

内容简介

本书是一本学习 OpenMP 编译原理和实现技术的入门级教材。内容分成三篇，第一篇是并行计算及 OpenMP 编程的基础内容，第二篇是 OpenMP 编译及其运行环境，第三篇是实践内容。在第二篇中，以一般编译器常见结构为主线，通过结合详细的 OMPI 源代码分析向读者介绍 OpenMP 编译器的工作原理及其实现技术，其中包括词法分析、语法分析、AST 树的结构、AST 树的生成及相关操作、OpenMP 编译制导指令的代码变换、OpenMP 线程与 OS 线程库的接口、运行环境等细节。OpenMP 编译制导指令的变换是 OpenMP 编译的核心内容，需要将 OpenMP 制导指令的语义功能利用操作系统的线程库来实现，分成并行域管理问题、任务分担和同步问题、变量数据环境问题三个核心内容。第三篇的四章给出了常见编译器、性能测试工具以及 OMPI 源代码的框架分析。

本书是国内第一本关于 OpenMP 编译器工作原理和实现细节的尝试。读者对象是研究 OpenMP 编译技术的研究人员和高校师生，作为入门的初步阅读材料，也可以作为研究生和高年级本科生学习并行语言编译技术相关课程的辅助参考书。由于作者才疏学浅，书中难免不少错漏，欢迎读者指正。联系邮件：lqm@szu.edu.cn。

前言

编写《OpenMP 编译原理及实现技术》教材是深圳大学“计算机科学与技术国家特色专业建设点”的建设内容之一。该教材和相应课程的设计目的有三点：衔接本科《编译原理》课程、扩展 OpenMP 并行语言编译的知识、增强学生的动手实践和编程能力，书中以 OpenMP 的一个开源编译器——OMPi 作为分析对象，做到理论与实践紧密结合，为进一步学习和研究打下必要的基础。

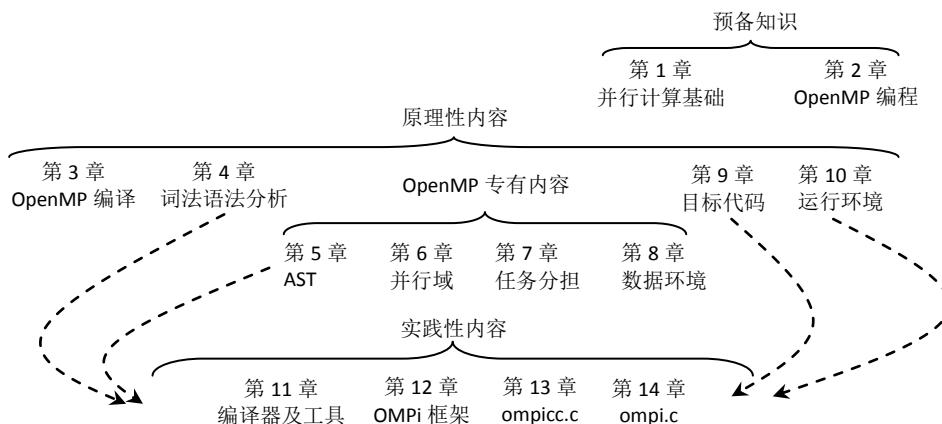
本书读者虽然不要求对编译原理有深入理解，但是还是需要先对编译有初步认识、具备基本概念。对于关心实现技术的读者，建议下载 OMPI 1.0.0 的源码并进行同步阅读。

书中第一篇基础部分共两章，分别讲述并行平台和 OpenMP C 编程作为预备知识。对于没有接触过并行计算技术的读者可以作为一个补充阅读，第 2 章的 OpenMP 编程基本上将相关的语言要素作了全面的介绍，没有特殊需要一般不再需要去阅读 OpenMP 的标准。

第二篇编译部分共八章，主要讲述 OpenMP 的编译器以及运行环境。第 3 章介绍 OpenMP 编译器基本框架。第 4 章介绍词法分析和语法分析，主要以 Lex 和 Yacc 工具实现 OpenMP 词法和语法分析为主要内容。第 5 章介绍 OpenMP 编译中使用的 AST 中间表示。第 6 章、第 7 章和第 8 章介绍 OpenMP 编译中的并行域管理、任务分担和同步、变量数据环境三大问题，这是 OpenMP 编译的核心所在。第 9 章讲述目标代码生成，主要是如何利用“框架”来实现 OpenMP 翻译的技术。第 10 章以 OMPI 运行环境为例讲述相应的运行环境，。

第三篇实践部共有四章内容。第 11 章给出了几种常见的 OpenMP 编译器以及性能测试工具，用于读者测试自己设计的或修改的编译器性能。第 12 章、13 章和第 14 章是关于 OMPI 编译器工作流程和框架的内容，并有两个主要的上层源文件的代码分析，如果希望在 OMPI 的基础之上进行增强改进，那么这 3 章的内容将有所帮助。

对学习或课程的安排可以分成不同流程，比如根据教学偏向于原理性还是实践性、是否具有并行计算基础等情况，选取书中部分章节作为参考材料，书中各章倾向性情况安排如下：



虽然书中有大量的代码，但是读者在第一遍阅读时可以只作粗略浏览，第二遍阅读时再仔细阅读代码。由于源代码阅读往往需要参考不同章节的内容，因此书中有大量的交叉引用说明。

如果对 OpenMP 编译有一定了解，并希望掌握 OMPI 的具体编码实现技术，可以从第 12 章开始阅读，将整体框架看完，再根据需要返回第二篇阅读编译细节。

限于作者的学识水平，书中难免有不少错误和不足，恳请读者批评指正。作者在编写本书时深感“抛砖引玉”一词不仅仅是场面上的客套话，更是作者的内心体会和期望。

致谢

本书得以顺利完稿，是许多人的共同努力！

在此首先要感谢陈国良院士的支持和帮助。陈院士于 2009 年到深圳大学主持计算机与软件学院的教学科研的全面工作，作者作为其高性能团队成员有幸参与了众多高性能计算的科研工作，包括参与全国产万亿次个人高性能计算机 KD-60 研制、作为核心人员研制了 SD-1 PHPC 以及正在进行基于龙芯处理器的 SD-30 十万亿次全国产化高性能计算机的研制等等。在这些工作中，作者萌发了编写 OpenMP 编译方面的书籍，陈院士表示认可和支持并给出了非常有价值的意见和建议，这就是本书得以编写并顺利完稿的最初原因。其次需要感谢明仲教授，在本书的最初构思、规划和资料整理的初期，明仲教授不仅阅读了初步的稿件材料并给出了大量的指导和修改意见，对后期稿件也提出了许多建设性意见。

刘刚老师编写了第一章的部分内容和第二章的大部分内容，毛睿老师参与了多个章节的编写和订正工作，陆克中老师阅读了初期的稿件并给出了有益的意见。正因为有了这几位老师的贡献，使本书的编写质量和水平得以提高。另有两位研究生参与了本书的编写，孔畅同学完成了第 11 章的所有实验并编写了该章内容，刘成健同学是该书的第一位真正意义上的读者，并参与了稿件查错订正工作。

对于深圳大学高性能计算团队中给本书编写工作提供了各种帮助的老师和同学，不能一一尽数，在此一并表示衷心的感谢！

目录

第一篇 基础.....	1
第 1 章 并行计算基础.....	2
1.1 基本概念.....	2
1.2 并行计算平台.....	3
1.2.1 典型结构.....	3
1.2.2 SMP.....	5
1.2.3 NUMA	7
1.2.4 GPU.....	9
1.2.5 Cluster.....	10
1.3 并行程序设计技术.....	13
1.3.1 并行程序设计	14
1.3.2 OpenMP.....	16
1.3.3 MPI.....	16
1.3.4 CUDA.....	17
1.3.5 HPF.....	18
1.4 小结	18
第 2 章 OpenMP 编程基础.....	19
2.1 OpenMP 基本概念.....	19
2.1.1 执行模式	19
2.1.2 OpenMP 编程要素.....	20
2.2 OpenMP 编程	22
2.2.1 并行域管理.....	23
2.2.2 任务分担	24
2.2.3 同步	32
2.2.4 数据环境控制	38
2.3 小结	44
第二篇 OpenMP 编译.....	46
第 3 章 OpenMP 编译.....	47
3.1 OpenMP 编译系统	47
3.1.1 编译系统	47
3.1.2 目标语言	48
3.2 OpenMP 编译器结构	50
3.2.1 功能模块	50
3.2.2 工作流程	54

3.3 编译优化.....	54
3.4 小结	54
第 4 章 词法与语法分析	56
4.1 Lex 工具	56
4.1.1 Lex 的正则表达式.....	57
4.1.2 Lex 使用方法.....	59
4.2 OpenMP/C 的词法分析.....	60
4.2.1 C 语言单词.....	61
4.2.2 OpenMP 单词.....	61
4.2.3 OpenMP 与 C 语言公用单词.....	62
4.3 scanner.l	62
4.3.1 全局声明段.....	63
4.3.2 模式匹配规则段	64
4.3.3 补充函数段.....	69
4.3.4 scanner.c.....	72
4.3.5 scanner.h	72
4.4 Yacc 工具.....	73
4.4.1 Yacc.....	73
4.4.2 Yacc 文件实例	75
4.5 OpenMP/C 语法分析	78
4.6 小结	81
第 5 章 AST 的创建	82
5.1 中间表示.....	82
5.1.1 两种中间表示形式	82
5.1.2 中间表示的选择	83
5.2 AST 节点数据结构.....	84
5.2.1 语句节点	84
5.2.2 类型说明节点	87
5.2.3 声明节点	88
5.2.4 表达式节点	89
5.2.5 OpenMP 制导节点	90
5.3 AST 节点维护函数.....	95
5.4 AST 的创建.....	96
5.4.1 语法制导翻译	96
5.4.2 例 1 OpenMP 的 for 节点	99
5.4.3 例 2 C 语言 while 语句	100
5.4.4 Helloworld.c 的 AST.....	101
5.5 符号表	103

5.5.1	字符串表	104
5.5.2	符号表	105
5.5.3	符号表操作	106
5.5.4	作用域管理	107
5.6	小结	107
第 6 章 并行域管理		108
6.1	并行域及其嵌套	108
6.2	并行域管理	110
6.2.1	线程无关接口	110
6.2.2	线程的供给	111
6.2.3	线程层次关系	111
6.2.4	并行域代码封装与标识	113
6.2.5	任务分担问题	113
6.3	目标代码形式	114
6.4	OMPi 的并行域管理	115
6.4.1	ORT 统一界面	115
6.4.2	并行域代码变换	117
6.4.3	线程管理与控制	118
6.4.4	总览	122
6.5	小结	123
第 7 章 任务分担与线程同步		125
7.1	for 制导指令	125
7.1.1	for 任务分担	125
7.1.2	循环变量分解原则	126
7.1.3	目标代码功能	127
7.1.4	目标代码形式	128
7.1.5	OMPi 的 for 制导指令	129
7.2	sections 制导指令	135
7.2.1	sections 任务分担描述	135
7.2.2	section 划分原则	136
7.2.3	目标代码功能	136
7.2.4	目标代码形式	136
7.2.5	OMPi 的 sections 制导指令	138
7.3	single 制导指令	139
7.4	nowait 问题	140
7.5	归约操作	141
7.6	线程同步	143
7.6.1	atomic	143

7.6.2	<code>critical</code>	143
7.6.3	<code>master</code>	144
7.6.4	<code>ordered</code>	144
7.6.5	<code>nowait</code>	145
7.6.6	<code>flush</code>	145
7.6.7	<code>barrier</code>	146
7.7	小结	146
第 8 章 数据环境控制		147
8.1	共享与私有	147
8.1.1	非全局变量的共享	148
8.1.2	变量的私有化	150
8.1.3	<code>threadprivate</code> 子句	150
8.1.4	基于进程的问题	151
8.2	并行域边界处理	151
8.2.1	<code>private</code> 变量	151
8.2.2	<code>threadprivate</code> 变量	151
8.3	OMPI 数据环境控制	152
8.3.1	共享变量	152
8.3.2	私有变量	153
8.3.3	线程专有变量	156
8.3.4	归约变量	159
8.4	小结	160
第 9 章 产生目标代码		162
9.1	源代码变换	162
9.1.1	变换流程	162
9.1.2	支撑函数	164
9.2	AST 变换	164
9.2.1	拼接及创建函数	164
9.2.2	变换函数集	165
9.2.3	OpenMP 节点变换	168
9.2.4	<code>parallel</code> 变换	172
9.2.5	<code>for</code> 变换	175
9.2.6	<code>sections</code> 变换	177
9.2.7	数据环境的处理	178
9.3	代码优化	180
9.4	AST 输出	181
9.4.1	OMPI 的 AST 输出	181
9.4.2	OpenMP 节点输出	182

9.5 小结	185
第 10 章 运行环境.....	186
10.1 重要数据结构.....	186
10.1.1 ORT	186
10.1.2 线程池与 EECB.....	187
10.1.3 任务分担结构	188
10.1.4 共享变量结构	189
10.2 初始化与退出.....	191
10.2.1 ORT 初始化	192
10.2.2 EELIB 初始化	194
10.3 并行支撑函数.....	196
10.3.1 线程状态管理	196
10.3.2 并行域管理.....	198
10.3.3 任务分担	203
10.3.4 同步	212
10.3.5 变量的数据环境	213
10.4 OpenMP 的 API	214
10.4.1 API 函数.....	214
10.4.2 ICV 变量.....	216
10.4.3 引用与链接	217
10.5 环境变量.....	217
10.6 小结	218
第三篇 实践篇.....	219
第 11 章 编译器及测试工具.....	220
11.1 常见 OpenMP 编译器	220
11.1.1 GCC 编译器	220
11.1.2 OMPI 编译器	222
11.1.3 Omni 编译器	224
11.2 性能测试工具.....	226
11.2.1 EPCC Microbenchmark	226
11.2.2 NSA Parallel Benchmark	233
11.2.3 SPEC OMP2001	234
11.2.4 LLNL	234
11.3 小结	235
第 12 章 OMPI 框架分析.....	236
12.1 工作流程.....	236
12.2 OMPI 的处理步骤.....	237
12.3 代码转换.....	241

12.4 进程问题.....	242
12.4.1 全局变量	243
12.4.2 非全局共享变量	244
12.5 运行环境.....	244
12.5.1 初始化	244
12.5.2 并行域的处理	245
12.5.3 任务分担	245
12.5.4 同步	246
12.5.5 线程专有变量	247
12.5.6 与 EELIB 的接口	248
12.6 源代码文档结构.....	249
12.7 后续阅读建议.....	249
12.8 小结	250
第 13 章 ompicc.c 源码分析	251
13.1 ompicc 工作流程	251
13.2 变量声明及参数处理.....	252
13.3 编译部分.....	262
13.3.1 文件名处理	263
13.3.2 预处理	264
13.3.3 代码变换	265
13.3.4 C 编译	266
13.4 链接部分.....	267
13.5 主函数部分.....	269
13.5.1 参数及配置函数	269
13.5.2 main 函数	271
13.6 配置文件.....	274
13.7 运行参数与选项.....	277
13.7.1 环境变量	278
13.7.2 命令行参数	279
13.7.3 配置文件	282
13.8 小结	282
第 14 章 ompi.c 源码分析	283
14.1 ompi 工作流程	283
14.2 ompi.c	284
14.2.1 变量声明及辅助函数	284
14.2.2 main 函数	287
14.2.3 错误处理	293
14.3 ort.defs.....	293

14.4 ompi.h	296
14.5 小结	298

图表索引

图 1.1 Flynn 分类法的四种计算机	2
图 1.2 P-M-NIC 结构示意图	4
图 1.3 两路 8 核 SMP 结构.....	5
图 1.4 两路 8 核 SMP 软件检测结果.....	6
图 1.5 Loongson 3A CPU	6
图 1.6 AMD Magny-Cours 的 32 核 NUMA 系统	7
图 1.7 8 路 32 核处理器软件检测结果	8
图 1.8 由 4 颗 loongson 3A 构成的 NUMA.....	9
图 1.9 Fermi 结构示意图.....	10
图 1.10 天河 1 号超级计算机及其结构示意图	11
图 1.11 天河 1 号软件架构示意图	12
图 1.12 SD-01 及其软硬件结构示意图	13
图 2.1 fork-join 并行执行模型	19
图 3.1 编译系统示意图	46
图 3.2 运行系统库函数的语义作用	47
图 3.3 多种实现方案示意	48
图 3.4 OpenMP 编译与 C 编译任务划分	48
图 3.5 编译器通用结构	49
图 3.6 编译器前后端划分	51
图 4.1 OMPI 中 Lex 的作用示意图	56
图 5.1 循环语句的抽象语法树及三地址码表示	82
图 5.2 “int var_x;”语句的 AST 结构	85
图 5.3 “var_x=var_y+var_z;”语句的 AST 结构	85
图 5.4 分支语句的 AST 结构	85
图 5.5 int 类型说明节点	87
图 5.6 “unsigned long” 的类型说明节点	87
图 5.7 枚举类型的类型说明节点	87
图 5.8 变量声明列表节点	88
图 5.9 OpenMP 构造节点	90
图 5.10 OpenMP 子句节点的两层链表结构	93
图 5.11 OpenMP 构造的 AST 树结构	93
图 5.12 语法树的节点及运算	96
图 5.13 OpenMP for 制导指令的 AST	98
图 5.14 while 语句的 AST	100
图 5.15 hello.c 的 AST 结构	102
图 5.16 allsymbols 数据结构示意图	103

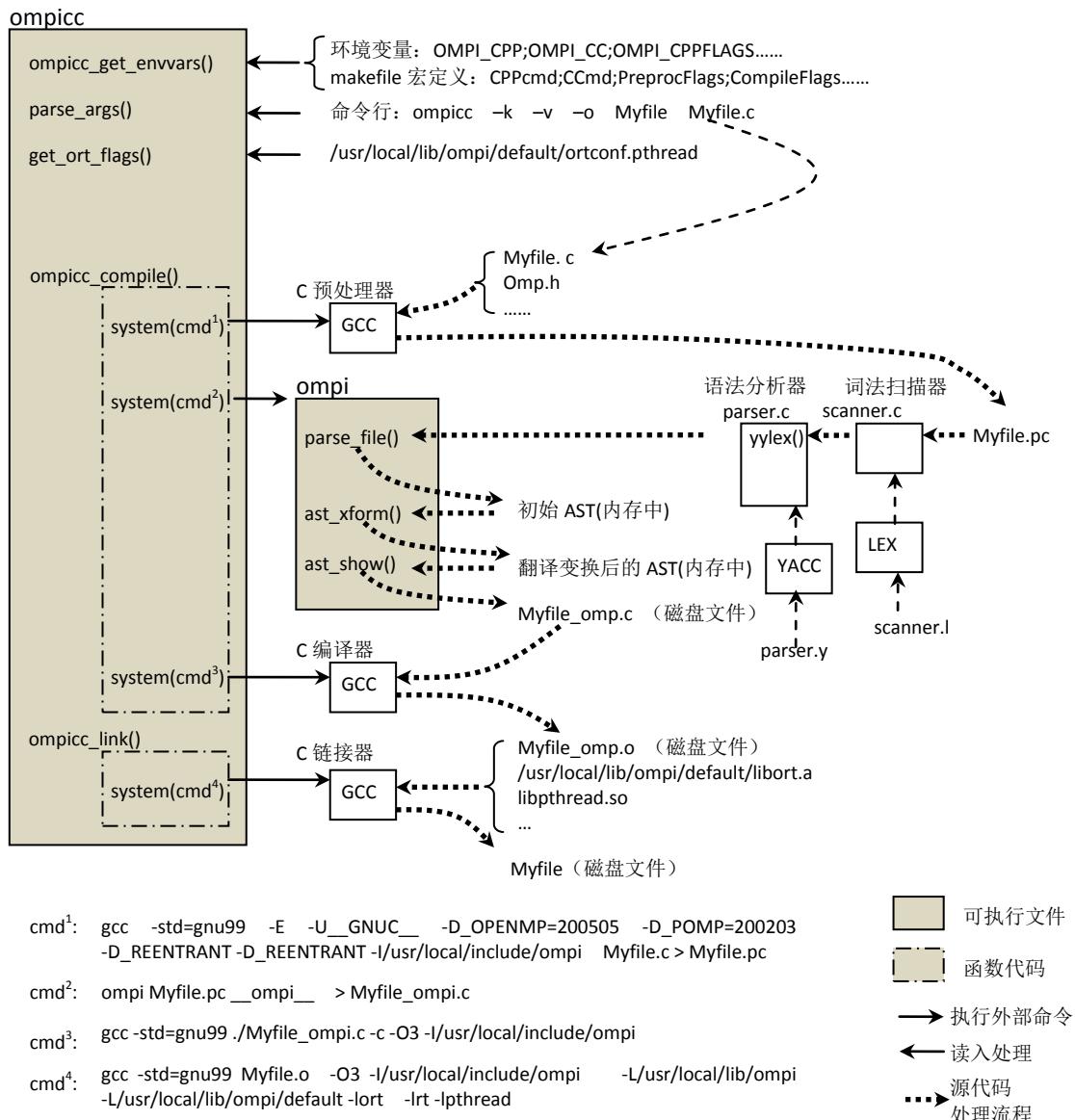
图 5.17 symtab->table 的数据结构示意图	105
图 6.1 并行嵌套执行图	109
图 6.2 线程无关接口形式	110
图 6.3 多个 nowait 任务分担域	113
图 6.4 parallel 的目标代码框架	114
图 6.5 ORT 与 EELIB 接口示意图	115
图 6.6 EE 统一接口	116
图 6.7 线程池的结构	119
图 6.8 线程组关系图	120
图 6.9 ECB 构成层次关系	121
图 6.10 线程池对象与 ECB 关系图	122
图 6.11 ECB 线程控制总览（2 层嵌套）	123
图 7.1 for 循环变量示意图	126
图 7.2 for 任务分担的目标代码框架	129
图 7.3 for 循环静态调度	132
图 7.4 动态调度示意图	133
图 7.5 sections 任务分担的目标代码框架	137
图 8.1 Linux 进程虚实地址映射	148
图 8.2 通过指针在线程之间共享堆栈变量	149
图 8.3 threadprivate 变量的作用	150
图 9.1 BlockList()功能示意图	165
图 9.2 AST 变换函数集及其层次关系	166
图 9.3 变换的递归调用	168
图 9.4 OpenMP 节点变换的函数调用关系	169
图 9.5 atomic 的 AST 变换示意图	171
图 9.6 parallel 构造的 AST 节点	173
图 9.7 并行域外壳代码的 AST 子树结构	174
图 9.8 并行域核代码封装前的 AST 结构	174
图 9.9 并行域内的线程任务函数 LIFO	175
图 9.10 for 构造的 AST 节点	176
图 9.11 循环体代码变换后的 AST 结构	176
图 9.12 变换前 sections 的 AST 子树结构	177
图 9.13 作用域层次变化与 OpenMP 变量引用	179
图 9.14 AST 输出函数及其基本调用层次	182
图 10.1 共享变量传递过程	191
图 10.2 EE 抽象接口	199
图 10.3 任务分担接口函数	206
图 10.4 omi.h 函数分类关系	215

图 11.1 同步制导指令测试.....	230
图 11.2 互斥制导指令测试.....	231
图 11.3 调度制导指令测试.....	232
图 12.1 OMPI 工作流程.....	236
图 12.2 用-k-v 参数执行 ompicc	238
图 12.3 获得预处理结果.....	238
图 12.4 预处理结果文件 omp-hello.pc	239
图 12.5 增加的运行库支撑例程.....	240
图 12.6 新增函数_thrFunc0_	240
图 12.7 新增函数__original_main.....	240
图 12.8 被更换后的 main 函数.....	240
图 13.1 ompicc 工作流程	252
图 13.2 OMPI 编译的完整处理流程	263
图 13.3 编译选项字符串及环境变量	278
图 13.4 OMPI 内部通用的参数链表结构	280
图 13.5 ompicc 命令行参数及内部数据结构	281
图 14.1 ompi 的代码转换流程.....	283
图 14.2 新的 main()函数	286
图 14.3 rtplib_onoff 子树	290
图 14.4 rtplib_defs 子树	290
图 14.5 omp.h 头文件子树	291
图 14.6 AST 整形后的源代码结构.....	292

表 1.1 三种编程模式之比较.....	15
表 2.1 OpenMP API 函数	21
表 2.2 private 和 threadprivate 区别	41
表 2.3 归约操作符与归约变量初值	43
表 4.1 Lex 的正则表达式.....	56
表 4.2 正则表达式举例	57
表 4.3 标记声明举例	57
表 4.4 scanner.l 中的变量及函数归类	70
表 5.1 OpenMP 构造的类型	90
表 5.2 OpenMP 子句的类型	92
表 6.1 嵌套于非嵌套实现上的差异	111
表 6.2 基于线程的 EELIB 接口函数表	117
表 9.1 制导指令与变换函数对应关系	169
表 10.1 ICV 变量	216

表 10.2 ICV 变量的操作函数	216
表 11.1 Loongson3A NSA 测试结构	234
表 13.1 OMPI 命令行选项	280
表 14.1 ompi.c 重要变量列表	284

OMPI 系统总图



第一篇 基础

第一篇的内容是并行计算和 OpenMP 编程的基础。由于并行计算和 OpenMP 编程在大多数高校的本科课程中都没有讲授,因此基础篇中选择这两个部分作为内容,而其他如编译原理、操作系统、C 语言编程等预备性基础内容在本科课程都有讲授,不需要在本书中重复。

第一章包括并行软硬件平台和相应的模型,关于并行计算平台的内容首先介绍了典型结构并略为详细地介绍了 SMP、NUMA、GPU 和 Cluster 四种常见并行计算机,关于并行程序设计技术则介绍了相关的计算模型、编程模型和 PCAM 设计方法,并略微详细地介绍了 OpenMP、MPI、CUDA 和 HPF 四种常见编程技术。

第二章比较全面地介绍了 OpenMP 编程的基础知识,包括 OpenMP 的基本元素和所有的制导指令方面的内容。关于基本元素分成了制导指令、API 函数和环境变量三个部分,制导指令首先介绍的是并行域方面的内容、然后是任务分担及同步的内容,最后是数据环境控制的内容。

第 2 章 OpenMP 编程基础

可以说 OpenMP 制导指令将 C 语言扩展为一个并行语言，但 OpenMP 本身不是一种独立的并行语言，而是为多处理器上编写并行程序而设计的、指导共享内存、多线程并行的编译制导指令和应用程序编程接口（API），可在 C/C++ 和 Fortran (77、90 和 95) 中应用，并在串行代码中以编译器可识别的注释形式出现。OpenMP 标准是由一些具有国际影响力的软件和硬件厂商共同定义和提出，是一种在共享存储体系结构的可移植编程模型，广泛应用与 Unix、Linux、Windows 等多种平台上。

本章讲述与 C 语言绑定的 OpenMP¹并行程序设计的基础知识，包括 OpenMP 基本要素、编译制导指令（Compiler Directive）、运行库函数（Runtime Library）和环境变量（Environment Variables）。

2.1 OpenMP 基本概念

首先来了解 OpenMP 的执行模式和三大要素。

2.1.1 执行模式

OpenMP 的执行模型采用 fork-join 的形式，其中 fork 创建新线程或者唤醒已有线程；join 即多线程的会合。fork-join 执行模型在刚开始执行的时候，只有一个称为“主线程”的运行线程存在。主线程在运行过程中，当遇到需要进行并行计算的时候，派生出线程来执行并行任务。在并行执行的时候，主线程和派生线程共同工作。在并行代码执行结束后，派生线程退出或者阻塞，不再工作，控制流程回到单独的主线程中。

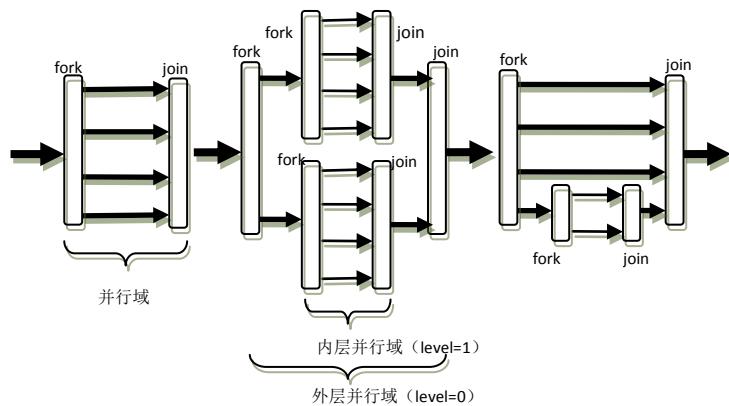


图 2.1 fork-join 并行执行模型

¹ 以后记为 OpenMP/C

OpenMP 的编程者需要在可并行工作的代码部分用制导指令向编译器指出其并行属性，而且这些并行区域可以出现嵌套的情况，如图 2.1 所示。下面对术语**并行域**（Parallel region）作如下定义：在成对的 fork 和 join 之间的区域，称为并行域，它既表示代码也表示执行时间区间。对**OpenMP 线程**作如下定义：在 OpenMP 程序中用于完成计算任务的一个执行流的执行实体，可以是操作系统的线程也可以是操作系统上的进程。

2.1.2 OpenMP 编程要素

OpenMP 编程模型以线程为基础，通过编译制导指令来显式地指导并行化，OpenMP 为编程人员提供了三种编程要素来实现对并行化的完善控制。它们是编译制导、API 函数集和环境变量。

编译制导

在 C/C++ 程序中，OpenMP 的所有编译制导指令是以#pragma omp 开始，后面跟具体的功能指令（或命令），其具有如下形式：

```
#pragma omp 指令 [子句[, 子句] ...]
```

支持 OpenMP 的编译器能识别、处理这些制导指令并实现其功能。其中指令或命令是可以单独出现的，而子句则必须出现在制导指令之后。制导指令和子句按照功能可以大体上分成四类：

- 1) 并行域控制类；
- 2) 任务分担类；
- 3) 同步控制类；
- 4) 数据环境类。

并行域控制类指令用于指示编译器产生多个线程以并发执行任务，任务分担类指令指示编译器如何给各个并发线程分发任务，同步控制类指令指示编译器协调并发线程之间的时间约束关系，数据环境类指令处理并行域内外的变量共享或私有属性以及边界上的数据传送操作等。

下面简单地介绍一下制导指令和相关的子句，现在不必完全弄懂它们的作用，只需大概了解即可，后面还将有详细的使用说明。此处的内容可以作为学习过程中的速查表。

- 1) 版本为 2.5 的 OpenMP 规范中的指令有以下这些：

- **parallel**: 用在一个结构块之前，表示这段代码将被多个线程并行执行；
- **for**: 用于 for 循环语句之前，表示将循环计算任务分配到多个线程中并行执行，以实现任务分担，必须由编程人员自己保证每次循环之间无数据相关性；
- **parallel for**: parallel 和 for 指令的结合，也是用在 for 循环语句之前，表示 for 循环体的代码将被多个线程并行执行，它同时具有并行域的产生和任务分担两个功能；
- **sections**: 用在可被并行执行的代码段之前，用于实现多个结构块语句的任务分担，可并行执行的代码段各自用 section 指令标出（注意区分 sections 和 section）；
- **parallel sections**: parallel 和 sections 两个语句的结合，类似于 parallel for；
- **single**: 用在并行域内，表示一段只被单个线程执行的代码；
- **critical**: 用在一段代码临界区之前，保证每次只有一个 OpenMP 线程进入；
- **flush**: 保证各个 OpenMP 线程的数据影像的一致性；

- **barrier**: 用于并行域内代码的线程同步，线程执行到 **barrier** 时要停下等待，直到所有线程都执行到 **barrier** 时才继续往下执行；
- **atomic**: 用于指定一个数据操作需要原子性地完成；
- **master**: 用于指定一段代码由主线程执行；
- **threadprivate**: 用于指定一个或多个变量是线程专用，后面会解释线程专有和私有的区别。

2) 相应的 OpenMP 的子句有以下一些：

- **private**: 指定一个或多个变量在每个线程中都有它自己的私有副本；
 - **firstprivate**: 指定一个或多个变量在每个线程都有它自己的私有副本，并且私有变量要在进入并行域或任务分担域时，继承主线程中的同名变量的值作为初值；
 - **lastprivate**: 是用来指定将线程中的一个或多个私有变量的值在并行处理结束后复制到主线程中的同名变量中，负责拷贝的线程是 **for** 或 **sections** 任务分担中的最后一个线程；
 - **reduction**: 用来指定一个或多个变量是私有的，并且在并行处理结束后这些变量要执行指定的归约运算，并将结果返回给主线程同名变量；
 - **nowait**: 指出并发线程可以忽略其他制导指令暗含的路障同步；
 - **num_threads**: 指定并行域内的线程的数目；
 - **schedule**: 指定 **for** 任务分担中的任务分配调度类型；
 - **shared**: 指定一个或多个变量为多个线程间的共享变量；
 - **ordered**: 用来指定 **for** 任务分担域内指定代码段需要按照串行循环次序执行；
 - **copyprivate**: 配合 **single** 指令，将指定线程的专有变量广播到并行域内其他线程的同名变量中；
 - **copyin**: 用来指定一个 **threadprivate** 类型的变量需要用主线程同名变量进行初始化；
 - **default**: 用来指定并行域内的变量的使用方式，缺省是 **shared**。
- 这些制导指令将会在后面的编程部分进行详细说明解释。

API 函数

除上述编译制导指令之外，OpenMP 还提供了一组 API 函数用于控制并发线程的某些行为，下面列出 OpenMP 2.5 所有的 API 函数：

表 2.1 OpenMP API 函数

函数名	作用
<code>omp_in_parallel</code>	判断当前是否在并行域中
<code>omp_get_thread_num</code>	返回线程号
<code>omp_set_num_threads</code>	设置后续并行域中的线程个数
<code>omp_get_num_threads</code>	返回当前并行区域中的线程数
<code>omp_get_max_threads</code>	获取并行域可用的最大线程数目
<code>omp_get_num_procs</code>	返回系统中处理器个数
<code>omp_get_dynamic</code>	判断是否支持动态改变线程数目
<code>omp_set_dynamic</code>	启用或关闭线程数目的动态改变
<code>omp_get_nested</code>	判断系统是否支持并行嵌套
<code>omp_set_nested</code>	启用或关闭并行嵌套

omp_init(_nest)_lock	初始化一个（嵌套）锁
omp_destroy(_nest)_lock	销毁一个（嵌套）锁
omp_set(_nest)_lock	（嵌套）加锁操作
omp_unset(_nest)_lock	（嵌套）解锁操作
omp_test(_nest)_lock	非阻塞的（嵌套）加锁
omp_get_wtime	获取 wall time 时间
omp_set_wtime	设置 wall time 时间

后面将详细讨论这些函数的用法，表 2.1 可以作为学习中的 API 函数速查表。

环境变量

OpenMP 规范定义了一些环境变量，可以在一定程度上控制 OpenMP 程序的行为。以下是开发过程中常用的环境变量

1. **OMP_SCHEDULE:** 用于 **for** 循环并行化后的调度，它的值就是循环调度的类型；
2. **OMP_NUM_THREADS:** 用于设置并行域中的线程数；
3. **OMP_DYNAMIC:** 通过设定变量值，来确定是否允许动态设定并行域内的线程数；
4. **OMP_NESTED:** 指出是否可以并行嵌套。

ICV

OpenMP 规范中定义了一些内部控制变量 ICV (Internal Control Variable)，用于表示系统的属性、能力和状态等，可以通过 OpenMP API 函数访问也可以通过环境变量进行修改。但是变量的具体名字和实现方式可以由各个编译器自行决定。

2.2 OpenMP 编程

下面的内容按照功能进行划分，每个功能部分将可能综合使用编译制导、环境变量和 OpenMP API 函数这三种要素。

2.2.1 并行域管理

设计并行程序时，首先需要多个线程来并发地执行任务，因此 OpenMP 编程中第一步就是应该掌握如何产生出多个线程。如前面所实，在 OpenMP 的相邻的 **fork**、**join** 操作之间我们称之为一个并行域，并行域可以嵌套。**parallel** 制导指令就是用来创建并行域的，它也可以和其他指令如 **for**、**sections** 等配合使用形成复合指令。

在 C/C++ 中，**parallel** 的使用方法如下：

```
#pragma omp parallel [for | sections] [子句[子句]…]
{
    ...代码...
}
```

parallel 语句后面要用一个大括号对将要并行执行的代码括起来。

```
1. void main(int argc, char *argv[]) {
2.     #pragma omp parallel
3.         {
```

并行域的开始 (对应 fork)

```
4.         printf("Hello, World!\n");
5.     }
6. }
```

执行以上代码将会打印出以下结果

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

可以看得出 `parallel` 语句中的代码被“相同”地执行了 4 次，说明总共创建了 4 个线程来执行 `parallel` 语句中的代码。为了指定使用多少个线程来执行，可以通过设置环境变量 `OMP_NUM_THREADS` 或者调用 `omp_set_num_threads()` 函数，也可以使用 `num_threads` 子句，前者只能在程序刚开始运行时起作用，而 API 函数和子句可以在程序中并行域产生之前起作用。使用 `num_threads` 子句的例子如下：

```
1. void main(int argc, char *argv[]) {
2. #pragma omp parallel num_threads(8)
3. {
4.     printf("Hello, World!, ThreadId=%d\n", omp_get_thread_num());
5. }
6. }
```

执行以上代码，将会打印出以下结果：

```
Hello, World!, ThreadId = 2
Hello, World!, ThreadId = 6
Hello, World!, ThreadId = 4
Hello, World!, ThreadId = 0
Hello, World!, ThreadId = 5
Hello, World!, ThreadId = 7
Hello, World!, ThreadId = 1
Hello, World!, ThreadId = 3
```

从 `ThreadId` 的不同可以看出创建了 8 个线程来执行以上代码。所以 `parallel` 指令是用来产生或唤醒多个线程创建并行域的，并且可以用 `num_threads` 子句控制线程数目。`parallel` 域中的每行代码都被多个线程重复执行。和传统的创建线程函数比起来，其过程非常简单直观。

`parallel` 的并行域内部代码中，若再出现 `parallel` 制导指令则出现并行嵌套问题，如果设置了 `OMP_NESTED` 环境变量，那么在条件许可时内部并行域也会由多个线程执行，反之没有设置相应变量，那么内部并行域的代码将只由一个线程来执行。还有一个环境变量 `OMP_DYNAMIC` 也影响并行域的行为，如果没有设置该环境变量将不允许动态调整并行域内的线程数目，`omp_set_dynamic()` 也是用于同样的目的。

2.2.2 任务分担

当使用 `parallel` 制导指令产生出并行域之后，如果仅仅是多个线程执行完全相同的任务，那么只是徒增计算工作量而不能达到加速计算的目的，甚至可能相互干扰得出错误结果。因此在产生出并行域之后，紧接着的问题就是如何将计算任务在这些线程之间分配，并加快计算结果的生成速度及其保证正确性。OpenMP 可以完成的任务分担的指令只有 `for`、`sections` 和 `single`，严格意义上来说只有 `for` 和 `sections` 是任务分担指令，而 `single` 只是协助任务分担的指令。

我们对**任务分担域**定义如下：由 **for**、**sections** 或 **single** 制导指令限定的代码及其执行时间段，也就是说任务分担域和并行域的定义一样，既是指代码区间也是指执行时间区间。

for 制导指令

for 指令指定紧随它的循环语句必须由线程组并行执行，用来将一个 **for** 循环任务分配到多个线程，此时各个线程各自分担其中一部分工作。**for** 指令一般可以和 **parallel** 指令结合起来形成 **parallel for** 指令使用，也可以单独用在 **parallel** 指令的并行域中。用法如下：

```
#pragma omp [parallel] for [子句]  
    for 循环语句
```

先看看单独使用 **for** 语句时是什么效果：

```
1. int j = 0;  
2. #pragma omp for  
3.     for (j = 0; j < 4; j++) {  
4.         printf("j = %d, ThreadId = %d\n", j, omp_get_thread_num());  
5.     }
```

执行以上代码后打印出以下结果

```
j = 0, ThreadId = 0  
j = 1, ThreadId = 0  
j = 2, ThreadId = 0  
j = 3, ThreadId = 0
```

从结果可以看，4 次循环都在一个 **ThreadId** 为 0 的线程里执行，并没有实现并发执行也不会加快计算速度。可见 **for** 指令要和 **parallel** 指令结合起来使用才有效果，即 **for** 出现在并行域中才能有多个线程来分担任务。以下代码就是 **parallel** 和 **for** 一起结合使用的形式：

```
1. int j = 0;  
2. #pragma omp parallel  
3. {  
4.     #pragma omp for  
5.     for (j = 0; j < 4; j++) {  
6.         printf("j = %d, ThreadId = %d\n", j, omp_get_thread_num());  
7.     }  
8. }
```

执行以上代码会打印出以下结果：

```
j = 1, ThreadId = 1  
j = 3, ThreadId = 3  
j = 2, ThreadId = 2  
j = 0, ThreadId = 0
```

此时，循环计算任务被正确分配到 4 个不同的线程中执行，各自只需要执行一次循环（总的串行循环次数为 4，4 个线程每个线程承担 1/4，即 1 次）即可。

上面这段代码也可以改写成以下 **parallel for** 复合制导指令的形式：

```
1. int j = 0;  
2. #pragma omp parallel for  
3. for (j = 0; j < 4; j++) {  
4.     printf("j = %d, ThreadId = %d\n", j, omp_get_thread_num());  
5. }
```

如果并行域中有 4 个线程，执行后会打印出相同的结果：

```
j = 0, ThreadId = 0  
j = 2, ThreadId = 2  
j = 1, ThreadId = 1  
j = 3, ThreadId = 3
```

现在考虑另一个情况，在一个 `parallel` 并行域中，可以有多个 `for` 制导指令，如：

```
1. int j;  
2. #pragma omp parallel  
3. {  
4.     #pragma omp for  
5.     for ( j = 0; j < 100; j++ ){  
6.         ...  
7.     }  
8.     #pragma omp for  
9.     for ( j = 0; j < 100; j++ ){  
10.        ...  
11.    }  
12.    ...  
13. }
```

此时只有一个并行域，在该并行域内的多个线程首先完成第一个 `for` 语句的任务分担，然后在此进行一次同步（`for` 制导指令本身隐含有结束处的路障同步），然后再进行第二个 `for` 语句的任务分担，直到退出并行域并只剩下一个主线程为止。

for 调度

在 OpenMP 的 `for` 任务分担中，任务的划分称为调度，各个线程如何划分任务是可以调整的，因此有静态划分、动态划分等，所以调度也分成多个类型。`for` 任务调度子句只能用于 `for` 制导指令中，下面先来看看提供多种调度类型的必要性。

当循环中每次迭代的计算量不相等时，如果简单地给各个线程分配相同次数的迭代的话，会使得各个线程计算负载不均衡，这会使得有些线程先执行完，有些后执行完，造成某些 CPU 核空闲，影响程序性能。例如：

```
1. int i, j;  
2. int a[100][100] = {0};  
3. for ( i = 0; i < 100; i++ )  
4. {  
5.     for ( j = i; j < 100; j++ )  
6.         a[i][j] = i*j;  
7. }
```

如果将最外层循环并行化的话，比如使用 4 个线程，如果给每个线程平均分配 25 次循环迭代计算的话，显然 $i=0$ 和 $i=99$ 的计算量相差了 100 倍，那么各个线程间可能出现较大的负载不平衡情况。为了解决这些问题，适应不同的计算类型，OpenMP 中提供了几种对 `for` 循环并行化的任务调度方案。

在 OpenMP 中，对 `for` 循环任务调度使用 `schedule` 子句来实现，下面介绍 `schedule` 子句的用法。

1) schedule 子句用法

schedule 子句的使用格式为:

```
schedule (type [, size])
```

schedule 有两个参数: type 和 size, size 参数是可选的。如果没有指定 size 大小, 循环迭代会尽可能平均地分配给每个线程。

① type 参数

表示调度类型, 有四种调度类型如下:

- static
- dynamic
- guided
- runtime

这四种调度类型实际上只有 static、dynamic、guided 三种调度方式。runtime 实际上是根据环境变量 OMP_SCHEDULED 来选择前三种中的某种类型, 相应的内部控制变量 ICV 是 run-sched-var。

② size 参数 (可选)

size 参数表示以循环迭代次数计算的划分单位, 每个线程所承担的计算任务对应于 0 个或若干个 size 次循环, size 参数必须是整数。static、dynamic、guided 三种调度方式都可以使用 size 参数, 也可以不使用 size 参数。当 type 参数类型为 runtime 时, size 参数是非法的 (不需要使用, 如果使用的话编译器会报错)。

2) static 静态调度

当 for 或者 parallel for 编译制导指令没有带 schedule 子句时, 大部分系统中默认采用 size 为 1 的 static 调度方式, 这种调度方式非常简单。假设有 n 次循环迭代, t 个线程, 那么给每个线程静态分配大约 n/t 次迭代计算。这里为什么说大约分配 n/t 次呢? 因为 n/t 不一定是整数, 因此实际分配的迭代次数可能存在差 1 的情况, 如果指定了 size 参数的话, 那么可能相差 size 次迭代。

静态调度时可以不使用 size 参数, 也可以使用 size 参数。

① 不使用 size 参数

不使用 size 参数时, 分配给每个线程的是 n/t 次连续的迭代, 不使用 size 参数的用法如下:

```
schedule(static)
```

例如以下代码:

```
1. #pragma omp parallel for schedule(static)
2.     for(i = 0; i < 10; i++)
3.     {
4.         printf("i=%d, thread_id=%d\n", i, omp_get_thread_num());
5.     }
```

假设并行域中有两个线程, 上面代码执行时打印的结果如下:

```
i=0, thread_id=0
i=1, thread_id=0
i=2, thread_id=0
```

```
i=3, thread_id=0  
i=4, thread_id=0  
i=5, thread_id=1  
i=6, thread_id=1  
i=7, thread_id=1  
i=8, thread_id=1  
i=9, thread_id=1
```

可以看出线程 0 得到了 $i=0 \sim 4$ 的连续迭代，线程 1 得到 $i=5 \sim 9$ 的连续迭代。注意由于多线程执行时序的随机性，每次执行时打印的结果顺序可能存在差别，后面的例子也一样。

② 使用 size 参数

使用 size 参数时，分配给每个线程的 size 次连续的迭代计算，用法如下：

```
schedule(static, size)
```

例如以下代码：

```
1. #pragma omp parallel for schedule(static, 2)  
2.     for(i = 0; i < 10; i++)  
3.     {  
4.         printf("i=%d, thread_id=%d\n", i, omp_get_thread_num());  
5.     }
```

若使用两个线程的并行域，执行时会打印以下结果：

```
i=0, thread_id=0  
i=1, thread_id=0  
i=4, thread_id=0  
i=5, thread_id=0  
i=8, thread_id=0  
i=9, thread_id=0  
i=2, thread_id=1  
i=3, thread_id=1  
i=6, thread_id=1  
i=7, thread_id=1
```

从打印结果可以看出，0、1 次迭代分配给 0 号线程，2、3 次迭代分配给 1 号线程，4、5 次迭代分配给 0 号线程，6、7 次迭代分配给 1 号线程，…。每个线程依次分配到 2 次（即 size）连续的迭代计算。

3) dynamic 动态调度

动态调度是动态地将迭代分配到各个线程，动态调度可以使用 size 参数也可以不使用 size 参数，不使用 size 参数时是根据各个线程的完成情况将迭代逐个地分配到各个线程，使用 size 参数时，每次分配给线程的迭代次数为指定的 size 次。各线程动态的申请任务，因此较快的线程可能申请更多次数，而较慢的线程申请任务次数可能较少，因此动态调度可以在一定程度上避免前面提到的按循环次数划分引起的负载不平衡问题。

①下面为使用动态调度不带 size 参数的例子：

```
1. #pragma omp parallel for schedule(dynamic)  
2.     for(i = 0; i < 10; i++)  
3.     {  
4.         printf("i=%d, thread_id=%d\n", i, omp_get_thread_num());  
5.     }
```

如果并行域使用两个线程，打印结果如下：

```
i=0, thread_id=0
```

```
i=1, thread_id=1  
i=2, thread_id=0  
i=3, thread_id=1  
i=5, thread_id=1  
i=6, thread_id=1  
i=7, thread_id=1  
i=8, thread_id=1  
i=4, thread_id=0  
i=9, thread_id=1
```

由于没有指定 size 所以任务划分是按 1 此迭代进行的。

②下面为动态调度使用 size 参数的例子:

```
1. #pragma omp parallel for schedule(dynamic, 2)  
2.     for(i = 0; i < 10; i++)  
3.     {  
4.         printf("i=%d, thread_id=%d\n", i, omp_get_thread_num());  
5.     }
```

打印结果如下:

```
i=0, thread_id=0  
i=1, thread_id=0  
i=4, thread_id=0  
i=2, thread_id=1  
i=5, thread_id=0  
i=3, thread_id=1  
i=6, thread_id=0  
i=8, thread_id=1  
i=7, thread_id=0  
i=9, thread_id=1
```

从打印结果可以看出第“0、1”，“4、5”，“6、7”次迭代被分配给了线程 0，第“2、3”，“8、9”次迭代则分配给了线程 1，每次分配的迭代次数为 2。较快的线程“抢到”了更多的任务。

动态调度时，size 小有利于实现更好的负载均衡，但是会引起过多的任务动态申请的开销，反之 size 大则开销较少，但是不易于实现负在平衡，size 的选择需要在这两者之间进行权衡。

4) guided 调度

guided 调度是一种采用指导性的启发式自调度方法。开始时每个线程会分配到较大的迭代块，之后分配到的迭代块会逐渐递减。迭代块的大小会按指数级下降到指定的 size 大小，如果没有指定 size 参数，那么迭代块大小最小会降到 1。

例如以下代码:

```
1. #pragma omp parallel for schedule(guided,2)  
2.     for (i = 0; i < 10; i++)  
3.     {  
4.         printf("i=%d, thread_id=%d\n", i, omp_get_thread_num());  
5.     }
```

打印结果如下:

```
i=0, thread_id=0  
i=1, thread_id=0  
i=2, thread_id=0  
i=3, thread_id=0  
i=4, thread_id=0
```

```
i=8, thread_id=0  
i=9, thread_id=0  
i=5, thread_id=1  
i=6, thread_id=1  
i=7, thread_id=1
```

第 0、1、2、3、4 次迭代被分配给线程 0，第 5、6、7 次迭代被分配给线程 1，第 8、9 次迭代被分配给线程 0，分配的迭代次数呈递减趋势，最后一次递减到 2 次。

5) runtime 调度

runtime 调度并不是像前面三种调度方式那样是真实调度方式，它是在运行时根据环境变量 `OMP_SCHEDULE` 来确定调度类型，最终使用的调度类型仍然是上述三种调度方式中的一种。

例如在 unix 系统中，可以使用 `setenv` 命令来设置 `OMP_SCHEDULE` 环境变量：

```
setenv OMP_SCHEDULE "dynamic, 2"
```

如果程序中选择 runtime 调度，那么上述命令设置调度类型为动态调度，动态调度的 size 为 2。在 windows 环境中，可以在“系统属性|高级|环境变量”对话框中进行设置环境变量。

sections 制导指令

`sections` 编译制导指令是用于非迭代计算的任务分担，它将 `sections` 语句里的代码用 `section` 制导指令划分成几个不同的段（可以是一条语句，也可以是用`{...}`括起来的结构块），不同的 `section` 段由不同的线程并行执行。用法如下：

```
#pragma omp [parallel] sections [子句]  
{  
#pragma omp section  
{  
    ...代码块...  
}  
[#pragma omp section]  
...  
}
```

先看一下以下具有三个 `section` 的例子代码：

```
1. void main(int argc, char *argv)  
2. {  
3.     #pragma omp parallel sections {  
4.         #pragma omp section  
5.             printf("section 1 thread = %d\n", omp_get_thread_num());  
6.         #pragma omp section  
7.             printf("section 2 thread = %d\n", omp_get_thread_num());  
8.         #pragma omp section  
9.             printf("section 3 thread = %d\n", omp_get_thread_num());  
10.    }
```

执行后将打印出以下结果：

```
section 1 thread = 0  
section 2 thread = 2  
section 3 thread = 1
```

此时，各个 `section` 里的代码是被分配到不同的线程并发地执行。下面来看看在一个并行域内有多个 `sections` 的情况：

```
1. void main(int argc, char *argv)
```

```

2.  {
3.  #pragma omp parallel
4.  {
5.      #pragma omp sections
6.      {
7.          #pragma omp section
8.              printf("section 1 ThreadId = %d\n", omp_get_thread_num());
9.          #pragma omp section
10.             printf("section 2 ThreadId = %d\n", omp_get_thread_num());
11.         }
12.     #pragma omp sections
13.     {
14.         #pragma omp section
15.             printf("section 3 ThreadId = %d\n", omp_get_thread_num());
16.         #pragma omp section
17.             printf("section 4 ThreadId = %d\n", omp_get_thread_num());
18.     }
19.   }
20. }
```

执行后将打印出以下结果:

```

section 1 ThreadId = 0
section 2 ThreadId = 3
section 3 ThreadId = 3
section 4 ThreadId = 1
```

这种方式和前面那种方式的区别是，这里有两个 `sections` 构造先后串行执行的，即第二个 `sections` 构造的代码要等第一个 `sections` 构造的代码执行完后才能执行。`sections` 构造里面的各个 `section` 部分代码是并行执行的。与 `for` 制导指令一样，在 `sections` 的结束处有一个隐含的路障同步，没有其他说明的情况下，所有线程都必须到达该点才能往下运行。

使用 `section` 指令时，需要注意的是这种方式需要保证各个 `section` 里的代码执行时间相差不大，否则某个 `section` 执行时间比其他 `section` 过长就造成了其它线程空闲等待的情况。用 `for` 语句来分担任务时工作量由系统自动划分，只要每次循环间没有时间上的差距，那么分摊是比较均匀的，使用 `section` 来划分线程是一种手工划分工作量的方式，最终负载均衡的好坏得依赖于程序员。

single 制导指令

单线程执行 `single` 制导指令指定所包含的代码段只由一个线程执行，别的线程跳过这段代码。如果没有 `nowait` 从句，所有线程在 `single` 制导指令结束处隐式同步点同步。如果 `single` 制导指令有 `nowait` 从句，则别的线程直接向下执行，不在隐式同步点等待；`single` 制导指令用在一段只被单个线程执行的代码段之前，表示后面的代码段将被单线程执行，具体用法如下：

```
#pragma omp single [子句]
```

对于如下范例代码：

```

1.  #include <stdio.h>
2.  int main(int argc,void **argv )
```

```

3.    {
4.        #pragma omp parallel
5.        {
6.            #pragma omp single
7.                printf ("Beginning work1.\n");
8.                printf("work on 1 parallelly. %d\n",omp_get_thread_num());
9.            #pragma omp single
10.               printf ("Finishing work1.\n");
11.            #pragma omp single nowait
12.                printf ("Beginning work2.\n");
13.                printf("work on 2 parallelly. %d\n",omp_get_thread_num());;
14.        }
15.    }

```

对应的输出结果如下：

```

Beginning work1.
work on 1 parallelly. 0
work on 1 parallelly. 3
work on 1 parallelly. 2
work on 1 parallelly. 1
Finishing work1.
Beginning work2.
work on 2 parallelly. 1
work on 2 parallelly. 3
work on 2 parallelly. 2
work on 2 parallelly. 0

```

从上面的结果可以看出，在并行域内有多个线程并发执行，因此“`work on 1/2 parallel`”将由 4 个线程并发执行，但是对于使用 `single` 语句制导的“`Beginning work1/2`”以及“`Finishing work1`”的打印语句只有一个线程在执行。

另一种需要使用 `single` 制导指令的情况是为了减少并行域创建和撤销的开销，而将多个临近的 `parallel` 并行域合并时。经过合并后，原来并行域之间的串行代码也将被并行执行，违反了代码原来的目的，因此这部分代码可以用 `single` 指令加以约束只用一个线程来完成。

2.2.3 同步

在正确产生并行域并用 `for`、`sections` 等语句进行任务分担后，还须考虑的是这些并发线程的同步互斥需求。在 OpenMP 应用程序中，由于是多线程执行，所以必须有线程互斥机制以保证程序在出现数据竞争的时候能够得出正确的结果，并且能控制线程执行的先后制约关系，以保证执行结果的正确性。

OpenMP 支持两种不同类型的线程同步机制，一种是互斥锁的机制，可以用来保护一块共享的存储空间，使任何时候访问这块共享内存空间的线程最多只有一个，从而保证了数据的完整性；另外一种同步机制是事件同步机制，这种机制保证了多个线程制之间的执行顺序。互斥的操作针对需要保护的数据而言，在产生了数据竞争的内存区域加入互斥，可以使用包括 `critical`、`atomic` 等制导指令以及 API 中的互斥函数。而事件机制则控制线程执行顺序，包括 `barrier` 同步路障、`ordered` 定序区段、`matser` 主线程执行等。

critical 临界区

在可能产生内存数据访问竞争的地方，都需要插入相应的临界区制导指令，临界区编译制导指令的格式如下：

```
#pragma omp critical [(name)]
{ 需保护的代码段 }
```

其中的名字 name 不是必需的。例如以下代码：

```
1. int i; int max_num_x=max_num_y=-1;
2. #pragma omp parallel for
3. for (i=0; i<n; i++)
4. {
5.     #pragma omp critical (max_arx)
6.         if (arx[i] > max_num_x)
7.             max_num_x = arx[i];
8.     #pragma omp critical (max_ary)
9.         if (ary[i] > max_num_y)
10.            max_num_y = ary[i];
11. }
```

