系分析理论和技术的研究工作不断深入。1988 年 Banerjee 在《超级计算中的依赖关系分析》中详细阐述了依赖关系分析理论和技术的研究，成为这一领域的权威著作[49]。90 年代后，不断有学者陆续提出几种新的依赖关系分析算法来更好地分析复杂的依赖关系[50-53]。

3.1.2 科学计算程序的结构特征

程序文件分析主要是对主程序、模块文件、子程序文件等文件之间的依赖关系进行分析。程序分析是并行化的关键，在该过程中应尽量挖掘程序文件间存在

的潜在并行性，以便获得并行处理效率最高的全局优化方案[54]。

本文进行的软件系统编译过程并行算法设计，将以数值预报模式系统 WRF 和 YHGSM 模式为研究对象，对所设计的算法并行处理效率进行实验验证。

对于像数值预报系统这样的大规模软件系统进行程序分析，首先必须要了解所研究模式系统存在的程序文件种类，然后利用依赖关系分析算法进行其间依赖关系的识别。在 Linux 系统下模式系统的源程序文件、目标文件等后缀名如表 3.1 所示。

表 3.1　源程序文件后缀规范

文件类型	后缀名
C 语言源程序文件	.c
C++ 语言源程序文件	.C, .cc, .cpp, .cxx, .c++
Fortran77 语言源程序文件	.f, .for
Fortran90/95 语言源程序文件	.f90
汇编语言程序文件	.s
目标文件	.o
头文件	.h
Fortran90/95 模块文件	.mod
静态链接库	.a
动态链接库	.so

数值预报系统 WRF 和 YHGSM 模式的主要程序文件都是用 Fortran 语言编写的，所以对程序文件的依赖关系分析主要集中在 Fortran 程序语言的构造特征上。Fortran 语言编写的程序文件共有四种程序单元：主程序、子程序、块数据单元以及模块文件。一般完整的程序文件由一个主程序文件和若干个子程序或模块文件组成，主程序文件可以引用各个子程序文件和模块文件。一个标准的 Fortran 语言源程序文件结构如图 3.1 所示[55]，其中 programA、moduleB、subroutineC 都是独立的源程序文件：

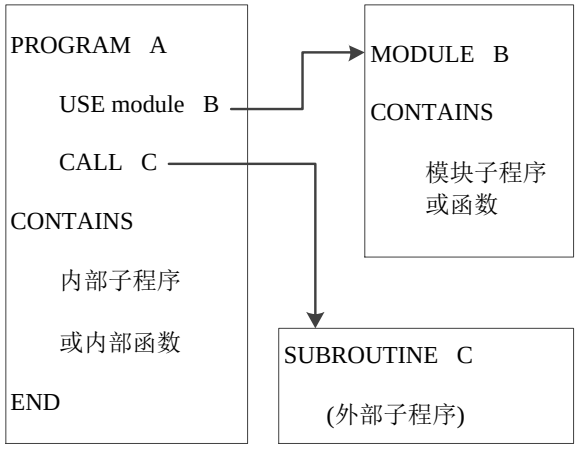

图 3.1　Fortran 程序的标准结构

模块是 Fortran 语言发展中新增加的一种程序单元，可以在主程序单元之外独立编写，有着独特形式，主要用于实现数据封装、特性继承、操作重载、公私属性等面向对象特性，使得程序更加安全、可靠、高效。模块中可声明常量、变量、数组、数据块、派生类型、接口界面块、模块函数、模块子程序，模块各程序单元可以被其他程序单元甚至自身引用，可看成外部子程序功能的扩展。特别需要注意的是：每个模块文件可以独立编写，只要依赖关系得到满足便可以与主程序文件、外部子程序文件一起编译、链接。模块定义一般格式为[55]：

<模块定义> → MODULE <模块名>

[<模块数据及模块子程序名声明>]

[CONTAINS

{<模块子程序定义>}]

END MODULE [<模块名>]

模块的程序文件内包含各种变量、数组等实体的类型说明语句以及派生类型定义和接口块，但是没有可执行语句，这意味着模块不能被直接执行，只能供其它程序单元引用。模块中的 CONTAINS 语句很少使用，当为某一个派生类型规定新的操作符时，可以将实现这些新操作的过程作为模块的内部过程置于 CONTAINS 之后，以便供各外部过程共享，一般模块文件中很少使用 CONTAINS 语句。

任何程序文件引用模块文件只需要引用该模块文件名即可，即在本程序文件内部的说明部分前加上 USE 语句，便可以将模块程序文件的内容扩展到本程序文件中来。USE 语句一般格式为[55]：

<USE 语句> → USE<模块名>{<数据对象名|模块子程序名> => <别名>}

只要在程序文件中使用了 USE 语句就可以在本程序文件中引用这些模块，相当于把这些模块文件中的程序内容都移植到本程序文件内。此时，模块文件中出现的变量和内部过程，都可以与引用该模块文件的程序文件中的变量共享。USE 语句还有其他引用方式，比如在 USE 前加上 ONLY 选项可以只共享模块文件中一部分变量，实现只取模块中部分实体与引用该模块的程序文件共享，其它实体没有共享关系。

Fortran 的子程序文件划分为两类：内部子程序和外部子程序。内部子程序包含在主程序文件 CONTAINS 结构中，必须与主程序文件一起保存，只能在本程序文件内部被调用执行，一起编译。当修改主程序文件内容或其内部子程序内容后，整个程序文件都需要重新编译，这在开发大规模应用软件系统过程中将会极大地降低程序开发效率，所以在大型的软件系统中应该尽量避免使用内部子程序。

为了发掘程序文件间存在的并行性，这里分析的子程序是独立于主程序之外

的外部子程序。外部子程序不必与主程序保存在一个源程序文件中，独立于主程序文件，可以独立编译生成 OBJ 文件，并且可与其他 OBJ 文件一起构建可执行程序文件。使用外部子程序可实现软件系统程序文件的并行化设计与编写，允许多人同时设计编写主程序和外部子程序，有利于完成大规模应用软件系统的开发与维护。

外部子程序包括外部函数程序文件和外部子程序文件两种。外部子程序文件内可以包含 CONTAINS 结构，允许定义使用自己的内部子程序，子程序名只是用来标识，不代表任何值，外部函数和外部子程序定义的一般格式为[55]：

<外部函数> → [<类型>] FUNCTION <函数名> ([<形式参数表>])

{< 说明语句 >}
{< 执行语句 >}
[CONTAINS
{< 内部子程序 >}] } 函数程序体

END [FUNCTION [<子程序名>]]

<外部子程序> → SUBROUTINE <子程序名> ([<形式参数表>])

{< 说明语句 >}
{< 执行语句 >}
[CONTAINS
{< 内部子程序 >}] } 子程序体

END [SUBROUTINE [<子程序名>]]

引用外部子程序，只要对所引用的子程序进行声明即可，声明时需给出子程序名，如 USE <子程序名>。

虽然可以将子程序和模块程序内容写进同一个源程序文件中，但是因为每次修改源程序内部一个程序时，就需要重新编译整个程序文件，耗时长而且效率低，所以把所有程序内容写进同一个源程序文件不符合软件系统的模块化设计思想。在设计大规模应用软件系统时，由于程序越长编译的时间就越长，一般常采用模块化编程方法，将一个程序划分成不同模块，写入不同源程序文件中。在 Fortran 语言中，模块文件和子程序文件是一种完全具有程序功能的基本程序块，可以作为一个独立程序文件被独立定义和编写，这种特性很适合系统程序设计的模块化。在大规模应用软件系统中，设计出独立的模块和子程序是至关重要的，不仅利于团队分工、高效工作，同时也便于后期维护、升级和移植。在设计系统中独立的程序文件时，要求模块和子程序文件的代码不应影响其它的程序文件，但由于主程序文件对模块文件和子程序文件的调用以及模块文件之间互相调用关系的存在，使得各程序文件之间又存在着依赖关系。因此，为了构架整个大规模应用软件系

统的程序文件依赖关系，需要对程序文件进行详细分析，下面将介绍几种分析软件系统的程序文件常见的几种依赖关系。

3.1.3 预处理依赖

定义 3.1：设程序文件 A 和 B 是软件系统中的两个源程序文件，若 A 中使用 include 指令导入了 B 中的内容(如#include “B.h”)，则称 A 直接包含依赖于 B。

文件包含是指一个程序文件将另一个程序文件的内容全部包含进来，被包含文件称为头文件，如图 3.2 所示[55]，程序文件 A 包含了头文件 B.h，即程序文件 A 依赖于程序文件 B.h。显然，若程序文件 A 包含依赖于程序文件 B，程序文件 B 包含依赖于程序文件 C，则程序文件 A 包含依赖于程序文件 C，所以包含依赖是具有可传递性。

```
//程序文件A
PROGRAM  A
. . .
#include “B.h”
. . .
. . .
END PROGRAM  A
```

图 3.2　Include 程序文件图

通常头文件中包含公用的宏定义与全局函数的声明，引入头文件通常会引起函数或变量的预处理依赖。在程序中使用文件包含有以下好处：

(1)可以更好地适应软件系统程序文件模块化的要求。因为组成软件系统的各个模块文件和子程序文件常按照团队开发习惯分开放置在多个文件中，通过文件包含既可以被其他程序文件所使用又可以保持软件系统程序文件存放方式，大大提高了程序文件的灵活性和可移植性。

(2)在团队开发软件系统过程中会累积一些常用的符号常量和宏定义，并将其单独编写成程序文件。通过使用文件包含，可以很方便地导入这些常用的子程序和程序文件，避免重新编写，提高程序编写效率。

(3)利于软件系统程序修改、更新和维护。当需要更新被包含的文件内容或者修改其内部的错误时，只需要修改对相应的程序文件即可，不必修改所有包含该文件的程序文件，有利于提高工作效率。

3.1.4 引用依赖

模块是 Fortran 语言的重要功能，用于实现数据封装、特性继承、操作重载、公私属性等面向对象特性，便于把大型的程序文件分解为多个小的程序文件单元，

使得整个软件程序可以在利用模块的功能基础上展开；而且在编写引用模块的程序文件时，可以不用考虑模块的具体程序细节，只需引用模块的功能即可，这样使得编写大型软件系统时对程序结构功能有更直观的认识。

由于主程序文件对模块文件、子程序以及模块文件彼此间的频繁引用，使得引用依赖是程序文件之间存在的最常见的依赖关系。若要建立软件系统的程序文件整体的依赖关系图，那么引用依赖将会是最重要的分析过程。

程序文件间的引用依赖是相对于某个模块文件而言的。这种引用依赖是指程序文件引用点依赖于模块的定义点文件，而模块非声明点文件，可以分为直接引用依赖和间接引用依赖。直接引用依赖是指在程序文件中直接引用某一模块文件而产生的引用依赖关系，如图 3.3 所示[55]：间接引用依赖是指程序文件通过头文件的预处理依赖而对某一模块文件产生的引用依赖关系，如图 3.4 所示[55]，程序文件 A 预处理依赖头文件 B.h，但并不是包含依赖于头文件 B.h，而是通过头文件 B.h 引用依赖于程序文件 C。