在一个并行域内的 for 任务分担域中，各个线程逐个进入到 critical 保护的区域内，比较当前元素和最大值的关系并可能进行最大值的更替，从而避免了数据竞争的情况。critical 语句不允许互相嵌套。

atomic 原子操作

在 OpenMP 的程序中，原子操作的功能是通过 #pragma omp atomic 编译制导指令提供的。前面提到的 critical 临界区操作能够作用在任意大小的代码块上，而原子操作只能作用在单条赋值语句中。在 C/C++ 语言中，原子操作的语法格式如下所示：

```
#pragma omp atomic
x <binop>=expr
```

或者

```
#pragma omp atomic
x++ // or x-, --x, ++x
```

明显的，能够使用原子语句的前提条件是相应的语句能够转化成一条机器指令，使得相应功能能够一次执行完毕而不会被打断。下面是在 C/C++ 语言中可用的原子操作。

“+ * - / & ^ | <<>>”

值得注意的是，当对一个数据进行原子操作保护的时候，就不能对数据进行临界区的保护，OpenMP 运行时并不能在这两种保护机制之间建立配合机制。用户在针对同一个内存单元使用原子操作的时候，需要在程序所有涉及到该变量并行赋值的部位都加入原子操作的保护。

```
1. int counter=0;
2. #pragma omp parallel
3. {
4.     for (int i=0;i<10000;i++)
5.     {
6.         #pragma omp atomic      //atomic operation
7.         counter++;
}
```

```
8.         }
9.     }
10.    printf("counter = %d\n", counter);
```

由于使用 `atomic` 语句，则避免了可能出现的数据访问竞争情况，最后的执行结果都是一致的，执行结果总是为下面的数值（假设有两个并发线程）：

```
counter=20000
```

而将 `atomic` 这一行语句从源程序中删除时，由于有了数据访问的竞争情况，所以最后的执行结果是不确定的。下面是一个可能的执行结果：

```
counter=12014
```

该结果因数据竞争而出错。

barrier 同步路障

路障（`barrier`）是 OpenMP 线程的一种同步方法。线程遇到路障时必须等待，直到并行区域内的所有线程都到达了同一点，才能继续执行下面的代码。在每一个并行域和任务分担域的结束处都会有一个隐含的同步路障，执行此并行域/任务分担域的线程组在执行完毕本区域代码之前，都需要同步并行域的所有线程。也就是说在 `parallel`、`for`、`sections` 和 `single` 构造的最后，会有一个隐式的路障。

在有些情况下，隐含的同步路障并不能提供有效的同步措施。这时，需要程序员插入明确的同步路障语句`#pragma omp barrier`。此时，在并行区域的执行过程中，所有的执行线程都会在同步路障语句上进行同步。

```
1.      #pragma omp parallel
2.      {
3.          Initialization();
4.          #pragma omp barrier
5.          Process();
6.      }
```

上述例子中，只有等所有的线程都完成 `Initialization()` 初始化操作以后，才能够进行下一步的处理动作，因此，在此处插入一个明确的同步路障操作以实现线程之间的同步。

nowait

为了避免在循环过程中不必要的同步路障并加快运行速度，可以使用 `nowait` 子句除去这个隐式的路障。如下范例所示：

```
1.  #include <omp.h>
2.  int main()
3.  {
4.      int i,j;
5.      #pragma omp parallel num_threads(4)
6.      {
7.          #pragma omp for nowait
8.          for (i = 0; i < 8; ++i)
9.          {
10.              printf("+\n");
11.          }
12.      }
13.  }
```

```
12.         {         printf("      -\n");         }
13.     }
14.     return 0;
15. }
```

执行结果如下：

```
+
+
+
+
+
+
+
-
-
-
-
-
-
-
-
-
```

也就是说第一个 **for** 循环结束后，后一个 **for** 循环才开始。但是使用了 **nowait** 之后则出现以下结果：

```
+
+
-
-
+
+
-
-
+
+
-
-
+
+
-
-
+
+
-
-
-
-
```

此时线程在完成第一个 **for** 循环子任务后，并不需要同步等待，而是直接执行后面的任务，因此出现“-”在“+”前面的情况。

nowait 子句消除了不必要的同步开销，加快了计算速度，但是也引入了实现上的困难，这个将在后面编译器相关的章节中讨论。

master 主线程执行

用于指定一段代码由主线程执行。**master** 制导指令和 **single** 制导指令类似，区别在于，**master** 制导指令包含的代码段只由主线程执行，而 **single** 制导指令包含的代码段可由任一线程执行，并且 **master** 制导指令在结束处没有隐式同步，也不能指定 **nowait** 从句。语句格式如下：

```
#pragma omp master
```

下面是一个计算 $0 \sim 4$ 的平方数的程序：

```
1. #include <omp.h>
2. #include <stdio.h>
```

```

3. int main( )
4. {
5.     int a[5], i;
6.     #pragma omp parallel
7.     {
8.         #pragma omp for
9.         for (i = 0; i < 5; i++)
10.            a[i] = i * i;
11.         #pragma omp master
12.         for (i = 0; i < 5; i++)
13.             printf_s("a[%d] = %d\n", i, a[i]);
14.     }
15. }
```

输入结果为：

```
a[0] = 0
a[1] = 1
a[2] = 4
a[3] = 9
a[4] = 16
```

虽然上面的打印语句是在 `parallel` 并行域中，但是并没有被多个线程所执行，而是只有一个线程将逐个元素打印出来。

ordered 顺序制导指令

对于循环代码的任务分担中，某些代码的执行需要按规定的顺序执行。典型的情况如下：在一次循环的过程中大部分的工作是可以并行执行的，而特定部分代码的工作需要等到前面的工作全部完成之后才能够执行。这时，可以使用 `ordered` 子句使特定的代码按照串行循环的次序来执行。下面例子说明 `ordered` 子句是如何对结果产生影响的。

```

1. #include <stdio.h>
2. #include <omp.h>
3. int main( )
4. {
5.     int i;
6.     #pragma omp parallel
7.     {
8.         test(1, 8);
9.         #pragma omp for ordered
10.        for (i = 0; i < 5; i++)
11.            printf_s("iteration %d\n", iter);
12.    }
13. }
```

打印结果如下：

```
iteration 0
iteration 1
iteration 2
iteration 3
iteration 4
```

从结果可以看出，虽然在 `ordered` 子句之前的工作是并行执行的，但是在遇到 `ordered` 子句的时候，只有前面的循环都执行完毕之后，才能够进行下一步执行。上面的例子中将所有循环迭代都串行化了，实际上可以只将关键部分串行化，其代码框架如下：

```
14. #pragma omp parallel
15. {
16.     #pragma omp for
17.     for (i = 0; i < 100; i++)
18.     {
19.         一些无数据相关、可并行乱序执行的操作
20.         ....
21.         #pragma omp ordered
22.         一些有数据相关、只能顺序执行的操作
23.     }
24. }
```

这样一来，有些任务在并行执行，对于部分必须串行执行的部分才启用 `ordered` 保护。

互斥锁函数

前面分别介绍了互斥同步的两种方法：`atomic` 和 `critical`，除了上述的编译制导指令，OpenMP 还可以通过库函数支持实现互斥操作，方便用户实现特定的同步需求。编译制导指令的互斥支持只能放置在一段代码之前，作用在这段代码之上。而 OpenMP API 所提供的互斥函数可放在任意需要的位置。程序员必须自己保证在调用相应锁操作之后释放相应的锁，否则就可能造成多线程程序的死锁。表 2.1 中包含 OpenMP API 函数提供的互斥函数和可嵌套的互斥锁函数。

下面来看看互斥函数的使用例子：

```
1. #include <omp.h>
2. static omp_lock_t lock;
3. int main()
4. {
5.     int i;
6.     omp_init_lock(&lock);
7.     #pragma omp parallel for
8.     for (i = 0; i < 5; ++i)
9.     {
10.         omp_set_lock(&lock);
11.         printf("%d +\n",omp_get_thread_num());
12.         printf("%d -\n",omp_get_thread_num());
13.         omp_unset_lock(&lock);
14.     }
15.     omp_destroy_lock(&lock);
16.     return 0;
17. }
```

下面是其中一个正常运行的结果：

2 +

```
2 -  
3 +  
3 -  
1 +  
1 -  
0 +  
0 -  
4 +  
4 -
```

上边的示例对 `for` 循环中的所有内容进行加锁保护，同时只能有一个线程执行 `for` 循环中的内容。因此同一个线程的两次打印之间不会被打断。如果删除代码中的获得锁、释放锁的代码，因两条打印语句之间间隔太短，大多数情况下也是正确的，但偶尔可能输出如下错误结果：

```
3 +  
3 -  
1 +  
1 -  
4 +  
4 -  
2 +  
0 +  
0 -  
2 -
```

互斥锁函数中只有 `omp_test_lock` 函数是带有返回值的，该函数可以看作是 `omp_set_lock` 的非阻塞版本。

2.2.4 数据环境控制

现在读者已经能成功创建并行域生成多个线程、使用任务分担指令并行执行任务、利用同步指令控制并发执行中的互斥和顺序后，还需要了解的是如何使用数据环境（Data Environment）控制指令。由于是多线程环境，因此就涉及了共享变量和私有变量的两个基本问题，在此基础之上还有线程专有数据、变量的初值和终值的设定、归约操作相关的变量等问题。

通常来说 OpenMP 是建立在共享存储结构的计算机之上，使用操作系统提供的线程作为并发执行的基础，所以线程间的全局变量和静态变量是共享的，而局部变量、自动变量是私有的。但是对 OpenMP 编程而言，缺省变量往往是共享变量，而不管它是不是全局静态变量还是局部自动变量。也就是说 OpenMP 各个线程的变量是共享还是私有，是依据 OpenMP 自身的规则和相关的数据子句而定，而不是依据操作系统线程或进程上的变量特性而定。

OpenMP 的数据处理子句包括 `private`、`firstprivate`、`lastprivate`、`shared`、`default`、`reduction`、`copyin` 和 `copyprivate`。它与编译制导指令 `parallel`、`for` 和 `sections` 相结合，用来控制变量的作用范围。它们控制数据环境，比如，哪些串行部分中的数据变量被传递到程序的并行部分以及如何传送，哪些变量对所有并行部分的线程是可见的，哪些变量对所有并行部分的线程是私有的，等等。

共享与私有化

1) `shared` 子句

shared 子句用来声明一个或多个变量是共享变量。用法如下:

```
shared(list)
```

需要注意的是，在并行域内使用共享变量时，如果存在写操作，必须对共享变量加以保护，否则不要轻易使用共享变量，尽量将共享变量的访问转化为私有变量的访问。循环迭代变量在循环构造的任务分担域里是私有的。声明在任务分担域内的自动变量都是私有的。

2) **default** 子句

default 子句用来允许用户控制并行区域中变量的共享属性。

用法如下:

```
default(shared | none)
```

使用 **shared** 时，缺省情况下，传入并行区域内的同名变量被当作共享变量来处理，不会产生线程私有副本，除非使用 **private** 等子句来指定某些变量为私有的才会产生副本。

如果使用 **none** 作为参数，除了那些由明确定义的除外，线程中用到的变量都必须显式指定为是共享的还是私有的。

3) **private** 子句

private 子句用于将一个或多个变量声明成线程私有的变量，变量声明成私有变量后，指定每个线程都有它自己的变量私有副本，其他线程无法访问私有副本。即使在并行域外有同名的共享变量，共享变量在并行域内不起任何作用，并且并行域内不会操作到外面的共享变量。

private 子句的用法格式如下:

```
private (list)
```

下面便是一个使用 **private** 子句的代码例子:

```
1. int k = 100;
2. #pragma omp parallel for private(k)
3. for ( k=0; k < 10; k++)
4. {
5.     printf("k=%d\n", k);
6. }
7. printf("last k=%d\n", k);
```

上面程序执行后打印的结果如下:

```
k=6
k=7
k=8
k=9
k=0
k=1
k=2
k=3
k=4
k=5
last k=100
```

从打印结果可以看出，**for** 循环前的变量 **k** 和循环区域内的变量 **k** 其实是两个不同的变量。用 **private** 子句声明的私有变量的初始值在并行域的入口处是未定义的，它并不会继承同名共享变量的值。注意，出现在 **reduction** 子句中的变量不能出现在 **private** 子句中。

4) **firstprivate** 子句

私有变量的初始化和终结操作：OpenMP 编译制导指令需要对这种需求给予支持，即使用 **firstprivate** 和 **lastprivate** 来满足这两种需求。使得并行域或任务分担域开始执行时，私有变量通过主线程中的变量初始化，也可以在并行域或任务分担域结束时，将最后一次一个线程上的私有变量赋值给主线程的同名变量。

private 声明的私有变量不会继承同名变量的值，于是 OpenMP 提供了 **firstprivate** 子句来实现这个功能。**firstprivate** 子句是 **private** 子句的超集，即不仅包含了 **private** 子句的功能，而且还要对变量做进行初始化。其格式如下：

firstprivate (list)

先看一下以下的代码例子

```
1. int k = 100;
2. #pragma omp parallel for firstprivate(k)
3.     for ( i=0; i < 4; i++)
4.     {
5.         k += i;
6.         printf("k=%d\n",k);
7.     }
8. printf("last k=%d\n", k);
```

上面代码执行后打印结果如下：

```
k=100
k=101
k=103
k=102
last k=100
```

从打印结果可以看出，并行域内的私有变量 **k** 继承了外面共享变量 **k** 的值 100 作为初始值，并且在退出并行区域后，共享变量 **k** 的值保持为 100 未变。

5) **lastprivate** 子句

有时要将任务分担域内私有变量的值经过计算后，在退出时，将它的值赋给同名的共享变量（前面的 **private** 和 **firstprivate** 子句在退出并行域时都没有将私有变量的最后取值赋给对应的共享变量），**lastprivate** 子句就是用来实现在退出并行域时将私有变量的值赋给共享变量。**lastprivate** 子句也是 **private** 子句的超集，即不仅包含了 **private** 子句的功能，而且还要将变量从 **for**、**sections** 的任务分担域中最后的线程中复制给外部同名变量。其格式如下：

lastprivate (list)

举个例子如下：

```
1. int k = 100;
2. #pragma omp parallel for firstprivate(k),lastprivate(k)
3.     for ( i=0; i < 4; i++)
4.     {
5.         k+=i;
6.         printf("k=%d\n",k);
7.     }
```

```
8. printf("last k=%d\n", k);
```

上面代码执行后的打印结果如下：

```
k=100  
k=101  
k=103  
k=102  
last k=103
```

从打印结果可以看出，退出 `for` 循环的并行区域后，共享变量 `k` 的值变成了 103，而不是保持原来的 100 不变。

由于在并行域内是多个线程并行执行的，最后到底是将那个线程的最终计算结果赋给了对应的共享变量呢？OpenMP 规范中指出，如果是 `for` 循环迭代，那么是将最后一次循环迭代中的值赋给对应的共享变量；如果是 `sections` 构造，那么是代码中排在最后的 `section` 语句中的值赋给对应的共享变量。注意这里说的最后一个 `section` 是指程序语法上的最后一个，而不是实际运行时的最后一个运行完的。

如果是类（`class`）类型的变量使用在 `lastprivate` 参数中，那么使用时有些限制，需要一个可访问的，明确的缺省构造函数，除非变量也被使用作为 `firstprivate` 子句的参数；还需要一个拷贝赋值操作符，并且这个拷贝赋值操作符对于不同对象的操作顺序是未指定的，依赖于编译器的定义。

5) flush

OpenMP 的 `flush` 制导指令主要与多个线程之间的共享变量的一致性问题。用法如下：

```
flush [(list)]
```

该指令将列表中的变量执行 `flush` 操作，直到所有变量都已完成相关操作后才返回，保证后续变量访问的一致性。

线程专有数据

线程专有数据和私有数据不太相同，`threadprivate` 子句用来指定全局的对象被各个线程各自复制了一个私有的拷贝，即各个线程具有各自私有、线程范围内的全局对象。`private` 变量在退出并行域后则失效，而 `threadprivate` 线程专有变量可以在前后多个并行域之间保持连续性。

1) threadprivate 子句

用法如下：

```
#pragma omp threadprivate(list) new-line
```

下面用 `threadprivate` 命令来实现一个线程私有的计数器，各个线程使用同一个函数来实现自己的计数。计数器代码如下：

```
1. int counter = 0;  
2. #pragma omp threadprivate (counter)  
3. int increment_counter()  
4. {  
5.     counter++;
```

```

6.     return(counter);
7.   }

```

如果是静态变量也同样可以使用 `threadprivate` 声明成线程私有的，上面的 `counter` 变量如改成用 `static` 类型来实现时，代码如下：

```

1. int increment_counter2()
2. {
3. static int counter = 0;
4. #pragma omp threadprivate (counter)
5. counter++;
6. return(counter);
7. }

```

用作 `threadprivate` 的变量的地址不能是常数。对于 C++ 的类（class）类型变量，用作 `threadprivate` 的参数时有些限制，当定义时带有外部初始化则必须具有明确的拷贝构造函数。对于 windows 系统，`threadprivate` 不能用于动态装载（使用 `LoadLibrary` 装载）的 DLL 中，可以用于静态装载的 DLL 中，关于 windows 系统中的更多限制，请参阅 MSDN 中有关 `threadprivate` 子句的帮助材料。有关 `threadprivate` 指令的更多限制方面的信息，详情请参阅 OpenMP2.5 规范。

表 2.2 private 和 threadprivate 区别

	Private	Threadprivate
数据类型	变量	变量
位置	在域的开始或共享任务单元	在块或整个文件区域的例程的定义上
持久性	否	是
扩充性	只是词法的-除非作为子程序的参数而传递	动态的
初始化	使用 <code>firstprivate</code>	使用 <code>copyin</code>

2) copyin 子句

`copyin` 子句用来将主线程中 `threadprivate` 变量的值复制到执行并行域的各个线程的 `threadprivate` 变量中，便于所有线程访问主线程中的变量值，其格式如下：

`copyin (list)`

`copyin` 中的参数必须被声明成 `threadprivate` 的，对于 `class` 类型的变量，必须带有明确的拷贝赋值操作符。

对于前面 `threadprivate` 中讲过的计数器函数，如果多个线程使用时，各个线程都需要对全局变量 `counter` 的副本进行初始化，可以使用 `copyin` 子句来实现，示例代码如下：

```

1. int main(int argc, char* argv[])
2. {
3.     int iterator;
4.     #pragma omp parallel sections copyin(counter)
5.     {
6.         #pragma omp section
7.         {
8.             int count1;
9.             for ( iterator = 0; iterator < 100; iterator++ )

```

```

10.           count1 = increment_counter();
11.           printf("count1 = %ld\n", count1);
12.       }
13. #pragma omp section
14. {
15.     int count2;
16.     for ( iterator = 0; iterator < 200; iterator++ )
17.         count2 = increment_counter();
18.     printf("count2 = %ld\n", count2);
19.   }
20. }
21. printf("counter = %ld\n", counter);
22. }
```

打印结果如下：

```

count1 = 100
count2 = 200
counter = 0
```

从打印结果可以看出，两个线程都正确实现了各自的计数，而外部共享变量仍为 0。

2) copyprivate 子句

`copyprivate` 子句提供了一种机制，即将一个线程私有变量的值广播到执行同一并行域的其他线程。用法如下：

```
copyprivate(list)
```

`copyprivate` 子句可以关联 `single` 构造，在 `single` 构造的 `barrier` 到达之前就完成了广播工作。`copyprivate` 可以对 `private` 和 `threadprivate` 子句中的变量进行操作，但是当使用 `single` 构造时，`copyprivate` 的变量不能用于 `private` 和 `firstprivate` 子句中。

下面便是一个使用 `copyprivate` 的代码例子：

```

1. int counter = 0;
2. #pragma omp threadprivate(counter)
3. int increment_counter()
4. {
5.     counter++;
6.     return(counter);
7. }
8. #pragma omp parallel
9. {
10.    int count;
11.    #pragma omp single copyprivate(counter)
12.    {
13.        counter = 50;
14.    }
15.    count = increment_counter();
16.    printf("ThreadId: %ld, count = %ld\n", omp_get_thread_num(), count);
```

```
17. }
```

打印结果为：

```
ThreadId: 2, count = 51  
ThreadId: 0, count = 51  
ThreadId: 3, count = 51  
ThreadId: 1, count = 51
```

如果没有使用 `copyprivate` 子句，那么打印结果为：

```
ThreadId: 2, count = 1  
ThreadId: 1, count = 1  
ThreadId: 0, count = 51  
ThreadId: 3, count = 1
```

从打印结果可以看出，使用 `copyprivate` 子句后，`single` 构造内给 `counter` 赋的值被广播到了其他线程里，但没有使用 `copyprivate` 子句时，只有一个线程获得了 `single` 构造内的赋值，其他线程没有获取 `single` 构造内的赋值。

归约操作

`reduction` 子句主要用来对一个或多个参数条目指定一个操作符，每个线程将创建参数条目的一个私有拷贝，在并行域或任务分担域的结束处，将用私有拷贝的值通过指定的运行符运算，原始的参数条目被运算结果的值更新。`reduction` 子句的格式如下：

```
reduction(operator: list)
```

下表列出了可以用于 `reduction` 子句的一些操作符以及对应私有拷贝变量缺省的初始值，私有拷贝变量的实际初始值依赖于 `reduction` 变量的数据类型。

表 2.3 归约操作符与归约变量初值

操作符	初值
+ (加)	0
* (减)	1
- (乘)	0
& (按位与)	~ 0
(按位或)	0
\wedge (按位异或)	0
$\&\&$ (逻辑与)	1
$\ $ (逻辑或)	0

例如一个整数求和的程序如下：

```
1. int i, sum = 100;  
2. #pragma omp parallel for reduction(+: sum)  
3. for ( i = 0; i < 1000; i++ )  
4. {  
5.     sum += i;  
6. }  
7. printf( "sum = %ld\n", sum);
```

注意，如果在并行域内不加锁保护就直接对共享变量进行写操作，存在数据竞争问题，会导致不可预测的异常结果。如果共享数据作为 `private`、`firstprivate`、`lastprivate`、`threadprivate`、`reduction` 子句的参数进入并行域后，就变成线程私有了，不需要加锁保护了。

2.3 小结

本章介绍了 OpenMP 的三个基本要素和基本的编程方法。三个要素是编译指导指令、API 函数和环境变量，在编程应用中环境变量只在程序开始运行时做初始设置，而制导指令和 API 可以只运行中发生作用。关于 OpenMP 编程的内容分成 4 个部分，综合应用制导指令、API 函数和环境变量，分别解决如何控制多个线程的产生、如何利用多个线程对任务进行划分、如何保证线程间的互斥和同步以及如何控制变量在并行域或任务分担于中的共享属性等等。

第二篇 OpenMP 编译

第二篇将讨论 OpenMP 编译的原理和实现细节。首先，通过分析 OpenMP 编译系统的构成，明确编译系统中各个部分的目标和相互关系，界定出 OpenMP 编译的狭义定义和相应的目标代码形式。然后讲述 OpenMP/C 程序的词法分析和语法分析问题以及如何使用 Lex 和 Yacc 工具来完成这两项任务。对于编译过程的中间表示——AST 树的问题，这里讨论了 AST 的形式和结构，以及为了创建 AST 在语法分析过程中所执行的语义动作。接着讨论了目标代码的翻译变换，先分析源程序和目标程序在 OpenMP 编译制导指令上的语义差距，然后给出了各种翻译“框架”。此处涉及了并行域管理问题、任务分担和同步问题、变量数据环境问题，是 OpenMP 编译的最核心部分。最后讨论目标代码的生成技术和运行环境的支持。

由于编译原理是本篇的基础知识，因此读者需要对编译的基本理论有所掌握，编译理论中的数学模型和算法不在这里讨论，这些包括：上下文无关语法、正则表达式、有限自动机、编译中的并行性问题等等。

第3章 OpenMP 编译

编译过程是将某种程序设计语言（源语言）所编写的程序（源程序/代码）作为翻译和加工的对象，输出与之等价的另一种语言（目标语言）程序（目标程序/代码）。对于 OpenMP 编译过程来说，带有 OpenMP 制导指令的 C/Fortran 语言程序作为源程序，目标程序则是可由处理器硬件直接执行的程序。一般来说，编译器将源程序翻译成目标程序的时候，往往还借助于运行系统来配合运行，因此 OpenMP 编译将涉及编译器、源语言、目标语言、运行系统等多个要素。本书只讨论基于 C 的 OpenMP 程序的编译。

OpenMP 翻译（OpenMP translation）是狭义 OpenMP 编译器进行的主要工作，它指将高层 OpenMP 程序转换成较低层的多线程程序（还需进一步编译成可执行文件），运行时需要依赖于运行环境提供的特定运行库函数。

3.1 OpenMP 编译系统

我们先从介绍编译系统开始，讲述编译器和运行系统的关系，明确一个可运行的编译系统不仅仅是一个编译转换程序，同时还是运行环境的支撑系统。然后根据分级编译的思想，分析以标准 C 代码作为 OpenMP 编译目标代码的原因和优势。然后给出 OpenMP 编译器通常具备的典型结构和相关功能模块的作用。

3.1.1 编译系统

编译系统包括编译程序（编译器）和运行系统，如果推广到一般情况，编译和运行并不需要在同一计算机上（可以是异构的两台计算机），此时我们将源程序、目标程序、编译器、运行系统等等要素统一地用图 3.1 表示如下。

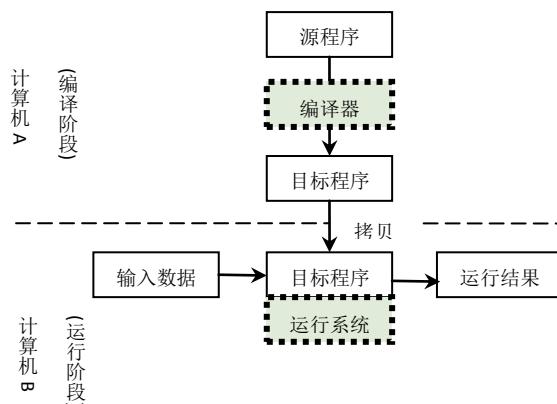


图 3.1 编译系统示意图

如果计算机 A 和计算机 B 属于不同的体系结构，那么这样的编译往往称为交叉编译。运行系统主要包含运行库、环境变量和一些配置文件等等。由于源语言程序中许多代码功能可能反复出现，因此可以先构建出具有这些常用功能的运行库，以此减轻编译时的负担。如果语义差距可以量化的话，图 3.2 表示了编译系统采用运行库和不采用运行库时的编译难度差别。

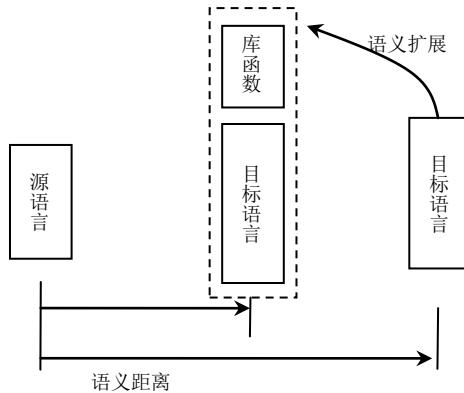


图 3.2 运行系统库函数的语义作用

从图 3.2 可以看出，运行系统的库函数相当于是目标语言的语义扩展，其功能强弱，直接影响到对源语言进行弥补语义差距的难度，因此设计一定的运行库将有利于简化编译器设计。

虽然大多数编译原理的课程重点都在讲述源语言到目标语言的语义差距的消除，并讲述了实现上述功能所需的词法分析、语法分析、中间代码生成和最终输出目标代码，但是实际应用中的编译器，大多都是以带有运行库这种形式来实现的，因此库的设计是可运行的编译系统的另一个重点。

编译器的设计必须和运行库的设计同时进行，或者说目标语言和运行库功能直接影响编译器的设计。

3.1.2 目标语言

下面我们需要确定 OpenMP 编译的目标语言是什么。从源语言到目标语言的编译过程可以经过多个中间语言的步骤来完成的，各个步骤可以集中处理本阶段的问题。比如 GCC 编译器对 C 语言的编译过程分为四步：

- 1、预处理（Preprocess）工作：处理宏定义，不管是通过-D 参数指定，还是在源码内部通过#define，或者使用了标准库、扩展库中的宏，都会替换为定义的值。
- 2、编译（Compile）阶段：将源代码（也就是预处理过的代码）编译为特定机器的汇编语言。
- 3、汇编（Assemble）阶段：将汇编源码汇编为机器码内容。此处如果有对外部的函数调用，则会预留未定义的地址以供最后一步链接阶段来填写。
- 4、链接（Link 阶段）：将链接对象文件链接为可执行文件，此过程比较复杂，需要链接很多外部的库文件，包括静态的，动态的。

从大的步骤上看，可以说第一步是将 C 语言程序编译成汇编语言程序，然后再将汇编语言程序编译成机器码。这是一个典型的分级编译的例子。

OpenMP 编译也可以分成多个步骤，在各个步骤完成不同的编译任务。我们将几种可能的编译实现方案用图 3.3 表示。

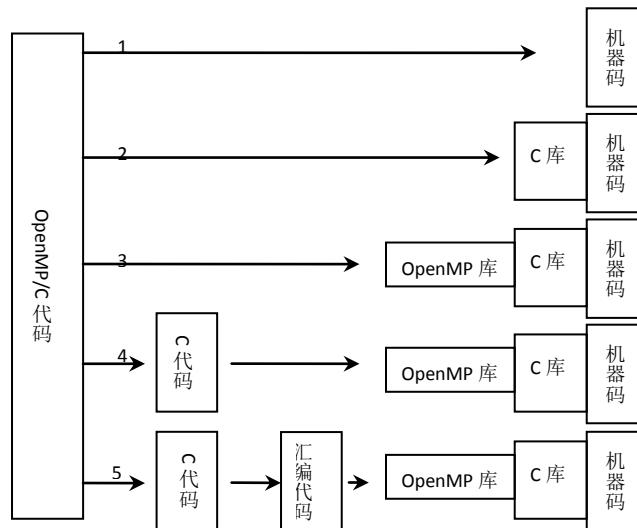


图 3.3 多种实现方案示意

前三种方式仅仅是在库的功能强弱上有所不同，但是编译目标语言都是机器码，也就是说编译器需要跨越“OpenMP 编译制导+ C 语言”到机器码的语义距离。第四种方式首先消除 OpenMP 编译制导的语义差距输出 C 代码，然后再将 C 代码编译成机器码，此时 OpenMP 制导指令的编译和 C 语言的编译任务各自独立开来。第五种方式则将 C 代码编译再进一步划分。后面两种实现方案的安排有利于分析 OpenMP 编译自身的问题。

本书主要讨论 OpenMP 的编译。常规 C 代码的编译已经在编译原理的课程中有详细的论述，因此下面将着重第四、第五种实现方案。这时候 C 代码的编译可以借助于现有的多种编译器。如果采用 GCC 作为其中的 C 语言编译器的话其过程将是第五种方案（虽然 GCC 本身具备有 OpenMP 编译能力，可通过命令行参数上的编译开关“-fopenmp”打开相应的编译能力），可用图 3.4 描述其过程。

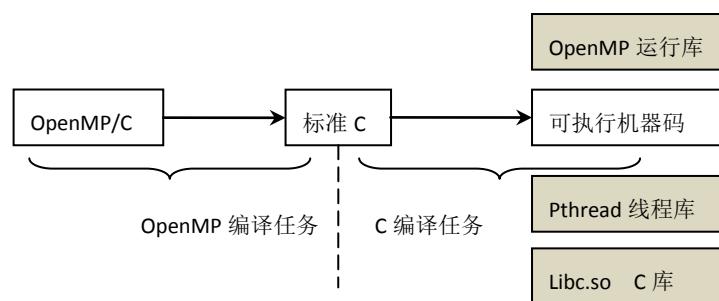


图 3.4 OpenMP 编译与 C 编译任务划分

这样一来，OpenMP 编译的目标代码为标准 C 代码，这些 C 代码再经过 C 编译器生成机器码形式的可执行文件，这个可执行文件运行中还需要 OpenMP 运行库等运行环境的支持。此时 OpenMP 编译在本书中称为狭义的 OpenMP 编译，如果没有特殊声明，后续章节中提到的 OpenMP 编译都是指这个概念。

3.2 OpenMP 编译器结构

本书狭义的 OpenMP 编译器的功能是将带有 OpenMP 编译制导的 C 代码翻译成标准 C 代码，因此它和所有编译器一样具有相似的结构，即有典型的八个部件构成，它们分别是词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成、信息表管理和错误处理，构成如图 3.5 所示的系统。

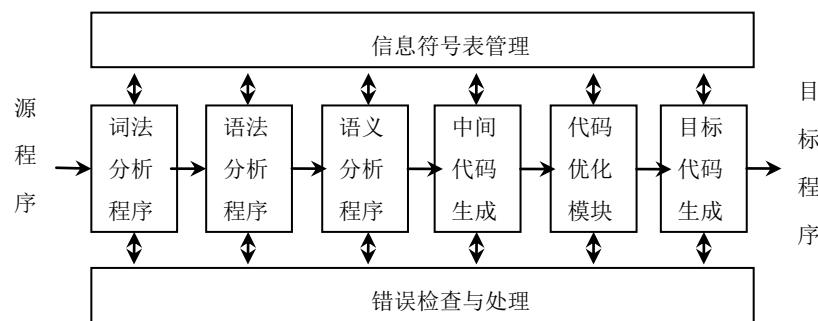


图 3.5 编译器通用结构

从逻辑上，上面的各个功能模块完成了编译过程中的一个步骤（phrase），每个步骤都将源程序从一种表示输出为另一种表示。下面就这八个功能模块和执行流程分别进行简单的分析讨论。

3.2.1 功能模块

除了符号表管理和错误处理在系统各处分散存在外，其他功能模块都是比较独立的。

词法分析程序

词法分析（Lexical analysis）或扫描（Scanning）是编译器最前端的输入模块，它将源代码文件中的一个长长的字符串，逐个识别为有意义的词素（Lexeme）或单词符号，并转变为便于内部处理的格式来保存。

通常词法扫描器的工作任务有：识别出源程序中的各个基本语法单位（通常称为单词或语法符号）；删除无用的空白字符、回车字符以及其他与语言无直接关系的非实质性字符；删除注释行；进行词法检查并报告所发现的错误。

对于识别出来的单词，可以用形如（Class, Value）的二元式来表示，其中 Class 是一个整数代码用来指示该单词的类别，Value 则是单词的取值。

由于 OpenMP 制导指令里面的关键词（保留字）和 C 语言的关键词有重叠的地方，比如“if”既是 C 语言条件语句中的关键字，又是 OpenMP 编译制导中的条件子句的关键字。因此在词法分析过程中，实际上要同时分析 C 语言单词和 OpenMP 编译制导的单词，并进行甄别。

OpenMP 制导指令中的关键词包括 `pragma`、`omp`、`parallel` 等等，数量并不多，因此 OpenMP/C 编译器的词法分析和普通 C 语言的词法分析大体相同。

语法分析程序

语法分析（Syntax Analysis）或解析（Parsing）程序需要借助于前面的词法分析，将词法分析输出的内部编码格式表示的单词序列尝试构建出一个符合语法规则的完整语法树。如果无法成功建立起一个合法的语法树，则由错误处理模块输出相应的语法错误信息。这棵树的叶子节点就是源程序中的各个单词，中间的节点则是程序设计语言的相关语法构造名。产生语法树的过程可以大致分为自顶向下和自底向上两大类。现有的最常用的两种方法是分别属于这两大类的递归下降法和算符优先法。前者根据语言中各语法范畴有文法递归定义的特点，用一组相互递归的子程序来完成语法分析；后者则利用各个算符间的优先关系和结合规则来制导语法分析，因而特别适合于分析各种表达式。这两种方法比较简单便于实现，许多编译器都或多或少地采用过这两种方法——对于表达式可采用算符优先分析法，对于语言的其他部分可以采用递归下降分析法。

对 OpenMP/C 程序代码的分析工作，需要根据 OpenMP 的语法规则和 C 语言的语法规则的相关文法描述来进行。通常的编程语言可以用前后文无关文法（CFG）或与之等价的 Backs-Naur 范式（BNF）来描述，从而整个语法分析过程能够按照此种描述机械地进行，所以可以直接借用工具软件来完成。

对于前面提到的语法树，可以真的用一个树形结构将所分析的语法成分表达出来，也可以是按照这个树形的语法树思路逐步进行语法分析而不用建立起整个语法树。

语义分析程序

语义分析（Semantic Analysis）是在前面进行了语法分析后，接着需要对编程语言的第二个特征属性——语义特征进行分析。语法特征描述的是各个语法元素之间的连接形式或结构，语义特征表征的是各个语法成分的含义和功能，包括这些语法元素的属性或执行时应进行的运算或操作。例如对 OpenMP 编译制导指令行的语法分析结束后，只是获得了相应的元素类型和排列顺序，但是具体是需要生成并行域还是需要进行同步操作，则是语义分析的范畴。

由于至今还没有找到一种公认的方法来系统地描述编程语言的语义，实际上使用的方法是：采用一种半机械化的方法来解决语义分析方面的问题。当前主流的方法是一种称为“语法制导翻译”的方法，这种方法把编译程序的语法分析和语义分析有机的结合起来同时完成。如果使用 Yacc 语法分析工具，可以方便地将语义动作结合进语法分析过程中。

OpenMP 编译过程中需要建立的 AST 中间表示就是在语法分析过程中结合语义动作来建立的。

中间代码生成

首先需要注意区分此处的中间代码与前面的分级编译中的“中间语言”的区别。虽然前面 3.1.2 小节中的 C 语言和汇编语言从广义上讲也是某种形式的“中间代码”，但是它们都是明

确的可使用的真实语言并属于编译器的输出目标代码。而此处说的中间代码指的是编译器未输出目标代码之前在内部使用的一种源代码的等价表示。

中间代码的生成是与语义分析紧密联系的。但是由于迄今为止未有公认的形式化系统来描述编程语言的语义，所以对中间代码的生成工作仍需要在一定程度上凭借经验来完成。如果采用语法制导翻译的编译器，通常做法是将产生中间代码的工作交给语义过程来完成。每当一个语义过程被执行以便对相应的语法结构进行语义分析时，它就根据此语法结构的语义，并结合在分析时获得的语义信息，产生相应的中间代码。目前常见的中间代码形式有逆波兰表示、三元式、四元式以及树形结构等等。因为树形的 AST 保留了源代码的语法层次结构，作为中间表示在源代码变换上具有优势。

这种先将源程序翻译为某种形式的中间代码，然后再将其翻译为目标代码的方法，可以使编译程序前后两部分功能更加单一，逻辑结构更加清晰，从而使得编译程序更易于编写与调整，也为代码优化和编译器移植创造了条件。通常将中间代码的生成作为编译器的前后端分界线，将中间代码生成之前的部分称为编译器的前端（Front End），中间代码生成之后的部分称为后端（Back End），这两部分可用图 3.6 表示。前端需要将源程序分解为多个独立的组成要素，并在这些要素上重建语法结构并输出其中间表示，其间收集的有关信息需要存放到符号表中。后端根据中间表示和符号表来构造和输出目标代码。

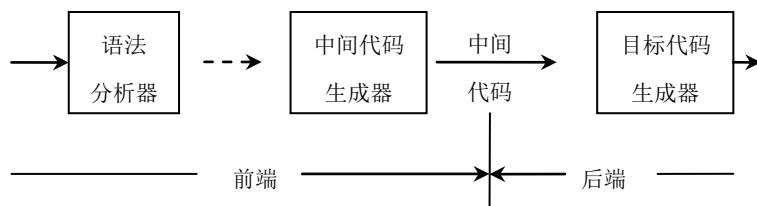


图 3.6 编译器前后端划分

如果能将编译器划分成前端和后端两个相对独立的部分，并以中间代码作为其信息交流的载体的模式来构建编译器，将会在编译程序的开发和维护带来许多好处。有些编译器集合是围绕一个精心设计的中间表示形式而创建的多个编译器，这些中间表示形式使得我们可以把特定语言的前端和特定目标机的后端相结合。使用这些编译器前端集合，加上某个目标主机的后端，可以为多种源语言建立该主机体系结构上的编译器。反过来可以一个语言的前端，配合上多种主机体系结构的后端，可以建立起同一种源语言在不同机器上的编译器。例如 Java 语言就是采用这种思想来实现的，对各种不同的计算机平台上的可执行文件都是相同的字节码（Byte Code）形式的中间表示，而 Java 虚拟机在解释（可以理解为逐条编译）和编译执行这些字节码的时候则是根据不同的底层体系结构而作不同的翻译工作，即一个前端对应多个后端。而 GCC（GNU Compiler Collection）则是多个前端对应一个后端的典型，GCC 中的 C、C++、FORTRAN 等多种语言的编译器前端是不同的，但是它们都输出一种所谓的抽象语法树 AST（Abstract Syntax Tree）形式的中间代码，然后共用统一的后端对 AST 形式的中间代码进行综合处理，最终产生相应的目标代码。有时也将前端和后端分别称为分析部分（Analysis）和综合部分（Synthesis）。

先将 OpenMP/C 程序分析成中间表示，然后在后端进行综合输出，可以采用不同的后端技术，有利于优化选择。另外，从宏观的角度上看，采用 C 作为中间语言，则可以将 OpenMP/C 程序的 OpenMP 前端配合上多种不同的 C 编译器后端。由于是源代码到源代码的变换，因此中间表示选用树形 AST 作为中间表示能有效分隔前后端，也更有利于翻译变换和输出。

代码优化程序

代码优化是为了提供更高质量目标代码，该工作常常在中间代码生成和目标代码输出之间插入一个代码优化处理的阶段来实现。根据目标代码的目标期望不同，优化方法也相应不同，有的是以运行时间为标准越快越好，有的是以存储空间开销为标准占用内存越少越好。优化工作可以分为与机器体系结构相关的优化和与机器体系结构无关的优化。

进行代码优化的代价是增加了编译程序本身的时间复杂度和降低可靠性（系统可靠性随复杂度上升而下降）。对速度的优化目标可能会不利于存储空间的优化目标，因此对各项优化目标应当进行权衡。如果只是对 OpenMP 程序进行源代码到源代码的变换，可以采用的编译优化技术并不多，很多传统编译器优化技术（特别是体系结构相关的优化技术）并不能在这里应用。但是对于运行环境，其可优化的地方就较多，例如对于 NUMA 架构的机器，OpenMP 线程的数据分配以及线程对处理器核的绑定等等，都是当前性能优化的热点。

目标代码生成

目标代码的生成是以语义分析（也可能加上优化处理）产生的中间代码作为输入的，它将中间代码翻译为最终形式的目标代码。这个翻译转换过程，需要确定源语言的各种语法成分（或中间代码的各种结构）对应的目标代码结构，该结构称为“框架”。框架比较固定，它是用目标代码形式描述的，但是有某些待定部分，需要在生成具体的目标代码时，根据各语法成分的确切形式和参数来确定的。对框架代码的要求做到：目标代码尽可能简洁、有较高的执行效率。由于我们选定 C 语言为 OpenMP 编译的目标代码，所以也要求相应的 C 框架代码要做到简洁和高效。

狭义的 OpenMP 编译只是源代码级的变换，因此编译原理课程中讲述的目标代码生成技术并不完全适用。例如为了生成可执行文件时，代码生成中的三个主要任务：指令选择、寄存器分配和指派、指令排序，在源代码转换工作中都不存在。

由于 OpenMP 并行语义是串行的 C 语言中没有的特性，因此需要在目标代码产生过程中进行翻译变换，通过源代码变换的同时结合底层的线程库来实现 OpenMP 的并行语义，这是 OpenMP 编译的翻译变换重点所在。本书的第 6 章、第 7 章和第 8 章将针对 OpenMP 并行语义的问题：并行域管理、任务分担以及变量数据环境进行分析讨论，这些部分构成了 OpenMP 编译的核心，其余章节与串行语言的编译没有本质区别。

信息表管理程序

在编译过程中总是需要收集、记录或查询源程序中出现的各种量的有关属性信息，因此编译程序需要建立和维护多个不同用途的表格（例如常数表、变量名、循环层次等等），这些表格统称为符号表。在编译过程中，造表和查表工作由一系列程序（或函数）来完成，它们并不是独立的存在而是安插在编译程序的相关功能代码中。

错误处理

由于编程人员不可避免的会写出有错误的代码，一个可用的编译器必须能够发现大多数常见错误，并能准确地报告出错误在源代码中的位置，否则就没有使用价值。由于编译系统的各个部分都可能需要程序来诊断问题，所以错误处理代码是广泛分布于编译器的各个角落，在需要的地方进行检查和诊断并报告错误所在。

3.2.2 工作流程

虽然典型编译器都有八个功能部件，但是图 3.5 所表示的只是它们的逻辑组织方式，并不代码执行上的先后顺序。在一个特定的实现中，多个步骤的活动可以被组合成一“遍”(Pass)或称为一“趟”的执行单位，每遍读入一个输入文件并产生一个输出文件。比如前端步骤中的词法分析、语法分析、语义分析，以及中间代码生成可以被组合成一个前端遍；代码优化可以作为一个可选的遍；最后是为特定目标机生成代码的后端遍。

3.3 编译优化

采用标准 C 代码作为输出的源代码级别的 OpenMP 编译虽然具有良好的可移植性和平台无关性，但是这种方案将 OpenMP 编译与后端的优化编译器分割开来，因此限制了线程级并行与指令集并行性开发之间的交互，使得很多优化方法无法实施。如果采用另一种实现方式，将 OpenMP 的翻译作为后端工作的一部分（牺牲了可移植性），此时由于 OpenMP 翻译、优化与面向指令级并行性的编译和优化可以共同在中间表示的层面上开展，可以灵活安排优化的相对顺序，也可以方便地在各种不同的分析、优化过程之间传递信息，因此可以获得比源代码级翻译更多的优化机会。但是即便采用源代码级的翻译，也可以在并行域的合并、冗余制导指令消除以及针对变量的数据属性进行优化。

另外在与运行库方面，也有许多优化技术，比如采用开销更小的线程库、针对 NUMA 架构利用局部性原理对数据分配和线程与处理器的绑定等等技术来提高性能。

本书作为入门级参考材料，并没有对性能优化进行讨论，需要读者自行阅读相关论文和书籍。

3.4 小结

本章介绍了 OpenMP 编译的狭义定义，简单分析了相应的编译器的基本构件：词法分析模块、语法分析模块、语义分析模块、中间代码生成、代码优化模块、目标代码生成以及符号表管理和错误检查。其中词法分析中要注意 OpenMP 与 C 语言共用关键字的区别，语法分析程序需要能在 C 语法规则之上识别 OpenMP 制导指令的语法，中间代码选取 AST 以便保留源代码的语法层次结构，目标代码生成中需要翻译 OpenMP 的并行语义。通过简单的介绍，在复习编译原理的基础之上初步了解 OpenMP 编译所涉及的几个问题，形成初步的概念。

第 4 章 词法与语法分析

在编译器的前端产生出中间代码的第一遍（Pass）步骤中，OpenMP 编译器和其他编译器很相似，就是通过词法分析和语法分析建立起抽象语法树 AST 中间代码表示，词法分析和语法分析的原理可以参考编译原理的教材，本章只讨论如何利用工具来实现 OpenMP 的词法分析和语法分析。

我们将基于两个开源工具 Lex 和 Yacc 来讨论，而该工具的原理以及可以适用的文法类型可以参考编译原理的书籍。在进行语法分析的同时可以编写相应的语义动作函数从而建立起 AST（其中又涉及 AST 树节点的设计问题），这些问题将在第 5 章讨论。

Lex 和 Yacc 是 UNIX 类操作系统中两个非常重要的、功能强大的工具。Lex 代表 Lexical Analyzer，Yacc 代表 Yet Another Compiler Compiler。如果能熟练掌握 Lex 和 Yacc，利用它们强大的功能可以非常容易地创建 FORTRAN、C 或其他编程语言的编译器。如果没有这样的工具，在开发程序的过程中遇到文本解析的问题时，例如解析 C 语言源程序、编写脚本引擎等等，就需要自己手动用 C 或者 C++ 直接编写解析程序，这对于简单格式的文本信息来说，不会是什么问题，但是对于稍微复杂一点的文本信息的解析来说，手工编写解析器将会是一件漫长痛苦而容易出错的事情。如果可以通过某种格式的文件来描述其词法或语法规则，再由工具软件来自动生成分析工具，那么将极大提高开发效率减少编程错误。

Lex 指词法扫描器，Yacc 指语法分析器，这是通用的说法。但是具体的实现将会有所不同，GNU 的 Lex 就是 Flex，GNU 的 Yacc 就是 Bison。为了统一，所以在后面的章节就只会用 Lex 来表示词法扫描器，用 Yacc 来表示语法分析器。

4.1 Lex 工具

本节要讨论编写某种的语言的编译器所用到的第一种工具——Lex，内容包括正则表达式、声明、匹配模式、变量，然后在后续小节继续讨论 Yacc 语法和解析器代码、以及如何把 Lex 和 Yacc 结合起来。

Lex 是一种生成词法扫描器的工具，扫描器是一种识别文本中的词汇模式的程序。这些词汇模式（或者正则表达式）定义在一种特殊的句子结构中。当 Lex 生成的扫描器在扫描接收到文件或文本形式的输入时，它试图将文本与正则表达式进行匹配。它一次读入一个输入字符，直到找到一个匹配的模式，Lex 就执行相关的动作（可能包括返回一个标记）。另一方面，如果没有可以匹配的正则表达式，将会停止进一步的处理，Lex 将显示一个错误消息。

因此需要对某种语言的待编译的源代码进行词法分析时，开发者首先可以用一个名为*.lex 的 Lex 文件（Lex 文件具有 .lex 或 .l 的扩展名）来描述其词法规则。这个 Lex 文件通过 Lex 公用程序（称 Lex 编译器）来处理，并生成 C 的输出文件。输出的 C 文件可以被编译为词法分析器的可执行版本。所以在词法分析中用户的主要工作就是编写 Lex 文件将词法规则描述清

楚，剩下的工作由 Lex 编译器完成，直到提供出一个可用的词法分析器的 C 文件。OMPI 利用 Lex 来完成 C 和 OpenMP 的词法分析，其工作流程如图 4.1 所示。

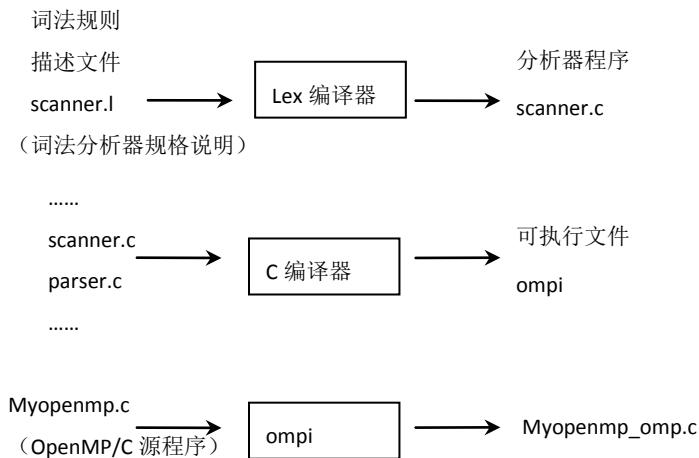


图 4.1 OMPi 中 Lex 的作用示意图

OMPI 的词法规则写在 scanner.l 文件（Lex 文件）中，经过 Lex 的编译输出为 scanner.c 文件，由于 OMPi 没有将 scanner.c 设计成独立运行的程序，因此它是和其他源代码一起工作的，由语法分析程序 parser.c 所调用。词法扫描器和语法分析器的功能集成在应用程序 ompi 中，ompi 不仅完成词法分析还完成语法分析等其他编译功能，图 4.1 例子 ompi 将输入 OpenMP/C 源文件 Myopenmp.c 作为输入直到输出编译后的目标代码 Myopenmp_comp.c，完成源代码到源代码的转换。也就是说它的词法分析输出结果对用户并不可见，如果需要输出结果可见，可以另行输出。

4.1.1 Lex 的正则表达式

由于开发者最重要的工作就是描述好词法规则，所以先来讨论这种统一的描述方法。对词法规则的描述正是通过表达式规则来说明的，所使用的正则表达式(Regular Expression)是一种使用元语言的模式描述。表达式由符号组成，符号一般是字符和数字，但是 Lex 中还有一些具有特殊含义的其他标记。表 4.1 给出了 Lex 的一些表达式规则。

表 4.1 Lex 的正则表达式

字符	含义
A-Z, 0-9, a-z	构成了部分模式的字符和数字。
.	匹配任意字符，除了 \n。
-	用来指定范围。例如：A-Z 指从 A 到 Z 之间的所有字符。
[]	一个字符集合。匹配括号内的 任意 字符。如果第一个字符是 ^ 那么它表示否定模式。例如: [abC] 匹配 a, b, 和 C 中的任何一个。
*	匹配 0 个或者多个上述的模式。
+	匹配 1 个或者多个上述模式。
?	匹配 0 个或 1 个上述模式。
\$	作为模式的最后一个字符匹配一行的结尾。

{}	指出一个模式可能出现的次数。例如: A{1,3} 表示 A 可能出现 1 次或 3 次。
\	用来转义元字符。同样用来覆盖字符在此表中定义的特殊意义, 只取字符的本意。
^	否定。
	表达式间的逻辑或。
"一些符号"	字符的字面含义。
/	向前匹配。如果在匹配的模版中的"/"后跟有后续表达式, 只匹配模版中"/"前面的部分。如: 如果输入 A01, 那么在模版 A0/1 中的 A0 是匹配的。
()	将一系列正则表达式分组。

扫描器的开发者可以利用上述规则来描述目标语言的词法规则, 表 4.2 给出了几个描述词法模式的例子。

表 4.2 正则表达式举例

正则表达式	含义
joke[rs]	匹配 jokes 或 joker。
A{1,2}shis+	匹配 AAshis, Ashis, AAshiss, Ashis 等。
A([b-e])?	匹配在 A 出现位置后跟随的从 b 到 e 的所有字符中的 0 个或 1 个。
[\t]*#"[\t]*"pragma"[\t]+"omp"[\t]+	匹配形如#pragma omp 的串

另外, 还可以在 Lex 文件中声明标记, Lex 中的标记声明类似 C 中的宏定义, 方便用户编程使用。每个标记都有一个相关的表达式。表 4.3 中给出了标记和表达式的例子。在后面我们以一个字数统计的程序为例说明 Lex 编程的时候, 其中就要用到这里讲到的如何声明标记的方法。

表 4.3 标记声明举例

标记	相关表达式	含义
number	([0-9])+	数字, 1 个或多个数字
chars	[A-Za-z]	字符, 任意字母字符
blank	""	空格, 一个空格
word	(chars)+	字, 1 个或多个 chars
variable	(字符)+(数字)*(字符)*(数字)*	变量名

从表 4.3 中可以知道此处定义了一个名为 number 的标记, 它代表了“数字”表达式——由 1 个或多个数字字符组成。同理也可以知道 chars 标记代表有字母字符所组成的字符串。

4.1.2 Lex 使用方法

使用 Lex 开发扫描器的编程过程可以分为三步: 首先在某个文件中以 Lex 可以理解的格式指定模式及相关的动作; 然后在这个文件上运行 Lex, 生成扫描器的 C 代码; 最后编译和链接 C 代码, 生成可执行的扫描器。如果扫描器是用 Yacc 开发的解析器的一部分, 只需要进行第一步和第二步。

现在需要看一看 **Lex** 可以理解的程序文件格式，一个 **Lex** 程序根据其作用可以分为三个段，形式如下：

声明部分

%%

规则部分

%%

辅助函数

这些段以 %% 来分界，第一段是声明段，有 C 和 Lex 的全局声明；第二段包括模式和动作(C 代码)；第三段是补充的 C 函数。如果作为独立扫描器使用，那么第三段中需要有 main() 函数。下面将利用 **Lex** 工具来生成一个用于字数统计的程序，并以此来分析 **Lex** 程序不同段的构成。

C 和 Lex 的全局声明

这一段中可以有 C 变量声明。以字数统计程序为例，需要声明一个整型变量，来保存程序统计出来的字数，后面还将进行 **Lex** 的标记声明。

字数统计程序的声明段：

```
1.      %{  
2.          int wordCount = 0;                      2字数统计值  
3.      %}  
  
4.      chars    [A-Za-z\ \.\"]  
5.      numbers ([0-9])+  
6.      delim    [\" \n\t]  
7.      whitespace {delim}+  
8.      words     {chars}+  
9.      %%
```

C 的声明部分起始于 "%{" 符号，终止于 "}" 符号，其间可以是包括 include 语句、声明语句在内的 C 语句。此例子中声明了一个初始值为 0 的用于字数统计值的整型变量 **wordCount**。Lex 的标记则不需要额外标注，此处声明了 5 种标记，都使用正则表达式，它们将在第二部分中被用到。第一个 **chars** 标记是字母、下划线、“.” 和引号，第二种是数字组成的字符串，第三种是分割符号，第四种是一个或多个分割符号、第五种是单词（1 个或多个 **chars**）。两个百分号标记指出了 **Lex** 程序中这一段的结束和第二段的开始。

Lex 的模式匹配规则

Lex 描述所要匹配的标记的规则，它出现在第二段。我们将使用 C 来定义标记匹配后的动作。下面是字数统计程序中的 Lex 标记匹配的规则。

```
10.     {words}      { wordCount++; /* increase the word count by one */ }  
11.     {whitespace} { /* do nothing */ }  
12.     {numbers}    { /* one may want to add some processing here */ }  
13.     %%
```

² 书中的代码中出现的中文，是作者所加的注释。

这里有三个项，每个项说明一个匹配规则和相应的动作。这里的匹配规则是利用了第一部分定义的标记，因此都用{...}括起来，当然更常见的是直接用正则表达式描述匹配规则。每个项的匹配规则后的动作也用一个{...}括起来。动作可以是在{...}里面的若干语句序列，也可以是调用第一段、第三段里面声明的函数。由于当前例子是用于字数统计的，所以每当匹配一个 words 标记则执行 “wordCount++” 从而实现字数统计的目的，而对于 whitespace 和 numbers 标记的匹配则不作任何事情。

辅助 C 代码

Lex 有一套可供使用的函数（其中必需之一就是 yywrap 函数）和变量。字数统计程序的第三段，也就是最后一段包含 C 的函数声明（有时是主函数）和实现。注意这一段必须包括的 C 代码段如下：

```
14. void main()
15. {
16.     yylex(); /* start the analysis*/          执行词法分析
17.     printf(" Number of words: %d\n", wordCount);    输出字数统计值，完成任务
18. }

19. int yywrap()
20. {
21.     return 1;
22. }
```

到这里已经初步讨论了 Lex 编程的基本元素，它足以编写简单的用于单词统计的词法分析程序，从该例子的分析中可以了解各部分的功能和作用。

4.2 OpenMP/C 的词法分析

OpenMP/C 的词法扫描需要处理 C 语言单词和 OpenMP 制导指令。OMPI 词法分析的关键是 scanner.l 和 scanner.h 文件。其中 scanner.l 经过 Lex 工具产生出 scanner.c 文件，这个 scanner.c 和 scanner.h 并不需要编译成可执行文件，而是以源代码的形式被后面的语法分析程序 parser.c 所使用。

由前面讨论已知 scanner.l 将由三部分组成的，其中第二部分要全面描述 C 语言和 OpenMP 编译制导指令的词法规则（或称为模式匹配规则）。

scanner.l 或 scanner.c 扫描的单词返回给 Yacc 的时候，是根据 parser.h 中的符号宏定义（例如 “#define TITLE 258”）来返回符号的。而 parser.h 是 bison 根据 parser.y 里面的 C/OpenMP 的 token 定义自动编号而来的。

4.2.1 C 语言单词

在 scanner.l 的第二部分中，既有 C 语言的符号也有 OpenMP 的符号，需要综合考虑。对于 C 语言的单词处理，其方法比较直接，大多数形如：

```
"int"           { count(); return(INT); }
```

或

```
{L}({L}|{D})* { count(); return( IdentOrType() ); }
```

它们的动作函数就是简单的返回对应的符号，例如遇到匹配“int”的字符串，那么返回符号 INT，如果遇到以字母开头、后面跟着 0 个到多个字母或数字的字符串，那么根据 IdentOrType() 函数的返回值判断为标识或类型。

4.2.2 OpenMP 单词

当遇到“section”这类 OpenMP 单词时，处理则略复杂一些。如果这样的单词不是出现在 OpenMP 制导指令行中，那么也只是当作普通 C 语言的单词来处理。所有其他 OpenMP 的关键字都相似，必须出现在 OpenMP 制导指令行中才有效。

```
1. "section" {
2.         count();           处理扫描行/列计数值
3.         if (on_omp_line)   如果是在 OpenMP 制导指令行中
4.             return(OMP_SECTION); 作为 omp 关键词返回
5.         else
6.             return IdentOrType(); 作为普通 c 语言标识符返回
7.     }
```

第一项是词法模式匹配规则，扫描时发现“section”字符串则匹配。第二项用{...}括起来是相应的动作。如果在 omp 的编译制导行中 (on_omp_line 为 1)，则返回具体 OpenMP 符号 OMP_SECTION，否则按 C 语言的标识符或者类型说明符返回（根据 IdentOrType() 的返回值）。其中 on_omp_line 是 OpenMP 制导指令行的标志，在扫描时发现“#pragma omp”或者“#pragma omp threadprivate”的时候设置的，只有两处：

```
1. [\t]*#"["[\t]*"pragma"[ \t]+"omp"[ \t]+ {
2.                                     count();
3.                                     on_omp_line = __has_omp = 1;
4.                                     return (PRAGMA_OMP);
5.                                 }
6.   [\t]*#"["[\t]*"pragma"[ \t]+"omp"[ \t]+"threadprivate"[ \t]* {
7.                                     count();
8.                                     on_omp_line = __has_omp = 1;
9.                                     return (PRAGMA_OMP_THREADPRIVATE);
10.                                }
```

也就是说只有在源代码中出现匹配的“#pragma omp”以及“#pragma omp threadprivate”时，才表明进入了 OpenMP 编译制导指令行的范围。那么同样，需要在退出编译制导指令行的范围时清除 on_omp_line 变量，这是在遇到回车符后清零的：

```
1. \n
2.         {
3.             count();
4.             if (on_omp_line)
5.             {
6.                 on_omp_line = 0; /* Line finished */
7.                 return ('\n');
```

```
7.          }
8.      }
```

也就是说编译制导指令不能跨行。

为了区分一个源代码中是否有 OpenMP 编译制导指令，可以用一个变量来标记。OMPi 使用 `_has_omp` 来标记是否有 OpenMP 制导指令，它在词法扫描的过程中设定的。该标志可以在后续代码变换的时候用到——如果未设置该标志，说明没有 OpenMP 语句，则不需要进行 OpenMP 的代码翻译。

4.2.3 OpenMP 与 C 语言公用单词

对于 `for`、`default`、`if`、`static` 这几个 OpenMP 制导指令和 C 语言语句中都会出现的单词，则根据它是否出现在 OpenMP 编译制导指令行中分别处理。`opmi` 是根据 `on_omp_line` 标志未来分别处理的，所以编译制导指令行以外的这类单词都按 C 的词法进行解释。

```
1. "if"           {
2.                 count();
3.                 if (on_omp_line)      在 OpenMP 制导指令行
4.                     return(OMP_IF);    返回 OpenMP 符号
5.                 else
6.                     return(IF);       返回 C 符号
7. }
```

在 OpenMP 制导指令行中返回 `OMP_IF` 符号，否则返回 C 语言符号 `IF`。虽然只是从字符串角度上无分区分它们，但是由于 `on_omp_line` 标志的存在，这两种符号还是可以区分开来的。

4.3 scanner.l

OMPi 使用 `scanner.l` 来描述词法分析器的规格说明，它由 OMPi 的开发者编写的（可以从 C 语言的 `Lex` 文件修改而来），然后交给 `Flex` 工具去生成词法分析器。下面对 `scanner.l` 的三个段作简要分析（按照文件的文本顺序进行），如果对具体的 `Lex` 编码不感兴趣可以跳过本小节。

4.3.1 全局声明段

`scanner.l` 文件的第一段是全局声明部分，下面将按照程序的顺序对代码进行编排，有些代码行没有展开分析。`scanner.l` 的开头是 6 个标记，其用途很直观。

1. D	[0-9]	数字
2. L	[a-zA-Z_]	字符
3. H	[a-fA-F0-9]	16 进制数
4. E	[Ee][+-]?{D}+	指数形式的浮点数
5. FS	(f F I L)	浮点类型
6. IS	(u U I L)*	整数类型

从下面的 “%{” 到下一个 “}%” 之间是 C 的变量声明部分，前面是一些头文件的说明。

```

7.    %{

8.    #include <stdio.h>
9.    #include <stdlib.h>
10.   #include <string.h>
11.   #include <ctype.h>
12.   #include "ompi.h"
13.   #include "ast.h"           /* For the yyval / ast types */
14.   #include "symtab.h"
15.   #include "parser.h"
16.   #include "scanner.h"

17. #define SCANNER_STR_SIZES 1024 /* All strings declared here are that long */

```

变量 `on_omp_line` 用于标识当前的扫描区域是否在 OpenMP 的制导指令行内，用于区分 OpenMP 和 C 公共的关键字。

```
18. static int on_omp_line = 0; /* Scanning an OpenMP #pragma line */
```

变量 `__has_omp` 用与标记扫描过程中是否发现有 OpenMP 编译制导指令，一旦发现有就置位。

```
19. int __has_omp = 0; /* True if we found at least 1 OpenMP #pragma */
```

下面的函数 `count()` 用于记录扫描过程中的行列计数值。`sharp()` 用于将#字符开头但不是 OpenMP 制导指令的行忽略掉（此时的`#include`、`#define` 等都已经被与处理程序展开，不再存在）。`gobbleline()` 用于将一行注释行（“//” 开头的行）忽略掉，`gobblecomment()` 用于将注释行（一行或多行）忽略掉。这些函数的实现将在 Lex 文件的第三段。

```
20. void count(), sharp(), gobbleline(), gobblecomment();
```

`IdentOrType()` 检查当前单词是不是 `TYPENAME` 里面的单词（使用 `symtab_get()` 函数），如果是则返回具体的 `TYPE_NAME`，否则按照 `IDENTIFIER` 返回。

```
21. #define IdentOrType() \
22.     ( symtab_get(stab,Symbol(yytext),TYPENAME) != NULL ? TYPE_NAME : IDENTIFIER )
```

由于词法扫描和语法分析使用了预处理后的文件，所以用下面的几个变量来跟踪原始文件名和原始文件的行号。预处理器会产生出类似于 “#1 “test.c”” 这样的标志，通过分析可以知道原始文件名为 `test.c`，原始行号是第 1 行。所以 `test.c` 将记录到 `origfile_name[]` 字符串中，而正在扫描的文件名会记录到 `thisfile_name[]` 字符串中。正在扫描的行号记录在 `thisfile_line` 变量中、`origfile_line` 变量中，而 `origfile_line` 是根据最近一次发现的标志（`marker_line` 的变量值）来推算获得的。`thisfile_column` 记录的当前扫描的列号。这些文件名和行号在指出编译错误的时候非常有用。

```
23. static char origfile_name[SCANNER_STR_SIZES]; // original file name
24. static char thisfile_name[SCANNER_STR_SIZES]; // the scanned file name
25. static int thisfile_line = 1, // Current line in our file (preprocessed)
```

```
26.         marker_line = 0,    // The line where the last marker was found
27.         origfile_line = 0; // Original file line the marker was referring to
28. static int thisfile_column = 0; // Column in the currently scanned line
```

变量start_token用于记录词法分析的初始规则。

```
29. static int start_token = 0; // For starting with a particular token
```

```
30. %}
```

```
31. %%
```

第一段和第二段的分割处

到此处“%}”出现的位置，C的声明结束，而%%出现的位置表示Lex文件的第一段结束、第二段的开始。

4.3.2 模式匹配规则段

第二段的一开始是关于初始规则的说明。如果已经设置过start_token变量，那么就将t取start_token的值，同时将0赋值给start_token避免再次设置。

紧接着初始规则说明后面的是OpenMP关键字的匹配规则、C语言关键字的匹配规则、标识符和常量匹配规则、运算符的匹配规则和分隔符的匹配规则。

```
32. %
33.     /* Trick to get an initial token (from the bison manual);
34.      * This is placed in the top of the produced yylex() function.
35.      */
36.     if (start_token)
37.     {
38.         int t = start_token;
39.         start_token = 0;           /* Don't do it again */
40.         return t;
41.     }
42. }
```

下面的两个规则用于忽略所遇到的C语言注释语句。当扫描器遇到“//”时，将调用gobbleline()函数直接跳到本行的结束处，忽略该位置到行末的所有内容。而遇到“/*”的时候，则调用gobblecomment()函数，忽略所有位于该符号和后面出现的“*/”之间的所有内容。所以注释行对于后面的parser而言是看不到的，因为它们不返回任何信息。这两个函数在scanner.l的第三段实现。

```
43.     "//"          { if (!on_omp_line) gobbleline(); }
44.     "/*"          { if (!on_omp_line) gobblecomment(); }
```

OpenMP 关键字

扫描过程中如果发现有“#pragma omp”或者“#pragma omp threadprivate”就说明代码中有OpenMP编译制导指令，此时设置__has_omp=1作为标志。

OpenMP的关键字只有在OpenMP编译制导指令行中才有效，如果“parallel”、“section”等单词出现在普通的C代码区域内只是作为普通的标识符或类型符（用IdentOrType()函数来区分标识符或类型符）。

这里需要能识别所有的OpenMP保留字包括（根据OpenMP版本差异会有所不同）：parallel、sections、nowait、ordered、schedule、dynamic、guided、runtime、section、single、master、critical、barrier、atomic、flush、private、firstprivate、lastprivate、shared、none、reduction、copyin、num_threads、copyprivate。

除了对“#pragma omp”或者“#pragma omp threadprivate”的扫描动作函数略有不同外，所有的扫描动作函数都做相似的事情：首先调用count()对扫描位置进行计数，然后判断是否在OpenMP制导指令行内，如果不是则调用IdentOrType()函数来区分标识符或类型符，如果是则返回相应的符号，这些符号包括：

OMP_PARALLEL、OMP_SECTIONS、OMP_NOWAIT、OMP_ORDERED、OMP_SCHEDULE、OMP_DYNAMIC、OMP_GUIDED、OMP_RUNTIME、OMP_SECTION、OMP_SINGLE、OMP_MASTER、OMP_CRITICAL、OMP_BARRIER、OMP_ATOMIC、OMP_FLUSH、OMP_PRIVATE、OMP_FIRSTPRIVATE、OMP_LASTPRIVATE、OMP_SHARED、OMP_NONE、OMP_REDUCTION、OMP_COPYIN、OMP_NUM_THREADS、OMP_COPYPRIVATE。

上面这些符号的宏定义在parser.h中，Lex通过在第一段C语言的#include “parser.h”包含此头文件，从而可以引用这些符号。

```
45.      /*
46.      * OpenMP tokens
47.      */
48.      [ \t]*#"["[ \t]*"pragma"[ \t]+"omp"[ \t]+ {                      #pragma omp 行
49.          count();
50.          on_omp_line = __has_omp = 1;      设置 OpenMP 标志
51.          return (PRAGMA_OMP);
52.      }
53.      [ \t]*#"["[ \t]*"pragma"[ \t]+"omp"[ \t]+"threadprivate"[ \t]* {    #pragma omp threadprivate 行
54.          count();
55.          on_omp_line = __has_omp = 1;      设置 OpenMP 标志
56.          return (PRAGMA_OMP_THREADPRIVATE);
57.      }
58.      [ \t]*#"["[ \t]*"line"  {
59.          sharp();                         删除无效 “#” 行
60.      }
61.      [ \t]*#""
62.      {
63.          sharp();                         删除无效 “#” 行
64.          "parallel"
65.          {
66.              count();
67.              if (on_omp_line)
68.                  return(OMP_PARALLEL);
```

```

68.                                else
69.                                return IdentOrType();
70.                            }
71. "sections"    {
72.                                count();
73.                                if (on_omp_line)
74.                                    return(OMP_SECTIONS);
75.                                else
76.                                    return IdentOrType();
77.                            }

.....
78. "copyprivate"   {
79.                                count();
80.                                if (on_omp_line)
81.                                    return(OMP_COPYPRIVATE);
82.                                else
83.                                    return IdentOrType();
84.                            }

```

上面的匹配规则对应的动作都很简单，基本上就是返回相应的符号。对于 OpenMP 和 C 公用的符号，则利用 on_omp_line 标记进行区分。

C 关键字

由于部分关键字在C语言和OpenMP编译制导指令之间有重叠，因此需要首先判断是否在 OpenMP 的制导指令行内，如果在 OpenMP 制导指令行内则按 OpenMP 的关键字处理，否则按 C 语言的关键字处理。

```

.....  

85. "auto"          { count(); return(AUTO); }           大多数符号可以直接返回  

86. "break"         { count(); return(BREAK); }  

87. "case"          { count(); return(CASE); }  

88. "char"          { count(); return(CHAR); }  

89. "const"         { count(); return(CONST); }  

90. "continue"      { count(); return(CONTINUE); }  

91. "default"       {                                         共用关键字需要甄别  

92.                                count();  

93.                                if (on_omp_line)  

94.                                    return(OMP_DEFAULT);  

95.                                else  

96.                                    return (DEFAULT);  

97.                                }  

98. "do"             { count(); return(DO); }  

99. "double"        { count(); return(DOUBLE); }  

100. "else"          { count(); return(ELSE); }  

101. "enum"          { count(); return(ENUM); }

```

```

102. "extern"           { count(); return(EXTERN); }
103. "float"            { count(); return(FLOAT); }
104. "for"               {
105.                   count();
106.                   if (on_omp_line)          公用的符号需要甄别
107.                       return (OMP_FOR);
108.                   else
109.                       return(FOR);
110.               }
111. "goto"              { count(); return(GOTO); }
.....
112. "while"             { count(); return(WHILE); }

```

下面的几个单词在分析动作中将返回特殊符号，有控制作用。

```

113.  /*
114.   * Hacks
115.   */
116. "__builtin_va_arg"   { count(); return(__BUILTIN_VA_ARG); }
117. "__builtin_offsetof" { count(); return(__BUILTIN_OFFSETOF); }
118. "__builtin_types_compatible_p" { count(); return(__BUILTIN_TYPES_COMPATIBLE_P); }

```

标识符、类型说明符、常量和字符串

除了前面的OpenMP和C的保留关键字以外，剩下的就是标识符、操作符和变量了。标识符或类型说明符必须是字母开头，后面跟若干个字母、数字或下划线。两者的区分工作是借助 `IdentOrType()` 函数完成的。

```

...
119. {L}{L}|{D}*           { count(); return( IdentOrType() ); }

```

常量包括数字和字符串两种。数字这里只有10进制和16进制，字符串就是被“”引号所包含的字母串。当匹配时，返回`CONSTANT`等符号。

```

120. 0[xX]{H}+{IS}?       { count(); return(CONSTANT); }
121. 0{D}+{IS}?           { count(); return(CONSTANT); }
122. {D}+{IS}?           { count(); return(CONSTANT); }
123. '(\\".|[^\\"])+'    { count(); return(CONSTANT); }

124. {D}+{E}{FS}?         { count(); return(CONSTANT); }
125. {D}*."{D}+{E}?{FS}? { count(); return(CONSTANT); }
126. {D}+."{D}+{E}?{FS}? { count(); return(CONSTANT); }
127. \"(\\".|[^\\"])*\"  { count(); return(STRING_LITERAL); }

```

操作符

此部分匹配规则说明是关于操作运算符的。只要符合匹配规则，那么就返回相应的符号编号。

```

128.    /*
129.     * Operators
130.    */
131.    ">="           { count(); return(RIGHT_ASSIGN); }
132.    "<="           { count(); return(LEFT_ASSIGN); }
133.    "+="           { count(); return(ADD_ASSIGN); }

134.    .....
135.    "=="           { count(); return(EQ_OP); }
136.    "!="           { count(); return(NE_OP); }

.....
137.    "?"            { count(); return('?'); }
138.    "..."          { count(); return(ELLIPSIS); }

```

分割符

分割符可以是常见的换行符、制表符等。对于OpenMP的编译制导行的结尾处出现换行符，那么on_omp_line标量将被清零。

```

139.    /*
140.     * Spaces, newlines etc.
141.    */
142.    [ \t\v\f]        { count(); }
143.    \n                {
144.        count();
145.        if (on_omp_line)
146.        {
147.            on_omp_line = 0; /* Line finished */
148.            return('\n');
149.        }
150.    }
151.    {/* ignore bad characters */}

152. %%
```

第二段和第三段的分割处

4.3.3 补充函数段

第三段不仅声明函数，更要实现它们。首先是必不可少的yywrap()函数，通常只是返回整数1。

```

153. int yywrap()
154. {
155.     return(1);
156. }
```

`sharp()`函数用于遇到“#”开头的行，且没有包含OpenMP编译制导指令时的动作函数，内部代码不在此讨论。

```
157. void sharp()
158. {
...
159. }
```

遇到C语言单行的注释行“//.....”则直接用`gobbleline()`处理掉，并且不返回任何东西。该函数简单地从当前扫描位置不断往后读入字符，直到出现换行符'\n'为止。然后更新当前扫描行的行号

```
160. void gobbleline()
161. {
...
162. }
```

遇到单行或多行的注释“/*.....*/”则调用`gobblecomment()`处理掉，不返回任何东西：

```
163. void gobblecomment()
164. {
...
165. }
```

`count()`里面需要跟踪扫描的行列号，也要处理制表符。`yytext[]`是当前扫描字符串，它是根据分割符在源代码中切割出来的。对于非空白单词，则拷贝到`yyval.name`中。

```
166. void count()
167. {
168.     int i, nonempty = 0;                      non empty 用于标记检查所扫描的单词是否为空
169.     for (i = 0; yytext[i] != 0; i++)           对本单词逐个字符检查
170.         if (yytext[i] == '\n')                  遇到换行符作为单词结尾
171.     {
172.         thisfile_column = 0;                   换行后列为 0
173.         thisfile_line++;                     换行后行号增 1
174.     }
175.     else                                     没有遇到换行符
176.         if (yytext[i] == '\t')                 遇到 tab 作为单词结尾
177.             thisfile_column += (8 - (thisfile_column % 8)); 跳到 tab 后面的列位置
178.         else                                 该单词直接结束，无换行符或 tab 符
179.     {
180.         thisfile_column++;                  每处理一个字符就列号增 1
181.         if (!isspace(yytext[i]))            只要有一个字符不是空白
182.             nonempty = 1;                  则该单词不是完全空白
183.     };
184.     if (nonempty)                          该单词不是完全空白
185.         strcpy(yyval.name, yytext);        将扫描单词 yytext 存放到 yyval.name 中
```

```
186. }
```

下面的函数将扫描器强制扫描一个字符串而不是扫描一个文件。

```
187. /* Set everything up to scan from a string */
188. void sc_scan_string(char *s)
189. {
190.     yy_scan_string(s);
191.     *origfile_name = 0;
192.     sc_set_filename("scanner_string_buffer");
193. }
```

下面是另外几个辅助函数。词法扫描器可以有不同的初始规则。初始规则的选择可以通过 `sc_set_start_token()` 来实现。这个函数根据传入的初始规则类型 `t` 来设置 `start_token` 变量，并且做好记录扫描位置的变量的初始化，将本文件的扫描行号 `thisfile_line` 赋值为 1，本文件的扫描列号 `thisfile_column` 赋值为 0，原始文件行号和相应的标记变量赋值为 0。

```
194. void sc_set_start_token(int t)
195. {
196.     start_token = t; thisfile_line = 1; thisfile_column = 0;
197.     marker_line = origfile_line = 0; }
```

给 `sc_set_filename()` 传入不同的文件名，可以设置被扫描的文件。该函数只是简单的将函数名拷贝到 `thisfile_name` 变量中。

```
198. void sc_set_filename(char *fn)
199. {
200.     strncpy(thisfile_name, fn, 255); }
```

下面的 `sc_original_file()` 函数根据是否存在有 `original_file` 字符串来返回文件名，如果有原始文件那么返回的是原始文件名，否则返回当前扫描的文件名。

```
201. char *sc_original_file()
202. {
203.     return (*origfile_name) ? origfile_name : thisfile_name; }
```

下面的 `sc_original_line()` 函数可以返回当前扫描行在原始文件中的行号。具体办法就是通过计算当前行与最近一次发现标记的行号 `marker_line` 之间的距离，加上标记行在原始文件中的行号。

```
204. int sc_original_line() { return(thisfile_line - marker_line + origfile_line); }
```

下面两个函数简单地返回当前行和列变量的值。

205. int sc_line()	{ return(thisfile_line); }	返回当前行号
206. int sc_column()	{ return(thisfile_column); }	返回当前列号

综合上面的分析，我们可以大致地将 `scanner.l` 负责的相关变量和函数分成几大类。

首先是关于初始规则的部分，它包括一个变量 `start_token` 用于记录初始规则的类型，`sc_set_start_token()` 用于设置选择初始规则。

其次是关于扫描位置跟踪的部分。扫描位置涉及当前文件和原始文件，因此相应的信息和操作函数也分成两部分。

当前位置的跟踪信息相关的变量包括： `thisfile_name[]` 用于记录当前扫描文件名， `thisfile_line` 用于记录当前文件中的行号， `thisfile_column` 用于记录当前文件中的列号。当前位置的跟踪信息相关的操作包括： `sc_line()` 和 `sc_column()` 分别用于获取当前行号为列号的操作， `sc_set_filename()` 用于设置当前文件名 `thisfile_name[]`， `count()` 用于记录和修改当前行号和列号。

原始文件的跟踪信息相关的变量有： `marker_line` 用于记录最近一次在当前文件中发现的标志所在的行号， `origianl_line` 用于记录最近发现的那个标记所记录的原始文件中的行号。原始文件的跟踪信息相关的操作有： `sc_original_file()` 用于获得原始文件名， `sc_original_line()` 用于获取当前扫描位置在原始文件中的行号。

第三部分是关于 OpenMP 出现的标志。相关变量有 `_has_omp` 用于记录当前文件是否出现了 OpenMP 指令， `on_omp_line` 用于记录当前行是否为 OpenMP 指令行。

第四部分是关于无效行的处理。包括： `gobbleline()` 忽略当前位置到行末的所有内容、 `gobblecomment()` 忽略所有被 /* 和 */ 括起来的注释内容， `sharp()` 用于忽略以 “#” 字符开头的但是没有 OpenMP 指令的空白编译制导行。

另外有 `sc_scan_string()` 等其他函数和变量。

可以用表 4.4 来做一个简单的归纳。

表 4.4 scanner.l 中的变量及函数归类

分类		变量	函数
初始规则		<code>start_token</code>	<code>sc_set_start_token()</code>
位置 跟踪	当前 文件	<code>thisfile_name</code> <code>thisfile_line</code> <code>thisfile_column</code>	<code>sc_line()</code> <code>sc_column()</code> <code>sc_set_filename()</code> <code>count()</code>
	原始 文件	<code>origfile_name</code> <code>original_line</code> <code>marker_line</code>	<code>sc_original_file()</code> <code>sc_original_line()</code>
	OpenMP 标志	<code>_has_omp, on_omp_line</code>	
无效行			<code>gobbleline()</code> <code>gobblecomment()</code> <code>sharp()</code>
其他			<code>sc_scan_string()</code> <code>IdentOrType()</code>

词法扫描器与语法扫描器是有交互的。当前单词是在执行 `count()` 的时候赋值到 `yyvar.name` 变量中的。 `yyin` 是扫描文件描述符， `yylex()` 是供调用的扫描函数。 `Yyin` 和 `yylex()` 对扫描器来说是外部变量和外部函数。

4.3.4 scanner.c

scanner.c 是 Lex 根据 scanner.l 产生的文件，它将 scanner.l 中用户编写的代码和声明的变量都转移过去，同时插入有大量 Lex 自动生成的代码。由于内容过于庞大不能在本书中列出，请读者自行阅读源代码。其中产生出来的代码中有一个 `yylex()` 函数，由语法分析器调用时 `yylex()` 将返回一个单词的符号。

4.3.5 scanner.h

scanner.h 文件被词法和语法扫描器的代码所使用，声明了扫描器的变量和函数。其中 `yyin` 和 `yylex()` 是 Lex 工具自己的内部变量和函数。`sc_set_filename(char *fn)` 用于设定被扫描的文件；`sc_original_file()`、`sc_original_line()`、`sc_line()`、`sc_column()`、`sc_set_start_token()` 被语法分析器所调用，用于不同的启动初始规则和不同的文件。`sc_scan_string()` 可以让扫描器临时强制扫描指定的字符串而不是扫描 `yyin` 输入文件。下面是 scanner.h 文件。

```
1. #ifndef __SCANNER_H__
2. #define __SCANNER_H__

3. #include <stdio.h>

4. extern FILE *yyin;           /* defined by flex */      扫描输入文件
5. extern int    yylex(void);          分析函数

6. /* Set this to the name of the file you are about to scan */
7. extern void   sc_set_filename(char *fn);        设定被扫描文件
```

`__has_omp` 用于标记是否出现 OpenMP 编译制导指令。

```
8. extern int   __has_omp;        /* True if > 0 OMP #pragmas where found */
```

下面四个函数用分别用于设置原始扫描文件、扫描行、当前行和列。

```
9. extern char *sc_original_file(void);      见 scanner.l 第 199 行
10. extern int   sc_original_line(void);       见 scanner.l 第 201 行
11. extern int   sc_line(void);                当前行号，参见 scanner.l 第 202 行
12. extern int   sc_column(void);              当前列号，参见第 203 行
```

该词法扫描器可以用于不同的目的，通过 `sc_set_starta_token()` 就可以启动不同的初始规则，实现不同的词法扫描的起点。

```
13. /* Special function to start scanning by returning some given token.
14.  * It is only used so that the parser can support mulitple start
15.  * symbols.
16.  */
17. extern void sc_set_start_token(int t);  用于设定不同的初始规则，见 scanner.l

18. /* Force the scanner to scan from the given string */
19. extern void sc_scan_string(char *s);      强制扫描指定的字符串
```

```
20. #endif
```

4.4 Yacc 工具

虽然只是用 **Lex** 也是可以完成语法分析的，但是需要编写复杂的 C 代码、维护大量用户定义的状态，以至于难以编写复杂的语法分析。如果存在复杂的结构、大量对上下文敏感的元素，那么 **Yacc** 是较好的选择。上下文敏感的元素是指在不同场合下有多种解释的单词或符号，比如 C 语言中的*，既可以是乘号（在两个表达式之间）也可以是间接访问（在指针变量前），如果只是用 **Lex**，用户需要维护当前状态是哪种才能正确处理。

在整个编译过程中 **Lex** 工具不是自己一次完成扫描，实际上它是和 **Yacc** 可以互动的，**Yacc** 在分析过程中根据需要向 **Lex** 要单词，也可以自己扫描输入文件，例如 “for(i=0;i<100;i++)” ，在语法匹配时，**Yacc** 向 **Lex** 要 for、i=0、i<100 和 i++，但是可以自己扫描 “(” 和 “)” 。实际上 **Lex** 返回“(”和“)”作为字符（编号小于 255），而所有被定义的标记都是取值大于 255 的。

4.4.1 Yacc

Yacc 代表 Yet Another Compiler Compiler，**Yacc** 的 GNU 版本是 **Bison**。它是一种工具，将任何一种编程语言的语法规则翻译成针对此种语言的 **Yacc** 语法解析器。它用巴科斯范式 (BNF, Backus Naur Form) 来书写源语言的语法规则。

在进一步阐述以前，让我们回顾一下什么是语法。在上一小节中，我们看到 **Lex** 从输入序列中识别标记。如果在查看标记序列时，可能想在某一序列出现时执行某一动作。这种情况下有效序列的规范称为语法。**Yacc** 语法文件包括这一语法规范，它还包含了序列匹配时编译器想要做的事。

为了更加说清这一概念，以英语为例。这一套标记可能是：名词、动词、形容词等等。为了使用这些标记造一个语法正确的句子，你的结构必须符合一定的规则。一个简单的句子可能是名词+动词或者名词+动词+名词（如 I care. See spot run.）。

所以在我们这里，标记本身来自语言（**Lex**），并且用 **Yacc** 来指定那些符合规范的标记序列，这种标记序列也叫语法。语法分析过程就是从初始符号开始，利用这些语法规则进行推导，经过各级非终结符号，最终推导出终结符号，从而形成一个符合语法规则的语法树。注意区分 **token** 和 **symbol**，**token** 是终结 **symbol**。

在 **Yacc** 文件里终结符号有三种类型：

- a) 命名标记：这些由 %token 标识符来定义。按照惯例，它们都是大写。
- b) 字符标记：字符常量的写法与 C 相同。
- c) 字符串标记：写法与 C 的字符串常量相同。例如，“<<”就是一个字符串标记。

Yacc 文件格式

如同 Lex 一样，Yacc 文件被%%分割成为的三个段，它们是：声明、语法规则和 C 代码。第一段中要写入 C/C++代码必须用%{和%}括起来；但是第三个段就可以直接写入 C/C++代码了，不需要任何的修饰；中间的那一段就是 Yacc 语法规则了。其文件格式如下：

```
1.    %{
2.        /* C declarations and includes */
3.    %}
4.        /* Yacc token and type declarations */
5.    %%
6.        /* Yacc Specification
7.           in the form of grammar rules like this:
8.        */
9.        symbol      :      symbols tokens
10.                   { $$ = my_c_code($1); }
11.                   ;
12.    %%
13.        /* C language program (the rest) */
```

第二段 Yacc 规则说明是将 Lex 提供的标记根据“语法”规则“粘合”到一起的地方。Yacc 语法规则说明了某种语言中标记的合法序列，一个符号可以有多种“合法”序列。

使用方法

通过 Yacc 来构造一个语法分析器的过程如下：

1. 在一个文件中说明语法：
 - a) 编写一个 .y 的语法文件（同时说明在这里要进行的 C 代码动作）。
 - b) 编写一个词法分析器来处理输入，并将标记传递给解析器。这可以使用 Lex 工具来完成，也可以自行实现一个 yylex()。
 - c) 编写一个函数，该函数通过调用 yyparse() 来开始解析。
 - d) 编写错误处理例程（如 yyerror()）。
2. 通过在语法文件上运行 Yacc 生成一个解析器。
3. 编译 Yacc 所生成的代码以及其他相关的源文件。
4. 将目标文件链接到适当的可执行解析器库。

按照惯例 Yacc 文件有 .y 后缀。在命令行如下调用 Yacc 编译器：

```
$ yacc <options> <filename ending with .y>
```

上面的命令将输出语法分析器的 C 源程序，如果带有 main() 函数则可以单独编译成可执行文件，否则可以与其他代码一起编译成可执行文件。

4.4.2 Yacc 文件实例

这里以解析一个格式为“姓名 = 年龄”的文件作为例子，来说明 Yacc 文件如何描述语法规则并实现目标功能。假设文件有多条纪录，每个记录有姓名和年龄，它们以空格分隔。下面将按照 Yacc 文件的三个段分别进行分析。

C 与 Yacc 的声明

第一段中 C 声明可能会定义动作代码中使用的类型、变量和宏定义，还可以包含头文件。每个 Yacc 声明段声明了终结符号和非终结符号的名称，还可能描述操作符优先级和针对不同符号的数据类型。一般由 Lex 返回这些标记，但是所有这些标记都必须在 Yacc 声明中进行说明，由 Yacc 自动将这些标记用整数编号，并保存在指定的头文件中。

在“姓名 = 年龄”文件解析的例子中我们感兴趣的是这些标记：NAME、EQ 和 AGE。NAME 是一个完全由字符组成的值，EQ 是等号“=”，AGE 是数字。于是声明段就会像这样：

```
1.   %{
2.   #typedef char* string; /* to specify token types as char* */
3.   #define YYSTYPE string /*a Yacc variable which has the value of returned token */
4.   %}
5.   %token NAME EQ AGE
6.   %%
```

类似 Lex，Yacc 也有一套变量和函数可供用户来进行功能扩展。YYSTYPE 定义了用来将值从 Lex 拷贝到解析器或者 Yacc 的 yyval（另一个 Yacc 变量）的类型。默认的类型是 int。由于字符串可以从 Lex 拷贝，类型被重定义为 char*。关于 Yacc 变量的详细讨论，请参考 Yacc 手册。

语法规则

Yacc 语法规则具有以下一般格式：

```
result : components { /* action to be taken in C */ };
```

在上面这行代码中，result 是规则描述的非终结符号。Components 是根据规则放在一起的不同的终结和非终结符号。如果匹配特定序列的话 Components 后面可以跟随要执行的动作。考虑如下的例子：

```
1.   param : NAME EQ NAME { printf("\tName:%s\tValue(name):%s\n", $1,$3);}
2.           | NAME EQ VALUE{ printf("\tName:%s\tValue(value):%s\n",$1,$3);}
3.           ;
```

该例子中 param 非终结符号对应两种合法序列“NAME EQ NAME”和“NAME EQ VALUE”，它们之间用“|”分割开来。如果上例中序列 NAME EQ NAME 被匹配，将执行相应的 {...} 括号中的动作。这里另一个有用的就是位置参数\$1 和\$3 的使用，它们引用了标记 NAME 和 NAME（或者第二行的 VALUE）的值。Lex 通过 Yacc 的变量 yyval 返回这些值。标记 NAME 在 Lex 代码是这样的：

```
1.       char [A-Za-z]
2.       name {char}+
3.       %%
4.       {name}    {     yyval = strdup(yytext);
5.                   return NAME; }
```

从以上代码可以看出一旦匹配 name 这个标记，则 yyval 内容从 yytext 中拷贝，同时返回 NAME 符号。

“姓名 = 年龄”的文件解析例子所用的 Yacc 规则段应该是这样的：

```

1.           file : record file
2.           | record
3.           ;
4.           record: NAME  EQ  AGE { printf("%s is now %s years old!!!", $1, $3);}
5.           ;
6.           %%
```

从上面代码可以看出 `file` 是非终结符号, 它可以由多个 `record` 排列而成, 而每条记录 `record` 则是形如 `NAME EQ AGE` 的标记序列。一旦匹配则打印 `NAME is now AGE years old!!!` 这样的字符串, 其中 `NAME` 和 `AGE` 在每条记录中有具体的取值。

附加 C 代码

现在再来看“姓名 = 年龄”例子的语法文件最后一段——附加 C 代码。这一段是可选的, 如果不需要的话可以没有。如果这个语法分析器单独使用, 它必须要有一个函数 `main()`, 通过调用 `yyparse()` 函数 (`Yacc` 中与 `Lex` 的 `yylex()` 有等效地位的函数) 来完成语法分析。

这一段还包括“姓名 = 年龄”文件解析例子的主函数, 它通过调用 `yyparse()` 开始语法分析:

```

1.         void main()
2.         {
3.             yyparse();
4.         }
```

一般来说, `Yacc` 最好提供 `yyerror(char msg)` 函数的代码。当解析器遇到错误时调用 `yyerror(char msg)`。错误消息作为参数来传递。一个简单的 `yyerror(char*)` 可能是这样的:

```

1.         int yyerror(char* msg)
2.         {
3.             printf("Error: %s
4.             encountered \n", msg);
5.         }
```

如果需要可以从变量 `yylineno` 获得语法分析中出错行的位置信息。

当 `Yacc` 文件准备好了以后, 就可以生成语法分析器的代码, 可能用到以下命令:

```
$ yacc -d <filename.y>
```

这生成了输出文件 `y.tab.h` 和 `y.tab.c`, 它们可以用 `UNIX` 上的任何标准 C 编译器来编译(如 `gcc`)。

结合 Lex 与 Yacc

到目前为止我们已经分别讨论了 `Lex` 和 `Yacc`, 并提到了在 `yyparse()` 中调用 `yylex()` 函数。现在来看一下它们是怎样结合使用的。

一个由 `Yacc` 生成的解析器调用 `yylex()` 函数来获得标记。`yylex()` 可以由 `Lex` 来生成或完全由自己来编写。对于由 `Lex` 生成的扫描器来说, 是可以很方便地和 `Yacc` 结合使用的, 每当 `Lex` 中匹配一个模式时都必须返回一个标记。`Lex` 中匹配模式时的动作一般格式为:

```

1.   {pattern} { /* do smthg*/
2.           return TOKEN_NAME; }
```

于是 Yacc 就会获得返回的标记 TOKEN_NAME。当 Yacc 用-d 选项来编译一个.y 文件时，会生成一个头文件，它对每个标记都有 #define 的定义。如果 Lex 和 Yacc 一起使用的话，头文件必须在相应的 Lex 文件 .lex 中的 C 声明段中包含它。

再回到“姓名 = 年龄”的文件解析例子中，看一看 Lex 和 Yacc 文件的代码。下面是语法文件 Name.y

```
1.      %
2.      typedef char* string;
3.      #define YYSTYPE string
4.      %
5.      %token NAME EQ AGE
6.      %%
7.      file : record file
8.      | record
9.      ;
10.     record : NAME EQ AGE {
11.         printf("%s is %s years old!!!\n", $1, $3); }
12.     ;
13.     %%
14.     int main()
15.     {
16.         yyparse();
17.         return 0;
18.     }
19.     int yyerror(char *msg)
20.     {
21.         printf("Error
22.             encountered: %s \n", msg);
23.     }
```

Lex 的解析器文件 Name.lex 的内容如下：

```
1.      %
2.      #include "y.tab.h"          这个是 Yacc 输出的文件
3.      #include <stdio.h>
4.      #include <string.h>
5.      extern char* yylval;        这个变量在语法分析器中声明
6.      %
7.      char [A-Za-z]
8.      num [0-9]
9.      eq       [=]
10.     name {char}+
11.     age      {num}+
12.     %%
13.     {name}{    yylval = strdup(yytext);
```

```

14.           return NAME; }
15.     {eq}   {  return EQ; }
16.     {age}  {  yyval = strdup(yytext);
17.               return AGE; }
18.   %%

19.   int yywrap()
20.   {
21.     return 1;
22.   }

```

作为一个参考，我们列出了 Yacc 生成的头文件 y.tab.h:

```

1. # define NAME 257
2. # define EQ 258
3. # define AGE 259

```

它们的取值都大于 255，因此和 Lex 直接返回的字符是可以相互区分的。对 Lex 和 Yacc 的讨论到此为止，下面来看看如何用 Yacc 完成对 OpenMP/C 代码的分析。

4.5 OpenMP/C 语法分析

OpenMP/C 语法分析需要重构语法树，如果成功则说明没有语法错误，否则说明有语法错误。可是仅仅以此为目的的语法分析没有任何作用，实际上我们进行语法分析重建 AST 是为了进行翻译变换，需要在语法分析后产生中间表示，这就需要语义动作的支持。但是由于编译目的不同这些语义动作差别很大，没有统一的内容，因此本小节不讨论语义动作函数（即使将编译目标限定于建立 AST 中间表示，也由于建立 AST 的语义动作代码比较复杂，相关的细节需要单独在第 5 章讨论）。可以统一讨论的就只有 Yacc 文件中的第二段的语法规则描述了，在此我们直接讨论 OMPI 的语法描述文件 parser.y。

OpenMP/C 的语法分析工作中不存在词法分析中 OpenMP 与 C 共用关键字的情况。比如词法分析中 if 可以匹配 OpenMP 的条件子句也可以匹配 C 的条件语句，因此需要加以判断甄别。在语法分析时，扫描器提供上来的符号不是 IF 就是 OMP_IF，因此是没有相互共用混淆的情况，也就是说在描述语法规则的时候，OpenMP 语法和 C 语言语法是相对独立的。但是两种语法共存于一个源文件中，它们必定有结合点，这个结合点的最上层，是在语句这个非终结符号上。在 OpenMP v2.5 规范中 OpenMP C and C++ Grammar 小节里面，有如下语法规则：

```

1. /* in C99 (ISO/IEC 9899:1999) */
2. block-item:
3.   declaration
4.   statement
5.   openmp-directive

6. statement:
7.   /* standard statements */
8.   openmp-construct

```

```

9.    openmp-construct:
10.   parallel-construct
11.   for-construct
12.   sections-construct
13.   single-construct
14.   parallel-for-construct
15.   parallel-sections-construct
16.   master-construct
17.   critical-construct
18.   atomic-construct
19.   ordered-construct

```

从 OpenMP 规范看，它是将 OpenMP 构造当作语句来扩展的，语句 **statement** 可以推导出普通语句（代码中的 **standard statements**）也可以推导出 **openmp-construct**，而后者可以推导出 10 种语法构造。与规范对应地在 OMPI 的 **parser.y** 文件中有以下代码：

```

1.   statement:
2.   labeled_statement
3.   {      $$ = $1;      }
4.   | compound_statement
5.   {      $$ = $1;      }
6.   | expression_statement
7.   {      $$ = $1;      }
8.   | selection_statement
9.   {      $$ = $1;      }
10.  | iteration_statement
11.  {      $$ = $1;      }
12.  | jump_statement
13.  {      $$ = $1;      }
14.  | openmp_construct // OpenMP Version 2.5 ISO/IEC 9899:1999 addition
15.  {      $$ = OmpStmt($1);
16.      $$->l = $1->l;
17.      }
18. ;

```

从上面代码可知语句这个非终结符号 **statement** 可以推导出七种非终结符号，其中六中对应于 C 语言的语句（具体见第 5 章），第七种为 OpenMP 构造 **openmp_construct**。因此 OpenMP 制导指令在 OpenMP/C 语言源程序中就是作为一种语句而存在，这是 OMPI 的处理方式，其他编译器可以用不同的处理方法。C 语言语法构造的描述不进一步讨论，下面对 OpenMP 的语法构造 **openmp_construct** 这个符号进一步考察，OMPI 中的 **parser.y** 中有如下代码：

```

1.   openmp_construct:
2.   parallel_construct
3.   {      $$ = $1;      }
4.   | for_construct
5.   {      $$ = $1;      }
6.   | sections_construct

```

```

7.      {      $$ = $1;      }
8.      | single_construct
9.      {      $$ = $1;      }
10.     | parallel_for_construct
11.     {      $$ = $1;      }
12.     | parallel_sections_construct
13.     {      $$ = $1;      }
14.     | master_construct
15.     {      $$ = $1;      }
16.     | critical_construct
17.     {      $$ = $1;      }
18.     | atomic_construct
19.     {      $$ = $1;      }
20.     | ordered_construct
21.     {      $$ = $1;      }
22. ;

```

从上面的代码描述可以知道 OpenMP 构造 openmp_construct 可以推导出 10 种构造的非终结符号，分别对应 parallel 构造、for 构造和 sections 构造等等。再进一步考察 parallel 构造有：

```

1.  parallel_construct:
2.    parallel_directive  structured_block
3.    {
4.      $$ = OmpConstruct(DCPARALLEL, $1, $2);
5.      $$->l = $1->l;
6.    }
7. ;

```

这个语法规则（符号的合法序列）指出 parallel 构造是由 parallel 制导指令（parallel_directive）后面跟着 C 语言结构块（structured_block）而构成的。这时候 structured_block 就返回到 C 语言的语法范畴，出现 OpenMP 与 C 的交互情况。那么对于 parallel_directive 这个非终结符号的语法规则描述有：

```

1.  parallel_directive:
2.    PRAGMA_OMP  OMP_PARALLEL  parallel_clause_optseq '\n'
3.    {      $$ = OmpDirective(DCPARALLEL, $3);      }
4. ;

```

上面代码表明 parallel_dirctive 是由符号 PRAGMA_OMP 后面跟着符号 OMP_PARALLEL 最后跟着 parallel 制导指令的子句序列。子句序列在 parser.y 中还有语法规则说明，但是 PRAGMA_OMP 和 OMP_PARALLEL 则不会有进一步的规则说明了，因为它们是词法分析器返回的终结符号（token 标记），例如 PRAGMA_OMP 的规则描述在 scanner.l 文件中：

```

1.  [ \t]*#"["[ \t]*"pragma"[ \t]+"omp"[ \t]+  {
2.                                count();
3.                                on_omp_line = __has_omp = 1;
4.                                return (PRAGMA_OMP);
5.                            }

```

这个规则说明出现形如“# pragma omp”的序列是 OpenMP 制导指令出现的标志。其中 PRAGMA_OMP 的编号在 Yacc 在编译 parser.y 时输出的 parser.h 文件中定义的。关于 OpenMP 的语法的其他信息，可以参考相应的标准 ISO/IEC 9899:1999（OpenMP Version 2.5），例如并行构造在 OpenMP 标准中的语法定义是这样的：

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
structured-block
```

根据 OpenMP v2.5 规范关于 OpenMP 和 C 语言的语法，很容易将规范中的语法描述转换成 Yacc 的语法规则描述，从而写出正确的 parser.y 文件。

4.6 小结

本章介绍了如何用 Lex 和 Yacc 来完成 OpenMP/C 代码的词法分析和语法分析。

关于词法分析的内容首先是 Lex 工具的使用方法，包括则正表达式的使用、Lex 文件的三段代码格式与功能等内容。然后分析了如何描述 OpenMP 和 C 代码的词法规则、如何区分公共关键字，并以 OMPI 的 Lex 文件为例详细说明了 OpenMP/C 词法规则的描述。

语法分析部分先是介绍了 Yacc 工具的使用，包括 Yacc 文件中三段代码的格式和功能、如何与 Lex 工具配合等内容。然后分析了 OpenMP/C 代码的语法规则描述。本章并没有介绍相关的语义动作，它将在第 5 章详细讨论。

通过本章学习并借助于 Lex 和 Yacc 的文档，读者应该能够对 OMPI 编译器源代码中的 scanner.l 和 parser.y 文件进行修改和增强，甚至是自行编写 OpenMP/C 的 Lex 和 Yacc 文件。

第 5 章 AST 的创建

编译器的前端构造出源程序的中间表示，后端才能根据这个中间表示生成目标程序。本章将讨论 OpenMP 编译中如何选择中间表达形式、AST 的结构、相关数据结构和函数功能。

AST 的建立是在语法分析的时候执行语义动作来完成的，这时涉及到 AST 节点类型及相应的节点创建、删除、输出等操作。OpenMP 程序 AST 的建立需要处理 C 语言的语法构造也需要处理 OpenMP 制导指令的语法构造。

作为中间表示，需要考虑其实现的复杂性，避免使用过多的数据类型，因此普通编程语言的 AST 节点的结构体往往分成语句节点、类型节点、声明节点、表达式节点四种，而对于 OpenMP 代码而言需要增加一种 OpenMP 节点。每种节点分成若干种类型，每个类型再分成几种子类型，从而覆盖所有的语法构造。本章将着重讨论这五种 AST 节点的设计和 AST 的建立过程。

5.1 中间表示

中间表示应该具有以下两个重要性质：易于生成、易于翻译为目标语言。具体到不同问题还会有具体的要求。比如在 OpenMP 编译原理的教学中，还需要便于将代码变换结果直观地展示出来。下面简单地介绍两种中间表示形式，并分析为何选择 AST 作为中间表示形式。

5.1.1 两种中间表示形式

编译器中常用的两种中间表示形式是：

1. 树形结构，包括语法分析树和（抽象）语法树。
2. 线性表示形式，特别是“三地址码”。

语法树

设计源代码到源代码的翻译器时，抽象语法树（Abstract Syntax Tree, AST）的数据结构是非常好的起点，因为它保留了源程序的层次化语法结构。在一个表达式的抽象语法树中，每个内部节点代表一个运算符，该节点的子节点代表这个运算符的运算分量。对于更一般化的程序设计语言任意一个语法构造，可以创建一个针对这个构造的运算符节点，并把这个构造的具有语义信息的组成部分作为这个运算符的运算分量。这时候的“运算符”并不仅仅指通常意义上的数学运算，所有语句构造都需要定义相应的“运算符”，比如定义 WHILE 作为语句 while 的运算符，DOWHILE 作为 do{...} while...语句的运算符等等。对于语句“do i=i+1;while (i< 100);”的抽象语法树可以表示成图 5.1-(a)所示。

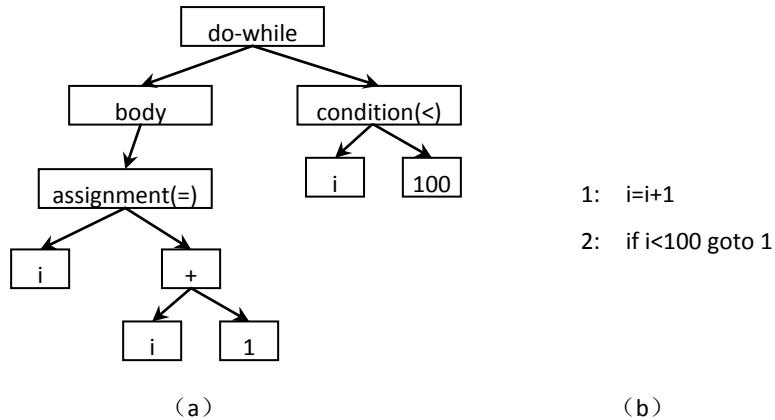


图 5.1 循环语句的抽象语法树及三地址码表示

抽象语法树有时也简称为语法树（Syntax Tree），与语法分析树很相似。区别在于，抽象语法树的内部节点代表的是程序的构造，而语法分析树内部节点代表的是非终结符号。文法中的很多非终结符号都代表程序的构造，但也有一部分是辅助符号（比如代表项、因子或其他表达式变体的非终结符号）。抽象语法树通常不需要这些辅助符号，因此会将这些符号忽略掉。为了区别它们，往往将语法分析树称为具体语法树（concrete syntax tree）。

三地址码

三地址码是由一个基本程序步骤（比如两个值相加）组成的序列。和树形结构不一样，它没有层次化的结构。对于语句“`do i=i+1; while (i<100);`”的三地址表示的中间代码如图 5.1-(b) 所示。这是一组“三地址”指令序列，具有形如其名字的指令形式： $z = x \text{ op } y$ ，其中 op 是一个二元运算符， x 和 y 是两个运算分量（变量地址）， z 是存放结果的地址。三地址指令最多只执行一个运算，通常是计算、比较或分支运算。

当需要对计算过程作优化时就需要这种表示形式。代码优化时可以将程序的三地址语句序列分割成多个不包含跳转语句的“基本块”，然后再使用各种优化技术。

5.1.2 中间表示的选择

通常编程语言的编译器在创建抽象语法树的同时生成三地址码序列。而且通常情况下这些编译器并不会创建出一棵存放了源代码所有程序构造的完整抽象语法树的数据结构，只是按照创建抽象语法树的规则“假装”构造了它，并同时生成三地址码序列。这些编译器在分析过程中只保存将要用于语义检查和或其他目的的节点及其属性，同时也保存了语法分析的数据结构，而不会保存整棵语法树。其目的是在构造三地址码序列时需要用到的那部分语法树的子树可用即可，一旦用完即可以释放掉。

本书选择 OMPI 作为 OpenMP 编译实现技术的分析对象是因为它“确实”构造了一棵完整的抽象语法树，对应了整个源代码的所有语法构造。由于按照狭义的 OpenMP 编译的定义，只需要进行源代码变换来弥补 OpenMP 制导指令与 C 语言之间的语义差距，而这些语义差距属于线程并发执行及相关问题，与具体的数学“计算”优化不直接相关，所以用于代码优化的三地址码表示在此处并没有优势。

综合而言，在本书讨论 OpenMP 的实现技术中选择构造了完整语法树的 OMPI 的原因是：

1. 本书讨论的 OpenMP 编译是狭义的定义，实现源代码到源代码的变换。而使用抽象语法树作为中间表示形式便于源代码到源代码的变换，抽象语法树保留了原程序的所有层次性的语法构造，很方便地从语法树还原到源代码。而三地址码与源代码差别很大，从三地址码序列还原成 C 代码的困难较大。
2. 由于最终的可执行文件产生是依赖于后续的 C 编译器，因此 OpenMP/C 程序中的大量优化任务不在 OpenMP 狹义编译范畴中，因此对三地址码的需求并不迫切。而可以将这些优化功能放在在后续的 C 编译器中，在那里产生三地址码的中间表示，进而完成代码优化。
3. 构造完整的抽象语法树便于教学和实践，通过输出抽象语法树可以清晰地看到源代码变换的中间过程和结果。

5.2 AST 节点数据结构

在使用语义动作来构建 AST 之前，需要先设计好 AST 节点的数据结构，设计的好坏将影响到 AST 创建和维护的难度或代码复杂度。

AST 上的节点是可以表达所有的语法构造元素的，但是可以用比语法元素的类型少的节点类型（主类型），并配合上节点的子类型来表达。比如可以将节点的主类型设计成只有以下五大类：语句、表达式、类型说明、声明、OpenMP 制导。下面先分析 OMPI 使用的 AST 节点数据结构，然后用具体例子说明如何使用这些节点来表示程序的各种语法构造。

5.2.1 语句节点

编程语言中有大量的语句，这些语句在 AST 中使用语句节点来表示。下面以 OMPI 的 AST 节点的设计来具体分析其实现细节。

OMPI 的 AST 中，语句节点使用 `aststmt_` 类型的结构体来表示，对应的指针类型为 `aststmt`。该结构体 `aststmt_` 可以分为 11 种类型，分别是 JUMP 跳转、ITERATION 循环、SELECTION 条件 / 选择分支、LABELED 标号、EXPRESSION 表达式、DECLARATION 声明、COMPOUND 复合语句、STATEMENTLIST 语句列表、FUNCDEF 函数定义、OMPSTMT OpenMP 节点、VERBATIM 维保节点。有的节点类型可能还可以继续细分为子类型。

```

1. struct aststmt_
2. {
3.     enum stmttype type;           类型
4.     int          subtype;         子类型（比如循环语句有 for、while 子类型等）
5.     aststmt      parent; /* Set *after* AST construction */
                           父节点
6.     aststmt      body; /* Most have a body (COMPOUND has ONLY body) */
                           语句体
7.     union
8.     {
9.         astexpr expr;           /* For expression & return statements */
                           用于表达式和返回语句

```