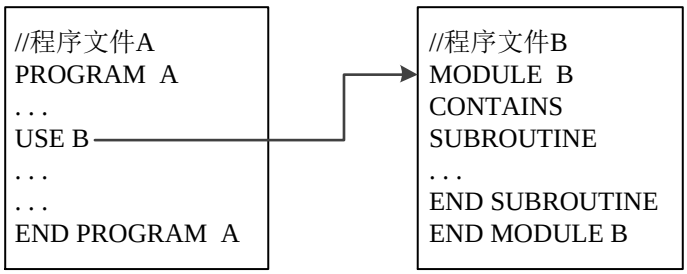

图 3.3　直接引用依赖

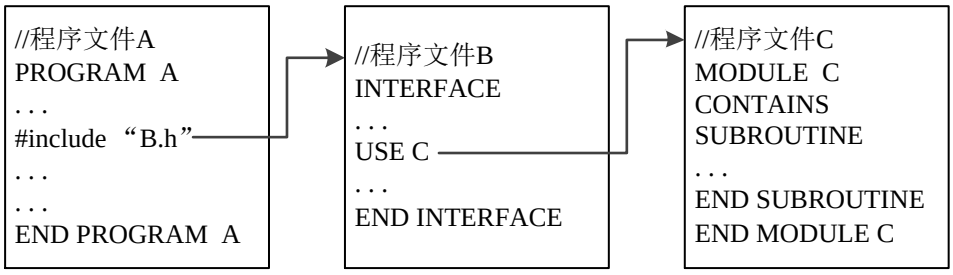

图 3.4　间接引用依赖

理解软件系统的程序文件间存在的依赖关系对建立更直观的系统程序文件依赖关系认识有重要意义，图 3.5 是软件系统中程序文件常用的几种依赖关系。

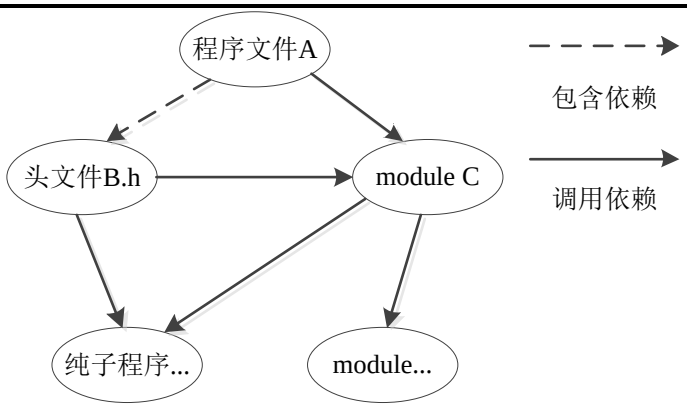

图 3.5　不同程序文件类型间的几种依赖关系

3.2 程序文件依赖关系算法设计

大规模应用软件系统的程序文件多、程序文件间的关系复杂，所以在分析软件系统的过程中，程序文件依赖关系的好坏直接影响软件系统编译过程并行化的效率。在设计依赖关系的算法时，将软件系统的每个程序文件看成一个节点，以程序文件间的引用语法为索引，建立程序文件间依赖关系。

在设计软件系统程序文件的依赖关系算法时，需要将软件系统的所有程序文件存放在同一个目录文件中，并以此目录文件地址为入口参数：首先读取所有模式系统的程序文件，建立一个程序文件目录来存储所有程序文件信息；其次根据程序文件目录信息建立每一个程序文件的依赖文件库和被依赖文件库，分别定义为程序文件的父节点和子节点；然后根据程序文件的引用语法对所有程序文件内容进行阅读分析，得出程序文件的依赖文件信息，并存储在每个程序文件的依赖文件库中；遍历所有程序文件后，再对程序文件的依赖文件库进行分析，对每一个程序文件的依赖文件库里的程序文件建立属于该程序文件的被依赖关系，并将程序文件存入被依赖文件库中。由此，便可以得到一个完整的程序文件依赖文件库和被依赖文件库信息链，这就是整个软件系统程序文件的依赖关系。具体分为以下四个步骤(采用 python 进行实现)：

1、获取软件系统的源程序压缩包，将所有源程序文件放入一个文件夹目录下，对所有程序文件进行分析，并将程序文件名存放在 module_names 中：

```
module_names = os.listdir(file_dir)
module_F_names = []
for mn in module_names:
    if mn.endswith('.F90'):
        module_F_names.append(mn.split('.')[0])
module_names = module_F_names
print module_names
```

2、根据程序文件名信息，对所有程序文件都建立该程序文件的依赖文件库和被依赖文件库，分别定义为该程序文件的父节点和子节点，其中父节点用来存储该程序文件的所有依赖文件。通过提取 module_names 里的程序文件名，扫描对应程序文件内容，定位关键字“USE”或者“use”，确定程序文件的依赖程序文件名，并存储在程序文件的依赖文件库中，遍历所有程序文件便可以得到整个软件系统程序文件的依赖文件库：

```
def __init__(self,name):
    if not isinstance(name,str):
        print "Name type error..."
        sys.exit(-1)
    self.name = name
    self.children = []
    self.parents = []
    self.keep = True
def add_child(self,child):
    self.children.append(child)
def add_parent(self,parent):
    self.parents.append(parent)
def find_parent(str):
    pattern = re.compile('(?<=[Uu][Ss][Ee] )\w+')
    parents = pattern.findall(str)
    parents = [c.lower() for c in parents]
    return parents
```

3、根据程序文件的依赖文件库，通过对程序文件依赖文件库里的程序文件数是否为零进行判断，依次对每一个依赖文件库里的程序文件建立该程序文件的被依赖文件库信息，并将被依赖的程序文件添加进去，然后弹出该依赖文件，继续判断依赖库程序文件数是否为零，若不为零则继续添加，直到依赖文件库文件数为零，遍历完 module_names 所有程序文件，便可以建立程序文件完整的依赖关系：

```
def del_child(self,child):
    if child in self.children:
        self.children.remove(child)
    else:
        print "No child %s in list"%child
def del_parent(self,parent):
    if parent in self.parents:
        self.parents.remove(parent)
    else:
```

```
            print "No parent %s in list"%parent
    def count_child(self):
        return len(self.children)
    def count_parent(self):
        return len(self.parents)
    def build_relation(file_dir):
        module_list = {}
        for m in module_names:
            module_list[m] = Module(m)
        for mn in module_names:
            f = open(os.path.join(file_dir,mn)+'.F90').read()
            parents = find_parent(f)
            if len(parents) > 0:
                for c in parents:
                    if (c in module_names) and (c not in module_list[mn].parents)
and (c != mn):
                        module_list[mn].add_parent(c)
                        module_list[c].add_child(mn)
        return module_list
```

4、最后将程序文件完整的依赖关系存储在 module_list 中，并打印出来：

```
    def print_relation(module_list):
        for n,m in module_list.items():
            print n,':',
            for c in m.parents:
                print c,'&',
            print '.'
```

本节中讨论的图是一种由顶点及连接两顶点的边所构成的有向图[56]，其中顶点集代表具体实物的集合，在图中每个顶点代表具体的程序文件；边集代表实物之间的相互关系，有向图中的边的指向表示连接的两程序文件间的依赖关系[57]。

使用有向无回图可以形象地将程序文件间的依赖关系表示出来，并且方便构造程序文件的分层。表示程序文件依赖关系的有向无回路图具体包括以下三部分：

1、顶点：顶点集合表示为$V(G) = \{v_1, v_2, v_3, v_4 \dots v_n\}$，其中 n 表示顶点的个数，与程序文件对应；

2、边：边集合表示为$E(G) = \{e_1, e_2, e_3, e_4 \dots e_n\}$,其中 n 表示边的条数，这里的边都是有向边，表示程序文件间的依赖关系，箭头的指向表示被依赖的关系。如果有一条从节点 A 到节点 B 的有向边，则表示程序文件 B 中某个 USE 后面所跟的模块名恰好在程序文件 A 中进行了定义；

3、顶点的度：顶点V_i的入度和出度分别是指以顶点V_i为终点和起点的有向边数目，分别代表程序文件间依赖与被依赖的程序文件的个数。

假设源程序文件分别为 A、B、C、D、E、F，依赖关系算法具体实现过程如图 3.6 所示。从图(a)中，通过读取源程序文件，得到程序文件名 A、B、C、D、E、F；然后通过每个程序文件的依赖分析，得到每个程序文件父节点文件，即它们的依赖关系库，如图(b)所示，程序文件 A、B 不依赖其他任何程序文件，程序文件 C 依赖程序文件 A，程序文件 D 依赖程序文件 A、B、E，程序文件 E 依赖程序文件 C，程序文件 F 依赖程序文件 D、E；由程序文件的父节点分别计算出每个程序文件的子节点，如图(c)所示，虽然程序文件 A、B 不依赖于任何文件，但是它们都有子节点，说明它们只是不依赖别的文件，但是被别的文件依赖，而程序文件 F 则是指依赖别的文件，不被其他文件依赖；最后得到图(d)，程序文件 A、B 没有依赖任何程序文件，程序文件 C、D 依赖于程序文件 A，程序文件 D 依赖于程序文件 A、B、E，程序文件 E 依赖于程序文件 C，程序文件 F 依赖于程序文件 D、E。

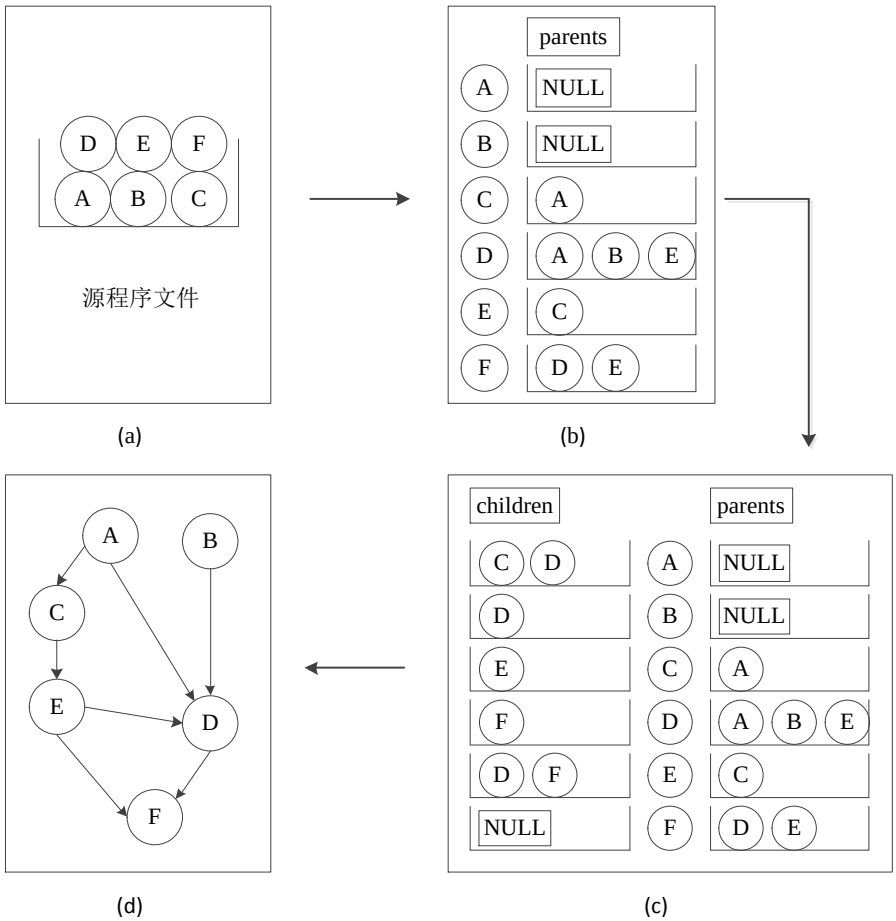

图 3.6　依赖关系算法设计图

3.3 程序文件的分层算法设计

为实现大规模应用软件系统编译过程的并行化，仅仅得到程序文件的依赖关系还不够。实现编译过程的并行化还需要将所有程序文件依据依赖关系进行分层，使得属于同一层次的程序文件间没有依赖关系，做到可以独立编译，同时为满足程序文件的依赖关系，必须保持程序文件编译的先后顺序，保证一个程序文件编译的最早执行时间必须在它依赖的所有其它程序文件编译之后，那么则要求前层次中包含了后层次中所有程序文件编译时所依赖的程序文件。因此，根据上一节得到的程序文件的依赖关系，可以构建程序文件依赖关系的有向无回路图，再根据图的层次构建方法，获得软件系统程序文件的分层结果。

从程序文件依赖关系的有向图中得到程序文件并行编译的层次关系，需要先置 i=1，再循环依次执行以下几个步骤[58]：

1、搜索入度为 0 的顶点集，并将顶点集对应的程序文件存入 i 层次内；

2、删除入度为 0 的顶点和以它们为起点的有向边，判断剩余的顶点数是否为 0，若不为 0，则由剩余的顶点重新得到一个依赖关系有向图，若剩余顶点为 0，说明已经得到了有向图的分层结果，退出；

3、i=i+1，同时对新的有向图重复步骤 1 和步骤 2，直至得到的有向图为空为止，此时将会得到 N 层的顶点分层结果。某一层程序文件的依赖程序文件全部都在该层的前面几层内，第一层程序文件都没有依赖程序文件，同层内的顶点对应的所有程序文件都没有依赖关系，可以并行编译。采用 python 实现的算法如下所示：

```
def del_child(self,child):
    if child in self.children:
        self.children.remove(child)
    else:
        print "No child %s in list"%child
def del_parent(self,parent):
    if parent in self.parents:
        self.parents.remove(parent)
    else:
        print "No parent %s in list"%parent
def count_child(self):
    return len(self.children)
def count_parent(self):
    return len(self.parents)
def analyze(module_list):
```

```
iter = 0
do_del = False
while len(module_list.keys()) > 0:
    iter += 1
    print "----------------No.%d-----------"%iter
    for n,m in module_list.items():
        if m.count_parent() == 0:
            m.keep = False
            do_del = True
    for n,m in module_list.items():
        if m.keep == False:
            for p in m.children:
                module_list[p].del_parent(n)
            print module_list.pop(n).name
    if do_del: do_del=False; continue
    else:
        print "----------------------------------------"
        print_relation(module_list)
        break;
```