```

10.    struct { aststmt init;
11.          astexpr cond, incr; } iteration;
12.          循环语句的初始、条件、增量操作三个部分
13.    struct { astexpr cond;           分支语句的条件
14.          aststmt elsebody; } selection;   分支语句的不成功分支(正常分支放在 body 里面)
15.    struct { astspec spec;
16.          astdecl decl;      /* dlist is for FuncDef */
17.          aststmt dlist; } declaration;     声明语句中的类型说明和声明部分
18.    symbol label;                /* For GOTO and labeled statements */
19.          标号, 用于跳转目标
20.    aststmt next;               /* For StatementList */
21.          用于将语句列表形成链表
22.    ompcon omp;                /* OpenMP construct node */
23.          当语句类型为 OMPSTMT 时, 记录 OpenMP 构造
24.    char *code;                /* For verbatim nodes */ 维保节点的字符串形式的代码
25.    } u;
26.    int l,c;                  /* Location in file (line, column) */
27.    symbol file;
28. };

```

根据变量名字可以推测出 `aststmt` 的成员变量 `type`、`subtype`、`parent`、`l`、`c`、`file` 的用途，它们是所有语句节点都必须设置的。大多数语句节点都有 `body` 的内容。变量 `l`、`c`、`file` 用于记录该语句所在的行、列和文件名。根据类型和子类型的不同，对联合体 `u` 的解释也是不同的：

1. 对于表达式和返回语句，`u` 是一个 `astexpr` 表达式类型的变量；
2. 对于循环语句，`u` 被解释为 `iteration` 结构体，描述了本循环的初始条件、结束条件和增量操作（`while` 循环不需要初始条件和增量操作）。此时循环体的代码将在 `body` 中存在；
3. 对于选择分支语句，`u` 被解释为 `selection` 结构体，用于保存成功分支条件和分支代码。对于分支不成功的代码则在 `body` 中存在。
4. 对于声明语句则解释为 `declaration` 结构体。如果是变量声明，则它的类型使用(`astspec` 类型说明节点) `spec` 指出，变量列表用语句节点 `astdecl` 类型的 `decl` 表示。如果是函数声明，则类型还是用 `spec` 指出，但是函数使用 `dlist` 声明（是 `aststmt` 类型）；
5. 对于标号语句则解释为 `symbol` 符号类型的 `label`，用于记录标号。而标号对应的语句则保存在 `body` 中；
6. 对于语句列表则解释为 `aststmt` 类型的 `next`，用于指向下一条语句。此时 `body` 指向当前语句；
7. 对于 OpenMP 构造块，此时节点类型为 `OMPSTMT`，`u` 被解释为 `ompcon` 类型的 `omp` 变量，用于记录该构造块。
8. 对于维保节点则解释为对应代码的字符串指针。

下面将对变量声明、赋值语句和条件语句为例说明相关数据结构。首先来看声明语句“`int var_x;`”，可以用图 5.2 所示的数据结构来表示。

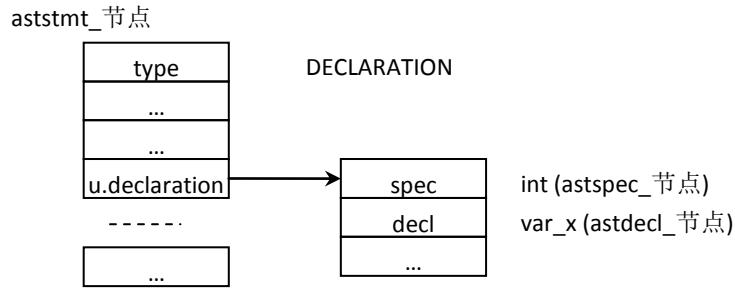


图 5.2 “int var_x;”语句的 AST 结构

由于“int var_x;”是一个语句，所以它需要用 `aststmt_` 节点来表示。这是一个变量声明语句，所以它的类型 `type` 为 `DECLARATION`，不需要子类型的限定。联合体 `u` 被解释为 `declararion` 结构体，并且这个结构体的 `spec` 成员指向一个对应于“int”的类型说明节点（`astspec_` 节点），另一个成员 `decl` 指向对应于 `var_x` 的声明节点（`astdecl_` 节点）。类型说明节点和声明节点都不属于语句节点，它们将在 5.2.2、5.2.3 小节讨论。

接着用一个赋值语句“`var_x=var_y+var_z;`”的例子来加深语句节点的认识，这条语句可以用图 5.3 所示的 AST 节点来表示。首先其类型应该是赋值语句，所以 `type` 为表达式 `EXPRESSION`，联合体 `u` 解释为类型是 `astexpr_` 的 `expr` 结构体，而 `astexpr_` 类型的节点可以参见 5.2.4 小节。

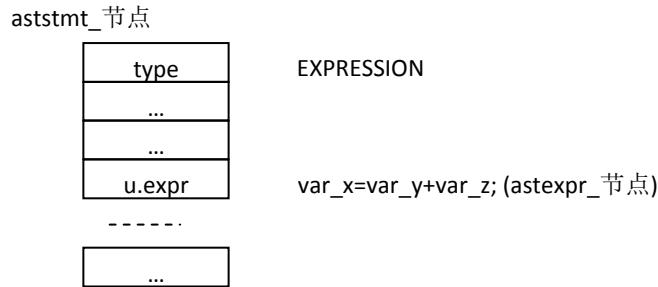


图 5.3 “var_x=var_y+var_z;”语句的 AST 结构

最后再以一条件分支语句“`if(a==b) then c=d;else e=f;`”为例，说明条件语句的 AST 节点数据结构，具体如图 5.4 所示。

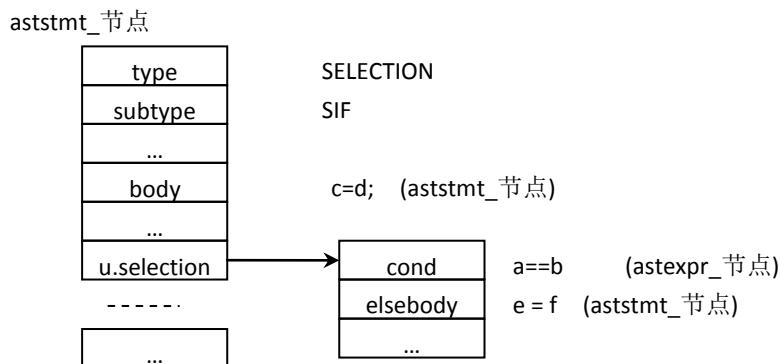


图 5.4 分支语句的 AST 结构

由于是分支语句，所以 `type` 为 `SELECTION`。但是分支语句有两种：条件分支和选择分支，所以需要用 `subtype` 标记为 `SIF`。对应于 IF 的成功分支的语句由 `body` (`aststmt_` 类型的节点)

表示。联合体解释为 `selection` 结构体，其条件部分用 `cond` (`astexpr_`类型的节点) 表示，不成功分支用 `elsebody` (`aststmt_`类型的节点) 表示。

5.2.2 类型说明节点

一个声明语句节点 `aststmt_` 包含两个部分，类型说明 (`specifier_`) 节点和变量声明 (`declarator_`) 节点，分别属于上层 `aststmt_` 结构的联合体 `u.declaration` 结构中的 `spec` 和 `decl`。`aststmt_` 结构参见 5.2.1 的分析。请注意区分类型为声明的语句节点 (`aststmt_` 节点) 和声明节点(`astdecl_`节点)及其关系。

OMPI 的类型说明节点是 `astspec_` 结构体，对应的指针类型为 `astspec`。结构体 `astspec_` 的 C 代码如下：

```
1. struct astspec_          类型说明节点
2. {
3.     enum spectype type;    6 种类型之一
4.     int      subtype;      27 种子类型之一
5.     symbol   name;        /* For SUE/enumlist name/user types */
                           结构体、联合体和枚举类型名
6.     astspec   body;        /* E.g. for SUE fields, lists */
7.     union
8.     {
9.         astexpr   expr;      /* For enum list */
                           枚举类型中，记录各个枚举值的列表
10.        astspec  next;      /* For Lists */
                           用于 “unsigned long” 类型这样的列表形成链表
11.        astdecl decl;      /* For structure specifiers (the fields) */
                           用于结构体的说明
12.    } u;
13.    int     l,c;           /* Location in file (line, column) */
                           所在行与列
14.    symbol  file;          所在文件
15. }
```

类型说明节点有 6 种类型，它们分别是:SPEC、STCLASSSPEC、USERTYPE、SUE、ENUMERATOR、SPECLIST，类型是通过 `type` 来说明的。当用作枚举类型时，联合体 `u` 将被解释为表达式类型的 `expr`；用作 SPECLIST 的多个类型说明符（例如 `unsigned long`）联合体 `u` 将被解释为类型说明指针，`next` 用于指出下一个类型说明；当用作结构体类型说明时联合体 `u` 将被解释为 `astdecl` 类型的 `decl` 声明节点。其他 `l`、`c`、`file` 和前面的节点用法相同。

例如 “`int var_x;`” 这个节点是使用 `aststmt_`类型的声明语句节点来表示的，见图 5.2 所示。其中的 `int` 是使用 `spec` 节点来保存的，这个 `spec` 就是这里的类型说明节点，此时 `type` 为 SPEC，`subtype` 为 SPEC_int，具体见图 5.5 所示。

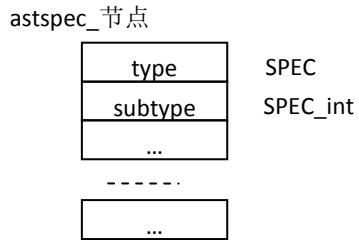


图 5.5 int 类型说明节点

对于象 “`unsigned long`” 这样的类型说明，则需要用下面图 5.6 所示的数据结构来表示。构造这样的列表使用的函数是 `Specifierlist()`。

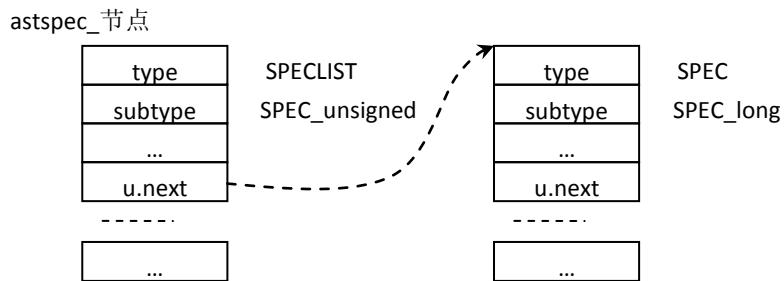


图 5.6 “`unsigned long`” 的类型说明节点

最后再用枚举类型作为例子 “`enum box { pencil, pen, chalk }`”，看看如何表示这样的类型说明。此时只需要一个 `astspec` 节点，其中的枚举值 `pencil`、`pen` 和 `chalk` 是在联合体 `u` 解释为 `decl`（是 `type` 为 `DLIST` 的声明节点，见 5.2.3 小节）来表示的，具体如图 5.7 所示。

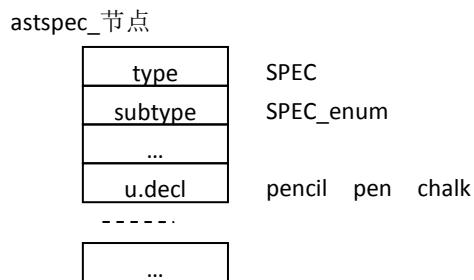


图 5.7 枚举类型的类型说明节点

5.2.3 声明节点

声明节点是在声明语句节点下的子节点，它和类型说明节点一起构成了声明语句的两个要素。`OMPI` 中的声明节点是 `astdecl_` 结构体（对应的指针类型为 `astdecl`）。有以下类型：DIDENT，DPAREN，DARRAY，DFUNC，DINIT，DECLARATOR，ABSDECLARATOR，DPARAM，DELLIPSIS，DBIT，DSTRUCTFIELD，DCASTTYPE，DLIST。声明节点 `astdecl_` 数据结构如下。

1. struct astdecl_
2. {
3. enum decltype type; 13 个类型之一
4. int subtype; 仅用于列表，4 个子类型之一

```

5.      astdecl      decl;      /* For initlist,initializer,declarator */
                           用于声明列表、初始化、声明符
6.      astspec       spec;      /* For pointer declarator */
                           用于指针
7.      union
8.      {
9.      symbol        id;      /* Identifiers */
                           用于记录标识符，例如 int a 中的 “a”
10.     astexpr       expr;      /* For initializer/bitdeclarator */
                           用于有初始化表达式的，例如 int a=1 和位声明
11.     astdecl       next;      /* For lists */
                           用于列表，例如 int a,b,c;
12.     astdecl       params;    /* For funcs */
                           用于声明函数时的参数，例如 int myfun(int a); 的 “int a”
13.   } u;
14.   int      l,c;           /* Location in file (line, column) */
15.   symbol file;
16. };

```

对于声明多个变量语句，例如“int a,b;”，其中变量声明部分的结构如图 5.8：

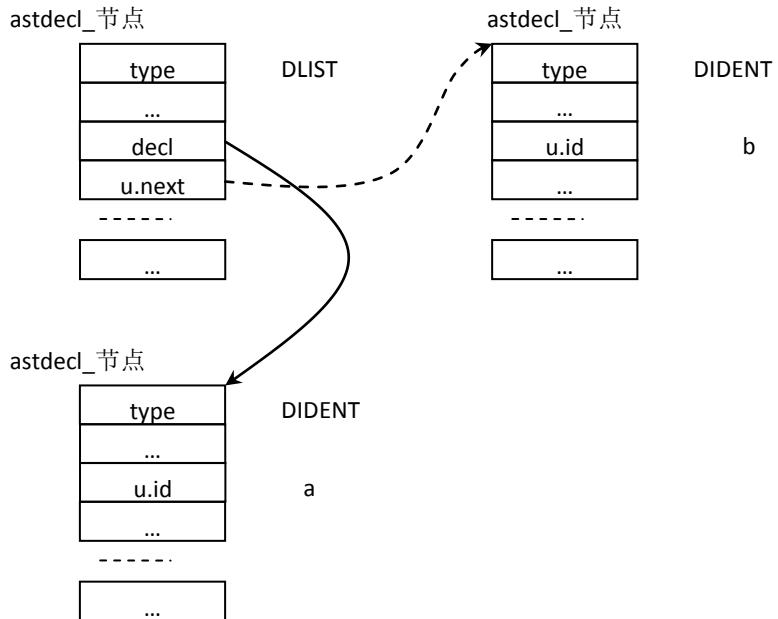


图 5.8 变量声明列表节点

5.2.4 表达式节点

对于表达式节点，OMPI 使用结构体 `astexpr_`（对应的指针类型为 `astexpr`）来表示，它可以有具体的以下 20 种子类型：IDENT 标识符，CONSTVAL 常量，STRING 字符串，FUNCCALL 函数调用返回值，ARRAYIDX 数组，DOTFIELD 结构体的子域（“.” 引用方式），PTRFIELD 结构体的子域（“->” 引用方式），CASTEXPR 类型转换的表达式，CONDEXPR 条件表达式，UOP 一元操作，BOP 二元操作，PREOP 前缀操作，POSTOP 后缀操作，ASS 赋值操作，COMMALIST 逗号分

割的列表， SPACELIST 空格分隔的列表， BRACEDINIT 括号的初始化， DESIGNATED， IDXDES， DOTDES。

在 OMPI 中表达式节点的结构体定义如下：

```
1. struct astexpr_
2. {
3.     enum exptype type;                                类型有 20 种, 见前面说明
4.     astexpr    left, right;                          表达式的左右两部分 (各自也是 astexpr_)
5.     int        opid; /* Used for operators */        操作类型 (如果表达式是 UOP, 则
6.     有 11 中, BOP 则有 19 种, ASS 则有 11 种)
7.     union {
8.         char   *str; /* Used by strings and constants */    如果表达式类型 type 为 CONSTVAL 常量或 STRING 字符串, 这个*str 来保存
9.         symbol sym; /* Used by identifiers/fields */       如果表达式类型 type 为 IDENT 标识符或 DOTFIELD, PTRFIELD, 用于保存相应的符号名
10.        astexpr cond; /* Used only in conditional exprs */  仅用于条件表达式, 保存用于判断真假条件的表达式
11.        astdecl dtype; /* Used only in casts & sizeof */    仅用于类型转换以及 sizeof 操作, 填写什么内容, 见后面 sizeof 节点生成
12.    } u;
13.    int      l,c;                                     /* Location in file (line, column) */
14.    symbol   file;                                    表达式所在行 (l) 列(c)
15. };
16.
```

表达式主要有操作符、左元素、右元素三部分构成，但是也有一元操作等特殊情况。联合体 `u` 根据 `type` 不同而作不同解释：对于常量或字符串，`u.str` 记录这个常量或字符串；如果是一个标识符的“.”或“->”域，则对应于 `u.sym` 符号；对于条件表达式则用 `u.cond` 来保存条件；如果用于类型转换及 `sizeof` 操作，则使用 `u.dtype`。

5.2.5 OpenMP 制导节点

OpenMP 的节点有三种，最上层的 `ompcon_` 构造节点，它可以有 `ompclause_` 和 `ompdir_` 两种子树。代码中先定义了子句 `ompclause_` (对应有 `ompclause` 指针)，然后是制导指令 `ompdir_` (对应有 `ompdir` 指针)，最后是 `ompcon_` (对应有 `ompcon` 指针) 结构。我们先从上层数据结构分析起。

ompcon

`ompcon_` 结构体是关于 OpenMP 相关代码的最上层结构，它包含有编译制导指令 `ompdir` 类型的 `directive` 成员以及具体的语句 `body` 成员，这个 `body` 代表的是该 OpenMP 构造内部的语句块。下面是 `ompcon_` 结构体的定义：

```
1. struct ompcon_
2. {
3.     enum dircontype type;                            制导指令的类型
4.     ompdir      directive;                         制导指令
```

```

5.     aststmt      body;      代码块
6.     aststmt      parent; /* The OmpStmt node the construct belongs to */
7.     int         l,c;      /* Location in file (line, column) */
8.     symbol file;
9. };

```

由于 OpenMP 构造包含两个部分，一个是 OpenMP 编译制导部分，另一个就是对应的功能语句块。OpenMP 构造节点的类型和 OpenMP 制导指令节点的类型是一致的，OpenMP 的构造节点的类型（制导指令类型也一样）及编号如表 5.1 所示，OpenMP 制导指令的类型共有 14 种。

表 5.1 OpenMP 构造的类型

编号	类型	制导指令
1	DCPARALLEL	parallel
2	DCFOR,	for
3	DCSECTIONS	sections
4	DCSECTION	section
5	DCSINGLE	single
6	DCPARFOR	parallel for
7	DCPARSECTIONS	parallel sections
8	DCMASTER	master
9	DCCRITICAL	critical
10	DCATOMIC	atomic
11	DCORDERED	ordered
12	DCBARRIER	barrier
13	DCFLUSH	flush
14	DCTHREADPRIVATE	threadprivate

OpenMP 构造节点可以用图 5.9 来直观地表示，directive 和 clause 节点见后面的分析，其中的功能语句块 body 的类型 aststmt 已经在前面讨论过。

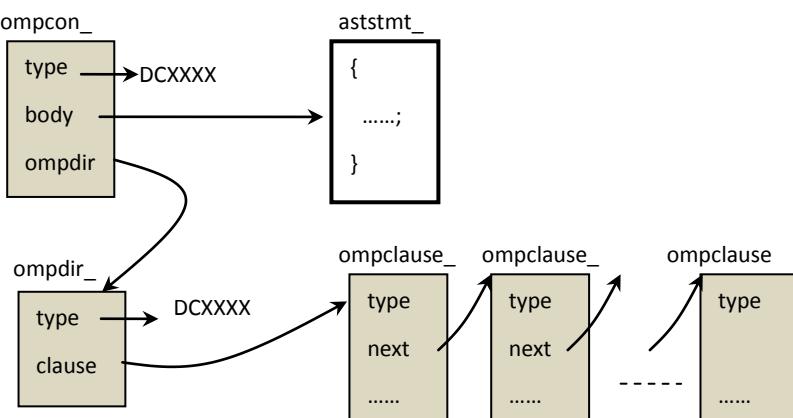


图 5.9 OpenMP 构造节点

ompdir

OpenMP 构造节点的一个重要成员是 directive，即 OpenMP 的制导指令。下面是关于 ompdir_ 结构体的定义。

```
1. struct ompdir_                      ompdir_ 属于 ompcon_ 下的元素
2. {
3.     enum dircontype    type;          14 种类型之一
4.     ompclause        clauses;        /* actually a clause list */
                                         可带有子句列表
5.     ompcon         parent;          /* The construct the directive belongs to */
                                         指向上层的 ompcon_ 结构体
6.     union {
7.         symbol  region;           用于 critical 区域的名称
8.         astdecl varlist;        /* For flush(), threadprivate() */
9.     } u;
10.    int      l,c;                /* Location in file (line, column) */
11.    symbol  file;
12. }
```

制导指令类型 type 和 OpenMP 构造的类型是一样的，具体可见表 5.1。一个制导指令需要记录其类型和相应的子句。对于 critical 制导指令，它的可选名称将保存在联合体 u.region 中；而对于 threadprivate 和 flush 制导指令则按照 u.varlist 进行解释，记录相应的变量声明。从这里可以看出，制导指令可以带有子句，并且由 ompclause 类型的 clauses 指出。下面马上讨论 ompclause 类型的节点。

ompclause

OpenMP 子句共有 15 种类型，有些类型还有子类型。子句节点的结构体定义如下。

```
1. struct ompclause_                  ompclause_ 属于 ompcon_ 下的元素
2. {
3.     enum clausetype    type;          15 种类型之一
4.     int              subtype;        13 种子类型之一
5.     ompdir        parent;          /* The directive the clause belongs to */
                                         用于指向上层的 astcon_ 结构体
6.     union {
7.         astexpr   expr;            用于 IF 子句的条件表达式
8.         astdecl  varlist;        用于 PRIVATE 或 SHARE 等子句的变量列表
9.         struct {
10.             ompclause elem;
11.             ompclause next;
12.         } list;                多个子句形成列表
13.     } u;
14.    int      l,c;                /* Location in file (line, column) */
15.    symbol  file;
16. }
```

如果带有条件子句 `if`, 那么对应的条件表达式将由联合体 `u` 解释为 `expr` 来表示; 如果子句是数据子句, 那么相应的变量列表将由联合体 `u` 解释为 `varlist` 来记录和表示; 如果有多个子句, 那么这些子句构成一个链表, 由联合体 `u` 解释为 `list` 结构体来表示, 其中 `list.elem` 用来记录一个子句项, 利用 `list.next` 指向下一个子句项。

下面是子句类型和子类型, 可以从类型名称上很容易地知道它对应的数据子句。但是最后一个类型 `OCFIRSTLASTPRIVATE` 并不是 OpenMP 的子句类型, 但是创建这个节点可以方便地跟踪那些同时在 `firstprivate` 和 `lastprivate` 子句中的变量。

表 5.2 OpenMP 子句的类型

类型	枚举号	子类型	子类型编号
OCNOWAIT	1		
OCIF	2		
OCNUMTHREADS	3		
OCORDERED	4		
OCSCHEDULE	5	OC_static	0
		OC_dynamic	1
		OC_guided	2
		OC_runtime	3
OCCOPYIN	6		
OCPRIvATE	7		
OCCOPYPRIVATE	8		
OCFIRSTPRIVATE	9		
OCLASTPRIVATE	10		
OCSHARED	11		
OCDEFAULT	12		
OCREDUCTION	12	OC_plus	6
		OC_times	7
		OC_minus	8
		OC_band	9
		OC_bor	10
		OC_xor	11
		OC_land	12
		OC_lor	13
OCLIST	14		
OCFIRSTLASTPRIVATE	15		

子句节点可以只有独立一项, 也可以是一个链表。如果是链表形式则有图 5.10 的组成形式。

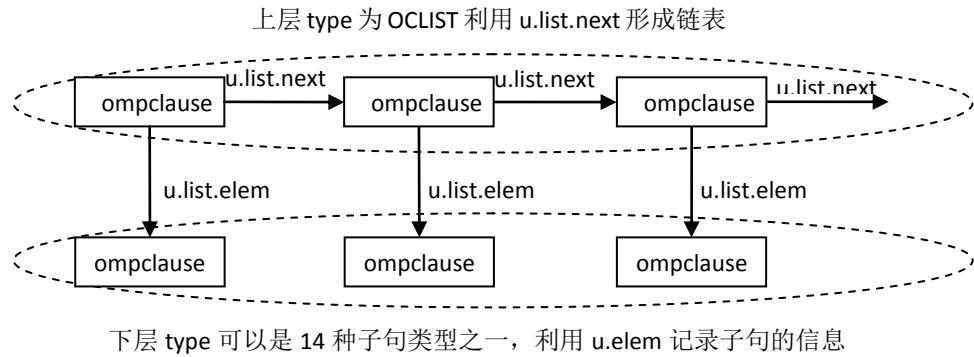


图 5.10 OpenMP 子句节点的两层链表结构

设想有如下代码：

```
#pragma omp parallel for private (a,b) if (Y>10) nowait
for(i=0;i<10;i++)
    a=b;
```

对应的 AST 树节点的构成如图 5.11 所示。

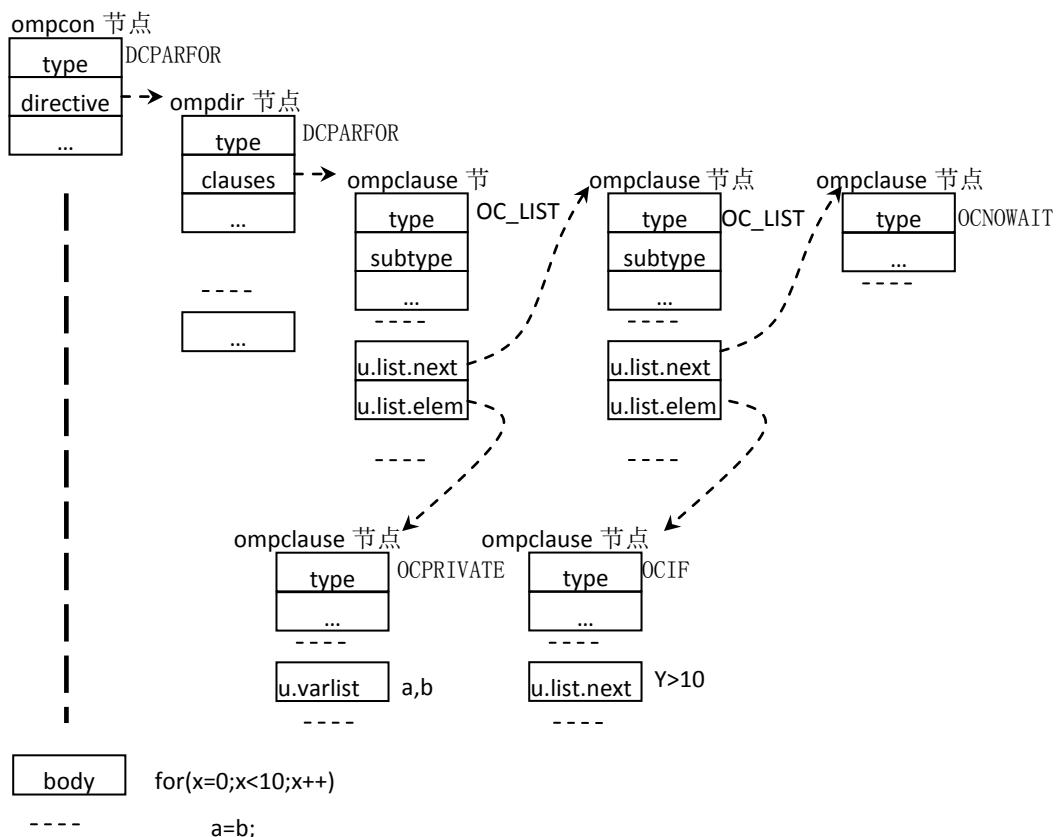


图 5.11 OpenMP 构造的 AST 树结构

最上层是一个 `ompcon` 节点，它有两个子节点分别对应于制导指令 `directive` 和功能语句块 `body`。其中 `body` 直接记录了内部的 `for` 语句，而 `directive` 则是一个类型为 `DCPARFOR` 的 `ompdir` 节点。制导指令节点里面还记录有子句节点，因此制导指令节点的 `clauses` 指向子句序列。此

处的子句序列有三项，这三项通过类型为 OC_LIST 的 `ompclause` 节点借助于 `u.list.next` 构成链表，各自有 `u.list.elem` 指向具体的子句节点——分别类型为 OCPRIVATE、OCIF 和 OCNOWAIT。

从框架上说，一个 OpenMP 构造对应一个 `ompcon` 节点，一个 `ompcon` 节点必须带有一个 `ompdir` 节点，一个 `ompdir` 节点可以带有零到多个 `ompclause` 节点。

5.3 AST 节点维护函数

由于在语法分析时执行语义动作从而构建出整个 AST，因此这些语义动作必然对 AST 树和节点进行各种创建和维护操作，出于设计的简洁性和代码的可读性，可以将这些节点维护函数独立出来，在语义动作代码中调用即可。

节点创建

节点创建函数是最常用的维护函数，在语法分析过程中每使用一次产生式基本上都对应着调用一次节点创建函数。对于前面的 5 种 AST 节点，创建函数也至少有 5 中。例如语句节点的创建函数 `Statement()` 的代码如下：

```
1. aststmt Statement(enum stmttype type, int subtype, aststmt body)
2. {
3.     aststmt s = smalloc(sizeof(struct aststmt_));
4.     s->type = type;
5.     s->subtype = subtype;
6.     s->body = body;
7.     s->parent = NULL;
8.     s->l = sc_original_line();
9.     s->c = sc_column();
10.    s->file = Symbol( sc_original_file() );
11.    return (s);
12. }
```

它分配内存给 `aststmt` 类型的结构体并将其指针保存在 `s` 中，根据传入的类型、子类型、语句体指针填写相关的成员变量（父结点指针暂时为空，等待以后写入）。

同理有类型说明节点、声明节点、表达式节点和 OpenMP 构造节点的创建函数：

```
astspec Specifier(enum spectype type, int subtype, symbol name, astspec body);
astdecl Decl(enum decltype type, int subtype, astdecl decl, astspec spec);
astexpr Astexpr(enum exprtype type, astexpr left, astexpr right);
ompcon OmpConstruct(enum dircontype type, ompdir dir, aststmt body);
```

虽然有上述 5 种节点的创建函数，就足以完成各种节点的创建，但是出于方便性和代码的可读性，也为了便于检查、减少编程错误，实际上对上述节点的创建函数进行二次封装，以针对不同的节点类型进行针对性修改。例如 GOTO 语句节点，可以用前面的 `Statement()` 函数来产生，也可以用下面的定制函数来创建：

```
1. aststmt Goto(symbol s)
2. {
3.     aststmt n = Statement(JUMP, SGOTO, NULL);
```

```

4.     n->u.label = s;
5.     return (n);
6. }

```

调用 `Goto()` 函数时，只需要传入跳转地址的符号 `s` 即可，然后在里面调用 `Statement()` 函数以 `JUMP` 类型、`SGOTO` 子类型来创建语句节点。这样一来只要跳转地址符号 `s` 不出错，那么创建的节点也就没有问题。而使用直接使用 `Statement()` 来创建节点，除了保证 `s` 正确外，还需要保证类型和子类型的参数要正确。

对于 OpenMP 构造节点，除了使用上层的 `OmpConstruct()` 外，还需要底层创建制导指令节点和子句节点的功能，它们是：

```

ompclause OmpClause(enum clausetype type, int subtype, astexpr expr, astdecl varlist);
ompdir OmpDirective(enum dircontype type, ompclause cla);

```

辅助函数

除了创建节点之外，还有许多其他辅助函数。比如记录父结点信息的功能等。下面以给 OpenMP 构造节点填写父结点语句的信息：

```

1. void ast_ompcon_parent(aststmt parent, ompcon t)
2. {
3.     t->parent = parent;
4.     ast_ompdir_parent(t, t->directive);
5.     if (t->body) /* barrier & flush don't have a body */
6.         ast_stmt_parent(parent, t->body);
7. }

```

要将 `t` 的父结点指针指向 `parent`，然后将本构造中的制导指令节点的父结点指向本构造体，同样语句体 `body` 的父结点也指向本构造体。

其他例如节点删除、子树拼接合并及变换等操作将在后面用到的时候再分析。

5.4 AST 的创建

AST 中间表示可以通过语法制导翻译过程来完成，语法制导翻译是通过向一个文法的产生式附加一些规则（或程序片段）而得到。语法制导的翻译方案（Translation Scheme）是一种将程序片段附加到一个文法的各个产生式的表示方法，在语法分析过程中使用一个产生式的时候，相应的程序片段就会被执行，这些程序片段的执行效果按照语法分析过程顺序组合起来，就得到这次分析/综合过程处理源程序后得到的翻译结果。

本节将给出 `for` 语句、`while` 语句以及一个完整的 `helloworld.c` 的 OpenMP 程序的 AST 构建细节。

5.4.1 语法制导翻译

语法制导翻译技术可以用于类型检查和中间代码的产生，在基于源代码变换的 OpenMP 编译中，需要用语法制导翻译技术来创建抽象语法树。使用语法制导定义（Syntax-Directed

Definition, SDD) 可以通过一组规则来将输入的源程序转换成一棵树。这些规则实际上是一个建立于语法树之上的 SDD，而其他更可能情况则是 SDD 建立在语法分析树之上。

语法制导定义

语法制导定义是一个上下文无关文法和属性及规则的结合。这里的规则指的是语义规则 (Semantic Rule)。属性和文法符号相关联，而规则是和产生式相关联的。如果 X 是一个符号而 a 是 X 的一个属性，那么用 $X.a$ 来表示 a 在某个标号为 X 的分析树节点上的值。由于语法分析树的节点往往用结构或对象来实现，所以 X 的属性可以用结构体的字段来实现。如果要创建 AST 树，那么这些非终结符号的属性就是 AST 树的节点。

语法制导翻译

语法制导翻译是通过向一个文法的产生式附加一些规则或程序代码而得到的。比如有如下产生式生成的表达式 expr :

```
expr -> expr1+term
```

expr 是两个字表达是 expr1 和 term 的和，此时可以利用 expr 的结构，用以下的伪代码来完成翻译 expr :

```
翻译 expr1;
```

```
翻译 term;
```

```
处理 +;
```

具体做法可以是下面这样的。先建立起 expr 的语法分析树：分别建立 expr1 和 term 的语法分析树，然后处理+运算符并构造得到一个和此运算符对应的节点。

这里需要注意两个与语法制导翻译相关的概念：

- 1) 属性 (attribute)：表示于某个程序构造相关的任意的量。属性可以使多种多样的，比如表达式的数据类型、生成的代码中的指令数目或为某个构造生成的代码中的第一条指令的位置等等都是属性的例子。由于使用文法符号（终结符号和非终结符号）来表示程序的构造，所以将属性的概念从程序构造扩展到表示这些构造的文法符号上。
- 2) (语法制导的) 翻译方案 (translation scheme)：翻译方案是一种将程序代码附加到一个文法的各个产生式上的表示法。当在语法分析过程中使用一个产生式的时候，相应的程序代码就会被执行，这些程序代码的执行效果按照语法分析过程的顺序组合起来，得到的结果就是这次分析/综合处理原程序得到的翻译结果。

翻译方案及语义动作

我们可以为表达式或任意的构造创建语法树，图 5.12 是运算符 op (及其运算分量 E1 和 E2) 和 while 语句 (expr 表示条件表达式， stmt 表示语句体) 的语法树。

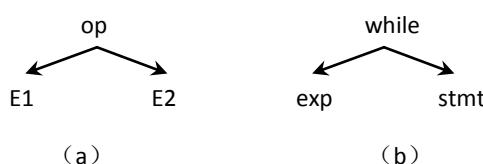


图 5.12 语法树的节点及运算

所有的非终结符号都有一个综合属性 `node`, 该属性表示相应的抽象语法树节点, 在 OMPI 中这个 `node` 属性是由结构体(分别是 `aststmt_`、`astspec_`、`astdecl_`、`astexpr_`、`ompcon_`、`ompdir_` 和 `ompclause_`) 来实现的。

接下来看看 OMPI 的语法制导翻译方案的一些细节。下面是语法分析器里的循环语句的语法描述片段, 参见 `parser.y`。

```
1. iteration_statement:  
2.     WHILE '(' expression ')' statement  
3.     {  
4.         $$ = While($3, $5);  
5.     }  
6.     | DO statement WHILE '(' expression ')' ';'  
7.     {  
8.         $$ = Do($2, $5);  
9.     }  
10.    .....  
.....
```

语法符号 `iteration_statement` 是非终结符号, 代表一个循环语句。它可以产生出 `while`、`do-while`、`for` 等循环。对应于 `while (expr) stmt` 的产生式, 为了产生语法树的翻译方案所插入的代码非常简单。`$$=While($3,$5)` 中的第一个`$$`代表本节点, 它是 `While()`函数的返回值。而 `While($3,$5)` 中的`$3` 代表表达式 `expression`, `$5` 代表 `statement`。那 `While()`函数就必须能根据 `$3` 和 `$5` 构建出一个类似于图 5.12-(b) 的语法树。关于 AST 节点创建的函数, 我们在 5.3 小节的“节点创建”已经分析过创建语句节点的 `Statement()` 和创建跳转节点 `Goto()` 的函数。下面是 `While()`函数的代码。

```
1. #define While(cond,body)  
2.     Iterationstatement (SWHILE,NULL,cond,NULL,body)
```

从此可以看出, `While()`函数创建 `while` 语句节点, 实际是利用宏定义间接调用了 `Iterationstatement()` 函数, 也就是说 `while` 语句和 `for`、`do-while` 等都共用相同的节点创建函数, 因为所有的循环语句都包含循环条件和循环体内的语句两部分语义内容。下面再来看看 `Iterationstatement()` 函数。

```
1. aststmt Iterationstatement(int subtype,aststmt init, astexpr cond, astexpr incr, aststmt body)  
2. {  
3.     aststmt n = Statement(ITERATION, subtype, body);  
4.     n->u.iteration.init = init; /* Maybe declaration or expression */  
5.     n->u.iteration.cond = cond;  
6.     n->u.iteration.incr = incr;  
7.     return (n);  
8. }
```

从 `Iterationstatement()` 的代码可以看出, `while` 语句的节点是语句节点 `aststmt`, 其类型为 `ITERATION`、子类型为 `SWHILE`, 此时代表 `while` 的语句节点的联合体 `u` 被解释为 `iterations`。

`u.iteration.cond` 是传入的条件，而 `u.iteration.init` 和 `u.iteration.incr` 则是给 `for` 语句使用的，因此传入 `NULL` 作为这两个参数。

5.4.2 例 1 OpenMP 的 for 节点

下面以 OpenMP 的 `for` 语句为例分析建立 AST 的语义动作，其语法规则和语义动作如下：

1. `omp_for_specific_iteration_statement:`
2. `FOR '(' init_stmt ';' cond_expr ';' incr_expr ')' statement`
3. `{`
4. `$$ = For($3, $5, $7, $9);`
5. `}`
6. `;`

该语法描述指出 `omp_for_specific_iteration_statement` 可以产生出形如“`FOR '(' init_stmt ';' cond_expr ';' incr_expr ')' statement`”的语句，其中`$3` 代表的是非终结符号“`init_stmt`”，而`$5` 代表“`cond_expr`”这个非终结符号。

在语法分析出一旦出现 `for` 循环语句，则相应的语义动作也将被激发，即`$$ = For($3, $5, $7, $9)`。而 `For(w,x,y,z)` 在 `ast.h` 中只是简单的宏定义，其定义如下：

```
#define For(init,cond,incr,body) Iterationstatement(SFOR,init,cond,incr,body)
```

这是一个宏定义，实际上调用的 `Iterationstatement()`，而 `Iterationstatement()` 在上一小节刚分析过。

因此，相应的语义动作将是新建一个 `statement` 节点 `n`，它的类型为循环（`ITERATION`）、子类型是 `for(SFOR)`、循环体是 `body`，该节点 `n` 的成员变量联合体 `n->u.iteration` 保存了指向初始条件 `init`、结束条件 `cond` 和递增操作 `incr` 这三个非终结符号。此处的 `body(aststmt 类型)`、`init(aststmt 类型)`、`cond(astexpr 类型)`、`expr(astexpr 类型)` 各自有一个 AST 节点来描述，形成以下逻辑关系：

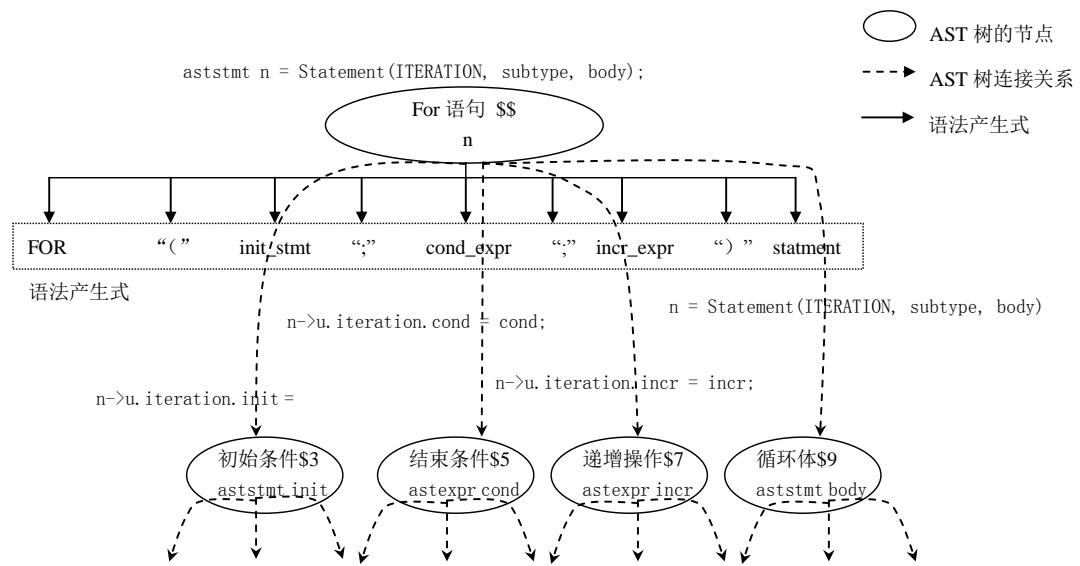


图 5.13 OpenMP for 带导指令的 AST

此时还不涉及符号表问题。`$$/$$3/$$5/$$7/$$9` 都是以指针形式存在的。其中 `for` 语句节点的创建：“`aststmt n = Statement(ITERATION, subtype, body);` ”，创建了一个类型为 `ITERATION` 的语句节点。

对于其中的 `init_stmt` 非终结符号`$3`，还有相应的语法规则：

```
1. init_stmt:
2.     IDENTIFIER '=' expression
3.     {
4.         if (checkDecls)
5.             check_uknown_var($1); /* Must do it! */
6.             $$ = Expression( Assignment(Identifier( Symbol($1) ), ASS_eq, $3 ) );
7.         }
8.     | init_expr_type IDENTIFIER '=' expression
9.     {
10.        astdecl inidec;
11.        $$ = Declaration($1,
12.                           inidec = InitDecl(
13.                               Declarator(NULL, IdentifierDecl( Symbol($2) )),
14.                               $4));
15.        if (checkDecls)
16.        {
17.            symtab_put(stab, Symbol($2), IDNAME)->isarray = 0;
18.            add_declarator_links($1, inidec);
19.        }
20.    }
21. ;
```

这里说明初始条件可以有两种产生式，分别对应有变量声明和没有变量声明两种。

5.4.3 例 2 C 语言 while 语句

接着再以 C 语言的循环语句为例说明 AST 构建。在 `parse.y` 中有如下语法规则和语义动作：

```
1. iteration_statement:
2.     WHILE '(' expression ')' statement
3.     {
4.         $$ = While($3, $5);
5.     }
6.     | DO statement WHILE '(' expression ')' ;
7.     {
8.         $$ = Do($2, $5);
9.     }
10.    | FOR '(' ';' ';' ')' statement
11.    {
12.        $$ = For(NULL, NULL, NULL, $6);
13.    }
```

```

14.     | FOR '(' expression ';' ';' ')' statement
15.     {
16.         $$ = For(Expression($3), NULL, NULL, $7);
17.     }
18.     | .....

```

C 语言的循环语句可以有多种生成式，假如我们遇到是的 while 循环，那么应该执行以下语法规则和语义动作：

```

1.      WHILE '(' expression ')' statement
2.      {
3.          $$ = While($3, $5);
4.      }

```

其中 While() 在 ast.h 中定义如下：

```
#define While(cond,body) Iterationstatement(SWHILE,NULL,cond,NULL,body)
```

而 Iterationstatement() 则是在 ast.c 中定义，已经在前面分析过。此时对应的 AST 树将按图 5.14 方式构建。

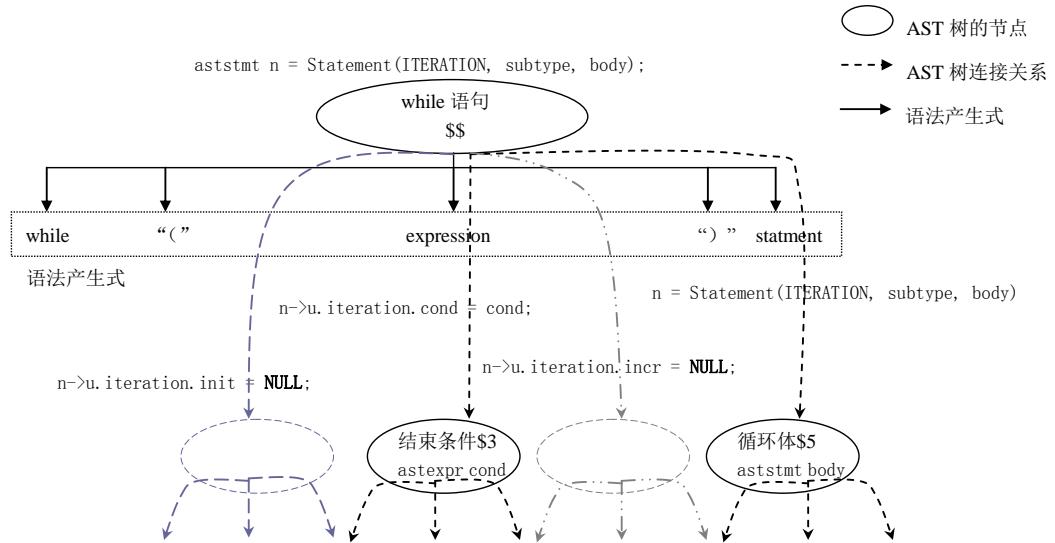


图 5.14 while 语句的 AST

这里也是使用了例 1 中的 ITERATION 类型的 statement 节点，但是子类不同，而且没有循环初始化部分和增量操作部分。

5.4.4 Helloworld.c 的 AST

有了前面的 AST 子树的概念后，现在来形成 OMPI 中一个完整的 AST 的直观认识，下面给出一个完整的程序例子来看看 AST 结构，有如下代码：

```

1. //a example for parallel directive
2. #include <omp.h>
3. int main((int argc, char** argv)
4. {
5.     #pragma omp parallel

```

```
6.         printf("hello world!  %n\n");
7.         return 0;
8.     }
```

因为在 `paser.y` 中关于初始规则的有关规则如下：

```
1. start_trick:
2.     translation_unit          /* to avoid warnings */
3.     | START_SYMBOL_EXPRESSION expression { pastree_expr = $2; }
4.     | START_SYMBOL_BLOCKLIST block_item_list { pastree_stmt = $2; }
.....
5. translation_unit:
6.     external_declaration
7.     {
8.         $$ = pastree = $1;
9.     }
10.    | translation_unit external_declaration
11.    {
12.        $$ = pastree = BlockList($1, $2);
13.    }
14. ;
```

所以一个源程序的最外层是一个 `BlockList()` 产生的 `STATEMENTLIST` 类型的语句节点，将很多 `external_declaration` 符号组织起来，后者又有规则如下：

```
1. external_declaration:
2.     function_definition
3.     {
4.         $$ = $1;
5.     }
6.     | declaration
7.     {
8.         $$ = $1;
9.     }
```

说明一个应用程序的外层是由声明和函数定义构成，到此已经可以看出程序的 AST 大体结构了，下面用图 5.15 表示其结构。

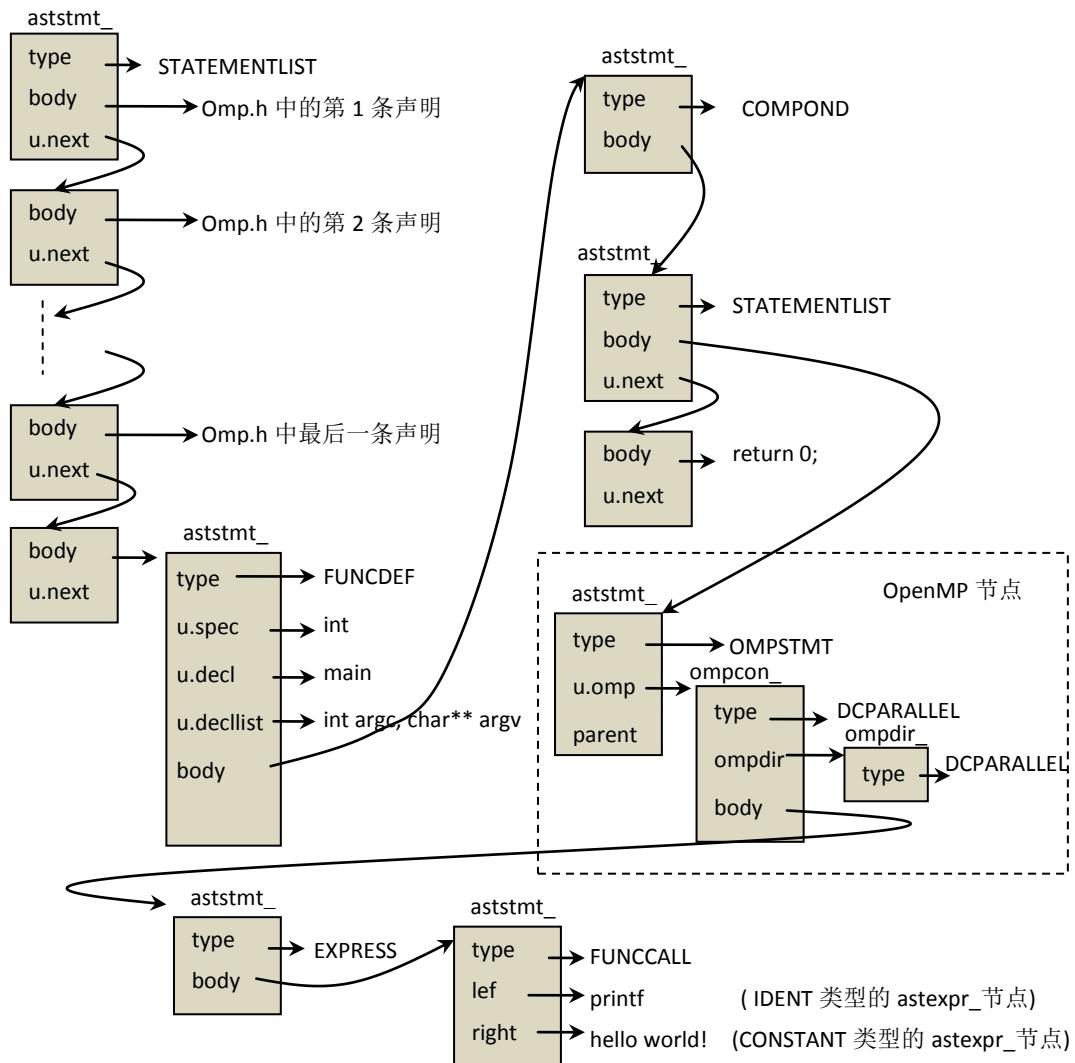


图 5.15 hello.c 的 AST 结构

5.5 符号表

由于语义分析过程中会遇到源程序中的大量语法符号所对应的各种名字（字符串），由于字符串的处理开销比较大，因此所有这些名字都将转换成符号来处理，此时只要用指针指向相应的字符串即可。这些符号组织成符号表，可以分别形成常数表、变量名表、函数名表等等，也可以组织成一个统一的表。表中每一个登记项记录一个名字字符串和相关信息，以反映该符号的属性和编译过程中的特征。由于需要经常频繁进行查表或填表访问，合理地组织符号表提高访问速度，将有利于提高编译的效率。

OMPI 的符号处理由 `symbol.c` 负责，所有的符号记录在 `ompi.c` 中的 `stab` 变量中，而所有符号的字符串经过散列变换后保存在 `allsymbols` 里面，因此可以用指针快速高效地访问到。对于一个名字字符串 `x`，可以通过 `symbol(x)` 来查找，如果找到则返回它的位置，如果没有找到则为该名字创建一个符号项。

符号的处理中要注意的是作用域(scope)的管理，同名符号在不同作用域上可能对应不同的类型，因此出现 `private(X)` 需要产生私有变量的时候，应该能根据作用域找到当前起作用的变量及其类型。

下面分别分析两方面的内容：符号表的存储结构、变量作用域问题。

5.5.1 字符串表

OMPI 对符号的管理分成两层，底层是记录了符号的字符串信息的表称作 `allsymbols` 数组，其元素称为 `symbol`，在 `symbol.h` 文件中有如下定义：

```

1.     typedef struct symbol_          符号
2.             {
3.                 char *name;           名字字符串指针
4.                 struct symbol_*next;   链接指针
5.             } *symbol;

```

`name` 指向相应的名字字符串，而 `next` 用来形成链表。

记录所有符号的字符串的数组 `allsymbols` 在 `symbol.c` 文件里面定义，它维护着所有符号的名字字符串。

```
1.     static symbol allsymbols[ALLSTSIZE];
```

这个数组结构和下面讨论的符号表 `symtab->table` 很相似，可用图 5.16 表示如下：

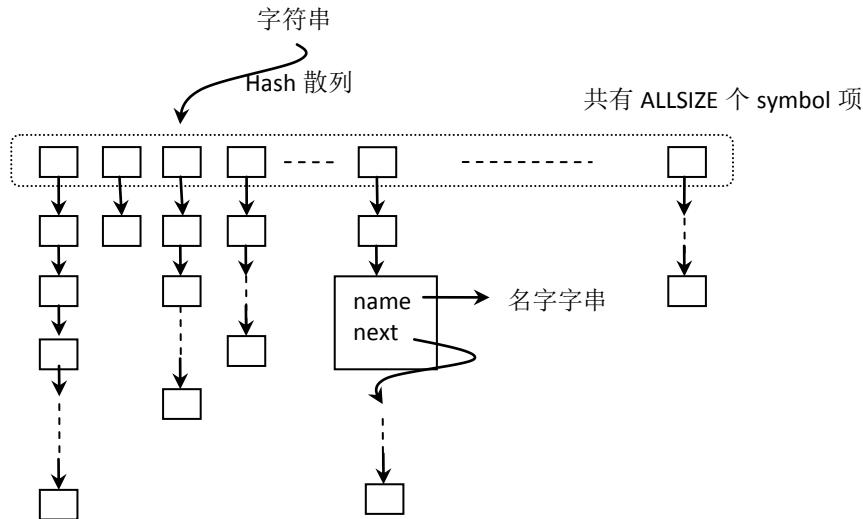


图 5.16 `allsymbols` 数据结构示意图

所有遇到符号的字符串都在这里记录，而符号表则不再关心字符串的问题，因此这个表只能填写不能删除。所有的 `symbol` 将按照 hash 散列值插入到 `allsymbols` 中，图 5.16 中每一个垂直链表对应相同的 hash 值，它们在 OMPI 中称作一个“桶”（bucket），符号 `symbol` 的 `next` 字段就是用来形成这个链表用的指针。

5.5.2 符号表

OMPI 对符号表的高层管理是使用 `syntab` 结构体，其中 `syntab->table` 是一个数组，记录了 `STSIZE` 个符号表项。Symbol.h 文件部分代码如下，首先是符号表的大小：

```
...  
1. #define STSIZE 1031 /* Prime */
```

符号表大小

然后是关于符号的名字空间问题。可以看出符号分成几种名字空间，namespace 有 IDNAME 标识符、TYPENAME 类型名、SUNAME 结构和联合体名、ENUMNAME 枚举名等 6 种。

```
2. typedef enum { IDNAME = 1, TYPENAME, SUNAME, ENUMNAME, LABELNAME, FUNCNAME }  
3.         namespace;
```

名字空间

再接着是符号表项指针和符号表项，符号表项是符号表的元素，它对符号 `symbol` 进行了封装，不同的名字空间的符号借助于成员 `space` 来标示。所有的符号都在符号表里。一个符号使用符号项来记录，下面是符号项 `stentry_` 的定义。

```
4.     typedef struct stentry_ *stentry;  
5.     struct stentry_ { symbol     key;           /* The symbol */      指向符号  
6.                     void     *vval;          /* General, to put anything */  
7.                     int      ival;          /* For int values (common case) */  
  
8.                     astspec   spec;          /* The specifier */  
9.                     astdecl   decl;          /* The bare declarator */  
10.                    astdecl   idecl;         /* initdeclarator (includes decl) */  
  
11.                    namespace space;        /* 1 table for all spaces */  
12.                    int      isarray;        /* Non-scalar */  
13.                    int      isthrpriv;       /* 1 if it is a threadprivate var */  
14.                    int      scopelevel;      /* The scope it was declared in */  
15.                    stentry   bucketnext;    /* for the bucket */  
16.                    stentry   stacknext;     /* for the scope stack */  
17.    };
```

符号表项里面的 `spec`、`decl` 和 `idecl` 分别指向某个变量符号的类型、声明和带初始化声明的 AST 节点，用于后面代码变换时克隆变量相关的代码。因为 `decl` 只指向声明，而声明这种非语句节点是没有父结点指针的，所以无法访问到初始化语句节点。大多数情况下只需要 `decl` 即可，如果带有初始化，那么需要使用 `idecl`。`vval` 和 `ival` 设计本意是可以存放一些随机目的的数据，实际上 OMPI 只在分析 OpenMP 数据子句的时候来保存类型（例如 `private`、`firstprivate` 等，参见 `x_clauses.c`），而 `vval` 用于保存归约操作的类型。`scopelevel` 用于记录本符号所处的作用域层次，`bucketnext` 用于形成具有相同 `hash` 值得符号的 FILO 链表，`stacknext` 则是根据所遇到符号的次序形成的 FILO 链表（堆栈）。

下面是符号表的定义，需要记录当前的变量作用域的层次。