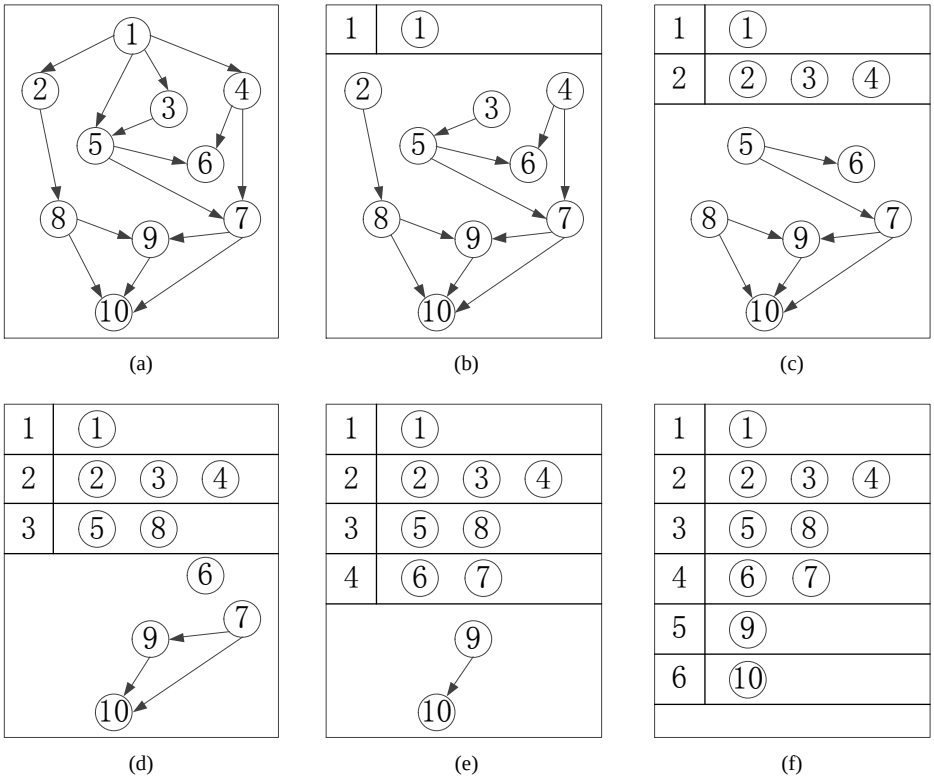

图 3.7　分层示意图

如图 3.7 所示，依据程序文件依赖关系的有向图提取出程序文件并行层次方法

的简化过程为：设程序文件的总数为 10 个，它们的依赖关系有向图如图(a)所示；从图(a)中可以看出只有 1 号顶点的入度为 0，所以第一次编译的是程序文件 1，即第一层只有一个程序文件；删除 1 号顶点和以 1 号顶点为起点的有向边得到图(b)；从图(b)中 2、3 和 4 号顶点的入度都为 0，所以第二层中包含 2、3 和 4 号程序文件；删除 2、3 和 4 号顶点以及以它们为起点的边，便得到新的有向图(c)，得出第三层包含 5 和 8 号程序文件；以此类推遍历所有顶点，直到删除所有的节点得到一个空的有向图，便可以得到的层次关系图(f)，从图中可以看出，通过对程序文件依赖关系有向图的分层算法分析，最终将程序文件的依赖关系有向无回路图转化成程序文件的层次图。从图中可以看出，对于大规模应用软件系统，随着程序文件规模的增大，可并行编译的程序文件将会增多，并发性更加明显。

3.4 程序文件编译过程的并行化设计

通过对程序文件的分层结果，得到了可以并行编译的每层程序文件，然后以此为基础构建每层程序文件编译的 makefile 文件，实现系统编译过程的并行化。下面将介绍 makefile 的一些基本构造知识，详细阐述系统编译并行化的 makefile 设计。

3.4.1 makefile 介绍

make 指令是 Unix 及其它操作系统上用于项目管理[59]的一个解释 makefile 中指令的命令工具。自 20 世纪 70 年代问世以来，make 命令工具得到了极大的发展，现在大多数的 IDE 都有 make 命令工具，比如： Linux 下 GNU 的 make，Delphi 的 make，Visual C++的 nmake。makefile 是指在 Unix 及其他操作系统下运行 make 指令是需要用到文件。

由于大规模应用软件系统一般都会包含成百上千个源程序文件，编译时如果每次都要键入编译指令命令的话，将会极大地降低工作效率，但是利用 makefile 就可以轻易解决这个问题。通过将整个软件系统的编译规则写进 makefile，可以实现编译过程的自动化，甚至可以实现更复杂的功能，因为 makefile 也执行操作系统的命令，像 shell 脚本一样让计算机执行要完成的任务。一旦在 makefile 中写入所有要编译的程序文件、目标文件和编译链接的规则，那么只需要在终端输入 make 命令，就可以实现整个软件系统的完全自动编译。而且，当用 make 处理某项规则时，若软件系统源程序文件有所修改，make 会找出必要条件和工作目标中所指定的文件，更新相应规则，如果必要条件中文件的更新时间在工作目标的更新时间之后，make 便会执行命令对该目标重新进行编译和链接，对其他没有改变的程序文件则不会进行编译，这可以大大节约软件系统重新编译的时间。因此，使用 make

和 makefile 工具可以有效提高软件系统开发的效率。

完整的 makefile 文件一般包含了以下五类内容，来说明各程序文件的编译方式和目标文件的生成规则：

1、显式规则。主要用于说明如何生成目标文件，包括目标文件、依赖文件以及生成的命令。

2、隐式规则。make 的自动推导功能，支持书写者可以较为粗糙地编写 makefile 文件。

3、变量定义。在 makefile 文件中可以定义一系列变量，这些变量在执行时可以扩展到相应位置，减小 makefile 编写难度，提高编写效率。

4、文件指示。主要是实现 makefile 的有效文件，包括引用另一个 makefile、指定 makefile 中的有效部分或者定义一个多行的命令。

5、注释。makefile 中可以用“#”字符进行行注释。

简而言之，makefile 描述了整个软件系统的编译规则，其中每项规则包含三个部分：目标文件、必要条件以及执行的命令。目标文件是指每项规则的目标文件，包括中间文件、可执行文件名或者空目标。必要条件是指每项规则的依赖关系，包括生成目标文件所需要的程序文件列表。命令是指每项规则的命令行，用于创建目标文件的一系列 shell 指令。makefile 基本的结构为[60]：

```
#注释以#号开头
include make.inc
target [target…] (要生成的文件)：[component…] (被依赖的文件)
[<TAB>命令 1]
[<TAB>命令 2]
…
[<TAB>命令 n]

clean :
        rm *
#可以使用“\”表示续行，“\”之后不能有空格
```

图 3.8　Makefile 基本的结构

在 makefile 中，每一个规则所包含的每条命令行必须独自占一行，每项规则可以有多个命令行，每行命令开头必须用 TAB 键进行分隔；冒号“：”表示目标文件的依赖文件，每行定义的一个目标文件后都有表示该目标文件的依赖文件列表，如“A：B”表示“A”文件的生成需要依赖“B”文件；可以用 include 关键字把其他 makefile 包含进来，如：include <filename>，那么被包含的文件在 makefile 执行时就会被读取在当前包含位置。

在大规模应用软件系统中，常会用到相同的编译选项去编译多个程序文件。如果为每个目标文件的编译都指定编译选项的话，那么软件系统的编译工作量将

十分巨大，但是宏定义的使用可以避免这种重复性的工作量，提高系统文件的编写效率，方便编写 makefile 文件。在 makefile 中用“=”号来定义宏，宏名一般使用大写，用“$(宏名)”来使用宏，如图 3.9 中[60]，FC 和 FFLAGS 就是 make 的宏定义，而$(FC)和$(FFLAGS)表示引用该值，此外还可以用“+=”追加宏的内容。

```
. . .
FC   =   ifort   #define a macro for name of compiler
FFLAGS    =  -O3 -ip -fp-model   #define a macro for the FC flags
. . .
test.o: test.f90
      $(FC)   -c   $(FFLAGS)   test.f90
. . .
```

图 3.9　Makefile 宏例子

Makefile 文件的宏定义变量的值不是不变的，有些一些宏的预定义变量的值在执行命令的过程中会发生相应的变化，比如$*、$@、$?和$<。其他 makefile 的预定义变量如表 3.2 所示[60]。

表 3.2　Makefile 中的预定义变量

变量名	含义
$*	不包含扩展名的目标文件名称
$+	所有依赖文件，以出现的先后为序，以空格分开
$<	第一个依赖文件名称
$?	所有修改日期比目标的创建日期晚的依赖文件
$@	目标的完整名称
$%	目标的归档成员名称
AR	归档维护程序的名称，默认值为 ar
CC	C 语言编译器的名称，默认值为 cc
CFLAGS	C 语言编译器的选项
CPP	C++语言预编译器的名称，默认值为$(CC) –E
CPPFLAGS	C++语言预编译器的选项
FC	FORTRAN 语言编译器的名称，默认值 f77
FFLAGS	FORTRAN 语言编译器的选项

3.4.2 编译过程并行化的 makefile 设计

编译开始时，用“$ make”命令可以激活 make 功能，读入所有的 makefile，包括被 include 包含的其它文件，然后初始化 makefile 文件中的所有变量，通过分析所有目标文件规则，为目标文件构建依赖关系链，并以此确定目标文件的生成规则，执行生成可执行文件的最终命令。在编译 makefile 文件时，可以使用“-f”指令选择编译指定的 makefile 文件，如“$ make -f mymakefile”，若选择多个进程

编译可以在 make 指令后面加上“-j N”，其中 N 是一个整数，代表所选择的进程数。makefile 编译命令还有一些其他编译选项如表 3.3 所示[60]。

表 3.3　常用命令行选项

命令行选项	含义
-f FILE	以指定的 FILE 文件作为 makefile
-C DIR	在读取 makefile 之前改变到指定的目录 DIR
-p	显示 make 变量数据库和隐含规则
-n	只打印要执行的命令，但不执行这些命令
-I DIR	当包含其他 makefile 文件时，可利用该选项制定搜索目录
-i	忽略所有的命令执行错误
-h	显示所有的 make 选项
-s	在执行命令时不显示命令

依据软件系统的程序文件分层信息，可以建立多层 makefile 文件，使得每一层的 makefile 负责本层程序文件的编译过程，由于每一层程序文件间没有依赖关系，因此在执行每层 make 指令时，可以设置多个进程数并行执行程序文件的编译过程，提高软件系统的编译效率。以第四章中将介绍的 YHGSM 模式为例，其程序文件可分为 15 层，下面介绍其多层 makefile 的构造过程：

1、首先根据程序文件编译要求，定义程序文件的编译变量文件 make.inc，如 FC= ifort，CC = mpicc -cc=icc，CPP = /lib/cpp –P 等；

2、其次在每层 makefile 文件中使用 include 工具，将 make.inc 文件包含进来，如“include make.inc”；

3、然后将所有层程序文件的编译信息写入每层的 makefile 文件中，如“{0}:\n\t$(FC) -o $(ROOTD)/{0}.o -c $(FCFLAGS) $(ROOTD)/module/{0}.F90”，其中每个程序文件被当做一个 make 的命令写入，该命令可独立在 shell 命令行执行，整个 makefile 文件内相当于包含了所有程序文件的编译命令；

4、执行 make 功能，进行编译，如是想要进行多线程编译，可以在 make 指令后加上“-j N”指令，实现编译过程并行化。

3.5 本章小结

本章是整篇论文算法设计的核心部分，作者关于如何实现软件系统程序文件并行化编译的算法工作在这一章得到了详细地体现。本章包括了实现全文目标的三个方面的算法设计，第一个算法设计是软件系统程序文件依赖关系的算法设计，首先介绍了程序文件依赖关系的基本概念，并从依赖关系概念出发对程序文件的结构特征进行分析，证明软件系统的程序文件设计正好满足进行并行化编译的基本格式要求，同时根据程序文件的结构特征总结了软件系统中存在的依赖关系类

别，最后以数值预报系统 YHGSM 模式为例，详细介绍了程序文件依赖关系的算法设计和部分关键算法程序，并阐述了如何以最终文件的依赖关系构建了整个软件系统程序文件依赖关系有向图的设计思路；第二个是根据文件的依赖关系如何实现程序文件分层的算法设计，通过有向图存储的方式，对有向图进行分层设计，逐步得到软件系统程序文件的分层结果，保证每一层的程序文件间没有依赖关系以及后一层所有程序文件的依赖关系都已存在前层的程序文件中；第三个是如何利用分层结果实现软件系统并行化编译，介绍了编译 make 的相关知识，以及并行编译时 makefile 文件构造的基本步骤，最后通过分层的程序文件完成软件系统进行并行化编译的方案设计。

第四章　大规模软件系统的并行化编译实验

本章将分别以两种大规模应用软件系统 WRF 和 YHGSM 模式为例，进行软件系统编译过程的并行化实验，并对实验结果进行分析。所有实验结果都是在一个服务器上进行得到的，该服务器含 16 个 Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz(cache 20480 KB) 处理器核，其操作系统为 Linux version 2.6.32-220.el6.x86_64，编译语言为 Intel FORTRAN Version 11.1。

4.1 数值预报模式系统 WRF 的并行化编译实验

4.1.1 WRF 模式简介

WRF 模式是新一代的高分辨率中尺度预报模式，由美国环境预测中心和美国国家大气研究中心等多家科研机构共同合作开发出来的，目前主要应用于有限区域的天气研究和业务预报[61]。该模式系统是一个区域格点模式，采用 Fortran90 程序语言编写，水平方向采用 Arakawa C 网格点，垂直方向采用地形跟随质量坐标，集数值天气预报、大气模拟以及数据同化于一体，能够更好地改善对中尺度天气的模拟和预报[62]。

WRF 模式的程序文件编译有着一套相当复杂的编译机制，要经过模式系统的配置和编译设置才可以开始模式的编译过程。

配置：运行 configure 脚本文件“./configure”来配置模式系统的编译环境。首先定位所需库和工具的路径，如 NetCDF、HDF 和 Perl，通过路径或系统环境设置来检查是否安装。之后，configure 脚本文件调用 UNIX 的 uname 指令来辨识将要编译的系统平台类型，选择系统处理器和编译器选项：serial、smpar、dmpar、dm+sm(PGI、INTEL、GNU…)，之后需要选择嵌套类型，通过选定系统配置参数来生成顶层目录的 configure.wrf 文件，该文件可以被“./clean –a”删除或者被下一次运行 configure 脚本文件重新覆盖。等以上环境参数配置完毕，便可以进行程序文件的编译安装步骤。

编译：在用 configure 脚本配置系统环境后，便可以使用 compile 脚本文件对 WRF 程序文件进行编译。WRF 模式系统编译的动力内核是由用户的环境设定来决定的，如果环境中没有设定模式的动力内核，模式将会默认选择 ARW。首先，compile 脚本文件执行文件检查，分析变量列表，复制 Registry.core 文件，然后调用 UNIX 的 make 指令来运行顶层的 Makefile 文件。顶层的 Makefile 文件负责 WRF 模式编译的其他部分，完成 WRF 模式各个子文件下的 make 编译目标。完整的编译顺序如下所示：

1、external 目录下的 make 编译：

a) 编译目录 external/io_{grib1，grib_share，int，netcdf}下关于 Grib Edition 1，binary 和 netCDF 实现的 I/O　API 的程序文件；

b) 编译目录 RSL_LITE 下生成通信层的程序文件（仅并行时）；

c) 编译目录 external/esmf_time_f90 下生成 ESMF 时间管理库的程序文件；

d) 编译目录 external/fftpack 下生成全球滤波器的 FFT 库的程序文件；

e) 编译其他定义在 configure.wrf 文件里的 external 目录下的程序文件；

2、编译目录 tools 下的程序文件来生成读取 Registry/Registry 文件和自动生成 inc 目录下文件的目标文件；

3、编译目录 frame 下 WRF 模式框架特定的模块文件；

4、编译目录 share 下与动力内核无关的中介层程序文件，包括 WRF 模式的 I/O 模块；

5、编译目录 phys 下 WRF 模式层物理过程的程序文件；

6、编译目录 dyn_core 下的模式特定动力核内中介层和模式层的子程序；

7、编译目录 main 下 WRF 模式主程序，创建符号链接生成可执行文件。

源程序文件（主要以.F 为主，在目录 external 下部分为.F90）要预先处理成为.f90 程序文件，然后再进行上述过程的编译操作。执行编译操作时，首先编译.f90 程序文件生成.o 目标文件，然后由生成的.o 文件组成 main/libwrflib.a 库文件，而 external 目录下则是生成各自的库文件，最后链接步骤将这些目标文件生成 exe 可执行文件。WRF 模式编译过程有并行编译选项，可以通过并行执行 make 指令，但是由于受到各个模块之间复杂的依赖关系的限制，在实际编译中，使用并行编译最多可获得 2 倍的加速[63]。因此，模式编译的环境变量参数 J 默认设置为 2，编译时间约为 30 分钟。若是机器环境不支持并行编译或者并行编译环境配置有问题，可重新设置环境变量参数进行串行编译：export J=“-j 1”，此时编译时间约为 50 分钟。如果在生成最终可执行文件时出现问题，则需要运行“./clean –a”指令，清除所有已生成的目标文件，然后重新进行配置编译，所耗时间周期长。

4.1.2 WRF 模式程序文件依赖关系

WRF 模式系统中有顶层文件目录结构：WRF 软件框架文件(frame)，WRF 模式系统文件(dyn_em、phys、share)，配置选项文件(arch、Registry)，公用程序(tools)，WRF 代码的接口安装包(external)[63]，3 个脚本文件：clean、compile、configure，实验文件目录 run、实例测试文件目录 test 等。

实验以 WRF 模式的 ARW 版本为对象，共对 579 个程序文件进行了分析，其中.f90 程序文件共 513 个，62 个.c 程序文件和 4 个.f 程序文件。通过程序文件间依

赖关系的分析，可以将 WRF 模式系统的程序文件可以分成 16 层，每层的程序文件数分别为：320、47、27、52、16、34、6、3、30、17、9、5、7、3、2、1。第一层程序文件包含了第二层所有程序文件所需要的依赖文件，每一层程序文件所依赖的文件都在其前面层次的程序文件中，并且同一层程序文件之间是完全独立的，相互之间不存在依赖关系。

4.1.3 WRF 模式的并行化编译实验

通过程序文件的分层结果，构建 WRF 模式程序文件的分层编译脚本，并进行参数配置，然后在含 16 个 Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz(cache 20480 KB)处理器核的服务器上进行 WRF 模式并行化编译，所用 Fortran 语言编译器为 Intel FORTRAN Version 11.1。

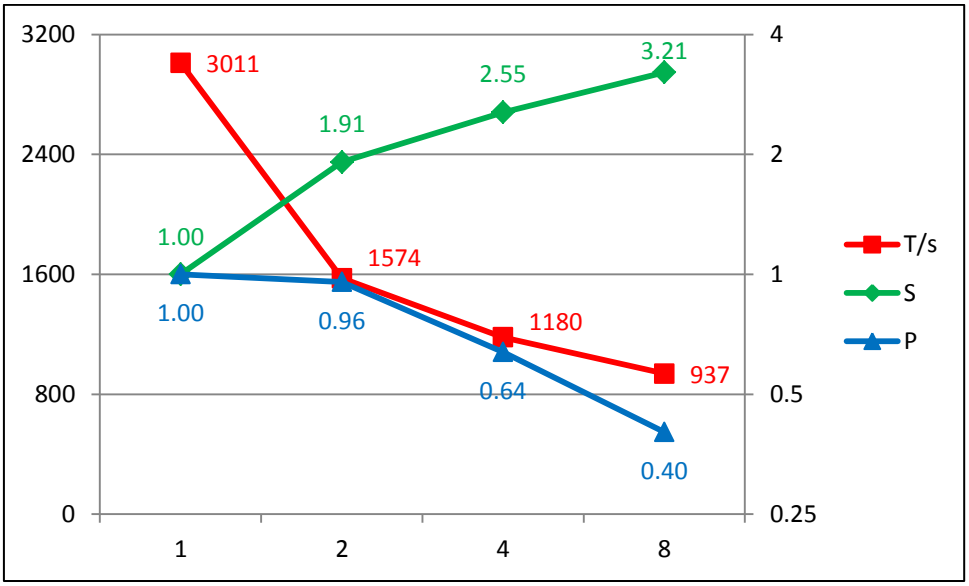

图 4.1　WRF 模式并行化编译的时间、加速比和并行效率折线图

图 4.1 是在不同进程下，WRF 模式系统并行化编译的折线图，其中 T 为编译时间，S 为加速比，P 为并行效率。图的左纵坐标是表示编译时间的坐标轴，单位为秒，右纵坐标是表示加速比、并行效率的坐标轴，为了放大并行效率的区别，右坐标轴采用对数刻度。

表 4.1　WRF 模式并行化编译的时间、加速比和并行效率表

进程数 / 项目	1	2	4	8
T/s	3011	1574	1180	937
ΔT/s	0	1437	1831	2074
S	1	1.91	2.55	3.21
P	100%	95.65%	63.79%	40.17%

表 4.1 是在不同进程下，WRF 模式系统采用本文并行化编译方式时的编译时

间、加速比和并行效率对比表，其中ΔT 表示不同进程数下并行编译时间与 WRF 模式串行编译时间差值的绝对值。从表中可以看出，WRF 模式串行编译时间为 3011 秒，当使用 2 个进程进行编译时，模式的编译时间为 1574 秒，比串行时间减少了 1437 秒，其中加速比为 1.91，并行效率为 95.65%，从图 4.1 中可以看出，2 个进程下编译时间迅速减少，并行效果十分明显。当使用 4 个进程进行编译时，编译时间降到 1180 秒，比串行时间减少了 1831 秒，加速比为 2.55，并行效率为 63.79%；当使用 8 个进程进行编译时，模式编译时间降到 937 秒，加速比为 3.21，但是并行效率只有 40.17%。编译时间的加速比随着进程数的增加而增加，说明在多进程下可以使没有依赖关系的程序文件同时进行编译，充分利用计算资源，缩短编译时间、提高编译效率；而并行处理的效率随着进程数的增加而降低，在 2 个进程时并行效率达到了 95.65%，但当使用 8 进程时，并行效率只有 40.17%。造成这种结果的原因可能是随着处理器个数的增加，由于程序文件分层较多，在不同进程下并行编译造成负载不均衡，而且由于每一层的程序文件数目不同造成层间编译的同步等待，使得程序文件的并行编译效率随着进程数降低明显。

表 4.2　WRF 模式两种编译方式的编译时间表

进程数 / 项目	1	2	4	8
T1/s	3011	1917	1654	1427
T2/s	3011	1574	1180	937
ΔT/s	0	343	474	490
T2/T1	100%	82.1%	71.3%	65.7%

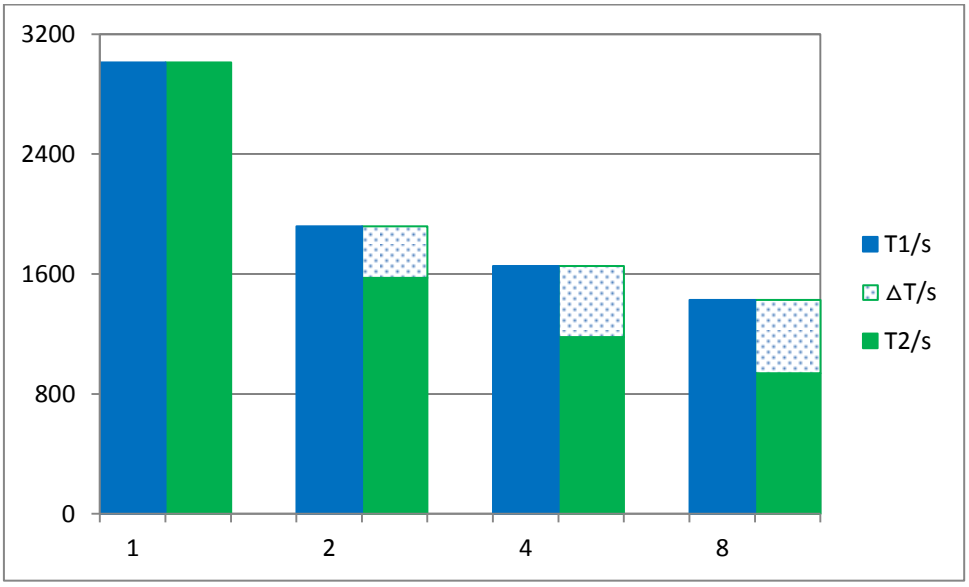

图 4.2　WRF 模式两种编译方式的编译时间柱状图

表 4.2 和图 4.2 分别是在不同进程下，WRF 模式系统现有编译方式与本文所提

并行化编译方式编译时间的对比表和柱状图，其中 T1 表示在现有编译方式下不同进程数时的编译时间，T2 表示采用本文并行化编译方式下在不同进程数时的编译时间，ΔT 表示两种编译方式的编译时间差值。从表 4.2 中可以看出，在 2 个进程下，现有编译时间比本文所提并行化编译时间多了 343 秒，在 4 个进程下，本文所提并行化编译时间减少了 474 秒，当进程数为 8 时，编译时间比现有编译时间减少了 490 秒，虽然 WRF 模式现有编译时间在多个进程下也会逐渐减小，但是没有本文所提并行化编译方式效果明显，造成这样的原因可能是因为 WRF 模式传统考虑的并行编译只考虑了纯子程序模块，这部分程序文件数目比较少，导致编译时间减少的速度随着进程数的增加而逐渐变慢。