```
18.     typedef struct syntab_ { stentry table[STSIZE];           STSIZE 个符号表项  
19.                     stentry top;        /* Most recent in scope */  
20.                     int      scopelevel;    /* Current scope level */  
21.     } *syntab;
```

所有遇到的符号都在这里记录，所有的 `symbol` 将按照地址指针的 hash 散列值插入到 `table` 中，图 5.17 中每一个垂直链表对应相同的 `hash` 值，它们在 OMPI 中称作一个“桶”（`bucket`），符号项 `stentry` 的 `bucketnext` 字段就是用来形成这个链表用的指针。注意这里的 `stentry` 并不关心符号的字符串，后者是在 `symbol` 结构中保存的。

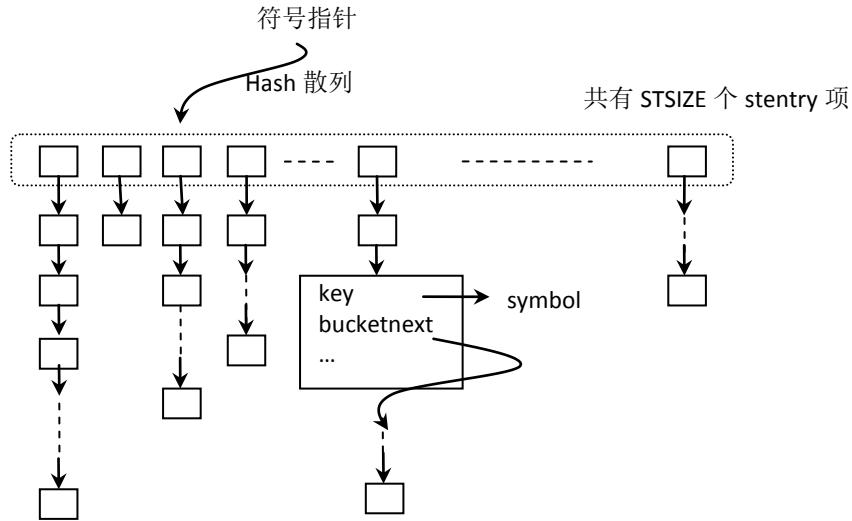


图 5.17 symtab->table 的数据结构示意图

5.5.3 符号表操作

符号表的常见操作分成两种，一种是关于符号名字字符串的操作，另一类是关于符号的操作。

名字字符串操作

这部分操作对应于 `allsymbols` 数组，有符号的插入、查找等。OMPI 使用 `Symbol()` 来根据字符串查找符号所在的位置，如果找不到则为该字符串创建一个符号插入 `allsymbols` 中，再返回它所在的位置。`Symbol_exists()` 则是判断一个字符串是否在符号表中。`Symbols_allfree()` 则将 `allsymbols` 的所有元素及其占用的空间清空，这个函数只在不再需要符号表时才调用。另外还有根据字符串来计算 `hash` 值得 `hash()` 函数。

符号表操作

符号表的操作常见的有插入 `symtab_put()`、释放 `freestentry()` 和删除 `symtab_remove()` 等操作。符号表操作中比较特殊的就是需要插入全局符号的时候，`symtab_insert_global()` 就是用于此目的的，此时不能按普通方式插入到 `hash` 值对应的“桶”的上面，而是需要遍历这个桶并插入到 `scopelevel=0` 的符号前。

5.5.4 作用域管理

每当语法扫描时遇到{}将产生新的作用域，不同作用域上的同名变量/符号是互不相关的相互独立的变量/符号。以 parser.y 的复合语句为例看看语法分析中的作用域处理：

```
1. /* ISO/IEC 9899:1999 6.8.2 */
2. compound_statement:
3. {
4.     '{' '}'
5.     '$$ = Compound(NULL);
6.     }
7.     | '{' { $<type>$ = sc_original_line()-1; scope_start(stab); }
8.         block_item_list '}''
9.     {
10.         $$ = Compound($3);
11.         scope_end(stab);
12.         $$->l = $<type>2; /* Remember 1st line */
13.     }
14. ;
```

此处复合语句 `compound_statement` 可以有两种产生式，第一种是{}里面没有语法构造的空语句，第二种是里面有 `block_item_list` 的复合语句。对于第二种情况，当遇到第一个“{”时需要用 `scope_start()` 往符号表里面插入名字空间为 IDNAME 的 “@scopper@” 符号，表示新作用域的开始，并且将 `syntab->scopelevel` 增 1 表示进入更深层的作用域。每个符号都会记录自己所在的作用域。可以对作用域进行命名，最外层是全局变量作用域 `scope 0`，然后是 `scope 1`，逐层编号等等。在任何时刻，第 i 层作用域上可见的变量在第 i+1 层代码中是可见的（除非在第 i 层声明了同名变量），反之第 i+1 层定义的变量对第 0 层到第 i 层的代码是不可见的。

如果在 `syntab` 里面出现同名符号，那么它们将有相同的 hash 值从而保存在相同的“桶”中，再根据“桶”的 FILO 的堆栈特性，查找是总是返回最里层作用域的符号，这个特性正好符合 C 语言作用域的特点。

语法分析中，每当退出一个作用域，则关闭次作用域同时删除该作用域内的符号。当语法扫描退出一个第 i 层的作用域时（遇到 “}”），该作用域的所有符号都要清除，这不需要对 `syntab` 进行扫描选出 `scopelevel=i` 的所有符号，而是顺着符号堆栈（由 `stacknext` 链表构成）逐个删除，因为最上层的就是最里层的作用域，直到遇到上次进入该作用域时插入的符号 `@scopper@` 为止。删除当前作用域内的符号可以由 `scope_end()` 来完成。

5.6 小结

本章首先解释了 OpenMP/C 编译使用 AST 作为中间表示的原因——AST 保留了语法层次结构便于进行源代码变换。然后结合 OMPi 编译器分析如何设计 AST 节点，详细分析了语句节点、表达式节点、类型说明节点、声明节点和 OpenMP 节点的具体数据结构。最后介绍语法制导

翻译技术，结合 **OMPI** 代码分析如何在语法分析中插入语义动作从而通过创建各种 **AST** 节点并构建出 **AST**。

关于符号表的处理也在本章介绍，不仅分析了符号表的数据结构，还分析如何随着语法分析进入不同作用域而动态的管理各种符号。

第 6 章 并行域管理

在以 AST 为基础进行源代码变换之前，需要先了解 OpenMP 制导指令的语义以及与 C 语言之间的语义差距，然后才是如何使用源代码级的翻译变换来消除这些差距。我们将在三个方面进行分析：并行域管理（第 6 章）、任务分担与线程同步（第 7 章）、变量数据环境控制（第 8 章）。

首先来考察并行域管理上的语义差距。OpenMP 应用程序的执行模式是 fork-Join。在每次进入并行域后将由 `fork` 操作产生出多个线程来执行计算任务；当退出并行域的时候执行 `join` 操作，除了主线程外所有线程都被撤销。如果出现嵌套的并行域，那么这些 `fork-join` 执行过程中将出现多层的线程树结构。由于并行域内往往需要执行任务分担，所以并行域的管理需要能记录这些层次关系、并行域内的协作关系、辅助变量作用域的判断等等。由于 C 语言本身并没有并发执行的概念，即使操作系统提供的类似 `pthreads` 库可以产生并发线程，也不能直接支持 OpenMP，因此这些差距必须通过源代码翻译变换和运行库函数进行消除。

本章将讨论 OpenMP 编译器将如何在标准 C 语言的基础之上，借助于运行库的支持来完成并行域代码的封装、线程层次、编号、协作等管理功能，并给出各自的代码翻译变换的框架形式。

6.1 并行域及其嵌套

在第 2 章中已经描述过并行域的概念。每当遇到 `parallel` 制导指令，将产生出多个 OpenMP 线程并发执行直到 `parallel` 制导指令限定的代码出口处才撤销。OpenMP 的 `parallel` 制导指令还允许嵌套（`nested`），但是需要调用 `omp_set_nested()` 函数修改 `ICV` 变量或者设置环境变量 `OMP_NESTED` 来启用嵌套特性，否则系统默认是不允许嵌套的。当系统不支持嵌套时，内层的并行域只有一个线程在执行。

以下面的代码为例来看看嵌套并行的执行情况：

```
1. #include <omp.h>
2. int main(void)
3. {
4.     omp_set_nested(1);
5.     #pragma omp parallel num_threads(2)
6.     {
7.         int level=0;
8.         printf("myid is %d @ level:%d\n",omp_get_thread_num(),level);
9.         #pragma omp barrier
```

```

10.         #pragma omp parallel num_threads(4)
11.         {
12.             level=1;
13.             printf("myid is %d @ level:%d\n",omp_get_thread_num(),level);
14.         }
15.     return 0;
16. }

```

该程序有内外两层并行域，由 gcc 编译后运行结果如下：

```

myid is 0 @ level:0
myid is 1 @ level:0
myid is 0 @ level:1
myid is 2 @ level:1
myid is 1 @ level:1
myid is 3 @ level:1
myid is 0 @ level:1
myid is 3 @ level:1
myid is 2 @ level:1
myid is 1 @ level:1

```

该程序通过“`omp_set_nested(1)`”来激活并行嵌套特性，外层的并行域指定线程数为 2 (`num_threads(2)`)，内层并行域的线程数目为 4 (`num_threads(4)`)。从执行结果上看，外层 (level 0) 的线程数目确实为 2，分别编号为 0 和 1；内层线程共有两组 8 个分别隶属于外层线程 0 和 1，并编号为 0、1、2 和 3。这时候的线程层次关系可以如图 6.1。

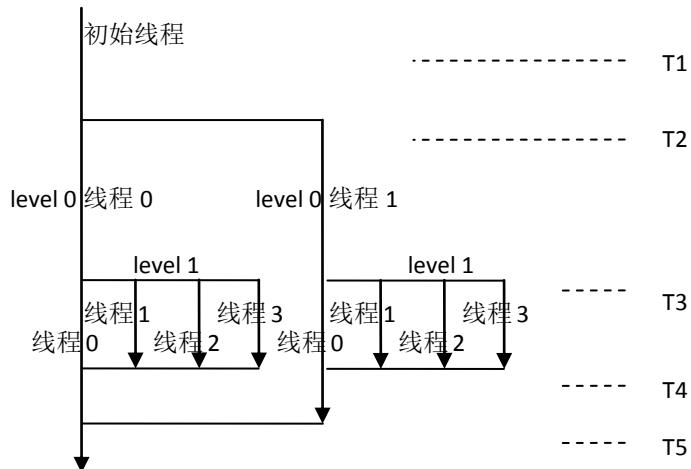


图 6.1 并行嵌套执行图

由此可以看出嵌套的并行域将会形成层次性结构，这些并行域中的线程结合负载分担语句将形成比较复杂的逻辑关系，使任务分担和变量数据环境控制更加困难，因此需要明确并行域管理所应该具备的基本功能以支持程序的正确运行。

6.2 并行域管理

并行域管理涉及以下五大问题。

首先是 OpenMP 线程的载体可以是多种形式的，因此并行域的管理需要能够屏蔽这些底层线程库的差异；

其次需要能够在恰当的时机产生或撤销线程，以满足 OpenMP 的 parallel 并行域的 fork-join 行为，为了提高效率可能采用线程池的方式来管理；

第三是线程间关系问题。由于并行域可能形成层次关系，外层 parallel 与内层 parallel 形成嵌套，并行域内线程也需要以线程 id 来相互区分，因此需要标记嵌套层数和“父子”关系、标记同一并行域内的线程“兄弟关系”；

第四需要对并行域代码的封装和标识，使得各线程能区分各自的工作；

第五是并行域内的线程往往需要进行任务分担，因此并行域管理必须提供任务和线程的映射信息等等。

下面将分析 OpenMP 并行域管理中的这五个问题，明确翻译后的目标代码应该具有什么功能，然后在下一小节给出目标代码的形式。

6.2.1 线程无关接口

作为一个通用的 OpenMP 编译器，应尽量考虑接纳不同的线程技术，不要限制于只是用一种具体的线程接口。所以应该抽象出 OpenMP 线程的概念，该抽象概念应该能满足 OpenMP 语法和语义，保证编译后的程序能正确执行。并行域的管理也应该分成两个部分，上层是满足于抽象 OpenMP 线程的部分，下层是与具体线程技术相关的部分。上层部分应该包含抽象、统一的线程供给、层次关系维护、并行域内任务分担信息的功能，底层应该包含具体线程的创建、同步等基本操作，如图 6.2 所示。

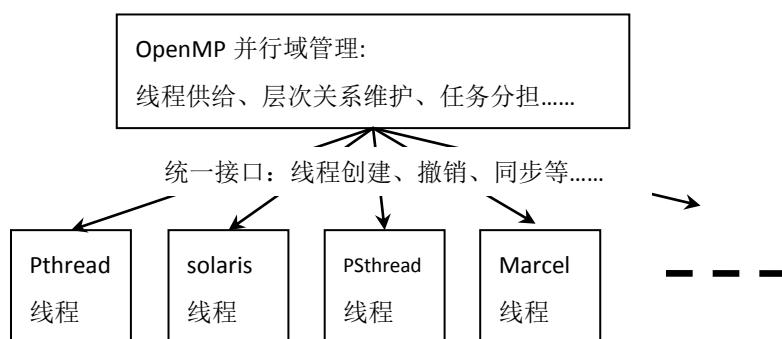


图 6.2 线程无关接口形式

编译器设计时在考虑统一接口这个问题上，还需要做以下决策：

1. 编译器对多种线程库的支持是动态还是静态方式。静态方式则是指编译器虽然具备支持多种线程库的能力，但是每次只能支持其中一种，比如在编译器配置和生成后不再改变。动态方式则是在需要的时候可以随时变换底层线程库，变换的时候不需要重新生成新的编译器。静态方式比动态方式容易实现。

- 编译器是否同时支持多种线程库，也就是说在一个 OpenMP 程序经过该编译器编译后，其并行域内的 OpenMP 线程是有一种还是多种具体的线程。同构系统内一般不需要在一个应用中支持多种不同的线程。

6.2.2 线程的供给

每当出现一个新的并行域，都应该为它提供足够的新线程，因此线程管理的第一个功能就是在需要的时候能供给线程，任务完成后能阻塞或撤销线程。

在实现该功能的时候应当考虑的一个问题是相应的开销和代价，要求以尽可能低的代价实现上述功能。根据不同的应用程序需求，可能对开销和代价的要求也不同。如果对执行时间有最高要求，那么需要线程创建和撤销时间最短，那么编译器可能会使用线程池的方法。提前生成若干线程，避免在需要的时候临时创建而浪费的时间开销，在需要的时候线程随时已准备好。更进一步地考虑时间响应问题，空闲未指派任务的线程是否阻塞也会影响响应速度。如果线程未指派任务时不阻塞而是空转，那么创建线程组后能立即执行相应的任务函数而不需要唤醒的时间开销。但是这样空闲的线程也将占用 CPU 资源并产生不必要的能耗。

第二个问题是线程数量的限制问题。首先，线程是否无限供给还是按处理器数目进行限制，这个是关于线程总数的限制问题，一般编译器应该限制最大的线程数量，以免生成超过合理数量线程。其次是关于是否允许嵌套并行的问题，如果编译器不支持嵌套，那么对于所有内层的并行域都不再提供新的线程，从而使得内部并行域都是串行完成的（可以进行任务划分，但是由一个线程完成）。允许嵌套将获得一定的灵活性，比如计算任务本身就有以下特性：外层的并行性较低，仅靠外层的并行度所产生的线程数量比可拥有的处理器（或处理器核）的数量要小，那么可以通过内层并行域产生出足够的线程来使用其他处理器。但是允许嵌套后的编译器实现技术比较复杂，需要在多个环节处理层间关系，反之不允许嵌套，那么在实现上会有所简化，在并行域的管理上速度有所提升。如果计算任务在外层并行域中已经创建了足够的并发线程，那么不支持并行嵌套有可能获得更好的性能。表 6.1 给出了这两者差异的简单比较。

表 6.1 嵌套与非嵌套实现上的差异

项目 类型	嵌套	无嵌套
实现复杂度	较复杂	较简单
层次支持	完全	部分
速度	较快	较慢
灵活性	高	低

6.2.3 线程层次关系

因为 OpenMP 语法上支持嵌套，所以无论编译器在线程供给上是否支持嵌套，都需要处理并行域的层次关系。在不支持嵌套的编译器中，内层并行域都是使用一个线程在执行，但是并不表示它可以忽视内层并行域。由于并行域会涉及变量共享或私有属性问题，直接忽视内层

并行域（将它们按串行代码处理）将可能引起错误。所以即使不提供内层并行域的线程供给，也要维护其层次关系，才能保证共享或私有变量问题、归约操作问题的正确性。下面来看一个例子以表明直接忽略内层并行域的制导指令引起的错误。

```
1. int i=10;
2. int j;
3. #pragma omp parallel firstprivate(i) reduction(+:j)
4. {
5.     i=i+10;
6.     j=i;
7. }
8. printf("j=%d\n",j);
```

如果上面的代码出现在外层的并行域之内，属于嵌套并行。假设当前线程的数目为 8，根据代码的本意，内层并行域执完之后将打印出“j=160”，即在内层并行域内每个线程都有 $j=20$ ，但是退出内层并行域时需要执行归约操作，使得外部的线程变量 j 将是 8 个线程私有变量 j 的求和。如果编译器不支持嵌套并行，想直接忽略掉内层的制导指令，那么对应的代码等效如下：

```
1. int i=10;
2. int j;
3. /* #pragma omp parallel firstprivate(i) reduction(+:j) */
4. {
5.     i=i+10;
6.     j=i;
7. }
8. printf("j=%d\n",j);
```

此时应当打印出“j=20”，出现了错误，因此即使在线程数量上不支持并行嵌套，也要正确处理内层编译制导指令。

线程层次关系的维护需要相应的数据结构来保存相关信息。在该数据结构中需要保存以下相关信息：

1. 线程的层次关系首先需要维护包括父子关系、兄弟关系等信息，例如使用一个整数来表示嵌套层次、通过指针指向父节点、由父结点管理兄弟节点等等。
2. 共享变量逐层能够由外向内层传递的问题，这个将在第 8 章讨论。
3. 编号标识（用 `omp_get_thread_num()` 获得的线程编号）问题，同一个线程在不同层次并行域中可能有不同的编号，而不同并行域上的不同线程可能会拥有相同的组内线程编号。
4. 最后是当内层并行域退出后，剩下的线程如何恢复外层并行域的相关信息的问题。

下面以图 6.1 为例。在 T_1 时间点，初始线程在串行执行任务；到了 T_2 时间点，`level 0` 的线程 0 号线程（记为 `level 0-0`）和 1 号线程（记为 `level 0-1`）处在同一个并行域中并分担 `level 0` 的计算任务；到了 T_3 时间点，`level 0-0` 变为 `level 1-0`，与 `level 1-1`、`level 1-2` 和 `level 1-3` 共同分担 `level 1` 的任务；而到了 T_4 时间点，`level 1-0` 又返回到 `level 0` 并行域变成 `level 0-0`；到了 T_5 时间点 `level 0-0` 结束任务分担返回到初始线程。

6.2.4 并行域代码封装与标识

将并行域内的代码进行封装和标识，将它们从原来串行代码中的一个结构块变换成为方便并发线程的执行的目标。一个线程在不同时刻可能执行不同的并行域的代码，因此线程和并行域代码不是一次性固定绑定的，所以这些封装好的代码还应该有标识或名字，运行线程在合适的时候找到它们。比较直接的方法是将并行域内的代码封装在一个函数中，根据名字就可以标识。具体形式在 6.4.2 和第 9 章中讨论。

6.2.5 任务分担问题

产生并行域的主要目的就是利用并发线程进行任务分担以加快运行速度。但是由于并行域嵌套问题，以及一个并行域内的多个带有 `nowait` 的任务分担域将引发任务分担上的难题。

当程序执行到一定时候，可能会有多个嵌套并行域同时存在，一个线程可能同时处于不同并行域中，那么当前线程应当和哪些线程协作来完成所分担的任务就是一个重要的问题。因此在前面层次结构所使用的数据结构中，应当填写上当前线程组正在对哪个任务进行分担执行。仍以图 6.1 为例，T2 时间点 level 0-0 和 level 0-1 分担的任务是不同于 T3 时间点 level 1-0、level 1-1、level 1-2 和 level 1-3 所分担的计算任务的。

还有一种情况。如果在一个并行域内有多个带有 `nowait` 子句的任务分担域，也由于一个线程需要确定到底在完成哪个任务，并行域管理需要提供相应的支持。以图 6.3 为例说明。

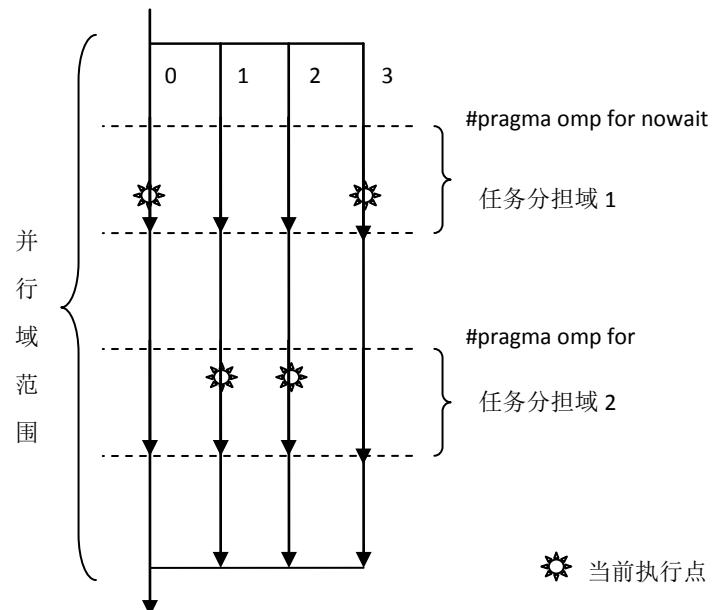


图 6.3 多个 `nowait` 任务分担域

由于线程 1、2 执行较快，跨越了第一个任务分担域进入到第二个任务分担域，而线程 0、1 仍然在第一个任务分担域。在并行域管理中需要提供相关的协助信息让这些线程能正确地分担任务。

如果知道了哪个计算任务是当前线程所需要分担的，那么如何在多个线程中分担则是需要考虑的第二个问题，如何分担任务的问题将在第 7 章中讨论。

6.3 目标代码形式

对于 OpenMP 的并行域管理功能的实现，一部分在代码变换中体现，另一部分在运行库中提供支持。对于并行域如此复杂的功能，全部依靠代码翻译变换来完成并不现实，因此许多功能就只好压入到运行库中。如此一来，编译工作关于并行域管理的部分反而变得简单，而运行库则需要负担更多的工作。此时在代码变换上只留下了启动并行域的代码（包括执行任务函数）、任务函数的封装两大任务。可行的目标代码形式如图 6.4：

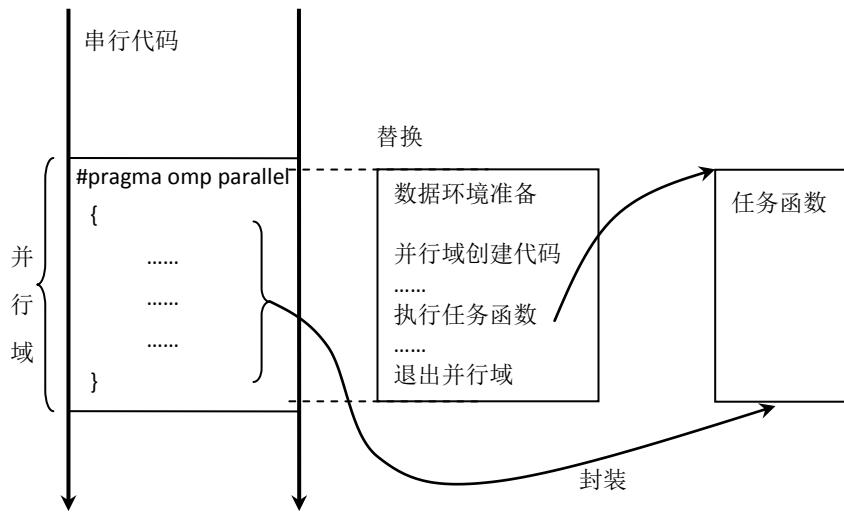


图 6.4 parallel 的目标代码框架

在 GCC 的 GOMP 中变换前后的代码形式如下：

```
1.      #pragma omp parallel          变换前的代码
2.      {
3.          body;
4.      }
```

下面是变换后的代码，分成两部分：

```
1.      void subfunction (void *data)    这个是任务函数
2.      {
3.          use data;
4.          body;
5.      }
.....
6.      setup data;                  数据共享、私有化等处理
7.      GOMP_parallel_start (subfunction, &data, num_threads);  创建并行域
8.      subfunction (&data);           调用任务函数
9.      GOMP_parallel_end ();       退出并行域
```

上面的代码变换与图 6.4 基本一致，GOMP 中的 `GOMP_parallel_start()` 和 `GOMP_parallel_end()` 等函数是 GOMP 运行库中的函数。OMPi 的实现略有不同。

这里只是给出实现 OpenMP 并行域管理的代码基本形式，与具体的 C 编程语言绑定的代码“框架”将在第 9 章给出。

6.4 OMPI 的并行域管理

在 OMPI 编译器中，围绕着一个统一接口 ORT/EELIB 和一个线程控制块 EECB 的概念，清晰地实现了 6.2 小节的 5 大功能。ORT（OMPI runtime）是一个抽象的 OpenMP 线程管理库，它通过 EELIB 接口来使用 EELIB 库提供的线程。因此可以对不同的线程库编写 EELIB 接口库，就可以在 OMPI 中使用各种线程。

由于并行域管理的大多数功能在 ORT 运行环境中，因此代码变换部分显得简单得多。

6.4.1 ORT 统一界面

OMPI 编译器将并行域管理交由 ORT（OMPI Runtime）负责，并且隔离了具体使用的底层线程库。

在 OMPI 编译器中，编程者看到的是称作 OpenMP 线程的实体，OMPI 实现这些 OpenMP 线程时却可以使用 POSIX 线程或其他体系结构相关的线程，甚至可以用重量级的进程来实现。所以在 OMPI 中有一个抽象的执行体（Execute Entities, EE）概念用于描述 OpenMP 线程，而所有的管理都是基于这些执行体的概念，因此与底层线程库的支持是无关的。各种线程技术通过 EELIB（Execute Entities Library）封装后给 ORT 使用。

ORT 和 EELIB 的相互关系可以用图 6.5 表示。可以看出 ORT 是 EELIB 的用户，它在并行域入口处向 EELIB 发出请求，调用 ee_request() 和 ee_create()，获取指定数量的线程。EELIB 在可用线程数目不足的情况下可能提供少于请求数量的线程。如果 EELIB 不支持嵌套并行，那么除了第一层的并行域有多个线程在执行外，所有内层的并行域只能由一个线程来执行（在内层并行域的入口处执行 ee_request() 将返回 0，表示没有其他可用线程）。主线程在并行域出口处调用 ee_waitall() 进入阻塞状态，直到所有并行域内的其他线程退出为止。

除了初始的主执行体（Master Entity）外，所有的执行体都是由 EELIB 提供的。EELIB 还负责提供三种锁：普通锁、嵌套锁和自旋锁，前两种锁在应用程序中通过 API 函数调用使用，自旋锁由 OMPI 的 ORT 模块内部使用。EELIB 只负责提供执行体和锁机制外什么也不管，剩下的全部由 ORT 负责。

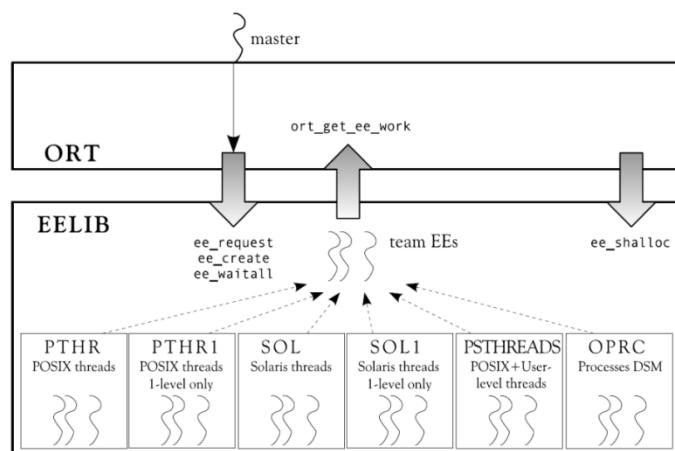


图 6.5 ORT 与 EELIB 接口示意图

从图 6.5 可以看出 OMPI 支持多种线程库，可以是 POSIX 线程、Solaris 线程、PStread 或进程等，加上是否支持嵌套并行，可以形成多种形式的组合，OMPI v1.0.0 提供了以下的线程接口：

- 1) 针对非嵌套并行进行优化的 POSIX pthreads 线程库，在 OMPI 中命名为 pthreads1；
- 2) 允许嵌套并行的 POSIX pthreads 线程库，在 OMPI 中命名为 pthreads；
- 3) 针对非嵌套并行进行优化的 Solaris 线程库，在 OMPI 中命名为 solthr1；
- 4) 允许嵌套并行的 Solaris 线程库，在 OMPI 中命名为 solthr；
- 5) 高性能的 PStread 线程库，在 OMPI 中命名为 psthread；

OMPI 并不能动态的改变所使用的底层线程库，而是必须在编译 OMPI 时指定使用哪种具体的线程。一旦指定后，只有一个 EELIB 的实现代码会生效（OMPI 源代码目录中的 lib 目录下有多个不同的目录对应不同线程技术）。由于基于线程和基于进程来实现 EE 执行体的差别比较大，所以 EELIB 提供的函数分别有 othr_XXXXX 的形式和 oproc_XXXXX 的形式，它们通过 lib/ort_priv.h 中的宏定义统一映射到 ORT 的 ee_XXXXX 函数中，基于线程的宏定义映射参见图 6.6，基于进程的映射则改为 oproc_XXXXX 的函数。

Lib/ort_priv.h 中的宏定义进行影射

```

#define ee_key_t          othr_key_t
#define ee_key_create     othr_key_create
#define ee_getspecific    othr_getspecific
#define ee_setspecific    othr_setspecific

#define ee_initialize     << othr_initialize >>
#define ee_finalize       << othr_finalize >>
#define ee_request        << othr_request >>
#define ee_create         << othr_create >>
#define ee_yield          othr_yield
#define ee_waitall        << othr_waitall >>

#define ee_lock_t          othr_lock_t
#define ee_init_lock      << othr_init_lock >>
#define ee_destroy_lock   << othr_destroy_lock >>
#define ee_set_lock        << othr_set_lock >>
#define ee_unset_lock     << othr_unset_lock >>
#define ee_test_lock      << othr_test_lock >>

#if defined(AVOID_OMPI_DEFAULT_BARRIER)
#define ee_barrier_t      othr_barrier_t
#define ee_barrier_init    othr_barrier_init
#define ee_barrier_destroy othr_barrier_destroy
#define ee_barrier_wait    othr_barrier_wait
#endif

```

Lib/ort.c 中进行调用 Lib/ee_pthreads/othr.c 中实现

图 6.6 EE 统一接口

表 6.2 是 ORT 与 EELIB 的接口函数列表。

表 6.2 基于线程的 EELIB 接口函数表

	函数名	功能
接 口 函 数	othr_initialize()	创建线程池、提供 EELIB 能力信息
	othr_finalize()	清理工作
	othr_request()	申请线程“配额”
	othr_create()	取得线程池中的工作线程并指派任务
	othr_waitall()	等待并行域内所有其他线程结束
	othr_init_lock()	创建并初始化一个锁
	othr_destroy_lock()	销毁一个锁
	othr_set_lock()	加锁
	othr_unset_lock()	解锁
	othr_test_lock()	测试

6.4.2 并行域代码变换

OMPI 将 `parallel` 指令指定的并行域代码封装成一个函数，然后由多个线程去执行以实现 `fork-join` 的执行模型。下面我们来看看 `parallel` 制导指令编译前后的代码。假如有以下具有 `parallel` 制导指令的代码：

```
1. #include <omp.h>
2. int main(void)
3. {
4.     #pragma omp parallel
5.     {
6.         printf("task1: myid is %d\n",omp_get_thread_num());
7.     }
8.     #pragma omp parallel
9.     {
10.        printf("task2: myid is %d\n",omp_get_thread_num());
11.    }
12.    return 0;
13. }
```

此处先后创建了两个并行域，域内的线程并发地打印输出自己的线程 ID 标识。对应上面的代码，经 OMPI 编译后的目标代码的第一部分如下：

```
1. static void * _thrFunc0_(void * __me)
2. {
3.     {
4.         printf("task1: myid is %d\n",omp_get_thread_num());
5.     }
6.     return (void *) 0;
7. }
.....
8. static void * _thrFunc1_(void * __me)
```

```

9.    {
10.    {
11.        printf("task2: myid is %d\n", omp_get_thread_num());
12.    }
13.    return (void *) 0;
14. }

```

此处可以看出，编译器将需要并发执行的任务（此处是打印信息）封装到一个线程任务函数中，两个并行域内的代码分别封装并命名为`_thrFunc0_`和`_thrFunc1_`。其次编译器还将产生第二部分代码，并在第二部分代码中产生多个线程来执行上面的`_thrFunc0_`和`_thrFunc1_`函数：

```

1. int __original_main(int _argc_ignored, char ** _argv_ignored)
2. {
3. {
4.     ort_execute_parallel(-1, _thrFunc0_, (void *) 0);
5. }
6. {
7.     ort_execute_parallel(-1, _thrFunc1_, (void *) 0);
8. }
9. return (0);
10. }

```

`ort_execute_parallel()`函数将启用多个线程来执行指定的`_thrFunc0_`和`_thrFunc1_`函数，从而实现`fork-join`中的`fork`操作，当这些线程执行完`_thrFunc0_`（或`_thrFunc1_`）函数经`ort_execute_parallel`函数返回后，只剩下主线程继续往下运行，从而结束并行域（`ort_execute_parallel()`含有`join`操作）。

此处`__original_main()`是 OMPI 编译器将初始源代码中的`main()`更名修改产生的函数，具体变换过程参见第 9 章。

6.4.3 线程管理与控制

OMPI 的线程管理与控制的功能主体在运行库 ORT 中。ORT 只能感知到执行体 EE (execute entity)，因此所有的并行域的管理也是基于 EE 的概念进行的，而执行体的描述则靠 EECB (EE Control Block) 执行体控制块来完成，EECB 有点类似于 OS 中的 PCB 的概念。另外作为真正的线程实体，OMPI 还采用线程池来管理，这样一来一个线程从线程池取下，可能因为该线程进入不同的并行域而具有多个不同的 EECB 控制块。

下面将讨论 OMPI 的 ORT/EELIB 运行环境在线程供给和层次管理中的作用。

线程池

为了避免每次进入和退出并行域的`fork-join`操作时都去创建和撤销线程，节约相应的时间开销，OMPI 实际上是维护了一个线程池。当需要的时候将线程从池中去取出，让它执行线程任务函数完成计算任务，任务完成后将线程放回线程池中。

线程池是一个链表队列，管理的是抽象的线程（注意与 EECB 相区分），与线程池相关的全局变量如下：

1. othr_pool_t *H = NULL;
2. volatile int plen; /* # threads in the pool-list */
3. othr_lock_t plock; /* A lock for accesing the pool-list */

H 指向线程池中线程链表的头节点，plen 用于记录当前线程池中保有的线程数目，plock 用于多个并发线程访问线程池（各自作为新的内层并行域的组长时）的互斥保护。

下面是线程池用来记录池中的一个线程所使用的数据结构 othr_pool_t：

1. typedef struct othr_pool_s {
2. void *(*workfunc)(void *); /* What function to execute */ 线程任务函数
3. void *arg; /* The function's argument */ 实际上是组长的 EECB
4. int id; /* A sequential id given by OMPI */
5. void *info;
6. int spin; /* Spin here waiting for work */ 空闲，自旋标记
7. struct othr_pool_s *next; /* Next node into the list */ 构成链表
8. } othr_pool_t;

所有可用的线程都有一个唯一编号 id（不同于 OpenMP 线程编号），它们通过 next 指针构成一个单向链表。当线程处于空闲状态未指派任务时 spin 自旋标志为有效，此时线程空转但是不进入阻塞，以便一旦有任务时可以快速进入工作状态。

线程池的逻辑结构如图 6.7 所示，EELIB 在初始化（ee_initialize()/othr_initialize()）的时候就建立了线程池，并用 H 指向第一个线程。所有线程在此处使用一个 othr_pool_t 类型的数据结构来记录，它们形成单项链表，共有 plen 个，使用 othr_lock_t 类型的全局变量 plock 作为访问互斥锁。

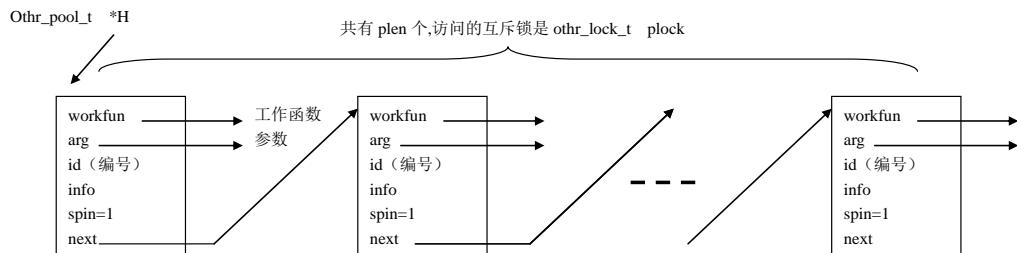


图 6.7 线程池的结构

othr 线程组中的线程是从线程池中取下来的，此时各个 othr 线程的任务函数可能相同也可能不相同，各自的组内 id 分别为 0 到 numthr（线程组的大小），而其中的 info 变量是共享的（并且由锁 rlock 来控制互斥访问），如图 6.8 所示。

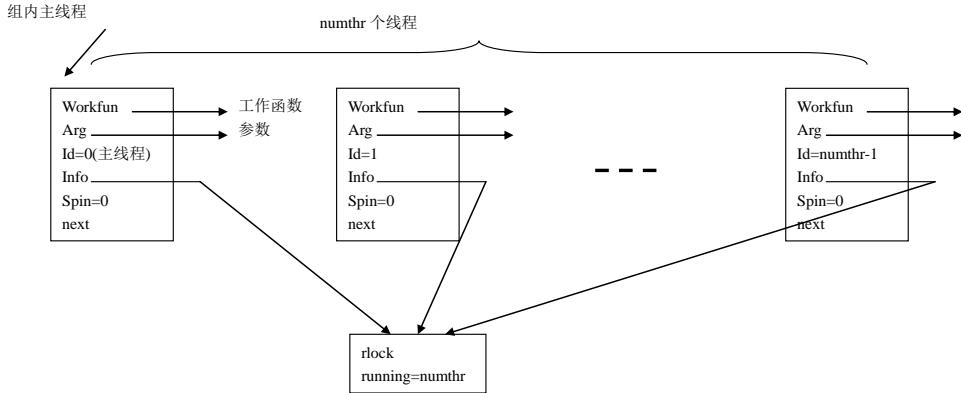


图 6.8 线程组关系图

EECB 及层次

EECB 是 OMPI 运行环境中非常重要的数据结构，大量操作都依赖于它所记录的信息。EECB 是一个与并行域相关的动态概念，它将跟随一个线程进入不同的并行域而变化，每当一个线程创建一个并行域后，它自己也成为新的并行域中的一个子线程，因此它自己也要切换到新生成的 ECB 上。而线程池节点数据结构是静态概念，它总是和一个底层线程固定联系的。下面是 ECB 的声明代码。

```

1. 1. typedef struct ort_eecb_s ort_eecb_t;
2. 2. struct ort_eecb_s {
3. 3.     ee_barrier_t      barrier;      /* Barrier for my children */
4. 4.     int    have_created_team; /* 1 if I have been a team parent in the past */
5. 5.     int    num_children;
6. 6.     void   *(*workfunc)(void *);    /* The function executed by my children */
7. 7.     ort_workshare_t workshare;    /* Some fields volatile since children snoop */
8. 8.     ort_cpriv_t      copyprivate;   /* For copyprivate; owner stores data here
9. 9.                                and the rest of the children grab it from here */
10. 10.    ort_eecb_t *parent;
11. 11.    int thread_num;        /* Thread id within the team */
12. 12.    int num_siblings;      /* # threads in my team */
13. 13.    int level;            /* At what level of parallelism I lie */
14. 14.    void *shared_data;    /* Pointer to shared struct of current function */
15. 15.    ort_eecb_t *sdn;       /* Where I will get shared data from; normally
16. 16.                                from my parent, except at a false parallel
17. 17.                                where I get it from myself since I am
                                the only thread to execute the region.
                                */
15. 15.    int mynextNWregion;   /* Non-volatile; I'm the only thread to use this */
16. 16.    int chunklb;          /* Non-volatile; my current chunk's start iteration;
                                changes for every chunk I execute through
                                ort_thischunk_lb(); only used in ordered_begin() */
17. 17.    int nowaitregion;     /* True if my current region is a NOWAIT one */

```

```

18.     /* Thread-library specific data */
19.     void *ee_info;           /* Handled by the ee library */
20. };

```

下面我们逐个简单地介绍 EECB 结构体成员的作用。

第 3~8 行的成员变量是给当前线程所创建的子线程所使用的。如果当前线程是一个并行域的发起者，那么本线程当前的 EECB 中的 barrier 用于其子线程同步时使用。它可以通过宏定义映射到 othr_barrier_t 的类型，最终由 pthreads 库的锁来实现。变量 have_created_team 用于记录自己是否创建了并行域中的线程组，1 表示创建了线程组。变量 num_children 用于记录自己所创建的子线程数目。函数指针 workfunc 指出子线程需要执行的任务函数。如果子线程需要进行任务分担，那么 workshare 变量记录了任务分担的信息。变量 copyprivate 用于 copyprivate 数据子句的情况，拥有数据的线程将 copyprivate 子句限定的私有变量保存在自己的 copyprivate 结构体中，其他同一个并行域中的子线程从这里获得数据，从而实现私有数据在并行域内的广播。

第 9~24 行的变量是线程自己使用的。指针 parent 指向自己的父节点线程(有可能是自己)。变量 thread_num 记录了自己的组内编号，由于不是全局编号，所以不同并行域中的线程会有相同编号，如果需要区分相同编号的线程则需要辨识它们的祖先。变量 num_siblings 记录了组内线程的数量。变量 level 记录了并行域的嵌套层次。指针 shared_data 指向当前线程的共享数据区。指针 sdn 指出共享数据所在的线程的 EECB，一般来说它将指向父节点线程，但是对于只有一个线程的并行域的情况就指向自己。变量 mynextNWregion 用于出现 nowait 子句的情况下。变量 chunklb 用于记录 for 循环的任务分担时的循环变量起点，仅用于动态调度和指导调度。变量 nowaitregion 用于标记当前处于带有 nowait 子句的任务分担域中。指针 ee_info 指向当前 EELIB 的基本信息。

图 6.9 是 EECB 构建层次关系的示意图，注意同一个底层线程在进入不同的并行域层次时将使用不同的 EECB。

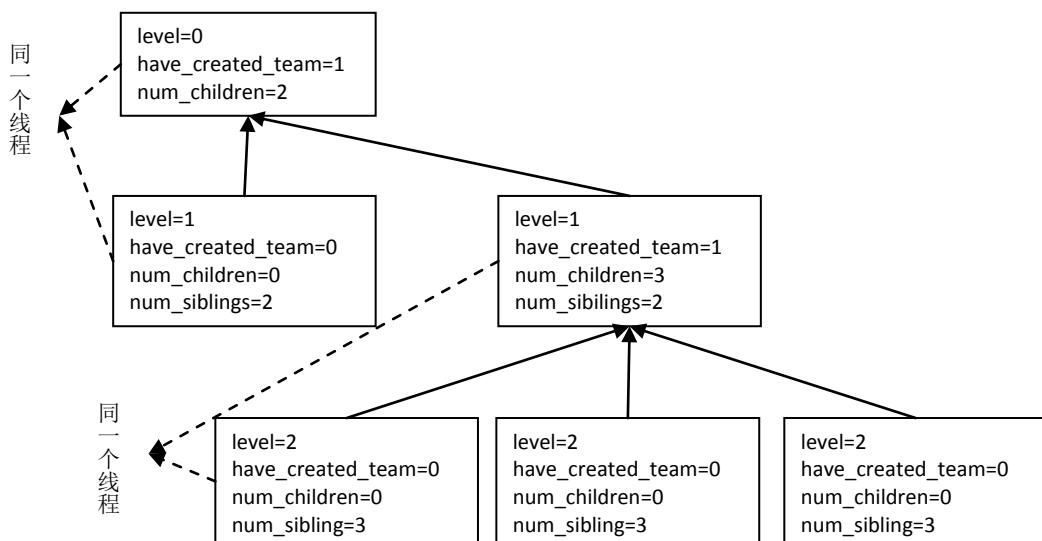


图 6.9 EECB 构成层次关系

图 6.10 用于说明线程池的描述对象和 EECB 之间的关系。

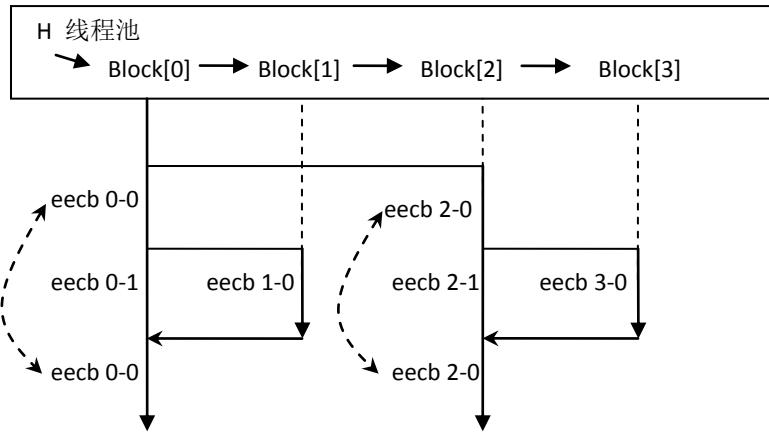


图 6.10 线程池对象与 ECB 关系图

其中 $\text{eecb } x-y$ 表示线程池中第 x 个线程正处于并行嵌套的第 y 层中，另有组内线程号未标示在图中。

供给与指派

当需要创建并行域时，作为线程组的创始者需要调用 `ee_request()`/`othr_request()` 向线程池提出请求，此时如果剩下的线程数目（`plen`）少于请求的数目，那么只将剩余的全部线程作为配额返回给请求者。OMPI 默认情况下不支持超过处理器数目的线程，因此一旦线程池中的线程全部分配出去之后，再请求新的线程将得不到满足。

当从线程池获得一定数量的线程配额，需要从线程池中将它们取下，并给它们指派任务。这些工作通过调用 `ee_create()`/`othr_create()` 来完成：从线程池 `H` 头指针处逐个取下所需要的线程；为该线程提供任务函数（线程池节点的 `workfunc` 成员变量）；将线程池节点的 `spin` 置为 0，从而使该线程脱离空闲状态以执行指派的 `workfunc` 任务函数。

任务分担信息

如果并行域内的任务分担控制指令没有 `nowait` 子句，那么任务分担信息是比较容易实现的，只需要对子任务进行标识编号即可完成任务分配。但是如果有 `nowait` 子句，快的线程可能跨越当前的任务分担域，进入到下一个任务分担域，这时候 OMPI 在运行环境中保持一个全局数据，用于记录这些同时发生的任务分担情况。对于无 `nowait` 的情况，任务分担问题在第 7 章讨论，而关于 `nowait` 情况的问题在 10.1.3 小节讨论。

6.4.4 总览

我们将线程池、EECB 层次结构以及执行流程综合起来，可以用图 6.11 来表示。图中实线表示线程的执行，虚线表示线程处于空闲状态（未执行任务）。当一个线程执行 `ort_execute_parallel()` 进而调用 `ee_create()` 函数后，将从线程池获得新的线程以生成新的并行域。进入新的并行域的各个线程将创建相应的 ECB 控制块，以维护其层次结构（线程池节点并不维护层次结构）。当并行域内的线程构建好 ECB 层次结构后，将调用相应的任务函数并发执行。当并行域内的线程执行完毕后，组长将调用 `ort_wait_all()` 函数等待组内成员到达后再继续往下运行，从而退出并行域。退出并行域后各个线程的 ECB 自动撤销，组长将恢复原先

的 EECB。当一个线程从线程池取出时，将退出原来的空转状态，进而调用 `ort_get_thread_work()` 获得任务函数。

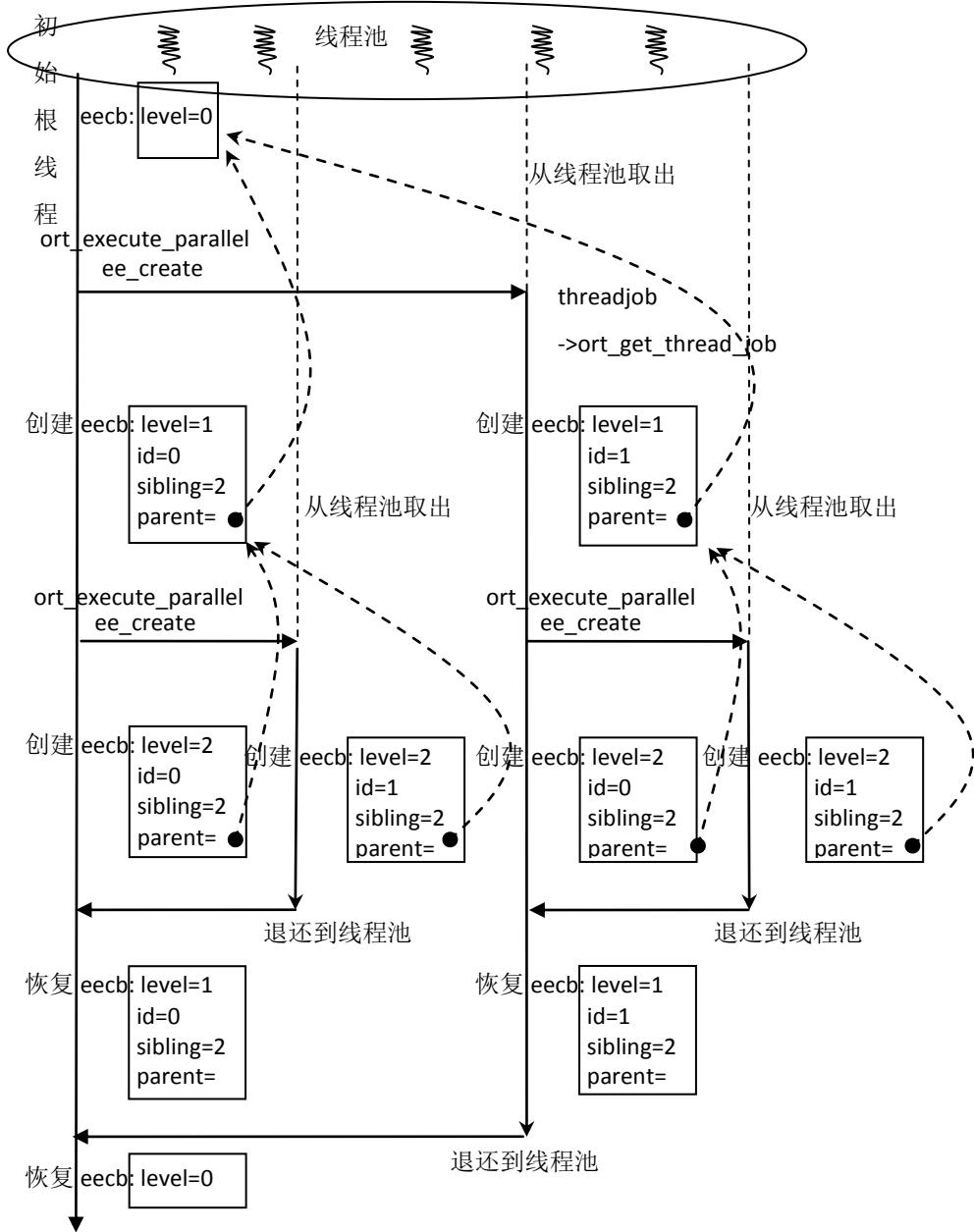


图 6.11 EECB 线程控制总览（2 层嵌套）

并行域中的 OpenMP 线程可以使用操作系统中的进程或线程来实现。每当遇到#pragma omp parallel 制导指令，那么编译器就需要将后面的代码用多个进程或线程来执行，而且还需要保证共享变量和私有变量的作用域不发生错误。

6.5 小结

本章分析了 OpenMP 并行域的管理以及如何用多线程 C 语言代码来实现其并行语义。首先分析了并行嵌套情况，了解并行域管理的复杂性，然后对并行域管理的 5 大问题分别讨论：线程无关接口、线程供给、线程层次关系、并行域代码封装及任务分担问题，并给出了目标代

码的基本形式。指出并行域管理的重点在于运行库，代码的编译变换工作并不复杂。最后以 **OMPi** 代码分析来讲述如何实现上述 5 大功能：利用 **ORT** 库和 **EELIB** 分层结构实现线程无关接口、给出了 **parallel** 制导指令变换的框架代码和任务函数的封装、分析了线程管理中的线程池和线程控制块 **EECB** 数据结构、如何利用 **EECB** 维护线程层次关系、线程供给和指派。任务分担信息将在 10.1.3 小节进一步讨论。

本章内容是 **OpenMP** 并行语义的核心所在，编译器必须提供高效的并行域管理，否则导致 **OpenMP** 的 **fork-join** 的执行模型效率低下。因此编译器尽可能采用高效的线程库并用线程池的办法来管理工作线程。

第 7 章 任务分担与线程同步

`OpenMP` 提供了用于任务分担的制导指令和线程同步制导指令。其中的任务分担制导指令有三种：`for`、`sections` 和 `single`，一般在并行域内使用。`for` 制导指令只用于 `for` 循环语句，编译器需要产生出多个线程分别执行部分循环变量对应的循环体计算任务。`sections` 语句用于多个能并发执行的复合语句块，每一个复合语句块都是一个 `section`。而 `single` 语句是在并行域内说明某一个复合语句块是只能由一个线程执行。同步语句用于协调并行域内线程的相互时序制约关系。

下面分别对三种制导指令和 7 种线程同步制导指令及如何用 C 代码来实现进行讨论，主要分析串行代码和并行代码之间的语义差距，具体的变换过程和方法将在第 9 章中讨论。

7.1 `for` 制导指令

串行代码中的 `for` 语句只是简单地将循环变量从初始值开始，逐个递增直到满足结束条件，其间对每一个循环变量的取值都将执行一遍循环体内的代码。所有这些循环体语句的执行是逐次串行地完成的。但是作为 `OpenMP for` 制导指令所指出的 `for` 循环而言，不同的循环变量取值所对应的循环体代码的执行是可以并发进行的。因此这两者之间的语义差别就需要通过 `OpenMP` 编译器来弥补。在经典的编译原理“龙书”³第二版的第十一章“并行性和局部性优化”的内容是这里的支撑原理，读者有需要可以参阅。由于 `OpenMP` 的并行性已由用户指出并负责其中的数据相关性问题，因此要比编译器自行挖掘并行性要容易得多。

`for` 制导指令相对 `sections` 和 `single` 而言，略为复杂一点，因为 `for` 制导指令由于调度策略和 `chunk` 大小的因素将会涉及到循环变量的划分问题。

7.1.1 `for` 任务分担

一个 `for` 循环语句实际上就是用所有的循环变量取值去执行循环体内的语句，如果我们将所有循环变量取值用 `i` 进行编号，设共有 `N` 个取值就有 $i=0,1,2,\dots,N-1$ ，那么它对应的循环体语句的计算任务就可以记为 { `taski` | ($i=0,1,2,\dots,N-1$) }。这样一来对 `for` 循环的任务分担实际上就是对循环变量进行划分，然后让各个并发线程各自负责一部分循环变量并完成相应的计算工作。

根据 `OpenMP` 标准，在 `OpenMP` 程序中能够进行负载分担的循环变量必须是整数类型，不能是其他类型的变量，因此对于循环变量描述比较简单。假设循环变量为 `i`，记计循环变量

³ Alfred 等, Compilers: Principles, Techniques, & Tools, 2nd, Pearson Education, 2006。中译版由机械工业出版社出版。

初值为 l_{init} , 循环步长为 l_{step} , 循环变量终止值为 l_{end} 。根据 l_{step} 的正负符号不同, 用式-1 分别定义循环变量上界 UB (Upper Bound) 如下:

$$UB = \begin{cases} l_{end} + 1, & \text{if } l_{step} > 0; \\ l_{end} - 1, & \text{if } l_{step} < 0; \end{cases} \quad \text{式 1}$$

图 7.1 是循环变量在数轴上的示意图, 上面是循环增量 l_{step} 为正的情况, 下面对应 l_{step} 为负的情况。

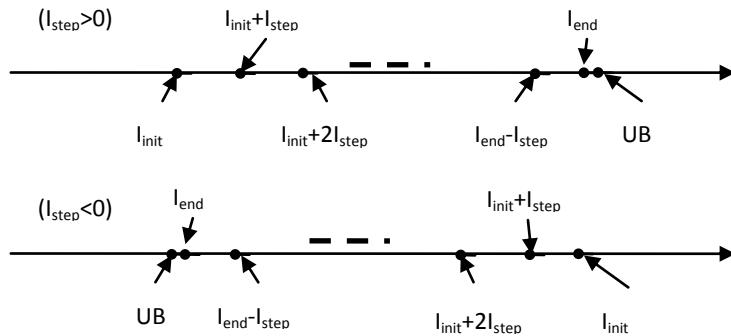


图 7.1 for 循环变量示意图

使用 UB 的好处是能够将多种 for 语句的形式统一成相对固定的形式, 即条件表达式中可以避免使用 “ \leq ” 或 “ \geq ”, 只需要用 “ $<$ ” 或 “ $>$ ”。例如 `for(l=linit; l<=lend; l=l+lstep)` 可以借助 UB 改写成 `for(l=linit; l<UB; l=l+lstep)`。这使得编译器所需要处理的情况和目标代码的形式有所减少, 简化其实现复杂度, 具体可见后面 OMPI 的实现部分。

为了描述循环变量的划分, 首先将所有循环变量的集合记为 Ψ , 无论 $l_{step}>0$ 还是 $l_{step}<0$ 都可以用式-2 表示:

$$\Psi = \{l_{init} + k * l_{step} \mid k = 0, 1, 2, \dots, \lfloor |UB - l_{init}| \rfloor / l_{step}\} \quad \text{式 2}$$

如果有 P 个线程的并行域要对这个 for 语句进行任务分担的话, 我们需要将 Ψ 分解成 P 个子集 (式-3), 记为 Ψ_p ($p=1, 2, \dots, P$) 而且满足子集间没有交集 (式-4), 即:

$$\Psi = \bigcup_{p=1}^P \Psi_p \quad \text{式 3}$$

$$\Psi_i \cap \Psi_j = \emptyset, \text{ if } i \neq j \quad \text{式 4}$$

各种调度方式的差别将造成 Ψ 分解的不同, 具体的分解方案受到调度方式 (static、dynamic、guided 和 runtime) 和 chunk 的大小所影响。静态调度的分解方案在 for 循环执行之前就已经产生, 而动态调度和运行时调度则在运行时实时产生出分解方案。

7.1.2 循环变量分解原则

这些循环变量子集的划分方法将直接影响任务完成的速度, 它主要体现在两个方面, 一个是任务的均衡程度, 另一个是循环体内的数据访问顺序与相应的 cache 冲突情况。所以循环变量的分解原则有:

- 1) 分解代价低;
- 2) 任务的计算量要均衡;
- 3) 尽量避免 cache 行冲突。

首先需要时间上快速的分解方案产生方法，因为这些代价都将计入并行开销中，高的开销意味着低的加速比。所以能用静态产生的方法尽量用静态分割方法。

其次要求分解方案产生的各个计算任务负载均衡。由于 `for` 语句隐含有路障，也就是说在 `for` 语句的入口处是线程 `fork` 操作，而出口处是 `join` 操作。如果因负载不平衡使得各个线程执行 `for` 循环体的时间不相等，那么最慢完成的线程将会造成其他所有线程空等的现象，这必将降低处理器效率和加速比。因此最理想的情况是所有线程同时到达 `for` 出口，然后执行 `join` 操作，没有线程需要等待其他线程到达这个同步路障。如果不能同时到达，那么也要尽量减少线程的等待时间。

静态地均匀划分循环变量来分解任务并不能总是保证负载均衡性，有时候可能还需要动态分解。假如各个循环变量对应的计算量不相同，那么即使循环变量子集划分时做到了完全均衡，也不能避免因计算量的差异造成的线程先后完成的现象（参见 2.2.2 任务分担小节的 `for` 调度）。另外即使计算负载完全均衡，但是由于处理能力的差异也会引起计算时间不等。对于这些情况，可以使用动态调度的分解机制，动态调度时，速度快的线程将会执行更多的任务，并保证各个线程的等待时间不超过一个 `chunk` 任务的执行时间。

第三个原则指的是循环变量划分方案尽量能让各个线程执行 `for` 循环体的代码时，充分考虑其中的代码访问和数据访问顺序与 `cache` 的关系。不过这个需要对循环体内部数据访问进行分析，当前编译器技术难以到达相应的能力。比较可行的办法是用户给与相应的提示，编译器根据相应的提示作一些优化。例如对一个简单的循环 `i` 变量（0,1,...98,99），循环体内语句所访问的都是 `x[i]`、`y[i]` 或 `z[i+1]` 的形式的变量，那么很大程度上可以判断 4 个处理器按照（0,4,8,12,16,...）、（1,5,9,13,17,...）、（2,6,10,14,18,...）、（3,7,11,15,19,...）的静态分解，就不如（0,1,2,3,...24）、（25,26,27,28,...）、（50,51,52,53,...）和（76,77,78,79,...）的分解方案好。前者和可能会引起较多 `cache` 行的竞争和 `cache` 的 ping-pong 效应、更低的 `cache` 命中率，假如处理器 1 在读写 `x[0]y[0]` 和 `z[1]`，处理器 2 在读写 `x[1]y[1]` 和 `z[2]`，那么 `x[0]` 和 `x[1]` 以及 `y[0]` 和 `y[1]` 都将可能在同一个 `cache` 行内，将因 ping-pong 效应引入不必要的数据传输和时延（无论 `cache` 一致性是采用何种协议来维护的）。若是对更复杂的数据访问情况，比如指针、结构体或对象的成员变量的访问等情况，不仅编译器自动分析的难度很大，就是人工分析也不一定有效。

7.1.3 目标代码功能

为了实现 `for` 制导指令的任务分担，编译器根据原来的串行代码产生出来的目标代码需要具备以下功能：

- 1) 记录串行循环变量的所有取值。具体方法可以是定义若干整型变量，分别保存循环变量的初始值、终值（或上界）以及递增步长。
- 2) 循环变量的划分功能。可以根据不同的调度机制，使用静态或动态的办法并按指定的 `chunk` 大小（如果指定了的话）进行划分。

- 3) 各线程能根据所分配的循环变量子集来执行各自的 `taski` (循环体内语句的计算任务)。由于此时的循环变量需要变成线程内部的私有变量，因此还涉及一些变量重新声明和辅助变量的定义。
- 4) 标记 `for` 任务分担代码的进入和退出信息。并行域内的多个线程可以根据相关的信息来确认自己所处位置和应该执行的代码。

并行域内的多个线程首先通过检测到 `for` 任务分担的标记从而知道各线程间需要协作完成一个 `for` 循环，然后通过分配给自己的循环变量子集来执行经过变换后的循环体内的代码，最终各线程完成自己的任务从而退出本次的 `for` 任务分担。默认情况下在 `for` 任务分担的结束处需要一个路障操作（除非显式地使用 `nowait` 子句说明）。

7.1.4 目标代码形式

编译器对 `for` 语句的编译，只需要完成任务分担，而不负责并行域的产生和管理。也就是说，如果没有 `parallel` 编译制导指令而直接使用 `for` 编译制导指令，一般 OpenMP 编译器将产生出串行执行的目标代码。所以一般情况下 OpenMP 编译器设计时可以不处理没有 `parallel` 语句包裹的 `for` 制导指令，也可以对 `for` 制导指令相关的代码进行任务分担，但是由于没有产生并行域，这些分解出来的任务实际上还是由一个线程逐个顺序地完成。无论哪种情况，前面讨论的循环变量分解方法都是适用的。

目标代码必须实现前面提到的四个功能，但是具体形式上是可以有多种实现方式的。

关于循环变量取值空间等问题可以通过声明变量并进行赋值的形式来完成，原始代码中的循环变量信息需要用共享变量来传入到各个线程中，而各个线程自己的循环变量子集可以采用私有变量形式来存储；变量的划分可以直接插入一段代码来完成，也可以将这些功能代码放到运行库中的某个函数中，目标代码中只出现相应的函数调用语句即可。

子任务 `taski` 的执行可以是将原来循环体内的代码略作修改而直接出现在目标代码中以结构块实现的（出于代码的可阅读性也可以封装成一个目标代码中的函数）。

`for` 任务分担区域的进入和离开标记可以采用对一些标记变量进行赋值的办法，或者通过一个封装好的函数来设置这些标记信息。参与任务分担的线程需要根据这些信息进行协调。

下面给出一种可行的静态调度的目标代码框架形式：

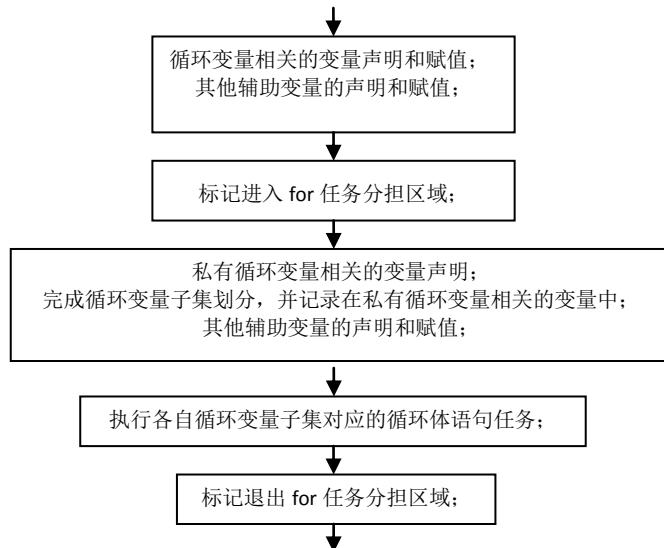


图 7.2 for 任务分担的目标代码框架

GCC 的 GOMP 对缺省调度模式的 for 制导指令变换前的形式如下：

1.	#pragma omp parallel for	变换前的代码
2.	for (i = lb; i <= ub; i++)	
3.	body;	循环体代码

下面是变换后的代码：

1.	void subfunction (void *data)	
2.	{	
3.	long _s0, _e0;	
4.	while (GOMP_loop_static_next (&_s0, &_e0))	获得当前循环变量的子集
	{	
5.	long _e1 = _e0, i;	
6.	for (i = _s0; i < _e1; i++)	
7.	body;	循环体代码
8.	}	
9.	GOMP_loop_end ();	
10.	}	

变换前的循环变量范围是 (lb,ub) , 而变换后的任务函数中每次通过 GOMP_loop_static_next() 取得循环变量的子集 (_s0,_e0) 。其中 GOMP_loop_static_next() 和 GOMP_loop_end() 是 GOMP 运行库中的函数。

7.1.5 OMPI 的 for 制导指令

OMPI 在编译 for 制导指令的时产生的目标代码主要有三部分：

1. 首先是循环变量相关的一些变量声明和赋值；
2. 其次是循环变量分解的代码；
3. 最后是做了修改的循环体。

下面先给出 OMPI 编译器对 `for` 制导指令的编译处理输出的目标代码样例，分析 OMPI 在不同调度机制下的循环变量分解方法，然后再讲述其代码变换和相应的运行库函数功能。

根据 OpenMP 标准，调度方式有三种，分别是 `static` 静态、`dynamic` 动态、`guided` 指导和 `runtime` 运行时调度四种。但是作为编译器，它应该处理五种情况：缺省（不带任何调度子句）、静态、动态/指导和运行时调度。缺省调度虽然是一种特殊的静态调度，但是由于缺省调度没有指定 `chunk` 大小，需要系统给出一个缺省的 `chunksize`，所以它实际上等效于不带 `chunksize` 的静态调度。

默认调度（无调度子句或 `schedule(static)`）

默认调度方式是针对于不带任何调度子句的 `for` 制导指令：

```
1. #pragma omp for
2.     for (i=0;i<=100;i++)
3.     {
4.         XXXXXXXX;
5.     }
```

经 OMPI 编译后的目标代码为：

```
1. {
2.     int i;                                //声明循环变量
3.     int from_=0, to_=0, step_;           //循环变量范围的有关变量
4.     struct _ort_gdopt_gdopt_;
5.
6.     step_=1;                            //循环变量变化步长
7.     ort_entering_for(1, 0, 0, step_, &gdopt_);
8.     if (ort_get_static_default_chunk(0, (100) + 1, step_, &from_, &to_))
9.     {
10.         for (i = from_; i < to_; i = i + 1) //i+1 对应的是循环增量
11.         {
12.             XXXXXXXX;
13.         }
14.     ort_leaving_for();
15. }
```

此时如果在外层使用`#pragma omp parallel`，那么这里的代码将作为线程任务并行执行。变量 `i` 由于声明在线程创建之后，是线程私有的循环变量（堆栈变量或自动变量）。各个线程所拥有的这个 `i` 循环变量具有不同的初值 `from_`，终值上界 `to_`，循环变量步长 `step_`。`From_` 和 `to_` 两个变量刚开始被赋予 0，然后在 `ort_get_static_default_chunk(0, (100) + 1, step_, &from_, &to_)` 的时候完成子集分解的，前三个传入的参数分别是初值 0、终值上界 `(100) +1` 和步长 `step_`（这些变量是在源代码语法分析时获得，在代码变换时填入），返回值 `from_` 和 `to_` 代表了本线程的循环变量子集起止范围。划分的方式是尽可能均衡地为每个线程分配一个连续的循环变量变化区间。

此处需要用到三个运行库函数。`ort_get_static_default_chunk()` 用于循环变量的分解，`ort_entering_for()` 用于任务分担的准备工作，而 `ort_leaving_for()` 用于退出 `for` 任务分担区域。

循环体内的代码“XXXXXXXX”被原封不动地保留，各个线程执行都是相同的代码段，只是各自采用不同的循环变量值而已。

静态调度（带 chunksize）

当静态调度指明了 chunksize 时，循环变量的划分必须以 chunksize 为单位进行。假设有如下的初始串行代码：

```
1. #pragma omp for schedule(static,5)
2.     for (i=0;i<=100;i++)
3.     {
4.         XXXXXXXX;
5.     }
```

经过 OMPI 编译输出的变换后将变换成如下的代码：

```
1. {
2.     int i;
3.     int from_= 0, to_= 0, step_;
4.     struct _ort_gdopt_gdopt_;
5.     int nchunks_, chid_, TN_, StCtN_;

6.     step_= 1;
7.     ort_entering_for(1, 0, 0, step_, &gdopt_);
8.     TN_= omp_get_num_threads();           获得线程数目
9.     StCtN_= step_* 5 * TN_;           计算循环变量子集步距，此处 5 是 chunksize
10.    ort_init_static_chunksize(0, (100) + 1, step_, 5, &nchunks_, &from_);
11.    from_-= StCtN_;
12.    to_= from_ + step_* (5);
13.    for (chid_= omp_get_thread_num(); chid_ < nchunks_; chid_ += TN_)
14.    {
15.        from_+= StCtN_;
16.        to_+= StCtN_;
17.        if ((100) + 1 < to_)
18.            to_= (100) + 1;           //边界的特殊处理
19.        for (i = from_; i < to_; i = i + 1) //对应于 1 个 chunk 的工作
20.        {
21.            XXXXXXXX;
22.        }
23.    }
24.    ort_leaving_for();
25. }
```

OMPI 中的带 chunk 参数的静态调度和默认的调度方式略有不同。由于 OMPI 按照每个线程每次处理一个 chunk 对应的任务，所以其外部循环用于控制哪一个 chunk 是当前任务，而内部循环用于控制当前 chunk 对应的循环变量。可以用图示的办法表示出这种情况下的循环变量划分方法（假设使用了 3 个线程），参见图 7.3。

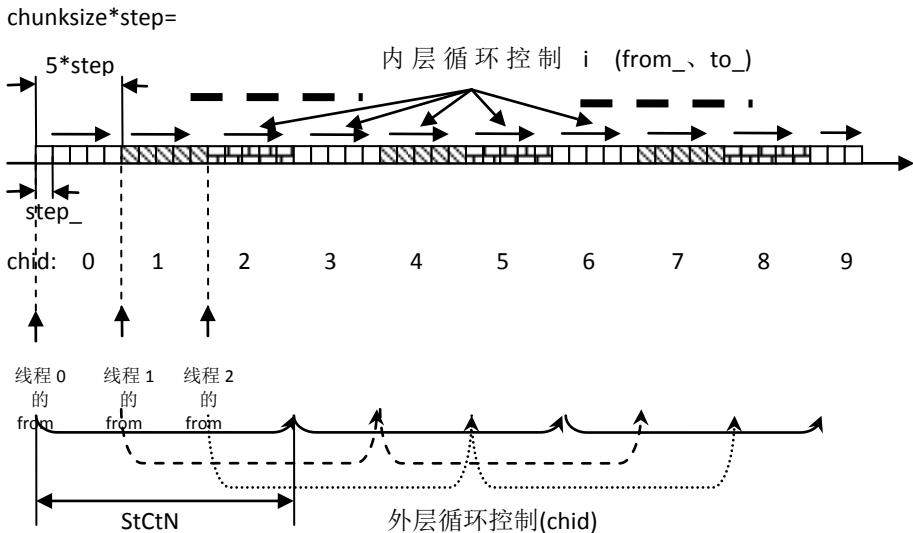


图 7.3 for 循环静态调度

对于整个循环变量的变化范围，根据 `chunksize` 的大小（此处是 5）划分成多个 `chunk` 块，每块包含 `chunksize` 个循环变量的取值（跨度为 `chunksize*step`），这些 `chunk` 块将从 0 开始逐个编号为 0、1、2...。每个线程将获得各自 `chunk` 的分配，这个是通过外层循环来实现的。每个线程分配的首个 `chunk` 块编号就是各自的线程编号，0 号线程获得的首块 `chunk` 编号为 0，1 号线程获得的首块 `chunk` 编号为 1，以此类推，因此由不同线程调用 `ort_init_static_chunksize()` 函数或返回的 `from` 起点是不同的。每个线程的 `chid` 按照线程数目来递增，因此 0 号线程获得 `chid` 为 0、3、6、9，1 号线程获得 `chid` 为 1、4、7，2 号线程获得 `chid` 为 2、5、8，这些 `chid` 的跨度为线程数目 `TN_`。每个线程获得的 `chunk` 块数可能是不相等的，这个需要通过 `ort_init_static_chunksize(...,&nchunks,...)` 函数来计算获得块数 `nchunks`。对于所分配到的每一个 `chid`，各个线程将用内部循环 `for (i = from_; i < to_; ...)` 来获得该 `chunk` 块对应的循环变量 `i` 的取值，进而执行循环体内的代码。由于最后一个 `chunk` 可能不完整，因此要特别对待。

动态调度及指导调度

前两种调度都是在代码变换后就已经确定的静态调度，下面来考察一下具有动态性的 `dynamic` 动态调度和 `guided` 指导调度，这两种调度方式在 OMPI 中按相同的方式来处理（区别在于调用 `ort_get_dynamic_chunk()` 函数还是 `ort_get_guided_chunk()` 函数）。假设有动态调度的 `for` 制导指令代码如下：

```

1. #pragma omp for schedule(dynamic,5)
2.     for (i=0;i<100;i++)
3.     {
4.         XXXXXXXX;
5.     }

```

OMPI 编译输出的变换后代码如下：

```

1. {
2.     int i;
3.     int from_=0,to_=0,step_;
4.     struct _ort_gdopt_gdopt_;           // 用于带有 nowait 子句情况的处理

```

```

5.
6.     step_= 1;
7.     ort_entering_for(1, 0, 0, step_, &gdopt_);
8.     while (ort_get_dynamic_chunk(0, 100, step_, 5, &from_, &to_, (int *) 0, &gdopt_))
9.     {
10.         for (i = from_; i < to_; i = i + 1)
11.         {
12.             XXXXXXXX;
13.         }
14.     }
15.     ort_leaving_for();
16. }

```

在动态调度中，循环变量的变化区间也是划分成多个 chunk 块的，每个块共有 chunksize 个取值。各个线程利用外层 while 循环借助于 `ort_get_dynamic_chunk()` 来获取一个 chunk 块，该 chunk 使用 `form_` 和 `to_` 变量来描述，然后用内层循环将该 chunk 块的所有循环变量 `i` 的取值去执行循环体代码。当外层 while 循环调用 `ort_get_dynamic_chunk()` 时返回值为 0 则表示所有 chunk 都已经被执行完毕，此时可以执行 `ort_leaving_for()` 退出 for 任务分担区域。

图 7.4 表示了动态的调度过程，此时线程 0、1、2 各自获得了一个使用私有变量 `from_` 和 `to_` 描述的 chunk 块，各自利用这些参数去执行循环体代码，一旦执行完毕将再次调用 `ort_get_dynamic_chunk()` 获得下一个 chunk。此时的分配不再是固定静态的，而是哪个线程完成所分配的 chunk 后将获得下一个 chunk。

在 OMPI 编译器中，guided 调度虽然调用了不同的 `ort_get_guided_chunk()` 函数，但是它在分配 chunk 的方法是相似的。

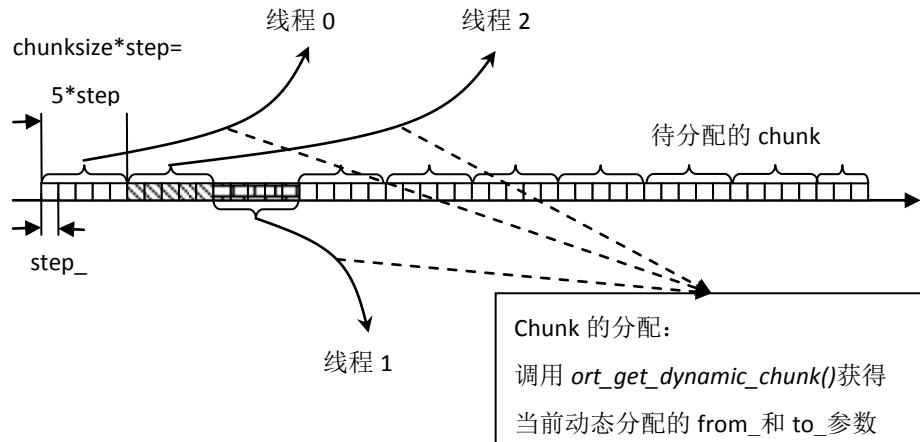


图 7.4 动态调度示意图

运行时调度

运行时调度实际上不是一种特定的调度，它只是指在运行时根据 ICVs 的设置来选择 static、dynamic 或 guided 调度方式中的一种。假设由运行时调度的 for 制导指令的初始代码如下：

```

1. #pragma omp for schedule(runtime)
2.     for (i=0;i<100;i++)

```

```

3.     {
4.         XXXXXXXX;
5.     }

```

经 OMPI 编译后输出的变换代码成如下代码：

```

1.  {
2.      int i;
3.      int from_= 0, to_= 0, step_;
4.      struct _ort_gdopt_gdopt_;
5.      int staticextra_= 0, chunkszie_;
6.      int (* get_chunk_)(int lb, int ub, int step, int chunksize, int * from, int * to, int * extra, struct
    _ort_gdopt_(* opt)); //此处只是声明函数类型

7.      step_= 1;
8.      ort_entering_for(1, 0, 0, step_, &gdopt_);
9.      ort_get_runtime_schedule_stuff(&get_chunk_, &chunkszie_);
10.     while ((*get_chunk_)(0, 100, step_, chunkszie_, &from_, &to_, &staticextra_, &gdopt_))
11.     {
12.         for (i = from_; i < to_; i = i + 1)
13.         {
14.             XXXXXXXX;
15.         }
16.     }
17.     ort_leaving_for();
18. }

```

OMPI 为 runtime 方式提供了一个统一的处理形式，dynamic、guided 和 static 共用一个代码框架，它们使用 while ((*get_chunk_)(0, 100, step_, chunkszie_, &from_, &to_, &staticextra_, &gdopt_)) 的统一形式来获得本次处理的 chunk 块，只是 get_chunk_ 函数指针可以指向 ort_get_dynamic_chunk、ort_get_guided_chunk、ort_get_runtimestatic_chunk 三者之一。Get_chunk_ 函数指针到底指向哪个具体的函数，是通过调用 ort_get_runtime_schedule_stuff(&get_chunk_, &chunkszie_) 来设置的。比如 ICVs 中指明静态调度，那么 ort_get_runtime_schedule_stuff() 函数根据 ICVs 中的调度类型有选择地将 get_chunk_ 指向 ort_get_runtimestatic_chunk()，此时与前面静态调度的工作流程完全一样，此时 while 循环只执行一次就完成。ort_get_dynamic_chunk() 和 ort_get_guided_chunk() 在前面已经讨论过。无论最终使用哪种调度方式，循环变量的区间还是划分成多个 chunk 的。

ort_get_runtime_schedule_stuff() 函数代码如下：

```

1. void ort_get_runtime_schedule_stuff(chunky_t *func, int *chunksize)
2. {
3.     *chunksize = ort->icv.rtchunk; /* -1 if not given */
4.     switch (ort->icv.rtschedule)
5.     {
6.         case _OMP_DYNAMIC:
7.             *func = ort_get_dynamic_chunk;
8.             if (*chunksize == -1) *chunksize = 1;

```

```

9.         break;
10.        case _OMP_GUIDED:
11.            *func = ort_get_guided_chunk;
12.            if (*chunksize == -1) *chunksize = 1;
13.            break;
14.        default:
15.            *func = ort_get_runtimestatic_chunk;
16.            break;
17.        }
18.    }

```

到此可以看出，**OMPI** 在静态调度（包括默认调度）中的代码与运行时调度中的静态调度部分代码应该可以重用的。

这些 **for** 制导指令变换后的代码中出现的运行库函数 `ort_entering_for()`、`ort_leaving_for()`、`ort_init_static_chunksize()`、`ort_get_static_default_chunk()`、`ort_get_dynamic_chunk()`、`ort_get_guided_chunk()` 等函数的实现将在第 10 章中讨论。

7.2 sections 制导指令

sections 制导指令是第二种任务分担语句，在多线程并发执行及分担计算任务上与 **for** 制导指令相似，但是由于 **sections** 的计算任务不是 **for** 循环体，因此计算任务负载和代码形式之间的差异可以非常大。

7.2.1 sections 任务分担描述

在 **sections** 制导指令里的代码包含多个可以并发执行的代码段，每个代码段都用 **section** 制导指令指明。检查这些代码段在并发执行时是否存在数据相关性并不是编译器的责任，这个应该由 OpenMP 编程者自己负责。

我们可以对所有的 n 个 **section** 进行编号，记为 S_1, S_2, \dots, S_n ，此时所有的 **section** 任务可以用集合 Ψ 来表示如式-5：

$$\Psi = \{S_i \mid i = 0, 1, 2, \dots, n\} \quad \text{式 5}$$

如果有 P 个线程的并行域要对这个 **sections** 语句进行任务分担的话，我们需要将 Ψ 分解成 P 个子集（式-6），记为 Ψ_p ($p=1, 2, \dots, P$) 而且满足子集间没有交集（式-7），即：

$$\Psi = \bigcup_{p=1}^P \Psi_p \quad \text{式 6}$$

$$\Psi_i \cap \Psi_j = \emptyset, \text{if } i \neq j \quad \text{式 7}$$

这些划分后的 **section** 子集可以由 P 个线程并发地完成以加快执行速度。所以这里也面临着和 **for** 制导指令相似的问题，就是如何进行划分才能获得最佳性能。

7.2.2 section 划分原则

`section` 的划分原则与 `for` 制导指令中的循环变量划分原则既相似又不同。此处的划分一样需要具有代价低、任务负载均衡的要求，以确保获得较高的性能。但是由于 OpenMP 程序中的 `for` 循环往往处理数组形式的数据对象，因此考虑 `cache` 冲突是很重要的问题，而在 `sections` 情况下由于代码差异性较大，经常是各自处理不同数据对象，因此由于线程的共享数据而引发的 `cache` 冲突不那么明显。另外一个问题是各个 `section` 的代码形式往往不具有可比性（`for` 循环中的每个循环变量取值对应的任务执行相同的代码从而具有相同的任务负载），所以一般很难判断任务的划分是否均衡。如果用户不能比较准确的判断出各个 `section` 的负载情况并合理划分，那么可能对性能有较不利的影响。

7.2.3 目标代码功能

为了实现 `sections` 制导指令的任务分担，编译器根据原来的串行代码产生出来的目标代码需要具备以下功能：

- 1) 记录所有 `section` 任务。具体方法可以是定义一个数组用于保存 n 个 `section` 代码段的相关信息。
- 2) 对 n 个 `section` 的进行划分和分配的功能。可以使用不同的调度机制，但是出于负载均衡的考虑往往用动态的办法进行划分。
- 3) 各线程能根据所分配的任务子集来执行相应的 `section` 代码段。需要一种根据 `section` 编号来判断执行哪个 `section` 代码段的机制，以及 `section` 代码的封装。
- 4) 标记 `sections` 任务分担代码的进入和退出信息。并行域内的多个线程可以根据相关的信息来确认自己所处位置和应该执行的代码。

并行域内的多个线程首先通过检测到 `sections` 任务分担的标记从而知道各线程间需要协作完成多个 `section` 代码段，然后静态或动态地获得自己所分配的 `sector` 号进而执行相应的 `section` 代码，最终各线程完成所有分配的任务从而退出本次的 `sections` 任务分担。默认情况下在 `sections` 任务分担的结束处需要一个路障操作（除非显式地使用 `nowait` 子句说明）。

7.2.4 目标代码形式

编译器对 `sections` 语句的编译，只需要完成任务分担，而不负责并行域的产生和管理。也就是说，如果没有 `parallel` 编译制导指令而直接使用 `sections` 编译制导指令，一般 OpenMP 编译器将产生出串行执行的目标代码。所以 OpenMP 编译器设计时一般可以不处理没有 `parallel` 语句包裹的 `sections` 制导指令，也可以对 `sections` 制导指令相关的代码完成任务分担，但是由于没有产生并行域，这些分解出来的任务实际上还是由一个线程逐个顺序地完成。无论哪种情况，前面讨论的 `sections` 任务分解方法都是适用的。

目标代码必须实现前面提到的四个功能，但是具体形式上是可以有多种实现方式的。

关于 `section` 任务编号的记录问题，可以用数组、链表等形式来实现，所保存的信息包括编号和代码起点位置（地址指针等）。`section` 任务的表示可以用函数也可以用复合语句来封

装，不管用复合语句还是函数来封装，都可以用 `switch/case` 语句或 `if` 语句来有选择地执行分配给自己的相关代码。

`sections` 任务分担区域的进入和离开标记可以采用对一些标记变量进行赋值的办法，或者通过一个封装好的函数来设置这些标记信息。参与任务分担的线程需要根据这些信息进行协调。

下面给出一种可行的动态调度的目标代码框架形式：

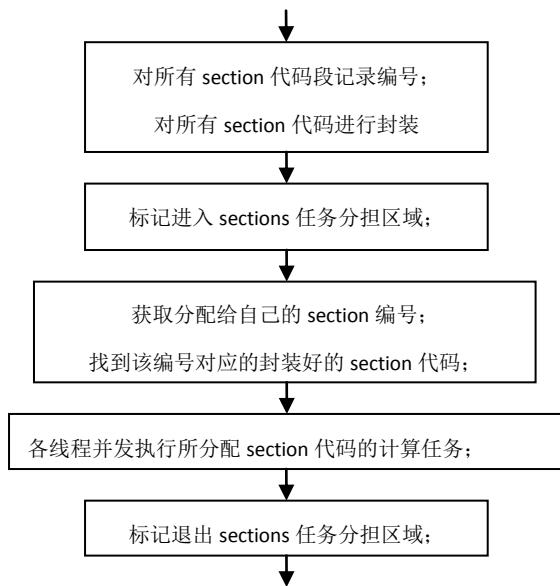


图 7.5 sections 任务分担的目标代码框架

下面看看 GCC 的 GOMP 对 `sections` 制导指令的变换形式。

```
1.      #pragma omp sections          变换前的代码
2.      {
3.          #pragma omp section
4.          stmt1;                  第一个 section
5.          #pragma omp section
6.          stmt2;                  第二个 section
7.          #pragma omp section
8.          stmt3;                  第三个 section
9.      }
```

下面是经过翻译变换后的代码

```
1.      for (i = GOMP_sections_start (3); i != 0; i = GOMP_sections_next ())  获取本线程应该执行的 section 编号
2.          switch (i)           根据编号选择不同的任务来执行
3.          {
4.              case 1:
5.                  stmt1;          第一个 section
6.                  break;
7.              case 2:
8.                  stmt2;          第二个 section
9.                  break;
10.             case 3:
```

```

11.           stmt3;          第三个 section
12.           break;
13.       }
14.       GOMP_barrier();      sections 隐含的同步操作。

```

GCC 的 GOMP 对 sections 的变换框架和 OMPI 的变换框架非常相似。

7.2.5 OMPI 的 sections 制导指令

OMPI 在编译 sections 制导指令时产生的目标代码主要有三部分：

1. 首先是记录 section 代码段相关信息的一些变量声明和赋值；
2. 其次是动态获得任务分配的代码；
3. 最后是用 switch 语句进行封装修改后的 section 代码。

下面先给出 OMPI 编译器对 sections 制导指令的编译处理输出的目标代码样例，分析它的分解方法，然后再讲述其代码变换和相应的运行库函数功能。

根据 OpenMP 标准，没有提供对 sections 任务分担的调度方式的设置以及相应的划分参数。但是作为编译器，它可以提供静态或动态的调度机制或者类似于 chunksize 的粒度信息。

下面是一个具有两个 section 段的 OpenMP 未编译的原始源代码。

```

1.     #pragma omp sections
2.     {
3.         #pragma omp section
4.         {
5.             XXXXXXXX;          section 任务 1
6.         }
7.         #pragma omp section
8.         {
9.             YYYYYYYY;          section 任务 2
10.        }
11.    }

```

经 OpenMP 编译后，将输出如下的目标代码：

```

1.     {
2.         int caseid_ = -1, inpar_;
3.
4.         if ((inpar_ = (omp_in_parallel() && omp_get_num_threads() > 1)) != 0)
5.             ort_entering_sections(1, 2);
6.         for (;;)
7.         {
8.             if (inpar_)
9.             {
10.                 if ((caseid_ = ort_get_section()) < 0)      获取任务编号
11.                     break;
12.                 }
13.             else

```

```

14.         {
15.             if ((++caseid_) >= 2)
16.                 break;
17.             }
18.         switch (caseid_)          根据任务编号执行相应任务
19.         {
20.             case 0 :
21.                 {
22.                     XXXXXXXX;           section 任务 1
23.                 }
24.                 break;
25.             case 1 :
26.                 {
27.                     YYYYYYYY;           section 任务 2
28.                 }
29.                 break;
30.             }                      //end switch
31.         }                      //end for
32.     if (inpar_)
33.         ort_leaving_sections();
34. }

```

首先例子中的两个 `section` 段分别用复合语句的形式进行封装在一个 `case` 语句里，参见上面第 18~30 行代码。每个线程所分配到的 `section` 编号用 `caseid_` 来记录，总的 `section` 数目为 2 (见第 5、15 行代码)。

OMPI 编译器在这里区分了外层是否有“`#pragma omp parallel`”制导指令形成的并行域，用 `inpar_` 变量来标记进入并行域，并根据这个变量来确定是采用串行方式执行还是并行方式执行。

如果 `inpar_` 为假则表示没有并行域，就按照串行方式执行当前的 `sections` 相应的代码。此时的 `for` 循环将根据第 15 行的 `++caseid_` 来遍历所有的 `section` 编号，逐个地完成第 18~30 行代码中的两个 `section` 段 (即 `XXXXXXX;` 和 `YYYYYYY;`)

如果 `inpar_` 为真那么表示已经建立了并行域，按照并行方式来执行，各个线程将动态地调用 `ort_get_section()` 来获得分配给自己的 `section` 编号 `caseid_`，然后将在后面的 `switch/case` 语句中有选择地执行相应的代码。

进入 `sections` 任务分担区域的时候先调用 `ort_entering_sections(1, 2)` 标记进入，退出时则调用 `ort_leaving_sections()`。

7.3 single 制导指令

`single` 制导指令用于并行域中某段代码只由一个线程执行的情况，编译器可以将这段代码封装到一个 `if` 语句的成功分支中，只有一个线程可以通过这个成功分支执行这段代码，其他线程将进入不成功分支 (如果没有 `nowait` 子句，进入不成功分支的代码需要等待成功分支的

线程完成才能继续往下执行）。**OMPI** 编译器利用 `ort_mySingle()` 来判断一个线程是否进入成功分支，该函数还具有同步路障功能。假设有关于 `single` 制导指令限定的代码如下：

```
1. #pragma omp single  
2.     XXXXXXXX;
```

经 **OMPI** 编译后将输出目标代码如下：

```
1. if (ort_mySingle(0))  
2. {  
3.     {  
4.         XXXXXXXX;  
5.     }  
6.     ort_leaving_single();  
7. }
```

线程进入 `ort_mySingle()` 函数后，只有最早调用该函数的线程返回值为 1（或者让线程 0 返回值为 1），其他线程将阻塞在这个函数中。当第一个线程执行完 `single` 限定的代码后调用 `ort_leaving_single()` 函数解除其他线程的阻塞或空转状态，于是所有线程同步地退出 `single` 区域（其他线程将进入不成功分支，而此时的不成功分支为空，所以继续往下执行代码）。

7.4 nowait 问题

对于 `for`、`sections` 和 `single` 任务分担的制导指令，在任务分担域结束时往往有隐含的路障同步（此时称为阻塞式任务分担），因此前面讨论的任务分担机制没有什么问题。但是如果出现 6.2.5 小节图 6.3 的情况，用 `nowait` 语句显式地要求先完成任务的线程可以跨越该路障（此时称为非阻塞式任务分担），进入到下一个任务分担域，而此时这个新的任务分担域里面所需的计算任务量在编译时是未知的，因此这个到来的执行体该如何划分和确定自己的计算任务量呢？这就引发线程应该承担哪个任务分担域的任务的问题。因此需要一个数据结构来跟踪各个线程各自处于那个任务分担域中，并为它们在分担任务时提供必要信息。

假如对于 `sections` 的任务分担是如下处理的，第 x 个 `section` 分配给第 x 个到达的线程，此时的 x 计数需要专门的变量来处理；如果 `single` 语句将任务分给第一个到达的线程，那么需要一个变量或标记表示已有线程承担该工作其他线程可以跳过，当最后一个线程通过时还需要对相应变量或标记进行复位。当遇到非阻塞的任务分担时，一个计数变量或标记是不够的。对于这些问题可以分成三种：

1. 在一个并行域内，有多种不同类型的任务分担制导指令；
2. 在一个并行域内，有多个同类型的任务分担制导指令；
3. 在一个并行域内，循环地使用一个任务分担制导指令。

对于第一种情况可以使用三个分别对应于 `for`、`sections` 和 `single` 的不同计数变量来处理。第二种情况要复杂的多，似乎也可以给每一个任务分担与创建一个计数变量来处理。**OpenMP** 编译器在编译源代码时，计算并行域内的任务分担域的数量，并创建相应数量的计数变量提供给运行环境使用，每个计数变量都有独立标识。上面的解决方式似乎比较简单，但实际上并不容易实现。如果在该并行域内有函数调用从而出现外部的孤立的任务分担制导指令，此时难以

统计任务分担域的数量，如果这个函数是外部的库函数那么几乎是不可能统计到这个任务分担域的。即便第二个问题解决了，第三个问题更复杂，例如：

```
1. #pragma omp parallel
2. {
3.     for (i = 0; i < 10; i++) /* This is not a parallel for */
4.     {
5.         #pragma omp single nowait
6.         {
7.             <code>
8.         }
9.         <code>
10.    }
11. }
```

这个 `single` 语句由于在循环语句内，因此被执行了 10 次。每个线程都将遇到这个 `single` 任务分担域 10 次，但是由于个线程执行速度的差异以及 `nowait` 的作用，使得在某个时刻各个线程进入任务分担域的次数（不是执行任务分担域内的代码）是不同的。因此这里虽然有一个任务分担域，但是不能只用一个计数变量来处理，而应该是 10 个。

对此类问题的常见的解决方法有两种，一是提供一个记账（bookkeeping）的方法，动态地分配用于任务分担域的数据结构、或者就像 Omni 编译器那样不允许 `nowait` 生效。前者开销太大，后者不够灵活效率低。

OMPI 采用一个折中的办法，它在执行体的父结点的控制块上维护了一个固定长度（长度为 `MAXWS`）的工作分担队列，用于记录各个活动的任务分担域的计帐信息。这样就不需要区分是不同的任务分担域还是循环执行同一个任务分担域，把上面三个问题统一起来了。这些记账信息包括：

- 1) 与制导指令相关的信息，如执行 `single` 代码的线程号，`sections` 区域里面剩下的 `section` 数量，`for` 区域里面循环上限和增量、执行体访问所需的锁等等信息；
- 2) 队列自身的信息，进入该并行域的执行体数目，离开该执行体的执行体数目等。

当活动任务分担域数量达到 `MAXWS` 时，就不再允许执行体继续进入到下一个任务分担域里面去，这时候相当于不允许新的 `nowait` 生效。但是如果活动的并行域还没有到达 `MAXWS`，则可以继续进入其他任务分担域，而且不需要动态分配相关数据的开销。所以它是介于任意数量的活动任务分担域的技术和完全不允许多个活动任务分担域之间的一个折中方法，避免了前者的动态分配操作开销（OMPI 早期 0.8.2 版本采用动态分配的办法，但是由于开销太大而在 0.8.3 版本开始采用固定办法）和后者人为地限制并行性能这两个问题。

OMPI 记账信息的实现在第 10 章讨论。

7.5 归约操作

在任务分担中，可能需要在这些并发线程之间执行某些归约操作。归约操作涉及并行域内的线程私有变量和共享变量（变量数据环境问题将在第 8 章中讨论），以 `sum` 求和归约为例，需要将所有线程的某个私有变量逐个求和，该过程可以串行也可以并行，然后将求和结果保存

在外部共享变量中。无论是串行还是并行求和，其计算过程的同步关系都需要小心处理，以避免因数据竞争而出现错误结果。如果采用串行求和，那么只要保证每次求和过程的原子性即可，使用互斥锁加以保护就可以实现，这是速度较慢但是比较简单的方式。如果采用并行求和可以采用两两求和的树形求解，理论上速度较快，但是归约操作只对一个简单的变量进行的话，涉及的线程不是非常巨大时可能并不需要并行求解，否则并行开销可能大于归约操作的计算时间。

编译器需要给归约变量生成两个变量，第一个是保存外部结果的共享变量，其次是各个线程自己的私有变量。然后编译器要能够输出执行归约操作的代码。在并发执行的过程中线程使用的是私有变量，退出任务分担域或并行域的时候完成归约操作并将结果保存在外部共享变量中。

如果一个带有 reduction 归约操作子句的代码段如下：

```
1. #pragma omp parallel reduction(+:k) //或者 parallel for/parallel sections 等
2. {
3.     XXXXXXXX;
4. }
```

此处假设变量 int k 在外部作用域已经声明（比如是全局变量或当前函数的前面声明过），经过 OMPI 编译器输出的目标代码如下：

```
1. {
2.     struct __shvt__ {
3.         int (* k);                                指向外部分作用域中的归约变量（共享变量）
4.     };
5.     struct __shvt__ * _shvars = (struct __shvt__ *) ort_get_shared_vars(__me);
6.     int k = 0;                                    内部作用域中的归约变量
7.     ...                                         //如果有 for/sections 等，将插入其他代码
8. {
9.     XXXXXXXX;
10. }
11. ort_reduction_begin(&_paredlock0);
12.     *(_shvars->k) += k;                      将内部变量归约操作值修改外部变量
13. ort_reduction_end(&_paredlock0);
14. ....
15. }
```

代码的前面几行用于将获取外部的共享变量 k（所有共享变量保存在一个结构体中，将该结构体指针传入即可），然后通过“int k=0;”声明了线程内部的私有变量（如果对变量使用了 copyin 子句等，还将有其他处理步骤，具体见第 8 章）。在功能代码执行后，最后几行的代码用于完成归约操作。归约操作的第一步是利用互斥锁保证求和的原子性，这通过调用 ort_reduction_begin(&_paredlock0) 来实现，其实就是对_paredlock0 进行加锁操作，然后就可以将私有变量加到共享变量中，然后对_paredlock0 解锁即可。如果有多个变量需要执行归约操作，那么各自会有自己的互斥锁，这些锁拥有统一的形式——_paredlockXX，其中 XX 是从 0 开始的编号。不同的归约变量使用不同的锁可以避免不必要的互斥，出于简化编译器可以只用一个锁来完成互斥同时保持正确性。

7.6 线程同步

OpenMP 为并行域中的代码提供了多种线程同步的制导指令，以保证必要的执行顺序和特殊要求。下面介绍的制导指令有些是用于并行域和各种任务分担制导指令行中，但 `ordered` 只能用于 `for` 制导指令中，`nowait` 用于 `for`、`sections` 和 `single` 指令行。

7.6.1 atomic

`atomic` 制导指令只能对一个简单标量的操作起作用，指定特定的存储单元将被原子更新。编译器需要给相应的操作进行互斥保护，最直接的办法就是使用锁，如果是使用 `pthreads` 库的线程，那么可以用 `pthread_mutex_t` 类型的普通互斥锁，也可以使用 `pthread_spinlock_t` 自旋锁。由于操作都是非常简单的操作，如果有自旋锁就可以避免因线程阻塞而引入的额外开销，所以只要系统提供自旋锁就应该使用自旋锁来作为 `atomic` 制导指令的实现技术。

为了编译该制导指令，编译器首先需要生成一个互斥锁，然后在进入相应的原子性的增减操作语句前调用加锁操作，而在退出相应操作时调用解锁操作。

在 OMPI 编译器中，它将为应用程序生成一个自旋锁保存在变量 `ort->atomic_lock` 中，然后将每个用 `atomic` 说明的原子操作都进行变换。下面以一个例子来说明编译前后的代码形式，假设有如下代码：

1. `#pragma omp atomic`
2. `XXXXXXX;`

经过 OMPI 编译后的目标代码如下：

1. `ort_atomic_begin();`
2. `XXXXXXX;`
3. `ort_atomic_end();`

其中的 `ort_atomic_begin()` 和 `ort_atomic_end()` 分别是对 `ort->atomic_lock` 进行加锁和解锁操作，而锁 `ort->atomic_lock` 的声明在应用程序初始化时完成的。由于 OMPI 对整个程序都使用同一个锁来对 `atomic` 制导指令保护的语句进行互斥操作，不同地方出现的 `atomic` 制导指令本来不需要互斥的也无法并发执行，因此这个约束条件是可以减弱的。可行的方法之一就是每次遇到一个变量的 `atomic` 制导指令就生成一个新的自旋锁并用编号实现区分，这样不同变量的 `atomic` 制导指令就可以并发执行同时确保操作的原子性。

7.6.2 critical

`critical` 与 `atomic` 制导指令相似，只不过 `critical` 可以对更大粒度的操作进行互斥保护。`critical` 制导指令表明指定的代码一次只能由一个线程执行，其他线程被阻塞在临界区外。OpenMP 编译器要做的工作有两项，一个是声明相关的互斥锁，另一个就是对 `critical` 制导指令所控制的代码使用加锁和解锁进行保护。如果不希望因为线程阻塞和调度引起过多的额外开销，可以使用自旋锁（例如 `pthreads` 库的 `pthread_spinlock_t` 类型的锁）。下面通过原始代码和经过 OMPI 编译后的目标代码对比来说明其过程，假设有初始代码如下：

```
1. #pragma omp critical
2. {
3.     XXXXXXXX;
4.     YYYYYYYY;
5. }
```

经 OMPI 编译后输出的目标代码如下：

```
1. ort_critical_begin(&_ompi_crity);
2. {
3.     XXXXXXXX;
4.     YYYYYYYY;
5. }
6. ort_critical_end(&_ompi_crity);
```

其变换过程非常直接，那就是将原有代码用加锁和解锁操作函数封装起来。而所用到的锁 `_ompi_crity` 是一个全局变量（初始化为 `pthread_spinlock_t` 类型的自旋锁），也就是说所有的 `critical` 制导指令指定的代码共用一个锁。这样也引出前面 `atomic` 中相同的问题，那就是互斥条件太强，可以减弱。通过声明多个互斥锁，在各自的 `critical` 出现的地方各自加锁和解锁可以避免这样的问题。

7.6.3 master

OpenMP 的 `master` 制导指令指定代码段只由主线程执行（其他的线程都将忽略这块代码区域），因此编译器需要对所有线程进行编号识别，并让编号为 0 的主线程来执行相关的代码。由于 OpenMP 程序可以通过 `omp_get_thread_num()` 函数来获得自己的线程编号，所以 OpenMP 运行环境必须在并行域中记录各个线程的编号，因此在编译 `master` 制导指令的时候就可以利用并行域处理中产生的编号来实现。这条指令没有相关隐式路障，所以不需要显式的路障操作。

假设有 `master` 制导指令的代码如下：

```
1. #pragma omp master
2.     XXXXXXXX;
```

经 OMPI 编译后输出的目标代码将如下：

```
1. if (omp_get_thread_num() == 0)
2.     XXXXXXXX;
```

此处的编译工作就是把需要由主线程执行的代码封装到 `if(omp_get_thread_num() == 0)` 的成功分支中。其他线程不执行这段代码，也没有路障阻碍其他线程往下继续运行。

7.6.4 ordered

OpenMP 的 `ordered` 制导指令指出其所包含 `for` 循环体内的代码的执行必须满足：

- 1) 在任何时候只能有一个线程执行；
- 2) 表示其封装的循环迭代将按相同顺序执行，就好象它们在一个串行处理器上执行一样。在本线程执行当前迭代的 `ordered` 部分代码之前，线程需要等待前面的迭代对应的 `ordered`

代码完成。指令 `ordered` 提供了在一个循环中需要按顺序执行时的微调方式，否则它就不必要。被 `ordered` 所限定部分只能出现在 `for` 或者 `parallel for` 制导指令行中。

编译器为了能够保证 `ordered` 的语义，必须将并行域中的各个线程所负责的循环变量范围做好记录，并以此作为确定执行顺序的依据。所以相应的代码应该具有以下功能：

- 1) 记录各线程所分担的循环变量范围；
- 2) 在进入 `ordered` 限定的代码之前需要进行同步协调，提供串行执行的能力；
- 3) 确保分得循环变量靠前的线程先执行；

下面根据 OMPI 的编译过程来讲述，假设在 `for` 制导指令所限定的代码中有一段初始代码如下：

```
1. #pragma omp ordered  
2.      XXXXXXXX;
```

经 OMPI 编译后输出的目标代码如下：

```
1. ort_ordered_begin();  
2.      XXXXXXXX;  
3. ort_ordered_end();
```

在 OMPI 编译器中，这种排序的控制是由当前并行域的父结点线程负责的，在当前并行域中的父结点线程中记录了当前已经完成的 `for` 循环变量的范围（或指出后续需要执行的循环变量范围），OMPI 用父结点线程控制块 EECB 中的任务共享数据中的 `next_iteration` 来保存。每个线程都需要记录自己负责的 `chunk` 编号或循环变量的当前起点位置，循环变量的起点信息保存在线程控制块的成员 `chunklb` 中，当调用 `ort_ordered_begin()` 的时候将会根据 `chunklb` 和父结点线程中 `next_iteration` 相比较就可以自动是否轮到自己运行，如果前面的线程还没有执行，那么当前线程需要阻塞或空转等待。当一个线程执行完该代码，调用 `ort_ordered_end()` 函数的时候将调整父线程的 `next_iteration` 变量，从而激活下一个线程的执行。通过这样的排序则可以保证执行顺序与串行执行相同。

7.6.5 nowait

`nowait` 子句只用于 `for`、`sections` 和 `single` 制导指令中，用于表示任务分担域中的线程在完成任务分担后，可以不用执行路障同步，而执行后续的代码。引入 `nowait` 之后，一个应用程序的线程可能处于多个任务分担域中，这样以来线程所分担的计算任务可能出于不同的任务分担域，它们需要明确知道自己当前所处在哪个任务分担域中，以便能正确地获取自己的应当分担的任务。当多个线程处在不同的任务分担域内时，EECB 为它们提供相应的信息以便个线程确定自己所处位置。

7.6.6 flush

`flush` 制导指令标识一个同步点，在此必须提供内存一致可视性的实现，线程可见的变量在此点被写回内存。编译器需要在出现 `flush` 语句的地方刷新 cache 内容到内存中，OMPI 编译器在遇到 `flush` 制导指令的时候，将插入一条语句来执行 `ort_fence()` 函数，它将会调用一条汇

编指令用于刷新 cache，根据硬件平台的不同可能调用 sync、cpuid、stbar 等指令。这个制导指令从功能上来说也可以归入到数据环境控制的内容。

7.6.7 barrier

barrier 制导指令用来同步一个线程组中所有的线程，先到达的线程在此阻塞，等待其他线程。barrier 语句最小代码必须是一个结构化的块。在出现 barrier 的地方增加一个语句 `ort_barrier_me()`，该函数根据是否创建了并行域作不同处理，如果没有并行域则可以直接返回，否则需要同步等待。

7.7 小结

本章讨论了 OpenMP 编译中的任务分担和线程同步问题。

任务分担是产生并行域的目的，利用并行域产生的并发线程组对计算任务进行分担，才能提高处理速度，因此如何描述计算任务并合理进行分割是任务分担的核心问题。本章给出了 `for`、`section` 和 `single` 三种任务分担制导指令的任务描述、划分原则和方法，并给出了用于翻译变换的“框架”代码。任务分担中的另一个难题是 `nowait` 子句引起的多个任务分担域问题，该问题主要通过对每个任务分担域提供记账信息来解决。

最后是线程同步制导指令的问题，它们的行为比较简单，因此代码变换也很容易，主要依靠互斥锁来实现。

第 8 章 数据环境控制

OpenMP 有多种数据子句用于控制数据的作用域（scope）及相关的使用情况，编译器需要能够正确处理它们。变量的作用域是指程序的一个区域，在其中对变量 `x` 的引用都指向特定的变量声明。如果通过程序代码就可以判断出一个声明的作用域，那么这个语言使用的是静态作用域（static scope），也称为词法作用域（lexical scope）。否则，这个语言使用的是动态作用域（dynamic scope），此时在程序运行中对一个变量 `x` 的引用可能指向 `x` 的若干声明中的一个。由于 C 语言使用静态作用域，其处理难度要比动态作用域小得多。

在 OpenMP 的作用域方面主要需要处理有关私有和共享变量、变量在进入和退出并行域边界时的变量拷贝及处理等问题，我们统称这些问题为数据环境（data environment）控制问题。除了 `threadprivate` 可以单独出现外，大多数数据子句必须在 `parallel`、`for`、`sections`、`single` 或它们的组合制导指令中出现。根据底层所采用的是线程还是进程的不同，对共享和私有的处理方法略有不同。

8.1 共享与私有

OpenMP 中默认的变量都是共享变量，除非使用数据子句将类型指定为私有变量。除了使用缺省的变量属性外，可以显式地用 `shared` 和 `private` 等子句说明其作用域属性。由于操作系统内一个进程产生的线程共享内存空间，但是拥有私有的堆栈，因此在共享区的全局数据是共享变量而在堆栈区的变量就是私有变量。如果使用进程则没有共享内存空间，所有变量都是私有变量，如果需要共享变量则需要向操作系统申请共享内存区。对于不在同一个操作系统中的进程如果要申请共享内存，则需要支持软件 DSM。

在操作系统内部的线程之间，OpenMP 数据子句声明的全局共享变量很容易实现。比如 Linux 中属于同一进程的线程，它们的内存映射关系可以用图 8.1 表示。

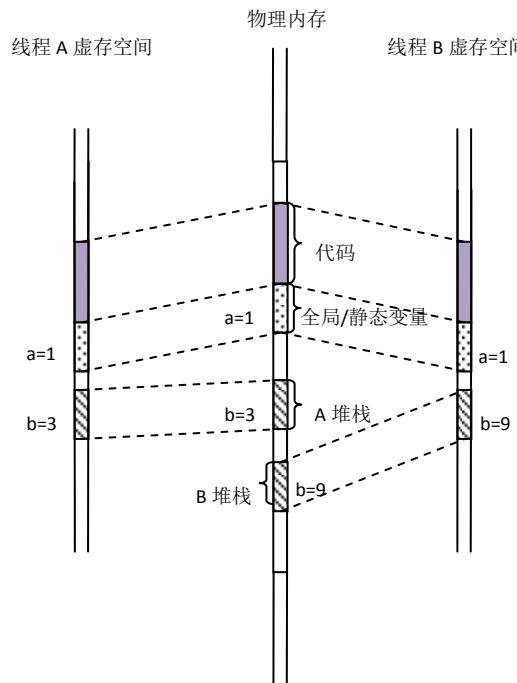


图 8.1Linux 进程虚实地址映射

在线程 A 和线程 B 中引用全局变量或静态变量 a 时，它们访问的是同一个变量，因此获得相同的取值（此例子中 a=1），但是访问堆栈变量或自动变量时，它们是在各自的私有堆栈中的，因此访问的不是同一个内存空间，取值也不必相同（线程 A 的 b=5，而线程 B 的 b=9）。

8.1.1 非全局变量的共享

但是在 OpenMP 程序中，使用 `shared` 子句的变量可以是全局变量或静态变量，也可能是函数内部的堆栈变量或自动变量，对于前者其共享性是线程本身所支持的，对于后者这需要编译器进行特殊处理。对于共享的非全局变量，这里称为 sng (shared non-global) 变量。我们以下面代码为例简要说明其原理。