表 4.3　WRF 模式两种编译方式的编译时间、加速比、并行效率表

进程数 / 项目	1	2	4	8
T1/s	3011	1917	1654	1427
T2/s	3011	1574	1180	937
S1	1	1.57	1.82	2.11
P1	100%	78.53%	45.51%	26.38%
S2	1	1.91	2.55	3.21
P2	100%	95.65%	63.79%	40.17%

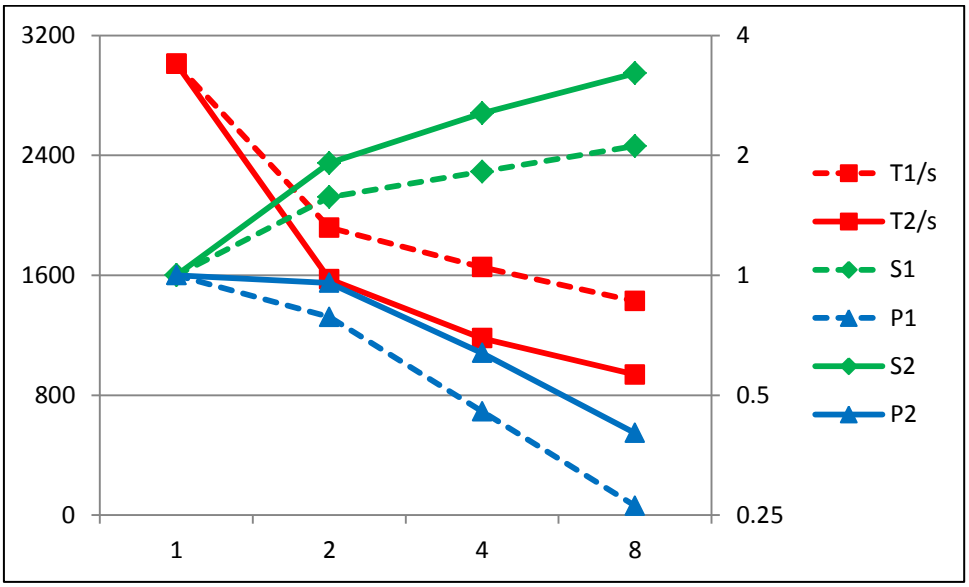

图 4.3　WRF 模式两种编译方式的编译时间、加速比、并行效率的折线图

表 4.3 和图 4.3 分别是不同进程下，WRF 模式系统现有编译方式与本文所提并行化编译方式的编译时间、加速比和并行效率的对比图表，其中图的左纵坐标是表示编译时间的坐标轴，单位是秒，右纵坐标是表示加速比、并行效率的坐标轴，为了放大并行效率的区别，右坐标轴采用对数刻度。

从图 4.3 中可以看出，随着进程数的增加，WRF 现有编译方式编译时间减少

的越来越慢，加速比和并行效率也是逐渐降低，而本文所提并行化编译方式的时间却下降程度明显，加速比和并行效率比传统方式的要好很多。从表 4.3 中得出，在 4 进程时现有编译方式的加速比只有 1.82、并行效率低至 45.51%，而此时本文所提并行化编译方式的加速比为 2.55、并行效率为 63.79%；当增至 8 进程时，现有编译方式的并行效率只有 26.38%，而本文所提并行化编译方式的并行效率为 40.17%，和现有编译方式 4 进程时的并行效率相当。

4.2 YHGSM 的并行化编译实验

4.2.1 YHGSM 模式简介

YHGSM 模式是我国自主研发的数值天气预报模式系统，用于制作全球的天气形势预报，为有限区域模式提供边界条件，同时它也是全球气象资料四维变分同化系统的重要组成部分。YHGSM 的程序设计语言主要为 FORTRAN 90，部分采用了 FORTRAN 77 和 C 语言，消息传递库为通用的 MPI，程序具有较好的可移植性。

模式系统的所有程序代码都处于目录 model 下，包括：主要程序代码根目录、变换相关代码根目录、陆面过程相关代码根目录、相关头文件所在目录、基本线性代数子程序所在目录、线性代数软件包目录、程序运行根目录、各种编译脚本目录、其它辅助目录。

YHGSM 模式进行编译时，需要先修改编译选项头文件 make.inc 的一些编译设置，然后在 model 目录下键入 mkbk/mymake，便可以执行模式的编译安装过程，串行编译时间约为 40 分钟，如果在生成最终可执行文件时出现问题，则需要运行“./clean –a”指令，清除所有已生成的目标文件，然后重新进行编译。YHGSM 模式也支持并行编译，通过修改 mymake 文件里的第二行“make –j N” 编译的环境变量 N 来实现多任务并行编译，但是模式系统暂时只考虑完全没有依赖关系的子程序文件的并行编译，这部分程序文件数目占整个模式总的程序文件比例小，因此随着进程个数的增加，并行效率会明显降低。

4.2.2 YHGSM 程序文件依赖关系

YHGSM 模式系统共有 3327 个源程序文件，其中.F90 程序文件有 1684 个，30 个.c 程序文件，1613 个.F 程序文件。Fortran 程序共 983181 行，其中 Blas 与 Lapack 共含有 415915 行，其余 567266 行。

经过对 YHGSM 模式程序文件的依赖分析和分层处理，YHGSM 模式程序文件可分为 15 层，每层程序文件分别为：1651、299、877、135、106、18、10、1、

1、161、23、18、11、5、11，其中独立子程序共有 1613 个。每一层程序文件依赖的所有程序文件都包含在其前面层次中，第一层程序文件包含了后面所有层次的程序文件所需要的依赖文件，同层间的程序文件没有依赖关系。

4.2.3 YHGSM 的并行化编译实验

图 4.4 和表 4.4 分别是在不同进程下，采用本文所提编译方式时，YHGSM 模式系统程序文件编译时间、加速比和并行效率的折线图和表格，图的左纵坐标是表示编译时间的坐标轴，单位是秒，右纵坐标是表示加速比、并行效率的坐标轴，为了放大并行效率的区别，右坐标轴采用对数刻度，T 表示并行化编译方式所用时间，ΔT 表示串行编译时间与不同进程数下并行编译时间差值的绝对值，S 表示加速比，P 表示并行效率。

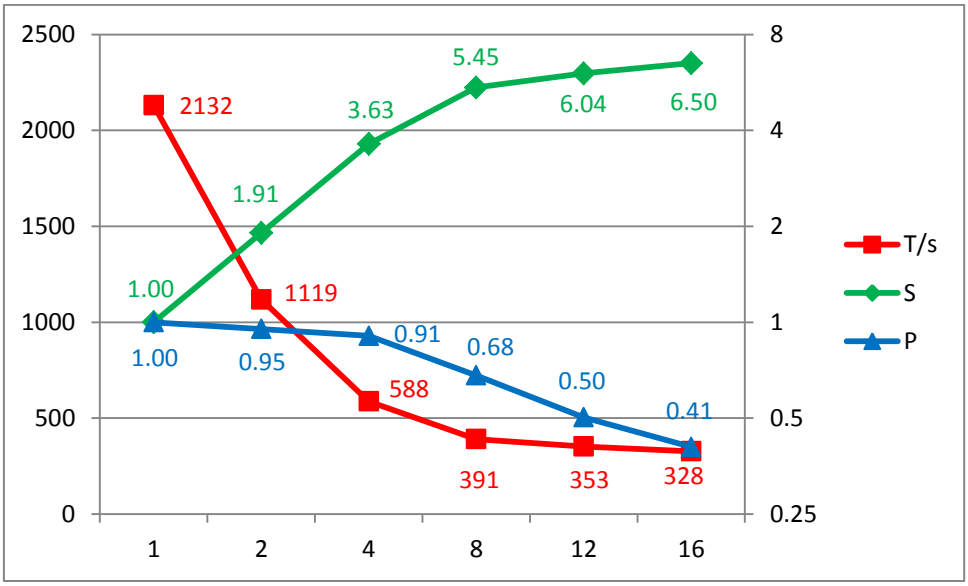

图 4.4　YHGSM 模式并行化编译的时间、加速比和并行效率的折线图

从图 4.4 中可以看出，采用本文所提编译方式时，YHGSM 编译时间随着进程数的增加而下降显著，在 8 进程时仍有较大的加速比和并行效率。YHGSM 模式的并行化编译方式具有充分利用计算资源，明显缩短编译时间，提高编译效率的效果。

表 4.4　YHGSM 模式并行化编译的时间、加速比和并行效率表

进程数 / 项目	1	2	4	8	12	16
T/s	2132	1119	588	391	353	328
ΔT/s	0	1013	1544	1741	1779	1804
S	1	1.91	3.63	5.45	6.04	6.50
P	100%	95.26%	90.65%	68.16%	50.33%	40.63%

从表 4.4 中可以看出，YHGSM 模式串行编译时间为 2132 秒，当使用 2 个进

程进行编译时，编译时间减少至为 1119 秒，比串行时间缩短了 1013 秒，其中加速比为 1.91，并行效率为 95.26%；当使用 4 个进程时，编译时间降到 588 秒，比串行时间减少了 1544 秒，加速比为 3.63，并行效率为 90.65%；当增至 8 个进程时，模式编译时间降到 391 秒，加速比为 5.45，并行效率为 68.16%，仍保持较高的加速比和并行效率；当增至 12 个进程时，编译时间降到 353 秒，加速比为 6.04，并行效率为 50.33%；当增至 16 个进程时，编译时间降到 328 秒，比 YHGSM 模式串行编译节省了约 85%的时间，加速比为 6.50，并行效率为 40.63%。

综上分析可知，YHGSM 模式并行化编译加速比随着进程数的增加而增加，同时并行处理的效率随着进程数的增加而逐渐降低，从 2 个进程时的 95.26%下降到 16 进程时的 40.63%。随着处理器的增加，在不同进程下多层次程序文件并行编译造成负载不均衡和层内程序文件编译造成的同步等待，逐渐影响程序文件的并行编译的效率。

表 4.5 和图 4.5 分别是不同进程下，YHGSM 模式系统现有编译方式与本文所提并行化编译方式编译时间的对比表和柱状图。T1 表示现有编译方式在不同进程数下的编译时间，T2 表示本文所提并行化编译方式在不同进程数下的编译时间，ΔT 表示两种编译方式的编译时间差值。

表 4.5　YHGSM 模式两种编译方式的编译时间表

进程数 / 项目	1	2	4	8	12	16
T1/s	2132	1334	790	585	582	578
T2/s	2132	1119	588	391	353	328
ΔT/s	0	215	202	194	229	250
T2/T1	100%	83.88%	74.43%	66.84%	60.65%	56.75%

从表 4.5 中可以看出，在 2 个进程下，YHGSM 模式现有编译时间比本文所提并行化编译时间多了 215 秒；在 4 个进程下，本文所提并行化编译时间减少了 202 秒；当进程数为 8 时，编译时间比现有编译时间减少了 194 秒；当进程数为 12 时，编译时间比现有编译时间减少了 229 秒；当进程数增加到 16 时，编译时间只为 328 秒，比相同进程数下现有编译时间减少了 250 秒。YHGSM 模式现有编译方式和本文所提并行化编译的编译时间在多个进程下都会逐渐减小，但是本文所提并行化编译方式的编译时间减少的速度比现有编译方式要快，特别是在增至 16 个进程时，现有编译时间比本文所提并行化编译多 250 秒，而本文所提并行化编译时间只有 328 秒，可见本文所提并行化编译方式在缩短编译时间、提高模式编译效率方面具有显著优势。