```

1. int functionX(void)
2. {
3.     int b=0;
4.     #pragma omp parallel shared(b)
5.     {
6.         int c;
7.         ....
8.         funcY(b);
9.         ....
10.    }
}

```

假如直接使用类似第 6 章的并行域封装，若不作其他变换产生出以下代码：

```

1. static void * _thrFunc0_(void * __me)
2. {

```

```

3. {
4.     int c;
.....
5.     funcY(b);
.....
6. }
7. return (void *) 0;
8. }

9. int functionX(void)
10. {
11.     int b=0;
12.     ort_execute_parallel(...,_thrFunc0,...);
13. }

```

此时 `functionX` 函数中的 `ort_execute_parallel` 确实可以用多个线程来执行 `_thrFunc0_` 函数，但是变量 `b` 却还存在很大问题。首先它没有在 `_thrFunc0_()` 函数中声明，所以语法上通不过。其次，即使在 `_thrFunc0_()` 函数中声明 `int b` 变量，那是线程堆栈私有变量与外部的 `b` 变量无关。第三种情况将变量 `b` 作为参数传入，可以保证变量取值是共享的，但是对变量的修改无法返回给调用者，因为此时的形参也是堆栈变量，函数返回后将丢弃。最后，如果将 `b` 的地址作为指针传入到 `_thrFunc0_` 里面，然后将里面的变量引用修改成指针变量形式，即 `funcY(b)` 变成 `funcY(*b)`，那么各个线程都将访问到同一个内存空间，从而实现非全局变量的共享，见图 8.2。

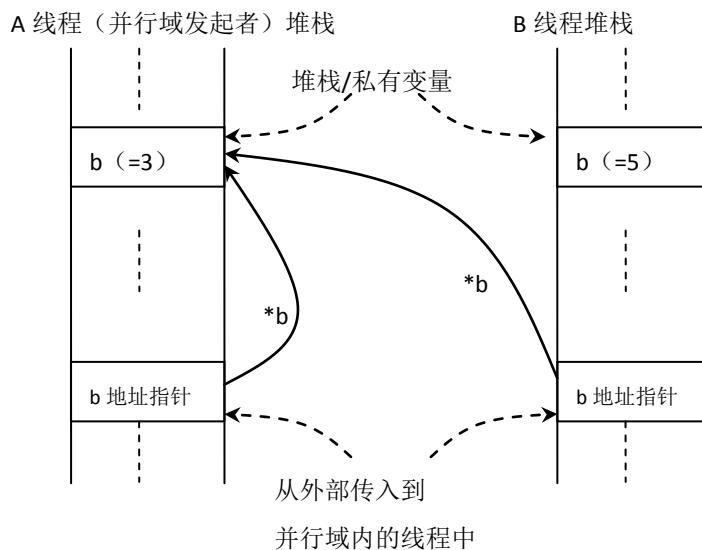


图 8.2 通过指针在线程之间共享堆栈变量

在 `_thrFunc0_()` 里面执行 `funcY(*b)` 时，无论是线程 A 还是线程 B 都将是访问的线程 A 的私有变量，从而实现了堆栈变量的共享。

8.1.2 变量的私有化

变量使用 `private` 子句后就必须处理为私有变量，即使一开始被声明为全局变量。变量的私有化很简单，只需要在并行域生成之后的代码中重新声明一次变量，那么该变量必然是线程的堆栈/自动变量，是私有的。对于采用基于线程的并行域时，对封装好的并行域代码，只需要在代码前段插入相应的变量声明即可。

变量的重新声明可以有两种方式。第一种方式是变量名称不变，仅仅是重新按照原来的类型声明一次；第二种方式是按原来的类型重新生成变量名，同时对并行域内的语句中的所有对应变量也应该作相应的更名。如果在语法分析过程中我们已经将相应的符号表按作用域进行区分，那么第二种方法也是非常容易的。

8.1.3 `threadprivate` 子句

`threadprivate` 不是前面讨论的简单共享或私有变量，它是介于两者之间的一个概念。首先 `threadprivate` 在线程之间是私有的，其次它还需要在同一个线程的不同的并行域上实现全局可见。假如声明了一个 `threadprivate` 类型的变量 `a`，那么其作用关系可用图 8.3 表示。

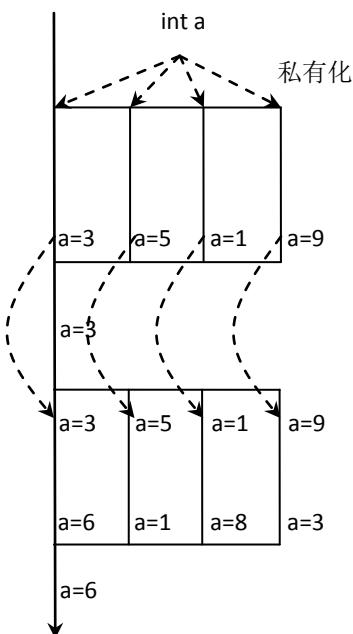


图 8.3 `threadprivate` 变量的作用

图 8.3 说明了一个全局变量经过 `threadprivate` 子句说明后，它将在各个线程上进行私有化。当各线程修改了 `a` 变量的值后，各线程相互间是不可见的。但是在退出并行域后再次进入一个并行域时，各个线程原来的 `a` 变量的值将被保留下来。当并行域结束时，主线程使用的 `a` 变量和并行域 0 号线程使用的是相同的变量 `a`。

由于 `threadprivate` 子句只能作用于全局变量或者静态变量，因此 `threadprivate` 特性可以比较容易地基于 `pthread` 线程中的“线程专有数据”的概念来实现。`pthread` 线程中特有的线程专有数据——**Thread Specific Data**（也有称线程私有数据、线程相关数据、线程存储数据等）。在多线程程序中，所有线程共享程序中的全局静态变量。现在有一全局变量，所有线程

都可以使用它，改变它的值。如果每个线程希望能单独拥有它，那么就需要使用线程专有数据了。表面上看起来这是一个全局变量，所有线程都可以使用它，而它的值在每一个线程中又是单独存储的，这就是线程专有数据的意义。

8.1.4 基于进程的问题

当并行域基于进程技术时，无论这些进程是在同一个操作系统范围内还是在不同的操作系统中，除非显式申请共享内存否则它们都是没有共享变量的。同一个操作系统内的进程可以通过向系统申请共享内存来实现共享变量。而不同操作系统上的进程则需要 sDSM 的支持来实现共享内存进而支持共享变量。

8.2 并行域边界处理

如果仅仅是将局部变量从线程私有声明为共享，或者将全局共享变量处理为私有变量而不做其他进一步的边界处理，那么它们的作用是非常有限的（将私有变量变为共享变量可以起到多个线程公用相同数据，而将共享变量处理为私有变量则几乎没有太大的使用价值），因此必须在私有变量在进入和退出并行域或任务分担域时作一定的处理。

当进入并行域时，私有数据需要初始化，根据变量所使用的数据子句类型不同，它们有不同的初始化方法。实际上我们常常需要将外部数据传入到并行域内的私有变量中，或者反过来将并行域内的私有变量传递到并行域外部进一步处理。因此 OpenMP 提供了多种相应的数据子句以完成相关操作，OpenMP 编译器需要仔细处理这些问题。

8.2.1 private 变量

对于仅仅使用 `private` 子句的变量，它们在进入并行域后的私有变量初始值不是主线程的副本。用 `private` 子句声明的私有变量的初始值在并行区域的入口处是未定义的，它并不会继承同名共享变量的值。但实际情况中有时需要继承原有共享变量的值，OpenMP 提供了 `firstprivate` 子句来实现这个功能。编译器所需要做的工作就是将外部同名变量值拷贝到私有变量上即可。反之，如果需要将并行域内的私有变量传递到外部同名变量中，可以使用 `lastprivate` 子句。出现在 `reduction` 子句中的参数不能出现在 `private` 子句中。

8.2.2 threadprivate 变量

数据子句 `copyin` 是将并行域外部的同名变量传递到并行域内部的同名 `threadprivate` 私有变量中，这个操作可以起到对并行域内部的私有变量进行初始化的作用。编译器对 `copyin` 子句的处理就是将主线程变量值拷贝到 `threadprivate` 变量中。反之，对于 `copyprivate` 子句内的变量，则用在需要在并行域内某个线程私有的 `threadprivate` 变量广播到其他线程的同名变量中。

编译器应该具备以下能力：

1. 为每个线程准备相应的变量用于保存线程专有数据;
2. 在进入并行域时保证对该变量设置初始值, 该初始值为上一个并行域结束时同编号线程中的变量值;

如果并行域的管理是基于 `pthreads` 线程库且采用线程池方式, 同时保证相同编号的 OpenMP 线程能够映射到同一个 `pthreads` 线程上, 那么使用 `pthreads` 的线程私有数据就可以保证拥有前面的两个能力。

`threadprivate` 对应的边界处理有 `copyin` 和 `copyprivate` 两种数据子句。对于 `copyprivate` 而言往往和 `single` 子句结合使用, 此时在 `single` 子句结束后需要进行同步 `barrier` 才能保证后续的数据广播能正确影响各个线程。

8.3 OMPI 数据环境控制

下面我们对 OMPI 的数据子句处里的实现细节进行分析, 只关注基于 `pthreads` 线程库的实现, 其他方式请自行阅读源代码进行分析、学习。

8.3.1 共享变量

如果 `shared` 子句作用于全局变量, 编译器可以不作任何处理。如果作用于自动变量, 那么需要一些额外处理。我们来看看 OMPI 编译器是如何处理自动变量在线程间共享的。

首先看看 `shared` 制导指令的编译前后变化。首先是未编译的源代码:

```

1. #include <omp.h>
2. int main()
3. {
4.     int a=1,b=2;
5. #pragma omp parallel    shared(a,b)
6.     {
7.         a=omp_get_thread_num();
8.         b=a;
9.     }
10.    return 0;
11. }
```

上述代码将 `main()` 函数中的自动变量 `a`、`b` 用制导指令声明为共享, 因此并行域内的 `a`、`b` 变量都是指向同一个内存单元。经过 OMPI 编译后输出的目标代码如下。

```

1. int __original_main(int _argc_ignored, char ** _argv_ignored)
2. {
3.     int a = 1, b = 2;
4.     {
5.         struct __shvt__ {
6.             int (* b);
7.             int (* a);
8.         } __shvars = {
```

```

9.         &b, &a
10.        };
11.        ort_execute_parallel(-1, _thrFunc0_, (void *) &_shvars);
12.    }

```

首先在自动变量 `a`、`b` 声明之后，`OMPI` 声明了一个类型为`_shvt_`的`_shvars` 变量，里面包含了所有共享变量的地址指针，此处是`*a`、`*b`，它们指向需要共享的自动变量 `a`、`b` (`&a`、`&b`)。然后使用 `ort_execute_parallel()`启动并行域以执行线程任务函数`_thrFunc0_()`，在各个线程的执行`_thrFunc0_()`的时候，它们的堆栈将是各自私有的，但是通过将`_shvars` 的指针传入各个线程，那么它们的变量`_shvars->a` 和`_shvars->b` 分别将指向主线程的私有变量 `a` 和 `b`，这样以来通过变量地址指针的传递实现了共享。此时的线程任务函数访问共享变量的形式如下：

```

1. static void * _thrFunc0_(void * __me)
2. {
3.     struct __shvt__
4.     {
5.         int (* b);
6.         int (* a);
7.     };
8.     struct __shvt__ * _shvars = (struct __shvt__ *) ort_get_shared_vars(__me);
9.     int (* b) = _shvars->b;
10.    int (* a) = _shvars->a;
11.    {
12.        (*a) = omp_get_thread_num();
13.        (*b) = (*a);
14.    }
15.    return (void *) 0;
}

```

首先，所有共享变量的指针变量`_shvars` 通过 `ort_get_shared_vars` 传入到`_thrFunc0_()`中，然后从中获取变量 `a`、`b` 的地址并赋给同类型的局部变量`*a`、`*b`，然后将并行域内的代码中所有对共享变量 `a`、`b` 的访问替换成对局部变量`*a`、`*b` 的访问即可。`Ort_get_shared_vars()`将在运行环境中进行分析，它的功能就是获取描述所有共享变量的`_shvars` 的指针。

全局变量作为共享变量使用时其实可以不作任何处理的，但是 `OMPI` 可能出于编程方面的考虑，还是将这种变量按照自动变量相同的方式来处理。这样一来就增加了不必要的变量拷贝，增加了额外的时延，虽然逻辑正确但仍是不合理的。

8.3.2 私有变量

通过在 C 语言结构块内声明变量，可以产生出作用域仅限于当前结构块内的私有变量，并行域或任务分担域内的私有变量就是这样实现的。

私有化

不论变量声明时是全局变量还是局部自动变量，`OMPI` 编译器对 `private` 子句的处理方式是相同的，都需要重新声明局部自动变量。

```
1. #include <omp.h>
```

```

2. int main()
3. {
4.     int a=1,b=2;
5. #pragma omp parallel private(a,b)
6.     {
7.         printf("a=%d    ,b=%d\n",a,b);
8.     }
9.     return 0;
10. }
```

该并行域经过 OMPI 编译输出以下代码：

```

1. static void * _thrFunc0_(void * __me)
2. {
3.     int b;
4.     int a;
5.     {
6.         printf("a=%d    ,b=%d\n", a, b);
7.     }
8.     return (void *) 0;
9. }
```

可以看到在 `_thrFunc0_()` 封装的并行域内，重新声明了相同类型的变量 `a` 和 `b`。从此也可以看出，即使原来的变量是全局共享变量，经过这样的处理也变成了私有变量。

`for` 循环变量是按照私有变量处理的，否则并行域内的并发线程争用循环变量则无法并发执行了。`private` 子句中可以放结构体变量，但是不能放结构体的成员。

firstprivate 和 lastprivate

为了将外部变量传入到并行域中以便对私有变量进行初始化，需用利用的是 `firstprivate` 数据子句，它与 `private` 的有些相似，但是还需要一个初始化赋值操作。

其处理工作包括：

- 首先，生成私有变量；
- 其次，将初始变量处理成共享变量并传入到并行域中；
- 最后，在进入并行域的一开始就对私有变量进行赋值。

例如对于如下的带有 `firstprivate` 制导指令的代码：

```

1. #include <omp.h>
2. int main()
3. {
4.     int i=0;
5. #pragma omp parallel firstprivate(i)
6.     {
7.         i=i+1;
8.     }
9. }
```

经 OMPI 编译变换后的并行域部分的目标代码形式如下：

```

1. static void * _thrFunc0_(void * __me)
2. {
3.     struct __shvt__ {
4.         int (* i);
5.     };
6.     struct __shvt__ * _shvars = (struct __shvt__ *) ort_get_shared_vars(__me);
7.     int i = *(_shvars->i);
8.     {
9.         i = i + 1;
10.    }
11.    return (void *) 0;
12. }

```

可以看出，对于 `firstprivate` 变量，在并行域中新生成了 `int i` 变量并且将初值设为外部传入的值 `(* (_shvars->i))`。而 `_shvars` 是利用 `ort_get_shared_vars()` 从线程控制块中获得的共享变量列表信息。而 `_shvars` 中的 `i` 的数值是在并行域创建之前就已经记录好的：

```

1. int i = 0;
2. {
3.     struct __shvt__ {
4.         int (* i);
5.         } _shvars = { &i };
6.     ort_execute_parallel(-1, _thrFunc0_, (void *) &_shvars);
7. }

```

也就是说并行域外部的 `i` 的数值保存在 `_shvars` 中，然后通过 `ort_execute_parallel()` 传入到并行域内的各个线程的。

`lastprivate` 只能用于 `for`（或者 `parallel for`）和 `sections`（或者 `parallel section`）子句中，用于将 `for` 循环中负责最后的一次迭代（或代码中排在最后的一个 `section`）的线程中的私有变量拷贝到并行域或任务分担域外部，用于影响后续计算。其原理和 `firstprivate` 类似，只不过赋值操作发生在退出并行域之前，而不是刚进入并行域的时候。假设有以下带有 `lastprivate` 的 OpenMP 代码：

```

1. #include <omp.h>
2. int main()
3. {
4.     int i=0,j;
5. #pragma omp parallel for lastprivate(j)
6.     for (i=0;i<100;i++)
7.     {
8.         j=i;
9.     }
10. }

```

经 OMPI 编译后输出的目标代码形式如下：

```

1. static void * _thrFunc0_(void * __me)
2. {

```

```

3.     struct __shvt__ {
4.         int (* j);
5.     };
6.     struct __shvt__ *_shvars = (struct __shvt__ *) ort_get_shared_vars(__me);
7.     int (* j) = _shvars->j;
8.     {
9.         int i;
10.        int (* _lap_j) = &(*j), j;
11.        .....
12.        ort_entering_for(1, 0, 0, step_, &gdopt_);
13.        .....
14.        if (last_iter_)
15.            *_lap_j = j;
16.        }
17.    return (void *) 0;

```

可以看出，`lastprivate` 和 `firstprivate` 在进入并行域时有些相似的，需要声明内部局部变量（第 10 行）。并且在第 7 行记录了外部变量的地址（`_lap_j`），在退出并行域之前将局部变量 `j` 的值保存到刚才记录的地址（`_lap_j`）对应的内存单元上。由于 `for` 并行域内有多个线程，它们都会退出并行域，但是只有一个负责最后一次迭代的线程能够通过 `if(last_iter_)` 检查并将结果保存。

而在 `__original_main()` 函数中的准备工作如下：

```

1.     int i = 0, j;
2.     {
3.         struct __shvt__ {
4.             int (* j);
5.             } _shvars = { &j };
6.         ort_execute_parallel(-1, _thrFunc0_, (void *) &_shvars);
7.     }

```

如果同时使用了 `firstprivate` 和 `lastprivate` 子句，那么就是在 `lastprivate` 的基础上再加上变量初始值的设置即可。例如在前面 `_thrFunc0()` 的第 10 行内容从 “`int (* _lap_j) = &(*j), j = *_lap_j;`” 替换成 “`int (* _lap_j) = &(*j), j = *_lap_j;`” 即可，此时的私有变量 `j` 将从外部获得初始值。

8.3.3 线程专有变量

OMPI 利用了 `pthreads` 的线程专有数据来实现 OpenMP 的 `threadprivate` 线程私有变量。

threadprivate

在单线程的程序里，有两种基本的数据：全局变量和局部变量。但在 `pthreads` 多线程程序里，还有第三种数据类型：线程专有数据（TSD: Thread-Specific Data）。它和全局变量很象，在线程内部，各个函数可以象使用全局变量一样调用它，但它对线程外部的其他线程是不可见

的。这种数据的必要性是显而易见的。例如我们常见的变量 `errno`，它返回标准的出错信息。它显然不能是一个局部变量，几乎每个函数都应该可以调用它；但它又不能是一个全局变量，否则在 A 线程里输出的很可能是 B 线程的出错信息。要实现诸如此类的变量，我们就必须使用线程数据。我们为每个线程数据创建一个键，它和这个键相关联，在各个线程里，都使用这个键来指代线程专有数据，但在不同的线程里，这个键代表的数据是不同的，在同一个线程里，它代表同样的数据内容。

OMPI 利用 `pthreads` 的线程专有数据来实现 OpenMP 的 `threadprivate` 变量的。由此一来 OMPi 需要为各个 `threadprivate` 生成线程专有数据即可。下面是一个带有 `threadprivate` 数据子句的代码：

```
1. #include <omp.h>
2. int tpa;
3. #pragma omp threadprivate(tpa)
4. int main()
5. {
6. #pragma omp parallel
7. {
8.         tpa=omp_get_thread_num();
9.     }
10.    return 0;
11. }
```

经过 OMPi 编译后输出的目标代码包括以下几个部分。

首先是线程专有数据的声明：

```
1. int tp_tpa_;
2. static void * tp_tpa_key_;
```

OMPI 将按照 `threadprivate` 变量的名称加上前缀 “`tp_`” 形成一个新的全局变量 `tp_tpa_`，同时准备了相应的键 `tp_tpa_key_`。

其次，在并行域的开始处需要获得该线程专有数据：

```
1. static void * _thrFunc0_(void * __me)
2. {
3.     int (* tpa) = ort_get_thrpriv(&tp_tpa_key_, sizeof(tp_tpa_), &tp_tpa_);
4. {
5.     (*tpa) = omp_get_thread_num();
6. }
7.     return (void *) 0;
8. }
```

在各个线程执行 `_thrFunc0_()` 的一开头，就执行 `ort_get_thrpriv()` 获得各自的线程专有数据，该函数实际上是 `pthreads` 库的 `pthread_getspecific()` 函数的一个封装，能够根据个线程的键值获取各自的专有数据。

copyin 和 copyprivate

如果需要将 `tpa` 声明为 `copyin`，即代码如下：

```
1. #include <omp.h>
```

```

2. int tpa;
3. #pragma omp threadprivate(tpa)
4. int main()
5. {
6. #pragma omp parallel copyin(tpa)
7. {
8.         tpa=omp_get_thread_num();
9.     }
10.    return 0;
11. }
```

编译后输出的部分目标代码如下。首先仍然是关于线程专有数据的全局变量声明：

```

1. int tp_tpa_;
2. static void * tp_tpa_key_;
```

其次在_thrFunc0_()函数中进行初始化：

```

1. static void * _thrFunc0_(void * __me)
2. {
3.     int (*tpa) = ort_get_thrpriv(&tp_tpa_key_, sizeof(tp_tpa_), &tp_tpa_);
4.     *tpa = tp_tpa_;
5.     ort_barrier_me();
6. {
7.     (*tpa)=omp_get_thread_num();
8. }
9. return (void *) 0;
10. }
```

上面第 4 行*tpa=tp_tpa_实际上就是将主线程的 tp_tpa_内容赋给各个线程的专有数据 *tpa。如果在并行域外部对相应变量进行了修改，那么相应的修改应该体现在主线程的专有数据中，因此在串行代码部分有相关的赋值计算：

```

1. int __original_main(int __argc_ignored, char ** __argv_ignored)
2. {
3.     int (*tpa) = ort_get_thrpriv(&tp_tpa_key_, sizeof(tp_tpa_), &tp_tpa_);
4.     (*tpa) = 10;
5.     {
6.         ort_execute_parallel(-1, _thrFunc0_, (void *) 0);
7.     }
8.     return (0);
9. }
```

此时实际上是将主线程的*tpa 赋值为 10，当后面并行域中的线程执行 copyin 的赋值时将从这里获得数据。

对于 copyprivate 而言，实际上相当于是对线程私有数据的广播。一般来说应该是在并行域中的 single 子句中出现，因为此时只有一个线程在修改 threadprivate 数据，它修改完后就可以广播给其他线程了。下面是带有 copyprivate 子句的代码：

```

1. #include <omp.h>
2. int tpa;
```

```

3. #pragma omp threadprivate(tpa)
4. int main()
5. {
6.     tpa =10;
7. #pragma omp parallel
8.     {
9. #pragma omp single copyprivate (tpa)
10.         tpa=omp_get_thread_num();
11.     }
12.     return 0;
13. }

```

经编译后输出的代码如下：

```

1. static void * _thrFunc0_(void * __me)
2. {
3.     int (* tpa) = ort_get_thrpriv(&tp_tpa_key_, sizeof(tp_tpa_), &tp_tpa_);
4. {
5.     if (ort_myself(0))
6.     {
7.         (*tpa) = omp_get_thread_num();
8.         ort_broadcast_private(1, tpa);
9.     }
10.    ort_leaving_single();
11.    ort_barrier_me();
12.    ort_copy_private(1, tpa, sizeof(tpa));
13.    ort_barrier_me();
14. }
15. return (void *) 0;
16. }

```

上面的 `ort_broadcast_private()` 函数将自己的线程专有数据进行保存以便后面其他线程进行拷贝。而 `ort_copy_private()` 将被所有线程所执行，将负责 `single` 代码的线程上的数据拷贝到自己的线程专有数据中。

8.3.4 归约变量

归约操作涉及两种变量，一个是在 `reduction` 所在的并行域或任务分担域内部，另一中是在其外部。归约操作实际上是将内部的变量值经过归约计算后赋给外部变量。所以一旦在并行域或任务分担域使用了 `reduction` 子句，那么该归约变量在内部自动变成私有变量，其私有化处理过程和 `private` 的方法相同。同时它有对应的一个共享变量传入到并行域内部以便将来把结果反映在并行域或任务分担域的外部。另外在退出并行域或任务分担域时的需要执行归约计算的操作并将结果保存到外部共享变量中，可参见 7.5 的归约操作。

在 `parallel` 制导指令中带有的 `reduction` 变量，不能在内层的任务分担制导指令中被声明成私有变量，而任务分担中的 `reduction` 变量在外部必须是共享变量。因此在代码变换过程中，

reduction 变量同时涉及外部的共享变量和内部的私有变量两种处理。在并行域外部的变量需要作为并行域内部的共享变量传入到并行域中，而并行域中的同名变量则需要进行私有化，在执行归约计算的时候需要将内部私有变量的结果反映到外部共享变量的变化上去。对于如下代码：

```
1. #include <omp.h>
2. int main()
3. {
4.     int k;
5.     #pragma omp parallel private(k)
6.     {
7.         int i=0;
8.         i=i+1;
9.         #pragma omp for reduction(+:k)
10.        for(i=0;i<10;i++)
11.            k=i;
12.    }
13.    return 0;
14. }
```

编译器应该提示错误，因为在外层声明为 **private** 的 **k** 变量在内层 **for** 任务分担是声明为 **reduction** 变量，违反了 OpenMP 规范的要求。OMPI 1.0.0 并没有对这个问题进行检查，而 gcc 则能成功检测该错误。

最后关于归约变量的初始值问题。在 OpenMP v2.5 规范文件中对于归约变量要求有明确的初始值，如表 2.3 所示，编译器应该正确设置这些初值。

8.4 小结

数据环境控制是通过 OpenMP 的数据子句来完成的，包括共享和私有属性、并行域边界上的数据处理两大问题。

本章分析了操作系统中线程、进程中的全局共享变量、堆栈私有变量等现象和原因。对于基于线程的技术，讨论了如何通过指针传递的办法将堆栈变量、局部变量变成线程共享变量的方法，以及通过在内部作用域声明同名变量的方法将共享变量变成私有变量的方法。对于 **threadprivate** 的特殊变量，需要利用 **pthreads** 库的线程专有数据来支持。相关的代码变换是通过 OMPI 的目标代码样例来讲述的。

本章另一部份内容是并行边界上的数据处理，主要包括将共享变量的值传入并行域内部同名变量中以及相反的操作、归约操作问题等。与此相关的 OpenMP 制导指令及子句有 **firstprivate**、**lastprivate**、**copyin**、**copyprivate**、**reduction** 等。由于它们的语义比较简单，书中直接展示了 OMPI 的目标代码例子。

第 9 章 产生目标代码

代码生成是以源程序的中间表示形式作为输入，并把它映射到目标语言的过程。如果目标语言是机器代码，那么必须为程序使用的每个变量选择寄存器位置，编译器的一个至关重要的方面就是合理分配寄存器已存放变量的值。但是狭义的 OpenMP 编译不需要涉及寄存器分配这些问题，只是源代码变换问题。

目标代码的生成是依赖于“框架”代码来实现的，我们在第 6 章、第 7 章和第 8 章中其实已经涉及了“框架”代码的初步形式，也大致了解了如何用多线程的 C 程序来实现并发的 OpenMP 程序。

本章讨论目标代码产生的方法，即如何根据前面的“框架”要求，对 AST 的各个子树的代码片断来重构产生出新的目标 AST 树的过程，这里近似于“手工”的翻译变换过程是非常细致和繁杂的。由于产生目标代码属于实现细节，因此本章将紧密结合 OMPI 编译器代码来具体讨论 AST 变换和 AST 输出。

9.1 源代码变换

OpenMP 以编译制导指令的方式出现在程序中，OpenMP 翻译模块的主要任务就是根据这些编译制导指令来对程序进行翻译，以表达编译制导指令规定的语义。作为一个标准，OpenMP 标准规定了用户可见的 OpenMP 程序行为，同时与 MPI 一样，OpenMP 程序也鼓励高效的实现。

由于狭义的 OpenMP 编译以 C 代码为目标代码，所以只需要处理 OpenMP 编译制导指令，而 OpenMP API 函数和环境变量则留给运行环境去处理。所有这些变换翻译都是以替换 AST 的 OpenMP 节点为基础的，将所有 OpenMP 节点替换成常规 C 语言的节点，即完成了编译工作中翻译的工作。

9.1.1 变换流程

变换的前提是已经有一个与源代码对应的完整的 AST，然后遍历此树，遇到需要翻译变换处理的节点和子树，则调用子树剪切、拼接等功能将它变换成“框架”代码所指定的形式，然后将变换好的子树插入挂接到 AST 中。代码中大多数 C 语言相关的节点和子树不需要变换，这些 C 语言结构块和语句往往整体进行处理，作为子树移动到合适的地方即可。因此翻译变换工作主要集中在 OpenMP 构造节点上，所有 OpenMP 子树需要裁减拼接和修改，有些子数还将产生多棵分离独立的子树分别插入到代码 AST 的不同地方。

AST 树的产生是由 OMPI 的 ompi.c 通过以下代码：“`ast = parse_file(filename, &r);`”构建出 AST 树的。`parser_file()`是语法分析其提供的函数，关于 AST 的产生可参考第 5 章。然后在变换之前需要进行简单的一些裁减、拼接等整形操作。最后才是进行 AST 的变换，其间可能

有 AST 子树或节点的删除、插入、移动或修改等操作。OMPi 的 ompi 应用程序将调用 `ast_xform()` 实现对 AST 的变换。其中 `ast_xform()` 实际上就是遍历整个 AST，然后根据所遇到的子树或节点的类型，调用相应的变换函数。

AST 变换的起点是 `ast_xform()`，实施的变换需要四大步骤来完成，这些步骤涉及如下函数：

- (1) `ast_stmt_xform()`;
- (2) `xform_thread_functions()`;
- (3) `place_thread_functions()`;
- (4) `place_globals()`

`ast_stmt_xform()` 完成对 AST 的变换，这是最复杂的工作之一。函数 `xform_thread_functions()` 是对被封装后并行域的线程任务函数的变换处理，而 `place_thread_functions()` 是将线程任务函数插入到调用函数之前。最后是调用 `sgl_fix_sglvars()` 和 `place_globals()` 处理全局共享变量。

下面是 `ast_xform()` 的代码。

```
1. void ast_xform(aststmt *tree)
2. {
3.     newglobals = NULL;
4.     thrfuns = NULL;
5.     newtail = NULL;
6.     ast_stmt_xform(tree);                                第一步骤 AST 变换
7.     xform_thread_functions(thrfuns); /* xform & add the new thread funcs */          对第一步产生的所有任务函数再进行变换
8.     place_thread_functions(thrfuns);                  将变换好的任务函数插入到合适的地方
9.     thrfuns = NULL;
10.    sgl_fix_sglvars(); /* Called before adding the tail, since it adds
11.                           something to it */
12.    if (tree != NULL && newglobals != NULL) /* Add the new globals */
13.    {
14.        // ast_stmt_xform(&newglobals);           /* Must do it ! (why??) */
15.        place_globals(*tree);                  添加新的全局变量
16.    }
17.    if (tree != NULL && newtail != NULL) /* Append @ bottom */
18.        /* Cannot do ast_stmt_xform(&newtail). See x_shglob.c */
19.        *tree = BlockList(*tree, newtail);
20.    if (newglobals != NULL)
21.        newglobals = NULL;
22.    if (newtail != NULL);
23.        newtail = NULL;
}
```

9.1.2 支撑函数

我们把前面三章提到的并行域、任务分担、同步和变量数据环境的变换工作根据复杂程度分成两大类型：

- 1) 简单变换。例如 `barrier` 这样的制导指令，只需要用单独一棵 AST 子树替换该节点即可；
- 2) 复杂变换。例如 `parallel` 的并行域产生，不仅需要替换一个 AST 子树，还需要在其他地方插入新的 AST 子树、对变量和变量数据环境进行处理；

目标代码的生成模块是在遍历 AST 树的时候完成变换的，针对上述两种类型的变换，必须提供以下四种功能函数集：

- 1) 遍历整个 AST 树的函数集；
- 2) 在遍历 AST 树的时候，根据 AST 节点的类型执行相应的变换操作的函数集；
- 3) 具有按照指定“框架”进行 AST 子树的裁剪拼接和构建能力的函数集；
- 4) 具有替换和插入 AST 子树的能力的函数集；
- 5) 根据 AST 输出字符文本形式的源代码文件的函数集。

9.2 AST 变换

根据前面的流程分析和支持函数的分类，下面将从 OMPI 的代码变换的框架分析入手，逐步讨论遍历与变换、拼接、替换和插入等函数集的实现细节。在代码实现中，遍历的功能并不是独立存在的，而是和变换或输出函数集的递归调用过程中实现的。例如通过变换函数集的递归调用而实现对 AST 的遍历，对输出函数集的递归调用也可以实现 AST 的遍历。因此对于遍历功能将不再单独讨论，而是结合变换和输出功能一并讨论的。

9.2.1 拼接及创建函数

AST 节点的创建和拼接是实现代码变换中按照预先设计的“代码框架”产生目标代码的基础。

节点创建函数集

在 AST 变换的过程中往往需要按照“框架”的需要创建出相应的 AST 节点，这类函数有很多，由于它和建立 AST 时所使用的节点创建功能完全相同，因此实际上就是前面 5.3 小节的“节点创建”函数集。

拼接函数

为了进行 AST 变换，需要一些 AST 常见操作的函数作为支持。这些操作包括 5.3 小节的节点创建等函数外，这里主要是语句节点的拼接函数 `BlockList()`。这个函数很简单，代码如下：

```
1. aststmt BlockList(aststmt l, aststmt st)
2. {
3.     aststmt s = Statement(STATEMENTLIST, 0, st);
```

```

4.     s->u.next = l;
5.     return (s);
6. }

```

这个函数将两个语句节点 `l` 和 `st` 构建成一个语句列表节点，具体关系如图 9.1 所示。

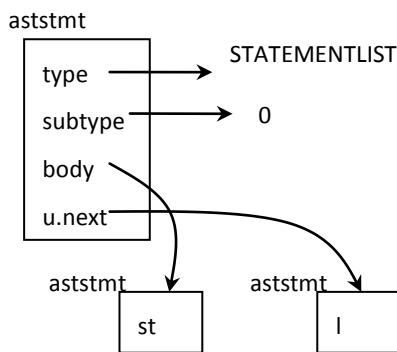


图 9.1 BlockList()功能示意图

拼接的关键是使用了一个类型为 `STATEMENTLIST` 的节点，该节点可以使用 `body` 指向一个语句节点而用 `next` 指向下一个节点。可以用多个 `STATEMENTLIST` 类型的节点来表示一串语句序列。该函数在前面创建 `AST` 时用来表示语句序列，而现在按照“框架”组装语句序列的时候也需要用到这个函数操作。

9.2.2 变换函数集

由于变换和遍历是联系在一起的，变换过程就是对整个 `AST` 进行遍历，然后根据节点的类型调用相应的变换函数，这些变换函数形如“`ast_XXX_xform()`”。由于源代码的根节点是一个 `aststmt` 节点，因此变换函数的起点是 `ast_stmt_xform()`，而 `ast_stmt_xform()` 函数里面也只是简单地根据节点类型再次调用各种 `ast_XXX_xform()` 函数。我们将这些变换函数分成普通语句的变换和 `OpenMP` 变换两大类，其层次关系如图 9.2 所示。

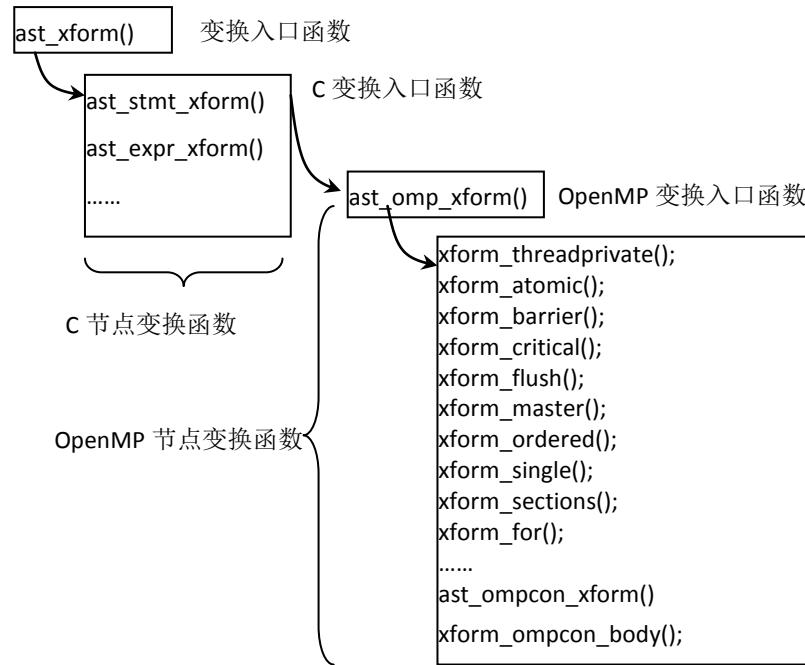


图 9.2 AST 变换函数集及其层次关系

入口函数 `ast_xform()` 首先调用的是 `ast_stmt_xform()`，该函数代码如下：

```

1. void ast_stmt_xform(aststmt *t)
2. {
3.     switch ((*t)->type)
4.     {
5.         case EXPRESSION:
6.             ast_expr_xform(&((*t)->u.expr));
7.             break;
8.         case ITERATION:
9.             ast_stmt_iteration_xform(t);
10.            break;
11.        case SELECTION:
12.            ast_stmt_selection_xform(t);
13.            break;
14.        case LABELED:
15.            ast_stmt_labeled_xform(t);
16.            break;
17.        case COMPOUND:
18.            if ((*t)->body)
19.            {
20.                scope_start(stab);
21.                ast_stmt_xform(&((*t)->body));
22.                scope_end(stab);
23.            }

```

```

24.         break;
25.     case STATEMENTLIST:
26.         ast_stmt_xform(&((*t)->u.next));
27.         ast_stmt_xform(&((*t)->body));
28.         break;
29.     case DECLARATION:
30.         xt_declaration_xform(t); /* transform & declare */
31.         break;
32.     case FUNCDEF:
33.         /* First declare the function */
34.         if (symtab_get(stab, decl_getidentifier_symbol((*t)->u.declaration.decl),
35.                         FUNCNAME) == NULL)
36.             symtab_put(stab, decl_getidentifier_symbol((*t)->u.declaration.decl),
37.                         FUNCNAME);
38.         scope_start(stab); /* New scope here */
39.         if ((*t)->u.declaration.dlist) /* Old style */
40.         {
41.             ast_stmt_xform(&((*t)->u.declaration.dlist)); /* declared themselves */
42.             xt_dlist_array2pointer((*t)->u.declaration.dlist); /* !array params */
43.         }
44.         else /* Normal; has paramtypelist */
45.         {
46.             xt_barebones_decl((*t)->u.declaration.decl);
47.             ast_declare_function_params((*t)->u.declaration.decl); /* decl manually */
48.             /* take care of array params */
49.             xt_decl_array2pointer((*t)->u.declaration.decl->decl->u.params);
50.         }
51.         ast_stmt_xform(&((*t)->body));
52.         tp_fix_funcbody_gtpvars((*t)->body); /* take care of gtp vars */
53.         if (processmode)
54.             ast_find_sgl_vars(*t, 1); /* just replace them with pointers */
55.         scope_end(stab); /* Scope done! */
56.         break;
57.     case OMPSTMT:
58.         ast_omp_xform(t); OpenMP 构造节点的变换
59.         break;
60.     }

```

OMPi 在遍历 AST 的时候，使用了大量的形如 `ast_XXX_xform()` 的函数，大多数只是起到遍历作用，并不真的进行变换，只有少量的函数进行了变换。特别是 `ast_omp_xform()` 函数，完成了对 OpenMP 节点的变换。注意到 `aststmt_` 节点有 11 种类型，而此处只处理了 9 种

(ITERATION、SELECTION、Labeled、EXPRESSION、DECLARATION、COMPOUND、STATEMENTLIST、FUNCDEF、OMPSTMT），JUMP 和 VERBATIM 两种语句节点不需要处理。

其递归调用过程可以用图 9.3 表示。

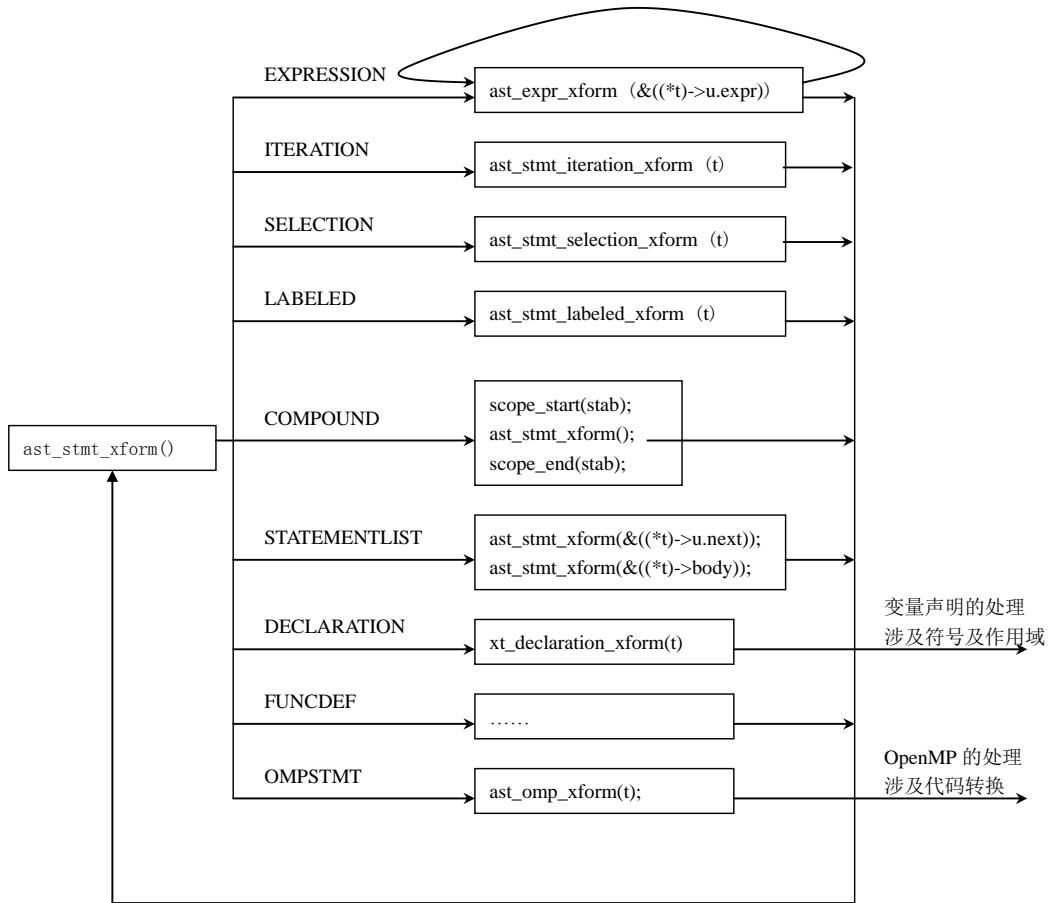


图 9.3 变换的递归调用

对于 ITERATION、SELECTION 和 Labeled 类型的语句节点，只是简单地对其内部成分再次调用 `ast_stmt_xform()`。对于用“{}”花括号括起来的 COMPOUND 复合语句，需要建立新的作用域（`scope_start` 函数），然后再调用 `ast_stmt_xform()` 对内部语句进行变换，最后退出该作用域（`scope_end` 函数）。对于语句列表，则是通过递归调用逐条进行处理。对函数定义的处理比较复杂，在此不深入讨论。对于声明语句使用的是 `xt_declaration_xform()`，而不是 `ast_XXX_xform()` 形式的函数。最后要注意的就是 OpenMP 节点的变换，相对其他节点的变换而言 OpenMP 节点的变换才是真正意义上的变换，相关入口函数是 `ast_omp_xform()`。

9.2.3 OpenMP 节点变换

当 `ast_stmt_xform()` 遇到 OpenMP 节点的时候，将进入特别的处理过程。OpenMP 节点变换函数 `ast_omp_xform()` 根据该 OpenMP 节点的类型，再进一步调用不同的变换函数进行处理。这些变换函数包括：`xform_threadprivate(t)`、`xform_atomic(t)`、`xform_barrier(t)`、`xform_critical(t)`、`xform_flush(t)`、`xform_master(t)`、`xform_ordered(t)`、`xform_single(t)`、`xform_sections(t)`、`xform_for(t)`。

此时的函数调用层次关系如图 9.4 所示。

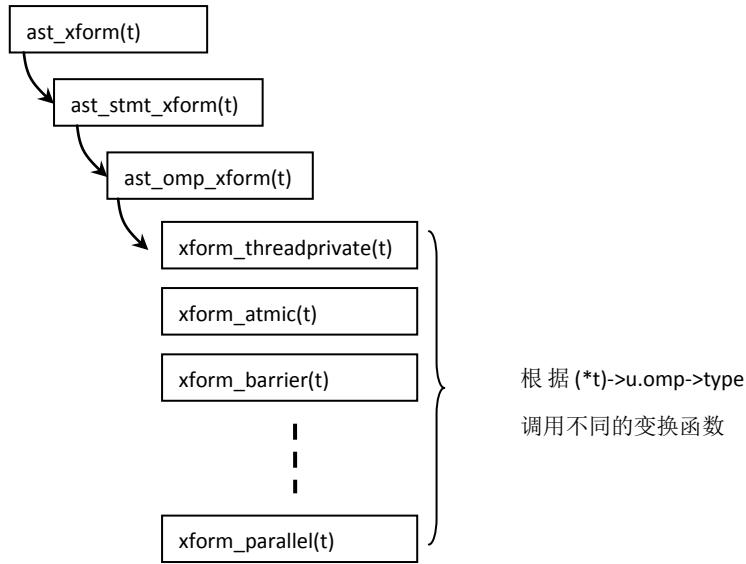


图 9.4 OpenMP 节点变换的函数调用关系

制导指令和变换函数的对应关系如表 9.1 所示。

表 9.1 制导指令与变换函数对应关系

制导指令	调用函数
DCTHREADPRIVATE	xform_threadprivate()
DCATOMIC	xform_atomic()
DCBARRIER	xfrom_barrier()
DCCRITICAL	xfrom_critical()
DCFLUSH	xfrom_flush()
DCMASTER	xfrom_master()
DCORDERED	xfrom_ordered()
DCPARALLEL	xfrom_ompcon_body()
	xform_parallel()
DCSINGLE	xfrom_ompcon_body()
	xform_single
DCSECTIONS	xfrom_ompcon_body()
	xfrom_sections()
DCFOR	xfrom_ompcon_body()
	xfrom_for()
DCPARSECTIONS	这两个制导的处理方法是：分割成两个独立的制导指令再分别使用 PARALLEL 和 FOR/SECTIONS 的处理
DCPARFOR	
Others	ast_ompcon_xform()

下面以 `atomic` 和 `parallel` 作为例子说明变换功能的实现细节，主要依靠 `BlockList()` 函数提供的 AST 节点/子树拼接功能和其他节点创建函数，按照预先设定的“模板框架”来对已有的 OpenMP 节点进行变换、替换和组装。

其中 `atomic`、`master`、`ordered`、`critical`（还涉及互斥锁的生成、编号）、`flush`、`barrier` 都比较简单，因此只给出 `atomic` 一个实现的细节讨论。但是对于 `parallel`，这是一比较复杂的过程，我们将会单独讨论，同样比较复杂的还有 `for` 制导指令和 `sections` 制导指令等。

atomic 变换

OpenMP 制导指令 `atomic` 的变换过程比较简单，已经在 7.6.1 小节中分析过。所代表的语义是原子性地完成指定的代码，使用 C 代码来描述的话可以用互斥来实现。因此将 `atomic` 所修饰的语句块前面加上互斥加锁操作，在代码块的结束处进行解锁即可。这两个操作如果封装成函数，那么变换操作就简化为给这个语句块的前后各插入一个函数，分别进行加锁和解锁即可。下面是 OMPI 的 `xform_atomic()` 函数的代码。

```

1. void xform_atomic(aststmt *t)
2. {
3.     aststmt s = (*t)->u.omp->body, parent = (*t)->parent, v;
4.     int      stlist; /* See comment above */
5.
6.     v = ompdir_commented((*t)->u.omp->directive); /* Put directive in comments */
7.     stlist = ((*t)->parent->type == STATEMENTLIST ||
8.               (*t)->parent->type == COMPOUND);
9.
10.    (*t)->u.omp->body = NULL; /* Make it NULL so as to free t easily */
11.    ast_free(*t); /* Get rid of the OmpStmt */
12.
13.    *t = BlockList(
14.        BlockList(
15.            BlockList(
16.                BlockList( v, Call0_stmt("ort_atomic_begin") ),
17.                s
18.            ),
19.            Call0_stmt("ort_atomic_end")
20.        ),
21.        linepragma(s->l + 1 - (!stlist), s->file)
22.    );
23. }

```

对于一段带有 `atomic` 编译制导的代码：

```

...
#pragma omp atomic
    XXXXX;
...

```

对应的 AST 子树结构如图 9.5-a 所示，首先这个构造是上级构造的一个部分，它本身是一个类型为 `OMPSTMT` 的语句节点(`aststmt` 节点)，因此联合体 `u` 解释为 `omp` 指向一个 `ompcon` 节点。`ompcon` 节点的 `ompdir` 指向制导指令节点，而 `body` 指向所修饰的语句块。

`xform_atomic()`首先将记录几个关键结构，第3行代码用`s`指向语句块（图9.5-a-①）用`parent`指向父结点（图9.5-a-②），第8行代码断开`body`指针（图9.5-a-③）。然后在第9行释放删除该节点（图9.5-a-④），对应于图中灰色方框。但是因为前面断开了`body`指针，语句块仍然可以通过`s`指针来访问到的，对应于图中粗黑线的方框。

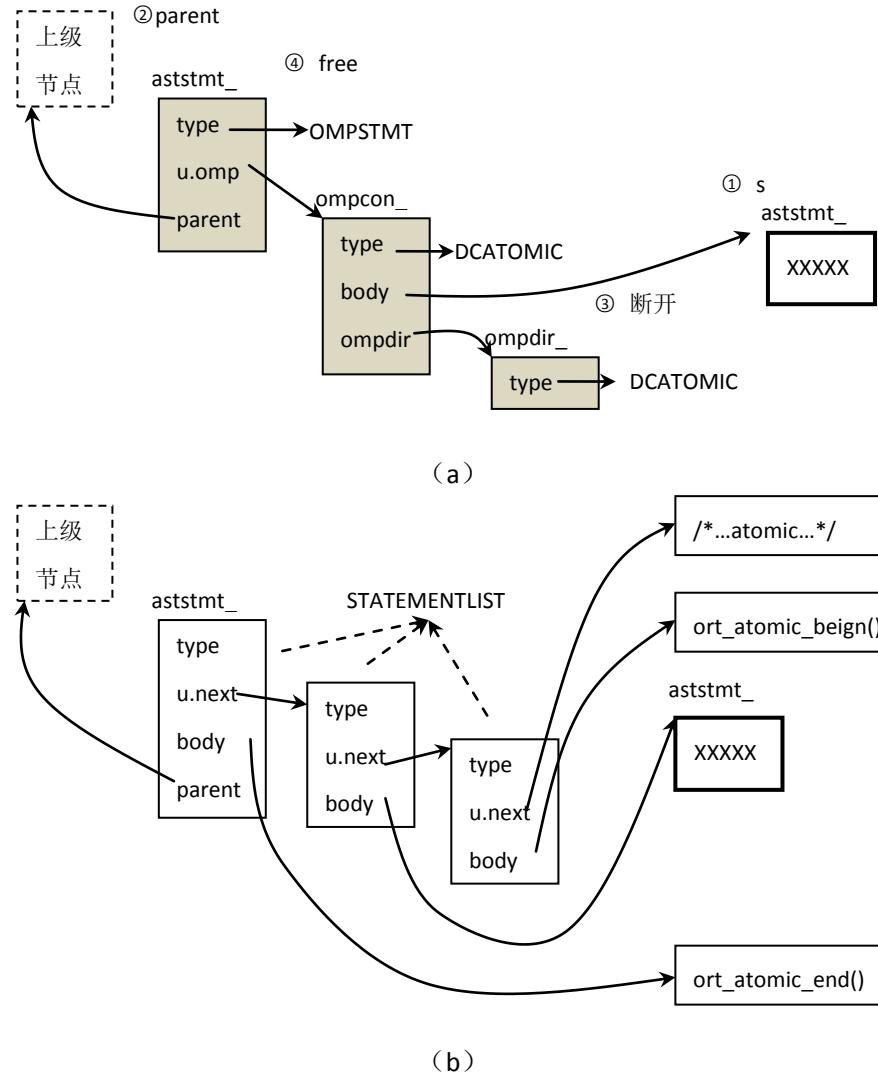


图 9.5 atomic 的 AST 变换示意图

第10行到19行的代码则多次利用`BlockList()`函数，将多条语句拼接成语句列表，逐条用结构体内的`next`指针连接起来。最终将图9.5-b这样的一个AST子树替换原来的节点（还是由原来的指针`t`引用），并且将新节点的`parent`指针指向原来保留的指针所指位置。

此时的代码实际上已经变换成了如下的目标形式（对应于图9.5-b）：

```

...
1. /* #pragma omp atomic */
2. ort_atomic_begin();
3. XXXXX;
4. ort_atomic_end();
...

```

从此可以看出代码变换的过程原理相对简单，就是按照“翻译框架”构造出所需的 AST 子树，但是实现上比较繁杂。而 `ort_atomic_begin()` 和 `ort_atomic_end()` 如何保证其内部代码的运行的原子性，是需要在运行环境中提供保障的，具体在第 10 章中讨论。

其他简单变换

其他简单的变换可以参考 `ast_xform.c` 中的 `xform_master()`、`xform_ordered()`、`xform_critical()`、`xform_flush()`、`xform_barrier()` 等函数，其代码变换过程与 `xform_atomic()` 很简单或类似，但是它们调用运行环境中函数不同。例如 `xform_ordered()` 插入的函数是 `ort_ordered_begin()` 和 `ort_ordered_end()`，`xform_critical()` 插入的函数是 `ort_critical_begin()` 和 `ort_critical_end()` 等，它们的变换任务比较简单，主要靠运行环境提供的库函数的功能来实现这些制导指令的语义功能。

9.2.4 parallel 变换

`parallel` 制导指令引出的是并行域，其变换比较复杂。它需要解决众多问题，包括并行域管理代码的产生、层次控制和变量数据环境等等，但是可以将复杂的行为留给运行环境，而编译部分的代码变换只需要完成以下三个功能：

- 1) 将并行域内的代码封装成线程任务函数，以便并发线程的执行；
- 2) 产生调用任务函数的代码；
- 3) 数据子句中的变量数据环境的变换处理；

上面的三各功能实际上是交织在一起的，在具体实现中并不是一个个独立模块，这些功能被安排成一系列的小的操作步骤，在 `OMPi` 中变换步骤安排如下：

- 1) 用 `aststmt` 类型的指针 `body` 记录并行域内语句体 `u.omp->body`。
- 2) 处理所有的数据子句内的变量，记录变量名及其数据环境类型，归约变量也需要记录。
- 3) 检查是否有 `if` 子句和 `numthread` 子句，并用 `astexpr` 类型的结构体分别记录 `if` 子句的条件表达式(`ifexpr`)和 `numthread` 的表达式(`numthrexp`)。
- 4) 检查是否有非全局共享变量，并处理缺省的变量数据环境类型（对应 `default` 子句）。
- 5) 完成私有变量和共享变量的声明，这些声明用四个 `aststmt` 类型的结构体来保存，它们是 `shvardecl`、`civardecl`、`prvardecl`、`fpvardecl` 分别对应于共享变量、`copyin` 变量、`private` 变量和 `firstprivate` 变量的声明。最终将这四种声明合并为两大类，共享的 `shvardecl` 和私有的 `prvardecl`。
- 6) 用可以产生并行域的代码替换现有节点，新节点形式如下：