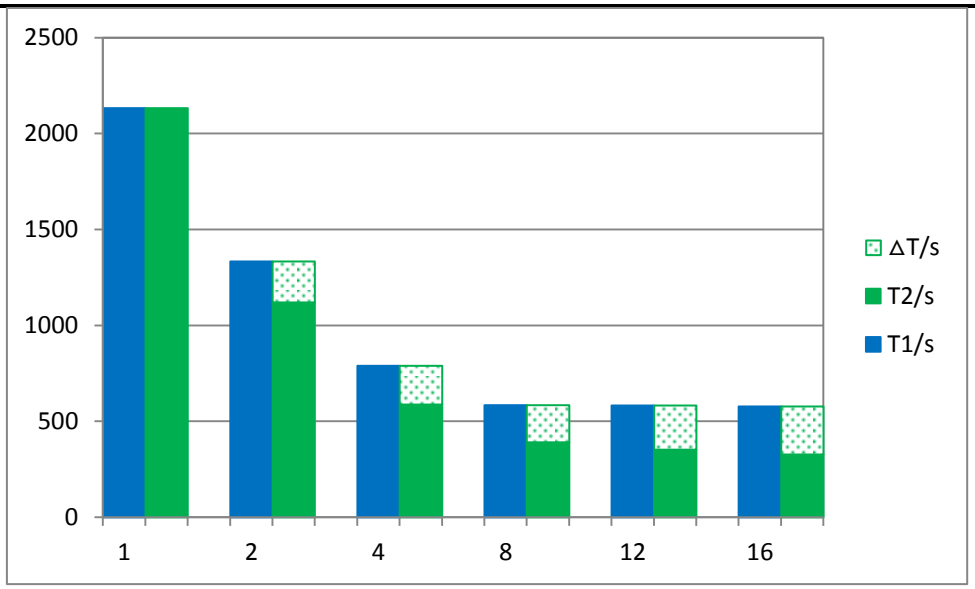

图 4.5　YHGSM 模式两种编译方式的编译时间柱状图

表 4.6 和图 4.6 分别是不同进程下，YHGSM 模式系统现有编译方式与本文所提并行化编译方式的编译时间、加速比和并行效率的对比图表，其中图的左纵坐标是表示编译时间的坐标轴，单位是秒，右纵坐标是表示加速比、并行效率的坐标轴，为了放大并行效率的区别，右坐标轴采用对数刻度，T1 表示现有编译方式在不同进程数下的编译时间，T2 表示本文所提并行化编译方式在不同进程数下的编译时间，S 表示加速比，P 表示并行效率。

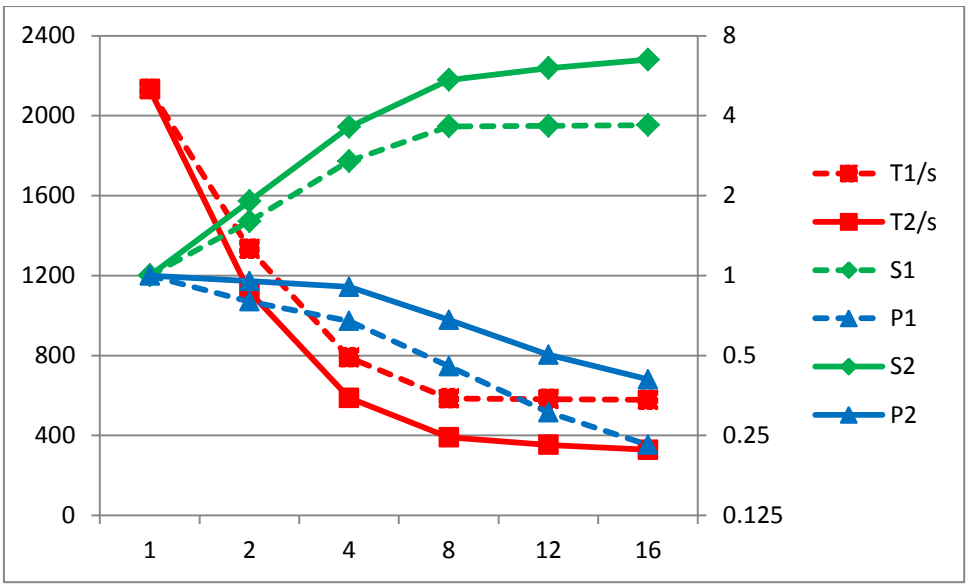

图 4.6　YHGSM 模式两种编译方式的编译时间、加速比、并行效率的折线图

从图 4.6 中可以看出，随着进程数的增加，在并行化编译方式下，YHGSM 模式编译时间下降的很明显，而模式现有编译方式编译时间减少得越来越慢，并行效率越来越低。从表 4.6 中可以看出，在 2 个进程时现有编译方式的加速比为 1.60、并行效率为 79.91%，在 4 个进程时加速比为 2.70、并行效率降到 67.47%，而此时

本文所提并行化编译方式的加速比为 3.63、并行效率为 90.65%；当增至 8 进程时，现有编译方式的并行效率只有 45.56%，而本文所提并行化编译方式的并行效率为 68.16%，比现有编译方式 4 个进程时的并行效率还高；当增至 16 进程时，现有编译方式的加速比只有 3.69，并行效率只有 23.05%，而本文所提并行化编译方式的加速比为 6.50，并行效率为 40.63%，仍保持较高的加速比和并行效率。可见，YHGSM 模式在本文所提并行化编译方式下，比现有编译方式更能发挥计算资源、提高编译效率。

表 4.6　YHGSM 模式两种编译方式的编译时间、加速比、并行效率表

进程数 / 项目	1	2	4	8	12	16
T1/s	2132	1334	790	585	582	578
T2/s	2132	1119	588	391	353	328
S1	1	1.60	2.70	3.64	3.66	3.69
P1	1	79.91%	67.47%	45.56%	30.53%	23.05%
S2	1	1.91	3.63	5.45	6.04	6.50
P2	1	95.26%	90.65%	68.16%	50.33%	40.63%

值得注意的是，随着进程数增加，YHGSM 模式并行化编译加速比渐渐呈现小幅度增加，并行处理的效率开始明显降低。说明在多进程下，层间的负载不均衡和层内的编译同步等待，越来越明显地影响模式并行编译的效率。

4.3 本章小结

本章以两种大规模应用软件系统 WRF 和 YHGSM 模式为研究对象，分别对两种模式系统程序文件进行了依赖分析和分层处理，并在此基础上进行了编译过程的并行化实验。实验结果显示，WRF 模式和 YHGSM 模式在多进程下使用并行化编译方式可以大大缩短编译时间，并行效率远高于 WRF 模式和 YHGSM 模式现有的编译方式，说明大规模应用软件系统并行化编译方法可以充分利用计算资源、提高编译效率。同时，并行化编译的并行处理效率随着进程数的增加而逐渐降低，可能是由于程序文件分层较多，导致在不同进程下并行编译造成负载不均衡，以及每一层的程序文件数目不同造成层间编译的同步等待，从而使得程序文件的并行编译效率随着进程数增加而降低。下一章将对影响并行化编译效率的可能因素展开优化实验。

第五章　软件系统程序文件并行化编译的优化设计

WRF 模式和 YHGSM 模式在多进程下进行并行化编译大大节省了编译时间、提高了编译效率，在软件系统编译过程方面表现出非常好的优化效果。但是由于分层较多导致的负载不均衡和每层程序文件数目不同造成编译同步等待，使得并行化编译方式随着进程数增加，并行效率逐渐降低。本章将以 YHGSM 模式为例，从软件系统的程序文件在层间的分配和层内的动态调度两个方面对程序文件的并行化编译方式进行优化。

5.1 程序文件在层间分配上的优化与实验

5.1.1 程序文件在层间分配上的优化算法

YHGSM 模式的所有程序文件经过程序依赖关系分析可以分为 15 层，每层的程序文件数目分别为：1651、299、877、135、106、18、10、1、1、161、23、18、11、5、11。虽然对 YHGSM 模式并行化编译取得了不错的效果，但是从模式的分层来看，每一层程序文件数目差异比较大，比如第 1 层有 1651 个程序文件，第 3 层有 877 个程序文件，而第 8 和 9 层都只有 1 个程序文件。在多进程下进行编译时，每层程序文件数目差距如此之大，造成的并行编译负载不均衡会直接影响并行处理效率，所以需要对程序文件在层间分配上进行优化，来提高并行化编译时的负载平衡性。

对层次间程序文件数目的优化，只能是调整在依赖关系有向图中无源无汇的程序文件，因为它们不被任何程序文件引用也不引用任何程序文件，可以在任何层次编译。这类程序文件在最初分层时，都被放入了第一层，那么现在可以将这些程序文件按照需求分配到其他程序文件严重偏少的层次里去，平衡每一层间所含程序文件数目的差距。从第一层程序文件中可以得到，第一层共有 1651 个程序文件，包含 1621 个 Fortran 程序文件和 30 个 C 程序文件，其中 30 个 C 程序没有引用关系，1613 个 Fortran 程序文件是独立子程序文件，在依赖关系图中无源无汇可以放在任何层次编译，剩余的 8 个 Fortran 程序文件被第二层程序文件所引用，不能调动层次。因此共有 1643 个程序文件可以用来均衡每层程序文件的数目差异，同时在优化程序文件数目时可以尽量将每层数目调整成进程数的整数倍，这样可以避免编译时进程浪费。通过 1643 个程序文件对模式层次进行优化，最后调整得到的每层程序文件数目依次为：168、300、880、168、168、168、164、164、164、164、164、164、164、164、163。

5.1.2 YHGSM 模式的并行化编译实验

通过重新调整 YHGSM 模式的层次后，然后重新生成编译脚本，再在服务器上进行 YHGSM 模式的并行化编译。图 5.1 是层间程序文件优化前后，在不同进程下 YHGSM 模式系统进行并行化编译的时间柱状图。T1 表示程序文件层间优化前的编译时间，T2 表示程序文件层间优化后的编译时间，ΔT1 表示现有编译与本文所提并行化编译优化前的时间差值，ΔT2 表示层间优化前后并行化编译时间的差值。表 5.1 是在不同进程下，YHGSM 模式系统在采用本文所提并行化编译方式时，层间程序文件优化前后编译时间的对比表。

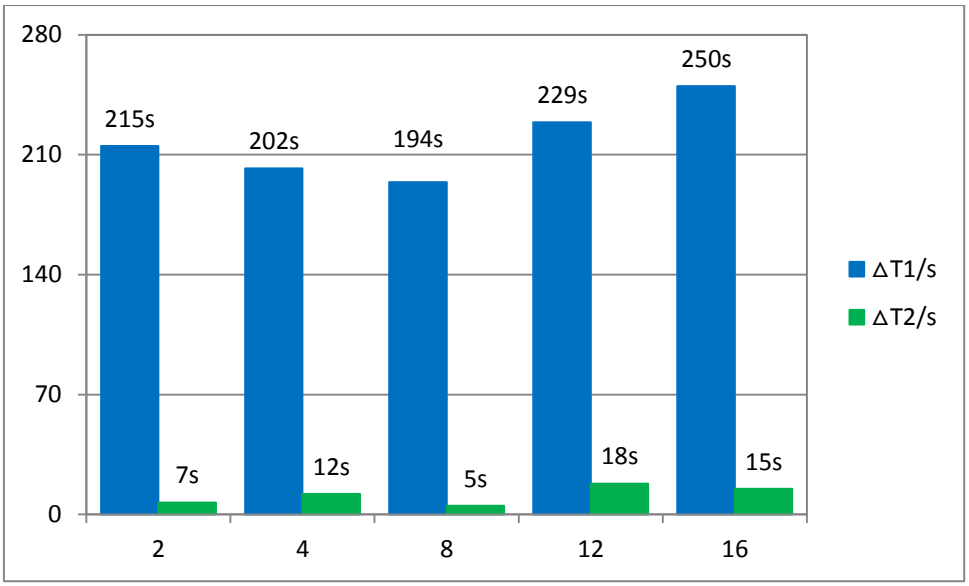

图 5.1　YHGSM 模式层间优化编译时间柱状图

表 5.1　YHGSM 模式层间优化编译时间对比表

项目 \ 进程数	1	2	4	8	12	16
T1/s	2132	1119	588	391	353	328
T2/s	2132	1112	576	386	335	313
ΔT1/s	0	215	202	194	229	250
ΔT2/s	0	7	12	5	18	15

从图 5.1 中可以看出，经过层间优化后，编译时间总体上是呈现减少的趋势，说明系统程序文件在层间的分配不平衡会对并行编译过程产生影响，从而影响并行编译效率。从表 5.1 中可以看出，在多进程下，优化层间程序文件分布起到了比较明显的作用，在 4 个进程下，层间程序文件优化后比优化前的编译时间总体优化了 12 秒，在 12 个进程下层间优化后的编译时间降到 313 秒，比优化前减少了 18 秒，说明调整层间程序文件分配的确起到了调节多进程负载平衡、优化编译时间的作用。

表 5.2　YHGSM 模式层间优化加速比、并行效率对比表

项目 \ 进程数	1	2	4	8	12	16
S1	1	1.91	3.63	5.45	6.04	6.50
P1	100%	95.26%	90.65%	68.16%	50.33%	40.63%
S2	1	1.92	3.70	5.52	6.36	6.81
P2	100%	95.86%	92.53%	69.04%	53.03%	42.57%

表 5.2 是在不同进程下，YHGSM 模式系统并行化编译方式层间优化前后加速比、并行效率的对比表，其中 S1、P1 分别是层间程序文件优化前的加速比和并行效率， S2、P2 分别是优化后的加速比和并行效率。从表中可以看出，优化后的加速比和并行处理效率都有所提高，4 个进程时加速比由 3.63 提高到了 3.70，并行效率从 90.65%提高到 92.53%；16 个进程时加速比由 6.50 提高到了 6.81，并行效率从 40.63%提高到 42.57%。总体上可以得出，在 4、8、12、16 进程下，层间优化后的编译时间加速比比优化前的提高了 0.2 左右，并行编译效率平均提高了 2%，可见层间优化对于改善软件系统并行化编译的并行效率还是很明显的。

5.2 程序文件在层内的动态调度优化与实验

5.2.1 程序文件在层内的动态调度优化算法

通过对 YHGSM 模式系统程序文件在层间分配上的调整，模式的并行编译效果有所改善，本节将在实现程序文件层间分配平衡的基础上，对同一层内的程序文件编译顺序进行调整，进一步优化软件系统的并行化编译方式。

由于程序文件编译的时间与其程序文件大小有直接的关系，一般大程序文件编译时间往往比小程序文件耗时久，所以程序文件的编译顺序也会影响编译效率。特别是对于大规模应用软件系统来说，由于它的规模大、程序文件多、各类文件大小差异明显，在多进程下编译时，不同大小的程序文件在不同的编译顺序会造成进程浪费，导致编译效率降低。比如，在 YHGSM 模式中，最大的程序文件有 10826 行，最小的程序文件只有 3 行，而且在同一层内程序文件间大小差异能达到 10817 行，由此造成的编译进程等待将会严重影响并行处理的效率，因此优化层内程序文件的编译次序对改善软件系统并行化编译的并行效率有重要意义，有利于优化多进程编译时的同步等待，从而有利于充分利用计算资源，提高编译效率。本节将对每一层内的程序文件编译顺序进行调整，让它们按照程序文件大小从大到小进行排列，实现在同层中的程序文件进行并行化编译时，先调度编译大程序文件，再调度编译小程序文件。

5.2.2 YHGSM 模式的并行化编译实验

图 5.3 是不同进程下，YHGSM 模式系统并行化编译层内优化前后编译时间的柱状图，其中ΔT1 表示现有编译与并行化编译优化前的时间差值，ΔT2 表示层内优化前后并行化编译时间的差值。表 5.3 是不同进程下，YHGSM 模式系统并行化编译层内优化前后编译时间的对比表，其中 T1 表示层内优化前的编译时间，T2 表示层内优化后的编译时间。

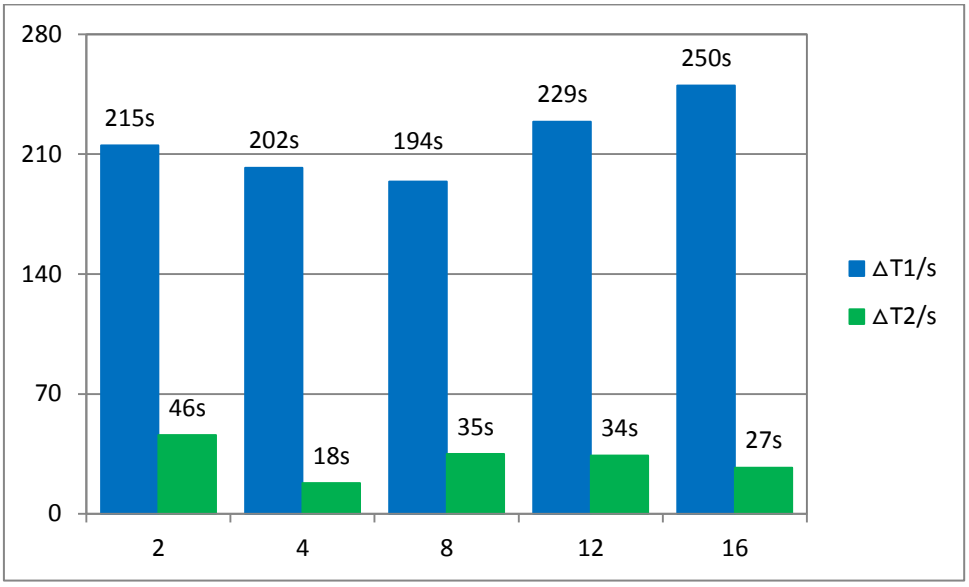

图 5.2　YHGSM 模式层内优化编译时间柱状图

表 5.3　YHGSM 模式层内优化的编译时间对比表

项目 \ 进程数	1	2	4	8	12	16
T1/s	2132	1119	599	391	353	328
T2/s	2132	1073	570	356	319	301
ΔT1/s	0	215	202	194	229	250
ΔT2/s	0	46	18	35	34	27

从图 5.2 中可以看出，经过层内优化后，编译时间总体上是呈现减少的趋势，虽然减少程度没有现有编译与本文所提并行化编译优化前的差值明显，但是符合在不同进程下编译时，程序文件在并行编译中消耗在进程等待上的时间量级。从表 5.3 中可以看出，在多进程下，优化层内程序文件编译顺序起到了缩短编译时间的作用，在 2 个进程下，编译时间总体优化了 46 秒；在 8 个进程下编译时间优化了 35 秒；当进程增加至 16 时，编译时间降为 301，比优化前并行化编译时间缩短了 27 秒，说明多进程下程序文件的大小差异对并行编译效果存在一定的影响，调整程序文件编译顺序可以很好地观察到这一点。

表 5.4 是不同进程下，YHGSM 模式系统层间优化前后的加速比、并行效率对比表，其中 S1、P1 分别为程序文件层间优化前的加速比和并行效率， S2、P2 为优化后的加速比和并行效率。从表中可以看出，优化后的加速比和并行处理效率

都有所提高，8 个进程时加速比由 5.78 提高到了 6.22，并行效率从 68.16%提高到 74.86%；16 个进程时加速比由 6.50 提高到了 7.08，并行效率从 40. 63%提高到 44.27%。在不同进程下，层内优化后的编译时间加速比比优化前有 0.1-0.5 的不同程度提高，并行效率平均提高了 4%，优化层内程序文件编译顺序对改善软件系统并行化编译效果、提高编译效率作用明显。

表 5.4　YHGSM 模式层内优化加速比、并行效率对比表

进程数 / 项目	1	2	4	8	12	16
S1	1	1.91	3.63	5.45	6.04	6.50
P1	100%	95.26%	90.65%	68.16%	50.33%	40.63%
S2	1	1.99	3.74	5.99	6.68	7.08
P2	100%	99.35%	93.51%	74.86%	55.69%	44.27%

5.3 本章小结

本章以大规模应用软件系统 YHGSM 模式为例，在第四章的基础上对可能影响并行化编译效率的因素展开优化实验，主要是通过对软件系统程序文件在层间分配和层内的动态调度上进行了调整设计，优化软件系统的并行化编译方式。结果显示，通过对程序文件在层间分配的调整，合理调动纯子程序文件，优化每层程序文件数目可以很好降低不同进程下并行编译造成的负载不平衡，对缩短编译时间有一定的促进作用；根据层内程序文件的大小，重新组织层内程序文件的编译过程，可以优化程序文件在层内的动态调度，降低并行编译时的同步等待，充分利用计算资源，提高编译效率。

结 束 语

目前，软件系统的编译方法多采用类似于按照串行路线进行一系列程序文件的编译，这种方法极为耗时，易造成计算资源浪费。尤其是在大规模应用软件系统中，由于大型应用软件系统规模大、程序文件数量多，导致软件系统的编译过程极为耗时，严重影响软件系统的工作效率。而且，国内外很少有关于实现大规模应用软件系统编译过程的并行设计，虽然有些软件系统考虑了程序文件编译的并行化，但是仅仅针对于没有任何依赖关系的纯子程序文件，例如数值预报系统WRF 模式和国内的 YHGSM 模式。但是在软件系统中，完全没有依赖关系的的纯子程序文件数目只占整体程序文件数的一小部分，所以随着并行进程数的增加，这种并行编译方式的并行处理效率会逐渐降低，不能充分发挥多核并行的优势，造成计算资源的浪费。

本文突破现有仅从纯子程序方面考虑并行编译的编译方式，尝试从图论的角度，通过分析软件系统所有程序文件间的依赖关系，利用有向图知识对软件系统的程序文件依赖关系进行分层，实现整个软件系统编译过程的并行化。因为大规模软件系统设计多采用模块化程序设计思路，将软件系统设计成由许多程序功能模块组成，不同模块负责实现不同的功能，这种程序设计结构具备了实现系统编译过程并行化的基础。因为软件系统的每个程序文件并不一定用到所有定义的模块，因此很多程序文件之间是相互独立的，这些程序文件的编译过程是可以轻易实现并行化的；而对于有依赖关系的程序文件，可以通过分析程序文件间的依赖关系，将每个程序文件看成一个节点，程序文件编译时两两之间的依赖关系用有向边来表示，从而构建这些程序文件的依赖关系有向无回路图，通过对图进行宽度搜索形成的层次关系将这些程序文件分成多个层，每层内的程序文件不存在相互之间的依赖关系，后面层内程序文件所有的依赖文件都在前层内，从而可以将第一层中的所有程序文件同时进行编译，等前一层编译结束则开始下一层中所有程序文件的编译，依次进行至最后一层，就可以完成整个模式系统的编译。

文章中为实现系统编译过程的并行化，主要工作内容有：(1)对软件系统的程序文件结构进行分析，提出分析程序文件依赖关系的算法，构造依赖关系有向图；(2)根据程序文件的依赖关系有向图，利用有向图存储的方式实现图的层次构造，生成程序文件可以并行编译的分层；(3)生成系统程序文件的分层编译脚本，并对脚本进行参数配置，实现系统编译过程的并行化；(4)对软件系统程序文件分层的情况进行层间分配和层内动态调度上的优化，探索可能影响并行编译效率的因子，进一步优化软件系统的编译过程。

在论文的第四章以两种大规模应用软件系统 WRF 和 YHGSM 模式为研究对象，

分别对两种模式系统进行了依赖分析和分层处理，并在此基础上进行了编译过程的并行化实验，结果显示 WRF 模式和 YHGSM 模式在多进程下使用并行化编译方式可以大大缩短编译时间，并行效率远高于 WRF 模式和 YHGSM 模式的现有编译方式，说明大规模应用软件系统并行化编译方法可以充分利用计算资源、提高编译效率。第五章以大规模应用软件系统 YHGSM 模式为例，在第四章的基础上，对软件系统程序文件在层间分配和层内的动态调度上进行了调整设计，对软件系统并行化编译方式进行了优化。结果显示，通过对程序文件在层间分配的调整，合理调动纯子程序文件，优化每层程序文件数目可以很好降低不同进程下并行编译造成的负载不平衡，对调节每层文件编译平衡、缩短编译时间有一定的好处；根据层内程序文件的大小，重新组织层内程序文件的编译过程，可以优化程序文件在层内的动态调度，降低并行编译时的同步等待，充分利用计算资源。

大规模应用软件系统的编译过程对提高软件开发维护的工作效率有着重要意义。本文通过对大规模应用软件系统程序文件设计的分析，提出基于依赖关系实现软件系统编译过程并行化的方法，并以数值预报系统 WRF 和 YHGSM 模式为例，取得了不错的效果。但是，在软件系统并行化编译的实验中，并行处理效率随着进程数的增加而降低，虽然通过优化程序文件在层间的分配和层内的动态调度，取得了一定的优化效果，但是效率随进程数增加而降低的现状不乐观。以后可以从如何实现动态调度编译任务和怎样减少程序文件的分层等方面，进一步探索优化软件系统并行编译的方法，提高软件系统的编译效率。

致　谢

转眼间两年半的硕士研究生学习阶段已经接近尾声了，开学之初的情节仍历历在目，如今已到了硕士毕业的时刻。回顾两年半科大生活的点滴，每一片段都历历在目，难以忘怀。在论文即将完成之际，心中满怀感恩。谨此感谢我的导师、亲人和朋友在我研究生学习的时期给予我鼓励和帮助。

首先特别感谢的是我的导师吴建平研究员！在两年半的研究生学习期间，吴老师在我的学习和生活等各个方面给了我无微不至的关心和莫大的鼓励，是吴老师细致耐心的鼓励和开导帮助我度过了研究生期间最沮丧的日子。吴老师严谨的治学态度、宽厚正直的为人和乐观豁达的处事方式深深地影响了我，从吴老师身上，我学习到了如何准确发现问题，如何切中重点解决问题，同时在生活中做到如何乐观看待很多暂时解决不了的问题，我想这些都会让我终身受益，能当吴老师的研究生真的是一件很幸运的事！

感谢王舒畅老师、张卫民老师、张理论老师、任开军老师、曹小群老师、赵延来老师、冷洪泽老师、银福康老师、刘柏年老师在硕士研究生期间给予的帮助、支持和关怀。

感谢实验室皇群博、朱孟斌、方民权、刘航、程海林、段博恒、吴应昂、孙敬哲、邢威师兄和余意、张泽、陈妍师姐，谢谢各位师兄师姐师姐在生活上给予的帮助和学习经验。感谢实验室同届的林士伟、邢德、赵盼盼和朱祥茹同学，大家一同走过研究生阶段的美好时光。感谢实验室孙亮、李松、李琰、王少可师弟们给实验室带来的欢乐。

在此我要特别感谢孙迪夫、赵盼盼、王贺、史浩、甘少多、时洋、赵云祥、邓皓文、徐叶茂在我平时生活中的关心和照顾，能在科大遇到这样一群有情有义、可爱交心的朋友真的很幸运，祝他们在以后的人生道路上一帆风顺！

最后，感谢我的父母和家人，在我成长的每个过程中凝聚着父母无私伟大的爱，他们给予了我精神上的巨大支持。特别是章子乐，她面对困难的坚韧、智慧和积极向上的心态给了我巨大鼓励！

参考文献

[1] Bell C, Dan B, Cote Y, et al. An Evaluation of Current High-Performance Networks[J]. Office of Scientific & Technical Information Technical Reports, 2003:28a.

[2] The Center for Programming Models for Scalable Parallel Computing[J]. Annual Report, 2008.

[3] Bikshandi G, Guo J, Hoeflinger D, et al. Programming for parallelism and locality with hierarchically tiled arrays[C]// ACM Sigplan Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, Usa, March. 2006:48-57.

[4] Manojkumar K, Jarek N. Memory Efficient Parallel Matrix Multiplication Operation for Irregular Problems[C]. Proceedings of the 3rd conference on Computing frontiers table of contents. 2006.

[5] D'Alberto P, Nicolau A. Adaptive Strassen's matrix multiplication[C]// International Conference on Supercomputing, ICS 2007, Seattle, Washington, Usa, June. 2007:284-292.

[6] Agerwala T, Martin J L, Mirza J H, et al. SP2 System Architecture. IBM Systems Journal, 1995, 34(2)：152-184

[7] Amza C, Cox A L, Dwarkadas S, et al. TreadMarks: Shared memory computing on networks of workstations[J]. Computer, 1996, 29(2):18-28.

[8] Blume W, Eigenmann R. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs[J]. IEEE Transactions on Parallel & Distributed Systems, 1992, 3(6):643-656.

[9] Blume W, Eigenmann R, Hoeflinger J, et al. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing[J]. IEEE Parallel & Distributed Technology Systems & Applications, 1994, 2(3):37-47.

[10] Blume W, Doallo R, Eigenmann R, et al. Parallel Programming with Polaris[J]. Computer, 1996, 29(12):78-82.

[11] 多核系列教材编写组.多核程序设计[M].清华火学出版社,2007.

[12] S. Borkar, Thousand Core Chips: A Technology Perspective[J]. 2007, 746-749.

[13] 武汉大学.多核架构与编程设计 [M]. 武汉大学出版社, 2010.

[14] Gidenstam A, Papatriantafilou M, Tsigas P. Allocating Memory in a Lock-free Manner[C]. Computing Science, Chalmers University of Technology, Tech. Rep. 2004.

[15] Cheadle A M, et al. Visualising Dynamic Memory Allocators[C]. Proceedings

of the 5th international symposium on Memory management, Ottawa, Ontario, Canada, ACM Special Interest Group on Programming Languages, 2006.

[16] Bjarne S. Thread-specific Heaps for Multi-threaded Programs [J]. ACM SIGPLAN No-tices, January 2001, 36(1).

[17] Danilo A, et al. SLA Based Resource Allocation Policies in Autonomic Environments[J]. Journal of Parallel and Distributed Computing, 2007, 67(3).

[18] Berger E, Zorn B, McKinley K. Reconsidering Custom Memory Allocation[C]. In Proc. Of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Seattle, WA, 2002:1-12.

[19] James R. Intel Threading Building Blocks[C]. O'Reilly, July 2007.

[20] Jeffrey S V. External Memory Algorithms and Data Structures: Dealing with Massive Data[J]. ACM Computing Surveys, June 2001, 33(2):209-271.

[21] Navarro J, et al. Practical, Transparent Operating System Support for Superpages[C]. In Proc. Of the 5th Symposium on Operating Systems Design and Implementation, Boston, MA, December 2002:89-104.

[22] Shameem Akhter, Jason Roberts. Multi-Core Programming: Increasing Performance through Software Multi-threading[M]. Intel Corporation, 2004.

[23] Wen-mei Hwu, et. al. Implicitly Parallel Programming Models for Thousand-Core Microprocessors, Design Automation Conference, Jun. 2006, 754-759.

[24] 李劲华.编译原理与技术[M].浙江大学出版社,2004.

[25] 陈火旺.程序设计语言编译原理[M].北京国防工业出版社,2000.

[26] Rastislav R ́oka. The design of a PLC modem and its implementation using FPGA circuits. Journal of electrical engineering electrical engineering.2002. 60(1).43-47

[27] Ashby S, Eulisse G, Schmid S, and Tuura LA. Parallel Compilation of CMS Software. Computing in High Energy Physics and Nuclear Physics, 2004, 2:590–593.

[28] Wolfe, Michael E. Improving Locality and Parallelism in Nested Loops. Ph.D. dissertation, Tech. Rept. CSL•TR-92-538, Comp, Systems Lab. Stanford Univ. Stanford, CA, Aug 1992.

[29] 沈志宇.并行编译方法[M].国防工业出版社,2000.

[30] Wolfe M J C. High performance compilers for parallel computing[J]. 1996.

[31] 龚雪容, 生拥宏,沈亚楠.串行程序并行化中计算代码与同步通讯代码的自动生成[J].计算机应用与软件, 2008, 25(1):91-92.

[32] Lo, Jack L and Susan Eggers. Improving Balanced Scheduling with Compiler' Optimizations that Increase Instruction-Level Parallelism. PLD195, 1995,

p151-162.

[33] P. Petersen and D. Padua Machine-independent evaluation of parallelizing compilers. Advanced Compilation Techniques for Novel Architectures, Jan. 1992.

[34] G. Lee, Clyde P. Kruskal, and D. J. Kuck. An empirical study of autoqatic restructuring of nonnumerical programs for parallel processors. Special Issue Parallel Processing of IEEE Trans. Computers. 1985: C-34(927-933).

[35] R. Cytron, D. J. Kuck, and A. V. Veidenbaum. The effect of restructuring compilers on program performance for high-speed computers. Special Issue of Computer Physics Communications devoted to the Proc. Conk on Vector and Parallel Processors in Computational Sci.. 1985: 37(39-48).

[36] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. IEEE Trans. Parallel Distributed Syst. 1998: 3(300-315).

[37] J.P. Singh and J.L. Hennessy. An empirical investigation of the effectiveness and limitations of automatic parallelization. Proc. Int. Symp. Shared Memory Multiprocessing, Tokyo. 1991.

[38] Brent R P. The Parallel Evaluation of General Arithmetic Expressions. [J]. ACM, 1974, 21(2):201-208.

[39] Mellorcrummey J. Center for Programming Models for Scalable Parallel Computing[J]. 2008.

[40] 邓倩妮译. Paheco, P. S. 并行程序设计导论. 机械工业出版社, 2012.8.

[41] Sun X H, Rover D T. Scalability of Parallel Algorithm-machine Combinations[J]. IEEE Trans. on Parallel and Distributed Systems, 1994, 5(6) : 599～613.

[42] Zhang X D, Yan Y, He K Q. Latency mtric: An Experimental Method for Measuring and Evaluating Parallel Program and Architecture Scalability. [J]. Parallel and Distributed Computing, 1994, 22: 392～410.

[43] MDuvall P, Matyas S, and Glover A. Continuous Integration: Improving Software Quality and Reducing Risk. USA: Addison-Wesley Professional, 2007. 23–44.

[44] 李保前.面向持续集成的并行和分布式构建方法研究[D]. 复旦大学,2012.

[45] 王朝瑞.图论[M].北京理工大学出版社,2004.

[46] 李明哲.图论及其算法[M].机械工业出版社,2010.

[47] 耿素云.离散数学.第 2 分册,集合论与图论[M].北京大学出版社,1998.

[48] S.PlimPton, B.Hendrickson, S.Bums, W.McLendon, Parallel algorithms of radiation transport On unstructured grids, in proceeding of supercomputing 2000.

[49] Banejee U. Dependence Analysis for Supercomputing. Norwell, Massachusetts: Kluwer Academic Publishers, 1988.

[50] Banerjee U. Data dependence in ordinary programs[D]. Department of Computer Science, University of Illinois at Urbana-Champaign, 1976.

[51] Kuck D L. Structure of Computers and Computations[M]. John Wiley & Sons, Inc., 1978.

[52] Banerjee U. Dependence analysis[M]. Springer Science & Business Media, 1997.

[53] Kuhn R H. Optimization and interconnection complexity for: parallel processors, single-stage networks, and decision trees[J]. 1980.

[54] Krste Asanovic, Rastislav Bodik, James Demmel and et al. A view of the parallel computing landscape [J]. Communications of the ACM - A View of Parallel Computing, 2009, 52(10): 56-67.

[55] 白云. FORTRAN95 程序设计[M]. 北京, 清华大学出版社, 2011.

[56] Zahn C T. Graph-theoretical methods for detecting and describing gestalt clusta-s [J].Computers, IEEE Transactions on, 1971,100(1): 68-86.

[57] 田沙沙. 基于图论的图像分割算法研究[D]. 重庆大学, 2011.

[58] 孙大群, 严义, 邬惠峰. 梯形图数据依赖关系分析与并行提取[J]. 计算机工程, 2014, 40(2): 67-70.

[59] Robert Mecklenburg 著, O'Reilly TaiWan 公司译, GNU Make 项目管理. 南京:东南大学出版社, 2006. 11-51.

[60] 徐海兵.GNU make 中文手册.2004-09-11.

[61] 邓莲堂. 新一代中尺度天气预报模式——WRF 模式简介[C].中国气象学会 2003 年年会"城市气象与科技奥运"分会论文集.2003.

[62] Skamarock W C, Klemp J, et al. A Description of the Advanced Research WRF Version 3 [Z]. Ncar Technical Note, NCAR/TN-475+STR, 2008.

[63] Weather research and forecast ARW version 3 modeling systems user's guide[J]. Colorado: NCAR, 2010 (2012-094) 1 1.

作者在学期间取得的学术成果

[1] 邢翔,吴建平,张镭.利用 WRF 模拟 SOCAL 春季大气边界层结构[C].第 32 届中国气象学会年会,2015.

[2] 邢翔,吴建平.大规模应用软件系统编译过程的并行化算法设计[C].湖南省第九届研究生创新论坛,2016.