```
{\n    <描述共享变量的结构体（如果有的话）>\n    if(前面第 2 步记录的 if 子句的条件表达式 ifexpr)\n        ort_execute_parallel(...);\n    else\n        ort_execute_serial(...);\n}
```

其中 `ort_execute_parallel()` 表示并行域的条件子句条件成立时将使用多个线程来并发执行，而 `ort_execute_serial()` 表示只用一个线程来执行并行域内代码。

- 7) 给第 1 步记录的 `body` 语句前面加上 `copyin`、`firstprivate` 等变量的初始化代码，在 `body` 的后面加上 `reductions` 变量（归约变量及其操作类型在第 2 步作了记录）的处理操作语句，最后在添加上 “`return (void*) 0;`” 语句，为后面第 8 步进行函数封装作准备。
- 8) 使用 `FuncDef()` 函数将第 7 步准备好的代码封装成一个线程任务函数，其编号通过预先调用 `new_thrfunc()` 获得并记录在全局变量 `_tfn` 字符串中。
- 9) 最后用 `xfrom_add_threadfunc()` 将线程任务函数挂接到一个 LIFO 队列上，等待所有变换完成之后再插入到 AST 树中。插入这些函数的工作是由 9.1.1 的四大步骤中的 (2) `xform_thread_functions` 和 (3) `place_thread_functions` 完成的。

可以看出，这些步骤地安排并不是必须严格顺序执行的，例如对于第 7、8 步的并行域内的代码封装可以提前到第 6 步之前。

AST 结构变换

对于 `parallel` 制导指令指出的并行域代码，它们所对应的 AST 将被拆解为两部分，一部分留在原来的位置我们称为并行域的外壳，另一部分称为并行核的代码将单独出来形成线程任务函数，插入到调用它的函数前面的位置。在外壳部分和并行核部分之间需要传递共享变量地址指针列表，在并行核里面还需要建立私有变量、处理归约操作。这些工作将由 `x_parallel.c` 中的 `xform_parallel()` 函数来完成。

首先来看看并行域外壳和核的拆解过程，假设有如图 9.6 所示的 `parallel` 制导指令 AST 结构：

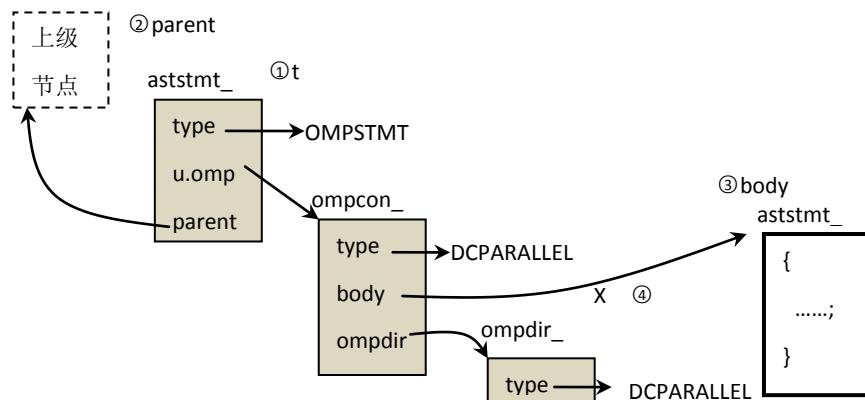


图 9.6 `parallel` 构造的 AST 节点

首先记录当前节点指针 `t` (图 9.6-①)、并行核节点指针 `body` (图 9.6-③) 以及父结点 `parent` 指针 (图 9.6-②)，然后断开指向并行核代码的指针 (图 9.6-④)，再释放 `t` 子树 (对应图 9.6 中灰色部分)。此时并行核代码仍可通过 `body` 指针访问。

接着需要用并行域外壳代码替换原来的 `t` 子树，此处我们只给出不带 `if` 子句的 `parallel` 制导指令的情况。在外壳程序中将给出并行线程需要执行的命名为 `_thrFuncXX` 的任务函数。

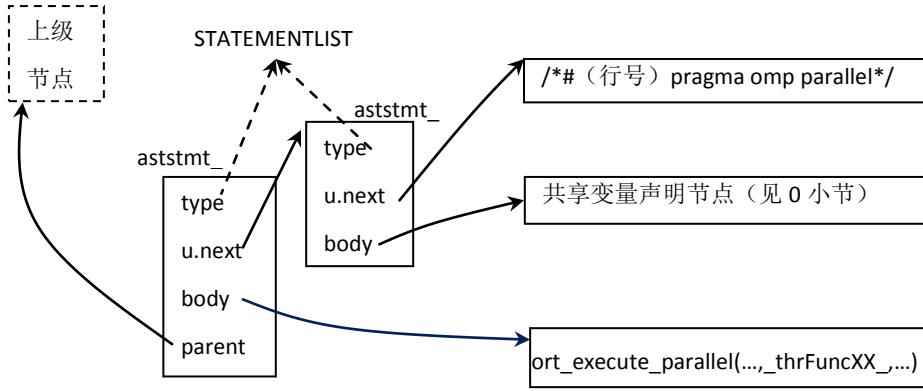


图 9.7 并行域外壳代码的 AST 子树结构

然后是并行域核代码的封装，也就是对图 9.6-④粗黑框部分代码封装成单独的线程任务函数。使用 BlockList()拼接函数和各种 AST 节点生成函数，按照图 9.8 的形式生成新的 AST 子树。最后再用“`static void * _thrFuncXX_(void * __me) { ... }`”将这棵子树封装成函数代码。此函数子树将暂时存放在图 9.9 所示的一个 LIFO 链表中，在全部任务函数生成好后一并插入到源代码的 AST 中。

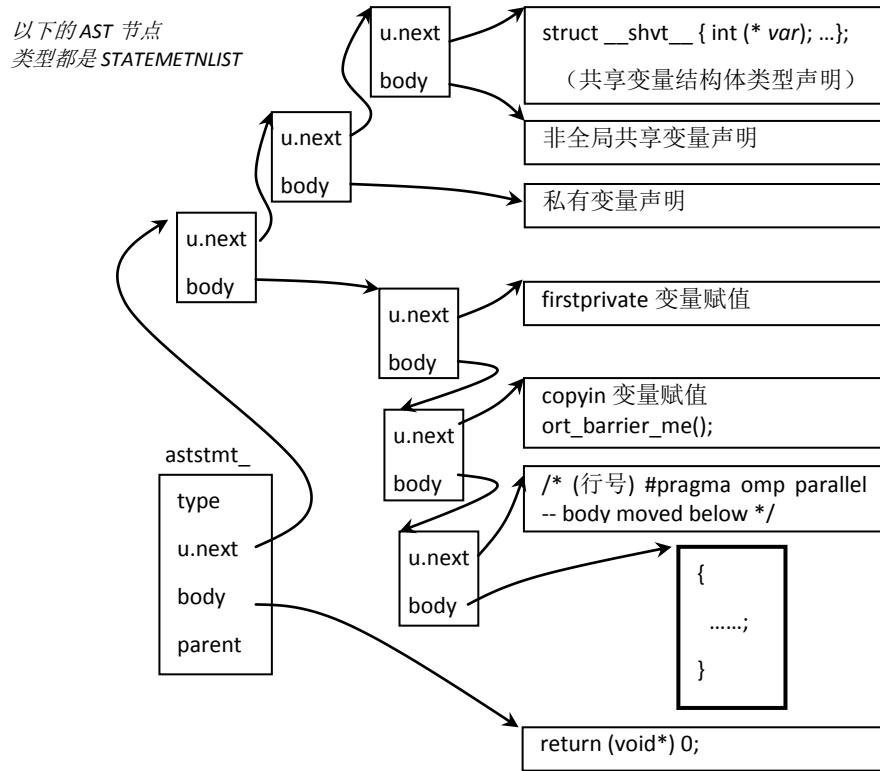


图 9.8 并行域核代码封装前的 AST 结构

可以看到新的 AST 子树中添加了很多关于变量声明和初始化的语句节点，这些语句的产生参见 9.2.7 的分析。

任务函数

由于在变换时，每次遇到并行域，都要将里面的功能代码用一个独立的任务函数来封装，这些函数被顺序命名，并且保存在一个列表中，以便将来插入到正确的地方。OMPI 将使用如

图 9.9 所示的一个 LIFO (Last In First Out) 来记录这些函数，每次完成一个任务函数的封装后就调用 `xfrom_add_threadfunc()` 将它插入到该链表中。

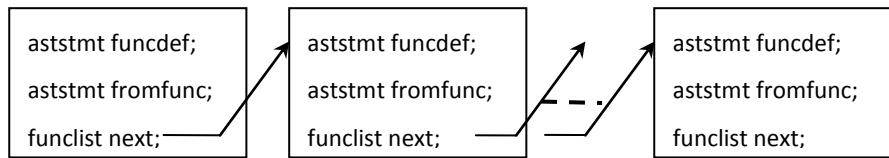


图 9.9 并行域内的线程任务函数 LIFO

这个 LIFO 的节点是类型为 `funclist` 的结构体，其中 `funcdef` 指向被封装的线程任务函数代码，`fromfunc` 是调用该任务函数的上级函数（封装好的任务函数必需插入到这些调用函数之前）。

前面 9.1.1 小节（变换流程）中提到的 `place_thread_functions()` 函数将遍历此 LIFO 链表，并逐个将这些任务函数插入到调用它们的函数之前。

9.2.5 for 变换

OpenMP 的 `for` 制导指令的变换也是比较复杂的，从 7.1 小节可知 `for` 变换根据调度模式的不同而有三种形式，每种调度的循环变量划分方式都不同，变换过程也需做相应的调整。`for` 制导指令本身与并行域的并发执行并没有什么直接关系，并行域的产生是 `parallel` 制导指令的责任。但是在 AST 中，OpenMP `for` 节点通常是在并行域内的，因此往往先对 `for` 节点完成变换后再进行外部的 `parallel` 节点的变换。

下面给出 `for` 变换的操作步骤：

- 1) 记录必要信息。用 `s` 指针记录 `for` 循环体的语句节点，用 `var` (`symbol` 类型) 记录循环变量，用 `cond` 指针记录循环终止条件的表达式，用 `incr` 指针记录循环增量操作的表达式，用 `lb` 和 `ub` (`astexpr` 类型) 记录循环变量的上下界，用 `step` 记录增量步长，用 `nw` 记录 `nowait` 子句，用 `sch` 记录调度子句，用 `ord` 记录 `ordered` 制导指令。记录调度类型到 `schedtype`，记录 `chunk` 大小到 `schedchunk`。上面一些信息将用于划分任务，划分原理参见 7.1。
- 2) 生成私有变量、归约变量等的声明语句节点；生成用于控制循环的 `from_`、`to_`、`step_` 等变量的声明节点；生成 `ort_entering_for()`、`ort_leaving_for()` 等函数调用的节点；生成 `firstprivate`、`lastprivate` 和 `reduction` 变量赋值的语句节点。
- 3) 生成带有任务分担能力的 `for` 循环节点，该 `for` 节点的循环体内的代码仍是旧的 `for` 语句内部的循环体代码。
- 4) 根据调度方式不同，生成能够计算 `from_`、`to_` 和 `step_` 变量的语句节点。
- 5) 将 1)、2)、3) 和 4) 拼接成一棵新的 AST 子树，删除原来旧子树。

下面先来看看变换前的 `for` 节点 AST 结构，仍以 7.1.5 的代码为例。

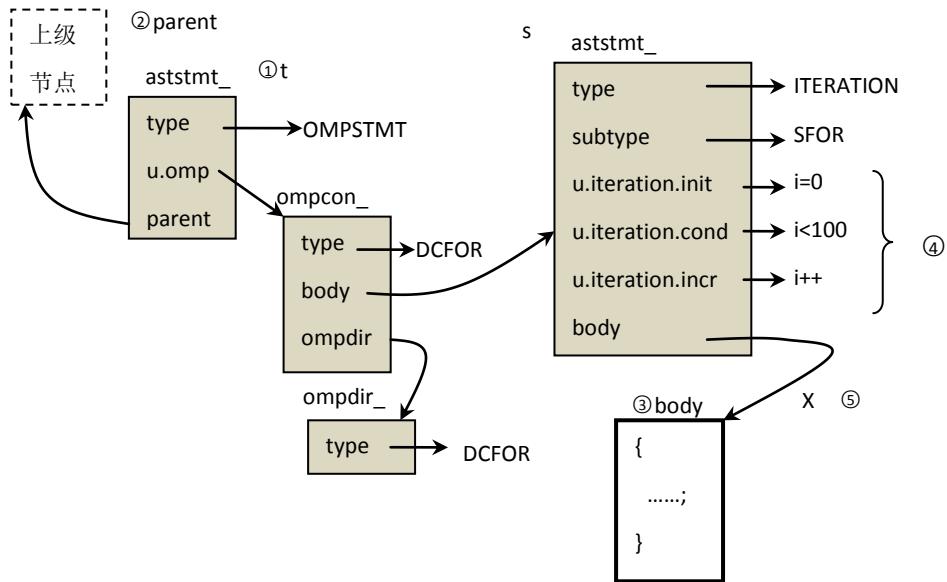


图 9.10 for 构造的 AST 节点

在变换之前，需要记录图 9.6 的 `t` 指针①、父结点 `parent` 指针②、循环体 `body`③和循环变量等参数④。变换后断开指针⑤（`OMPI 1.0.0` 断开的不是这个地方，导致有内存泄露），然后可以释放 `t` 子树①。

由于调度机制的不同，OpenMP for 子树变换也不相同，下面只以缺省的调度方式为例（由于 for 语句不能带 `shared` 子句，故例子中只有私有变量处理）分析其变换过程。

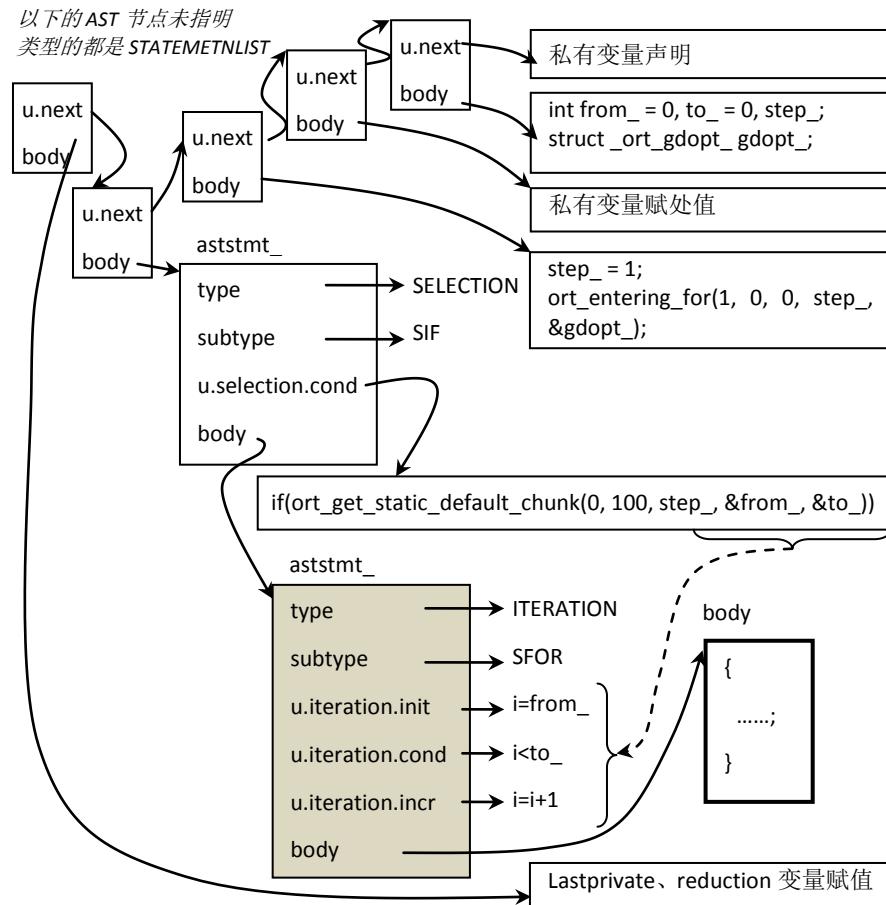


图 9.11 循环体代码变换后的 AST 结构

图 9.8 给出了循环体代码变换后的 AST 结构，可以看出该代码翻译变换所采用的“框架”模板相当复杂，如果考虑将动态调度、指导调度将更复杂。其中 if 节点调用的 `ort_get_static_default_chunk()` 函数为本线程获取了 `from_` 和 `to_` 变量，它将影响 for 节点所执行的循环变量变换范围，从而实现各个线程进行任务分担的目的。

9.2.6 sections 变换

`sections` 变换过程和 `parallel`、`for` 的变换采用的相同的技术，都是利用拼接函数和 AST 节点创建函数、按照第 7 章给出的框架，对 `sections` 子树进行裁减分割、变换、替换和拼接实现“框架”的要求，从而实现 AST 变换的。由于 `sections` 的 AST 子树比 `parallel` 和 `for` 制导指令的要复杂一点，下面的图 9.12 给出具有两个 `section` 的未变换前 `sections` 的 AST 子树结构。

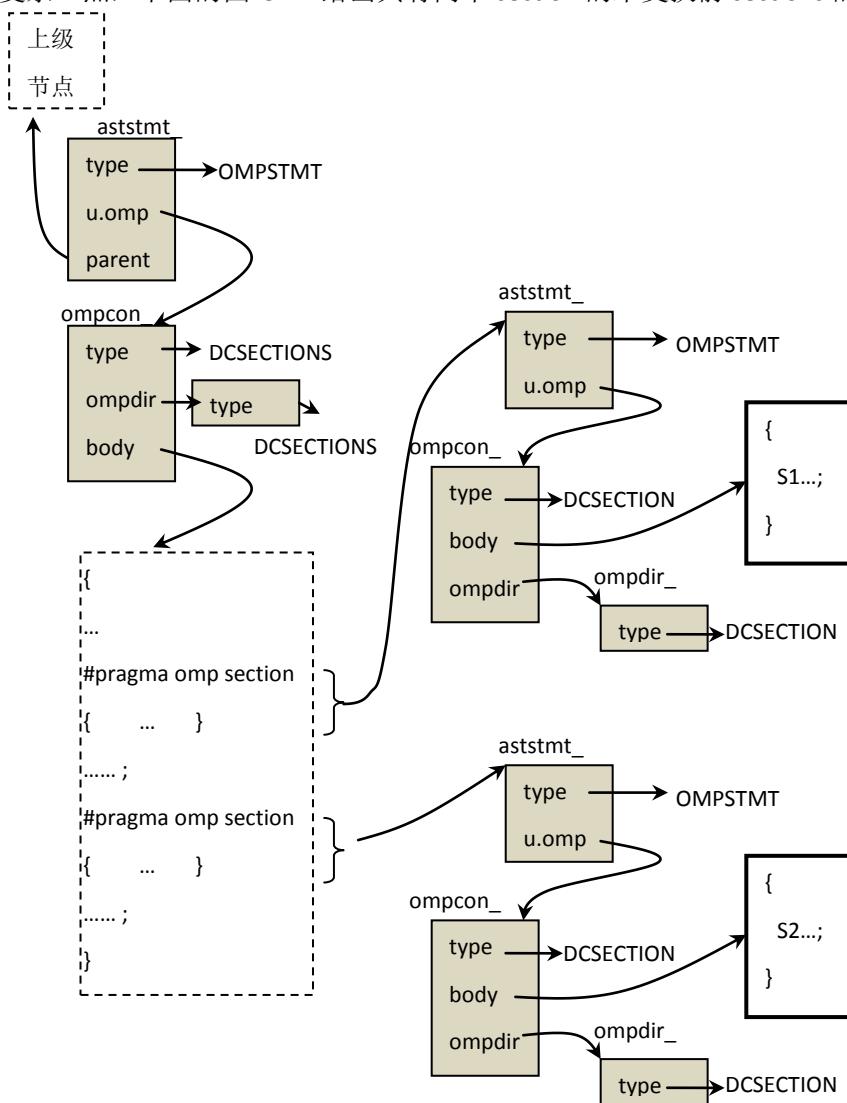


图 9.12 变换前 sections 的 AST 子树结构

由 `sections` 子树内部还有多个 `section` 要表示、层次较高，因此图 9.12 有一部分 AST 直接用源代码表示（虚线框中），以简化图的结构便于阅读。在代码变换中，除了 `S1` 和 `S2` 的代码

需要保留，其他节点都要释放删除，因此其裁剪过程略显复杂。然后是对 S1、S2 代码的变换和拼接，相关 AST 变换的代码和方法留给读者自行分析研究。

9.2.7 数据环境的处理

OMPI 的数据环境控制分成三个部分来讨论，它们是代码变换以及支撑代码变换的变量收集和作用域问题。

数据子句变换

在 OMPI 的数据环境处理中，数据子句的 AST 变换比较简单，主要是形成私有变量、共享变量、归约变量和线程专有变量等的声明和赋值语句。因此涉及的都是比较简单的 AST 节点的插入、删除操作。由于其变换操作比较简单，这些问题和基本的代码形式都已经在第 8 章变量数据环境控制中已经讲得比较清楚。

实际上述变换操作不是独立完成的，而往往是在 parallel、for、sections 等语句的变换处理中一并处理的。因此在 parallel、for、sections 变换中完成这些变量的声明和处理，需要以下两个基本功能来支持：

1. 各种类型变量的符号的收集；
2. 所收集各种变量符号的具体类型是什么；

下面分别就这两个问题进行分析。

变量收集

变量收集工作首先需要扫描当前并行域或任务分担域的数据子句，将所有在 private、firstprivate、copyin、shared、reduction 子句中的变量保存到一个符号表中。然后根据它们的类型分组并产生相应的变量声明语句及相关操作代码。

变量作用域与类型

在前面讨论的数据环境变换处理中，经常需要产生出变量的声明语句，这些变量是由 OpenMP 制导指令的数据子句中以变量名的形式引用的，比如 private(a,b,c,...)、shared(l,j,k,...)、reduction(x,y,z,...)语句等等。由于存在不同作用域上不同类型的同名变量，所以需要根据当前作用域找到变量的正确类型。下面以 private 为例说明如何借助作用域管理功能正确完成相关工作。假如有以下代码：

```
1. int a;                                作用域 (层 N)
.....
2. {   unsigned long a;                    作用域 (层 N+1)
3.     #pragma omp parallelfor private(a)    并行域 1
4.     for(i=0;i<10;i++)
5.     {
6.       .....
7.     }                                     作用域 (层 N)
.....
8. {   float a;                            作用域 (层 N+1)
```

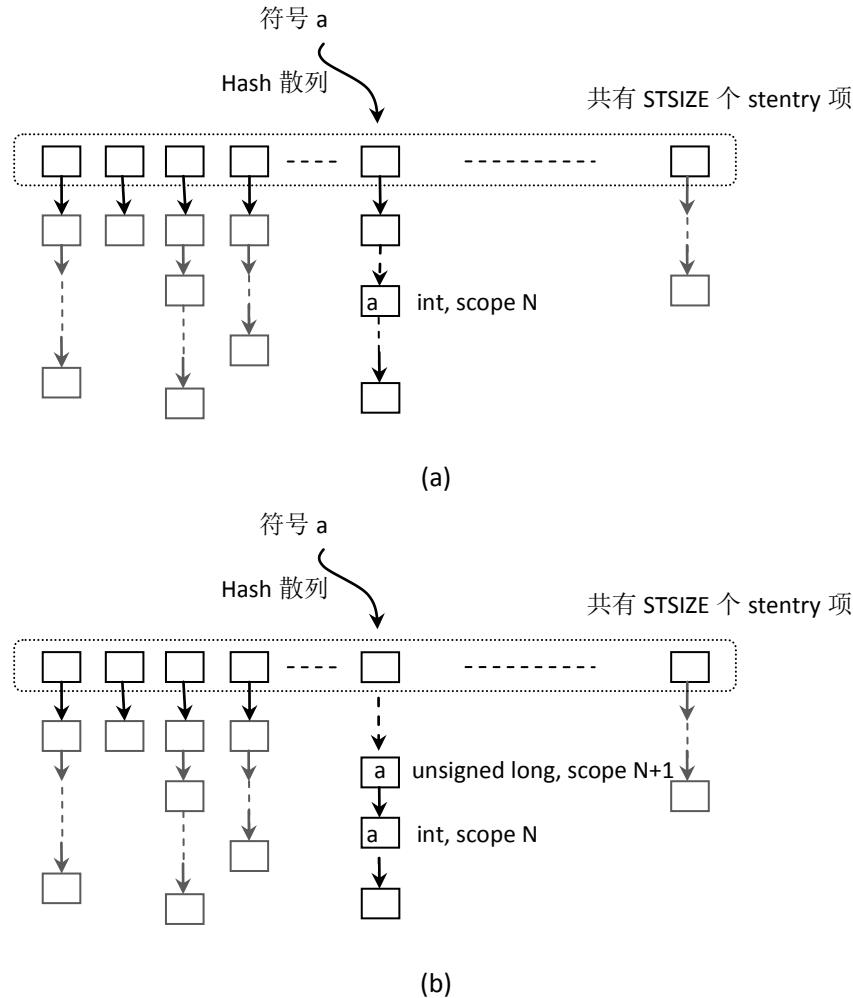
```

8.     #pragma omp parallel sections private(a)      并行域 2
9.     {
10.     .....
11. }
.....          作用域 (层 N)

```

上面的代码在两个并行域中都将 `a` 声明为 `private` 变量，因此在并行内部的代码中应当产生类似于“`unsigned long a;`”和“`float a;`”的声明语句，现在看看符号管理如何帮助它们找到正确的变量和变量类型。

随着对代码扫描的进行（也可以是对 AST 的遍历），在代码声明了 `int a` 之后，其符号表栈的情况如图 9.13-a 所示。此时符号表堆栈中符号 `a` 是整形 `int` 作用域层次为 `N`。



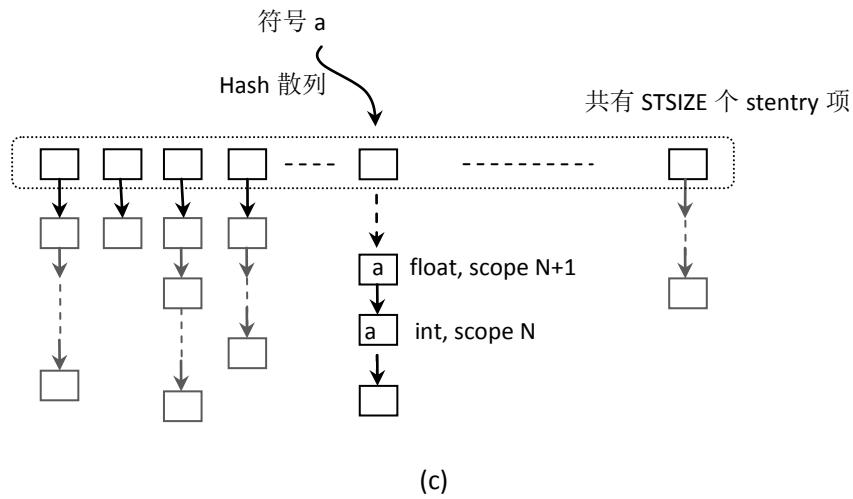


图 9.13 作用域层次变化与 OpenMP 变量引用

然后遇到第一个“{ ... }”结构块从而进入层 N+1 作用域，这里声明了 `unsigned long` 类型的同名变量 `a`，使得符号表变成如图 9.13-b 形式。然后遇到第一个 `parallel for` 制导指令，`private(a)` 指出需要生成私有变量 `a` 的时候，经符号表的查找得知当前起作用的的变量 `a` 是整形 `unsigned long`，因此该私有变量的声明为“`unsigned long a;`”。

接着退出第一个“{ ... }”的结构块，于是“`unsigned long`”类型的符号 `a` 随着清理 N+1 层符号项的操作而删除，符号表堆栈还原到图 9.13-a 状态。随后然后遇到第二个“{ ... }”结构块从而再次进入层 N+1 作用域，这里声明了 `float` 类型的同名变量 `a`，使得符号表变成如图 9.13-c 形式。然后遇到第一个 `parallel sections` 制导指令，`private(a)`指出需要生成私有变量 `a` 的时候，经符号表的查找得知当前起作用的的变量 `a` 是浮点 `float`，因此该私有变量正确的声明为“`float a;`”。

9.3 代码优化

在生成目标 C 代码的时候，可以在变换后的 AST 基础之上进行源代码级的优化，下面将介绍一些基本的优化方法。

OpenMP 程序运行时额外开销的很大一部分是因为并行域上的线程 fork-join 开销，因此并行域的合并可能对 OpenMP 程序性能有明显的作用。`fork-join` 是面向任务的模型，任务和线程之间是动态的关系，而 SPMD 模型则是面向执行者的，不存在线程组的产生和撤销操作及时间开销。并行域合并的优化思想是：通过将相邻的 OpenMP 并行域进行合并，减少线程组的产生和撤销，从而接近于 SPMD 的执行模式，提高程序运行速度。

为了实现并行域的合并，需要解决如下几个问题：

1. AST 的裁剪、拼接和修改等能力，这个与前面讨论的 AST 变化要求相同；
2. 确定并行域合并的边界。从原理上说，可以将 OpenMP 程序全部改造成只带有一个 `parallel` 制导指令的 SPMD 类型的程序，至少通过人工修改是可能的。但是实际上利用编译器进行优化是，并行域的合并很难跨越所在的函数。因此并行域的合并一般只限于该 `parallel` 制导指令所在的函数内部。

3. 保证同步行为不被破坏。由于并行块末尾有隐含同步操作，但是因合并之后并行域的边界撤销从而丢失了一些路障同步，需要在必要的地方插入必要的路障同步。
4. 串行代码的保护。相邻并行域之间的串行代码因并行域的合并而在多个线程中并发执行，从而可能引发问题。解决方法可以使用 `master`、`single` 制导指令保持其原来的语义，或者进行变换使得它可以并发执行的同时保证正确性。

另外还可以删除用户编写 OpenMP 程序时添加的不必要的冗余同步。冗余同步操作的判断依据是：删除该同步语句并不影响程序的结果。其中一种冗余同步是用户在带有隐含同步的制导指令处添加的路障同步，该冗余同步很容易消除。再进一步，可以修改不合理的 `for` 循环的嵌套顺序或 `chunksize`、尝试将 `for` 的动态调度修改为静态调度等等方法。

总体上来说这种将 OpenMP 翻译器与后端的优化编译器分割开来方案，限制了线程级并行与指令级并行性开发之间的交互，使得很多优化无法进行。感兴趣的读者可以阅读其他材料以了解 OpenMP 编译优化的问题。

9.4 AST 输出

完成 AST 的变换后，最终需要通过 AST 的输出还原成源代码文本文件。从形式上看，AST 的输出实际上是语法分析的逆过程，但是 AST 输出要比 AST 的建立要简单直接得多。

OMPI 根据 AST 输出源代码文件的入口函数是 `ast_show()` 函数，通过遍历 AST 来将 AST 节点还原成字符文本。然后，还需要将这些字符串文本输出到文件中，这个过程可以是用一个缓冲区保存这些输出的字符文本，然后再写到文件中，也可以是利用 `printf` 函数来输出字符文本，通过应用程序的输出 IO 将这些字符文本写到文件中。后者因为不需要在内存中管理一个巨大的字符缓冲区，因此更便于实现。

9.4.1 OMPI 的 AST 输出

遍历过程是通过递归调用输出函数来完成的，这些输出函数并不多，与 AST 节点类型是相对应的，函数名称和调用层次关系如图 9.14：

```

void ast_stmt_show(aststmt stmt);
    ↘ void ast_stmt_jump_show(astexpr tree);
    void ast_stmt_iteration_show(astexpr tree);
    void ast_stmt_selection_show(astexpr tree);
    void ast_stmt_labeled_show (astexpr tree);

-----
void ast_expr_show(astexpr tree);
    ↘ void ast_decl_show(astdecl tree);
    void ast_spec_show(astspec tree);
    void ast_ompcon_show(ompcon tree);
    ↘ void ast_ompdir_show(ompdir t);
    void ast_ompclause_show(ompclause t);

```

图 9.14 AST 输出函数及其基本调用层次

这些函数可以从名称上看得出来是对应什么类型的节点的输出，由于 AST 的最上层节点一定是一个语句节点所以 `ast_show()` 函数里面首先调用的就是 `ast_stmt_show()`。然后再根据语句节点类型的不同而调用 `ast_stmt_jump_show()`、`ast_stmt_iteration_show()`、`ast_stmt_selection_show()`、`ast_stmt_labeled_show()`、`ast_expr_show()`、`ast_ompcon_show()`。其中对于表达式语句节点还会调用 `ast_decl_show()` 和 `ast_spec_show()`。而对于 OpenMP 节点则还调用 `ast_ompdir_show()` 和 `ast_ompclause_show()` 来输出制导指令和相应的子句。

通过递归调用上述的函数将各个语法构造从 AST 节点还原到字符文本形式。由于是使用 `printf` 来输出字符串，因此这些字符串输出到标准输出文件中，即 `stdio` 上。通过应用程序的输出 IO 重定向可以直接将这些字符文本保存到文件中。例如变换程序名字是 `myompi`，该程序读入一个 OpenMP/C 源程序利用 `printf` 输出变换结果，那么通过简单的命令行：`myompi myprog.c > myprog_omp.c` 可以将变换结果保存在文件 `myprog_omp.c` 中。由于 AST 不是简单且规则的树，其节点的子节点数目不定，且子节点的顺序也没有严格规定，因此不能用简单二叉树的前序遍历、中序遍历、后序遍历算法进行遍历，但为了输出字符文本执行的基本上是深度优先遍历，每个节点的子节点的遍历顺序是根据该节点的语法元素的排列顺序来进行的。

9.4.2 OpenMP 节点输出

下面以 OpenMP 节点输出为例，看看 AST 节点是如何还原成字符文本的。当遍历过程中遇到一个 OpenMP 节点时，`ast_stmt_show()` 根据节点类型进一步调用 `ast_ompcon_show()` 来输出这个节点。而 `ast_ompcon_show()` 又将调用 `ast_ompdir_show()` 和 `ast_stmt_show()` 来先后输出相应的制导指令和并行域内代码（编译制导指令中除了 `barrier` 和 `flush` 没有相应代码外，其他制导指令都将修饰一定的代码）。制导指令的输出函数 `ast_ompdir_show()` 函数代码如下：

1. void ast_ompdir_show(ompdir t)
2. {

```

3.     printf("#pragma omp %s ", ompdirnames[t->type]);      输出相应的制导指令字符串
4.     switch (t->type)                                     下面几种制导指令还需输出其他内容
5.     {
6.         case DCCRITICAL:
7.             if (t->u.region)
8.                 printf("(%s)", t->u.region->name);           critical 可能需要输出带有的名字
9.             break;
10.        case DCFLUSH:
11.            if (t->u.varlist)
12.            {
13.                printf("(");
14.                ast_decl_show(t->u.varlist);                  flush 需要输出相应变量列表
15.                printf(")");
16.            }
17.            break;
18.        case DCTHREADPRIVATE:
19.            if (t->u.varlist)
20.            {
21.                printf("(");
22.                ast_decl_show(t->u.varlist);                  threadprivate 也要输出变量列表
23.                printf(")");
24.            }
25.            break;
26.        default:
27.            if (t->clauses)
28.                ast_ompclause_show(t->clauses);              如果有子句，则输出
29.            break;
30.        }
31.    printf("\n");
32. }

```

制导指令行的输出首先是制导指令然后是子句。上面的第一行语句 “`printf("#pragma omp %s ", ompdirnames[t->type]);`” 这个将打印出一行形如：

```
#pragma omp XXXXX
```

的字符，其中 XXXXX 是根据 `ompdirnames[t->type]` 而确定的（比如 parallel、for 等）。如果是 critical 制导指令且有名称则还继续打印名称，如果是 threadprivate 或 flush 制导指令并且带有变量声明，则调用 `ast_decl_show()` 继续打印这些变量声明（这些变量声明的外边用 (...) 括起来）。如果带有子句则使用 `ast_ompclause_show()` 进行打印输出。函数 `ast_ompclause_show()` 的代码如下：

```

1. void ast_ompclause_show(ompclause t)
2. {
3.     if (t->type == OCLIST)
4.     {
5.         if (t->u.list.next != NULL)

```

```

6.      {
7.          ast_ompclause_show(t->u.list.next);
8.          printf(" ");
9.      }
10.     assert((t = t->u.list.elem) != NULL);
11. }

12.    printf("%s", clauseenames[t->type]);
13.    switch (t->type)
14.    {
15.        case OCIF:
16.        case OCNUMTHREADS:
17.            printf("( "); ast_expr_show(t->u.expr); printf(" )");
18.            break;
19.        case OCSCHEDULE:
20.            printf("( %s%s", clausesubs[t->subtype], t->u.expr ? "," : " ");
21.            if (t->u.expr)
22.                ast_expr_show(t->u.expr);
23.            printf(" )");
24.            break;
25.        case OCDEFAULT:
26.            printf("( %s )", clausesubs[t->subtype]);
27.            break;
28.        case OCREDUCTION:
29.            printf("( %s : ", clausesubs[t->subtype]);
30.            ast_decl_show(t->u.varlist);
31.            printf(" )");
32.            break;
33.        case OCCOPYIN:
34.        case OCPRIVATE:
35.        case OCCOPYPRIVATE:
36.        case OCFIRSTPRIVATE:
37.        case OCLASTPRIVATE:
38.        case OCSHARED:
39.            printf("( ); ast_decl_show(t->u.varlist); printf(")");
40.            break;
41.        case OCNOWAIT:
42.            printf("nowait");
43.            break;
44.        case OCORDERED:
45.            printf("ordered");
46.            break;
47.    }
48. }

```

AST 中 OpenMP 节点的数据子句的数据结构请参考 5.2.5 小节。该函数首先判断是否为子句列表，如果是的话将逐个遍历。当处理的是单个子句时，首先子句命令字符串打印出来（比如 `if`、`num_thread` 或 `private` 等等），然后再将根据类型的不同而作差别处理。如果是 `if` 和 `num_thread` 子句，那么还需要将条件表达式或线程数目表达式打印出来；对于 `schedule` 制导指令，则需要打印形如 `(YYY:XXX)` 的字符串，其中调度类型 `YYY` 是根据子类型而定，而后面的 `chunksize` 表达式 `XXX` 则可能为空，如果不为空则使用 `ast_expr_show()` 打印出来；对于 `reduction` 制导指令则还需要输出归约操作和变量列表，形如 `(YYY:XXXX)`，其中 `YYY` 可能是`+`、`-`和`*`等操，`XXXX` 则是由 `ast_decl_show()` 打印出来的变量声明列表；如果是 `private`、`shared` 等数据子句，还需要用 `ast_decl_show()` 打印出变量列表；如果是 `nowait` 或 `ordered` 子句，那么需要打印出 `nowait` 或 `ordered` 字符串。

如果是对编译后的 AST 进行输出，那么不再会有 OpenMP 节点出现，因为所有 OpenMP 节点都经过替换后变成普通 C 语言的节点了。

9.5 小结

本章首先分析了源代码变换所应具备的各种支撑函数的功能，指出 AST 节点创建和拼接是代码变换的基础。然后讨论如何通过 AST 的遍历，逐个子树和节点进行变换，特别是 OpenMP 子树节点的变换。书中以 `atomic`、`parallel`、`for` 和 `sections` 制导指令为例，通过 AST 子树节点的变换图例，说明如何利用“框架”完成代码变换。其中 `parallel` 变换涉及两棵子树，分别对应于并行域外壳代码和核代码，这两个子树插入不同地方，外壳代码留在原处，而核代码封装成任务函数插入到更前面的位置。`for` 制导指令的变换也比较复杂，根据不同的调度机制有不同的变换“框架”，其关键是能够控制 `for` 循环的起止范围，在不改变循环体内代码的情况下实现任务分担。将 `sections` 制导指令的变换留给读者自行分析。数据环境控制问题也作了分析讨论，代码变换中很重要的一点是数据子句中变量的收集，从收集的变量符号确定其当前作用域的数据类型，然后在必要的地方生成相关的变量声明语句。

另一部分内容是 AST 的输出，通过遍历 AST 的各节点并根据语法构造的顺序打印对应的字符串，再通过 IO 重定向到文件中，可以将 AST 还原成源代码文件。

第 10 章 运行环境

编译后的 OpenMP 程序并不是直接运行在 C 语言环境下的，它还需要相应的运行库的支持。运行库的功能设计，是在编译系统设计时和编译器一起完成的。设计编译器必须知道运行系统到底能提供什么能力，才能确定出目标代码的形式，因此不同的运行库能力对应不同的目标代码形式。或者反过来先确定目标代码形式，然后再决定运行库的能力。

本章将讨论 OpenMP 程序运行环境中的环境变量问题、API 函数以及并行语义支撑函数，其中的并行支撑函数的形式和功能强弱与设计编译系统时的运行库形式和能力强弱有关，因此没有统一的要求，只能以具体的编译系统来讨论。这里讨论的内容也适用于其他系统，比如 GCC 的 OpenMP 编译器 GOMP 也是类似的，同样有类似的代码翻译变换和与之衔接的 gomp 库。GCC 编译器产生的 OpenMP 可执行文件需要 libgomp.so 动态库或 libgomp.a 静态库的支持。

下面先对 OpenMP 的运行环境 ORT 的重要数据结构和初始化进行简单讨论，然后分析其并行语义支撑函数的实现问题，最后给出 OpenMP API 和环境变量的支持。

10.1 重要数据结构

OpenMP 的运行环境中几个重要的数据结构：描述 ORT 的结构体、线程池、EECB、任务分担结构体、共享变量结构体。

10.1.1 ORT

运行环境的主要特性包括：

1. 保存在 ICV 变量中的 OpenMP 特性：

缺省线程组的大小；是否支持动态改变线程数目；是否支持嵌套并行；嵌套层数限制；处理器数目；当前 runtime 调度设置；堆栈大小等等。

2. 线程库能力：

对线程数目动态变化以及嵌套并行的支持情况；缺省线程数目；最大线程数目；最大嵌套层次等等。

下面先来看看 ORT 描述运行环境的数据结构 `ort_globals` 及其指针`*ort`。其中线程库能力将对 OpenMP 特性有所限制和约束。OpenMP 的 ORT 数据结构定义在 `ort.c` 文件中，具体如下：

```
1. struct {
2.     ort_icvs_t          icv;           内部控制变量
3.     ort_caps_t          eecaps;        线程库的能力
4.     volatile ee_lock_t  atomic_lock; /* Global lock for atomic */
```

```

5.     volatile ee_lock_t preparation_lock; /* For initializing user locks */
6. } ort_globals, *ort;

```

其中 icv 用于记录 OpenMP 的 ICV 变量， eecaps 记录了当前线程库的能力，另外还有两个互斥锁， atomic_lock 用于 atomic 制导指令的互斥。

内部控制变量 ICV 的数据结构如下：

```

1. typedef struct {
2.     int dynamic;      /* Per task */
3.     int nested;
4.     int rtschedule;   /* For runtime schedules */
5.     int rtchunk;      /* ditto */
6.     int nthreads;     /* default # threads for a team */
7.     int ncpus;        /* Global */
8.     int stacksize;
9.     int waitpolicy;
10.    int threadlimit;
11.    int levellimit;
12. } ort_icvs_t;

```

可以看出它与 OpenMP 标准中的 ICV 不完全一样，增添了一个其他变量在里面。线程能力描述的数据结构如下：

```

1. typedef struct {
2.     int supports_nested;
3.     int supports_dynamic;
4.     int supports_nested_nondynamic;
5.     int max_levels_supported; /* -1 if no limit, else >= 0. */
6.     int max_threads_supported; /* -1 if no limit */
7.     int default_numthreads;   /* the default # threads for a team */
8. } ort_caps_t;

```

该结构用于记录线程库对嵌套并行、动态改变线程数目、最大嵌套级别、最大线程数目、缺省线程组大小的信息。每种线程库在初始化时都要填写这个结构体，以提供必要信息给 ORT 使用。

10.1.2 线程池与 EECB

线程池和 EECB 已经在第 6 章讨论过。这里分析一下线程池数据结构、EECB、具体的线程三者是如何互动的。

首先 OMPI 的运行环境在初始化的时候先创建一批线程，比如是 pthreads 线程，此时每个线程用一个线程池的项 othr_pool_t 数据结构来描述，此时 othr_pool_t->workfunc 因未分配任务所以为空，而 othr_pool_t->spin 为 1 表示正在空转。它所描述的 pthreads 线程则在执行 threadjob() 函数，检测到 spin=1 因此调用 WAIT WHILE() 自旋空转。

当需要创建并行域而执行 `ort_execute_parallel()` 的时候将调用 `ee_create()` 修改线程池线程的 `othr_pool_t->workfunc` 和 `othr_pool_t->spin`, 为相应的线程提供任务函数并解除自旋标记。Pthreads 线程将因为 spin 标记解除而从 `WAIT WHILE()` 解脱出来, 这时候才会建立起 EECB。也就是说 EECB 是动态概念, 只在 `pthread` 线程分配了任务之后才建立起来的。

在底层线程完成了任务函数后, 返回到 `threadjob()` 函数中, EECB 直接弃之不用即可, 然后归还线程到线程池中, 并将 `othr_pool_t->workfunc` 清空和 `othr_pool_t->spin` 置位, 使得线程继续进行自旋空转。

10.1.3 任务分担结构

任务分担结构是和 `nowait` 子句相关的, 如果没有 `nowait` 子句引出来的麻烦, 也就不需要下面这些“记账”信息的数据结构。正如第 7 章中关于 `nowait` 问题的讨论, 虽然 Omni 编译器对 `for` 和 `single` 采用了忽略 `nowait` 子句、对 `sections` 采用静态任务划分的办法, OMPI 采用的是允许固定数目的 `nowait` 任务分担域同时生效。对于每个任务分担域, OMPI 使用了一个“记账”数据结构来跟踪当前并行域内活跃的任务分担域。对于一个并行域内的任务分担域, 满足以下条件就称为活动分担域:

1. 至少有一个线程进入该任务分担域;
2. 只要还有一个线程没有离开该任务分担域。

ORT 中有两种结构体用于任务分担的记账信息。在线程组长 EECB 里面有一个 `ort_workshare_t` 类型的 `workshare` 变量, 用于记录当前并行域内线程组的所有(一个或多个)任务分担信息。如果并行域内有 `nowait` 任务分担域, 则可能多于一个任务分担记账信息。而 `ort_workshare_t` 里面则是用 `wsregion_t` 类型的结构体来保存一个任务分担域的记账信息。

每当进入一个并行域时 `ort_execute_parallel()` 将会调用 `init_workshare_regions()` 对线程组长的 EECB 任务分担结构进行初始化, 该结构 `ort_workshare_t` 定义如下:

```
1. /* A table of simultaneously active workshare regions */
2. typedef struct {
3.     /* This is for BLOCKING (i.e. with no NOWAIT clause) regions. */
4.     wsregion_t blocking;
5.     /* This is for keeping track of active NOWAIT regions. */
6.     volatile int headregion, tailregion; /* volatile since all threads watch */
7.     wsregion_t active[MAXACTIVEREGIONS];
8. } ort_workshare_t;
```

变量 `blocking` 用于阻塞式的任务分担记账信息, `headregion` 和 `tailregion` 用于跟踪活跃的非阻塞任务分担域, `active[]` 数组可以保存 `MAXACTIVEREGIONS` 个活跃的任务分担域的记账信息。如果并行域内有多个 `nowait` 的任务分担域, 那么将会有多个有效的 `active` 元素, 每个对应一个任务分担域。

一个任务分担域的记账信息 `wsregion_t` 既可以用于阻塞式也可以用于非阻塞式的任务分担, 它是 `active[]` 数组的一个元素。其结构体类型定义如下:

```
1. /* For workshare regions */
2. typedef struct
```

```

2.  {
3.      ee_lock_t      reglock;           /* Lock for the region */
4.      volatile int empty;            /* True if no thread entered yet */
5.      volatile int notleft;          /* # threads that have not left yet */
6.      int           init;            /* 1 if the region was initialized */

7.      /* SECTIONS & FOR specific data */
8.      volatile int sectionsleft; /* Remaining # sections to be given away */
9.      ort_forloop_t forloop;       /* Stuff for FOR regions */
10. } wsregion_t;

```

其中 `reglock` 用于互斥访问, `empty` 标记还没有线程进入 (这个标记对 `single` 制导指令来说已经足够), `notleft` 记录了还未离开的线程数目, `init` 标记是否进行了初始化。对于 `sections` 和 `for` 制导指令, 各自有 `sectionsleft` 用于记录还未完成的 `section` 数目、`forloop` 记录未完成的循环变量取值范围。后者也很简单, 只是记录了当前循环变量的下界 (动态变化的) 和 `order` 顺序执行的信息。

```

1.  /* For FOR loops */
2.  typedef struct {
3.      /* lb is initialized to the loop's initial lower bound. During execution,
4.         * it represents the "current" lower bound, i.e. the next iteration to be
5.         * scheduled.
6.         * *** IT IS ONLY USED FOR THE GUIDED & DYNAMIC SCHEDULES ***
7.      */
8.      volatile int      lb;           /* The next iteration to be scheduled */
9.      ort_ordered_info_t ordering;   /* Bookkeeping for the ORDERED clause */
10. } ort_forloop_t;

```

顺序执行的制导指令 `order` 的记账信息如下, 包括指出下一个可执行的循环变量值 `next_iteration`、循环变量递增方向 `incr`:

```

1.  typedef struct {
2.      int      next_iteration; /* Low bound of the chunk that should be next */
3.      int      incr;          /* Is > 0 if FOR's step is positive */
4.      ee_lock_t lock;
5.  } ort_ordered_info_t;

```

10.1.4 共享变量结构

对于共享变量, 下面先讨论在应用程序的源代码中是如何体现的, 然后再分析运行环境中相关的数据结构。

目标代码中的共享变量

在进入并行域的时候应用程序将要调用 `ort_execute_parallel()`, 该函数的第三个参数 `void* shared` 就是一个共享变量的结构体, 这个结构体在并行域的外面形成后传递到并行域内部 (参见 8.3.1 小节)。为了便于说明仍将以 8.3.1 小节的例子来说明。

```

1.  int __original_main(int _argc_ignored, char ** _argv_ignored)

```

```

2. {
3.     int a = 1, b = 2;
4. {
5.     struct __shvt__ {
6.         int (* b);
7.         int (* a);
8.     } _shvars = {
9.         &b, &a
10.    };
11.    ort_execute_parallel(-1, _thrFunc0_, (void *) &_shvars);
12. }

```

在并行域外部，`__shvt__`类型的构体变量`_shvars`记录了共享变量的指针列表。下面再看看共享变量的内部形式：

```

1. static void * _thrFunc0_(void * __me)
2. {
3.     struct __shvt__ {
4.         int (* b);
5.         int (* a);
6.     };
7.     struct __shvt__ * _shvars = (struct __shvt__ *) ort_get_shared_vars(__me);
8.     int (* b) = _shvars->b;
9.     int (* a) = _shvars->a;
10. {
11.     (*a) = omp_get_thread_num();
12.     (*b) = (*a);
13. }
14. return (void *) 0;
15. }

```

在各个线程执行任务函数`_thrFunc0_`的时候在此获得共享变量地址列表`_shvars`，从而可以访问到共享变量。

从上面的代码看，应用程序中的代码主要是对非全局共享变量进行封装，将它们的指针保存在一个结构体中，然后将此结构体的指针传入并行域内从而实现共享。也就是说运行环境中调用`ort_execute_parallel()`只是负责一个指针的传递，然后再并行域内调用`ort_get_shared_vars()`获取该指针。

形成共享变量地址列表的结构体`__shvt__`的过程是需要符号表的帮助。在`ast_var.c`文件中有如下定义：

```
syntab      sng_vars = NULL, gtp_vars = NULL, sgl_vars = NULL;
```

其中`sng_vars`用于记录非全局共享变量，`gtp`用于记录`threadprivate`变量，而`sgl`用于记录全局共享变量。这三个变量都是符号表类型的，符号表的结构和操作函数可以参见5.5小节。

EECB 中的共享变量

在前的 `ort_execute_parallel()` 和 `ort_get_shared_vars()` 之间传递这些共享变量的地址指针之间，还需要 ECB 作为中介，其过程如图 10.1 所示。

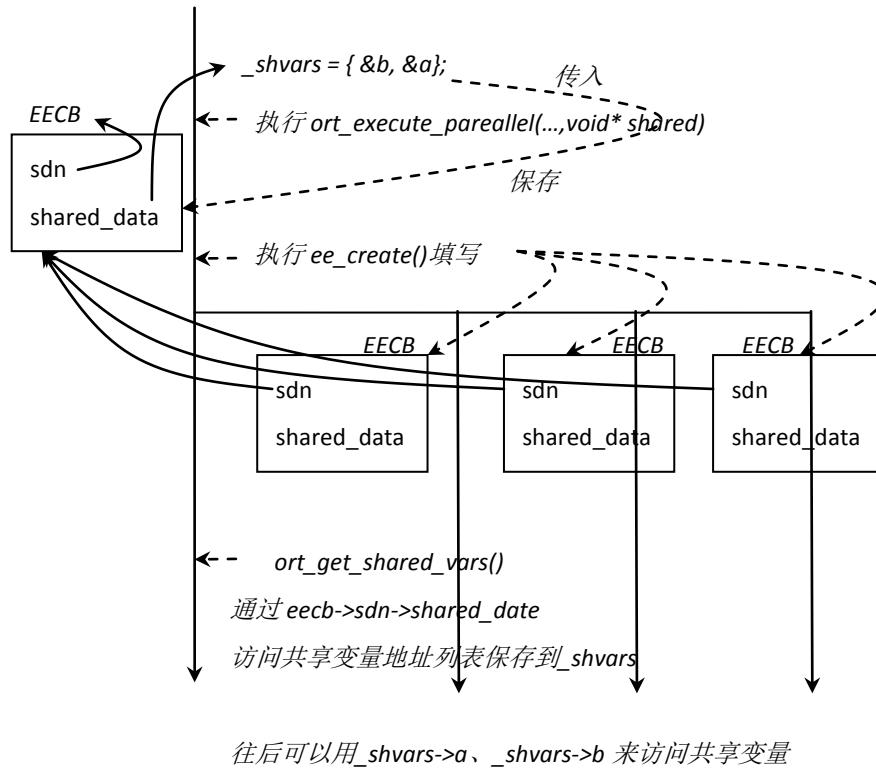


图 10.1 共享变量传递过程

首先通过在线程组长里面保存传入的共享变量地址列表结构到 `shared_data` 指针中，然后再创建线程组的 `ee_create()` 里面将线程组成员的 `eeccb->sdn` 指向线程组长的 ECB，接着用 `ort_get_shared_vars()` 将 `eeccb->sdn->shared_data` 保存到本地指针 `_shvars` 中，从而完成共享变量地址列表的传递过程。

10.2 初始化与退出

OMPI 编译出来的可执行文件在运行的一开始就执行 ORT 运行环境的初始化，因为输出的目标代码的 `main()` 函数非常简单：

```
1. /* OMPI-generated main() */
2. int main(int argc, char **argv)
3. {
4.     int _xval = 0;
5.
6.     ort_initialize(&argc, &argv);
7.     _xval = (int) __original_main(argc, argv);
8.     ort_finalize(_xval);
9.     return (_xval);
}
```

先是调用 `ort_initialize()` 进行初始化，才能执行原来程序的 `main()` 函数（被 `OMPI` 改名为 `__original_main()`），最后在退出时执行 `ort_finalize()` 清理环境。`OMPI` 运行环境的初始化工作分成两个层面，上层是 `ORT` 的初始化，底层是 `EELIB` 的初始化。初始化的目的就是为了给并行执行提供支持，因此线程池也在初始化的时候完成。

10.2.1 ORT 初始化

`ORT` 的初始化在可执行文件的一开始就要调用，否则后面的 `OpenMP` 行为无法执行。在经过 `AST` 变换后新增的 `main()` 函数里面将会调用 `ort_initialize()` 进行初始化并在退出时调用 `ort_finalize()`。

初始化所需要完成的工作内容有：

1. 读取环境变量进行 `ORT` 全局性设置，比如 `ICV` 变量的初始值；
2. 调用 `ee_initialize()` 对 `EELIB` 线程库进行初始化（建立起线程池，暂时为阻塞状态），如果 `EELIB` 能力低于环境变量的设置值则降低 `ORT` 能力；
3. 对初始线程自己的 `EECB` 线程控制块 `ort_master` 进行设置，并将它作为主线程专有的变量（`pthread` 线程专有变量，通过 `eecb_key` 访问）；

函数 `ort_finalize()` 在 `OpenMP` 程序退出前被调用，退出操作则比较简单，由于 `ORT` 层面没有什么需要处理的，它实际上执行的是 `ee_finalize()`，最终被 `ort_priv.h` 的宏定义映射到 `lib/ee_pthreads/othr.c` 文件中的 `othr_finalize()`（实际上是空函数），然后将程序的退出返回码 `exitval` 返回给调用者。`exitval` 是原来的主函数（现为 `__original_main`）的返回值，如果原来的主函数没有返回值那么 `ompi` 程序给出来的调用是 `ort_finalize(0)`。

下面是 `ort_initialize()` 函数的代码。

```
1.  /*
2.   * First function called.
3.   */
4.  int ort_initialize(int *argc, char ***argv)
5.  {
6.      if (ort_initialized) return 0;                                如果已经初始化过了，返回
7.      ort = &ort_globals;                                         ort 指向 ort_globals 结构体，全局变量
                                                               ort_globals，包含 ORT 的全局性的信息。
```

以下开始设置 `ICV` 的初始值。

```
8.      ort->icv.ncpus      = ort_get_num_procs(); /* Default ICV values */
                                                               该函数在 sysdeps.c 文件中实现，获取处理器数
9.      ort->icv.nthreads    = -1; /* No preference for now */
10.     ort->icv.rtschedule = _OMP_STATIC;
11.     ort->icv.rtchunk    = -1;
12.     ort->icv.dynamic    = 1;
13.     ort->icv.nested     = 0;
14.     ort->icv.stacksize  = KBytes(256);
```

获取环境变量并设置 ICV。

```
15.     ort_get_environment();          /* Get environmental variables */  
         获取 OpenMP 环境变量值，并修改 ICVs
```

执行 EELIB 的初始化。

```
16.     /* Initialize the thread library */  
17.     if (ort->icv.nthreads > 0) ort->icv.nthreads--; /* Need 1 less for eelib */  
           此时 ort->icv.nthreads 只可能在 ort_get_environment()中赋值  
18.     if (ee_initialize (argc, argv, &ort->icv, &ort->eecaps) != 0 )  
19.         ort_error(1, "cannot initialize the thread library.\n");  
           如果使用 pthreads 库，EELIB 的初始化 ee_initialize()映射到 othr.c 中的 othr_initialize()
```

自己作为主线程，完成以下初始化工作，包括自己的 EECB 内容填写。

```
20.     if (__MYPID == 0)             /* Everybody in case of threads */  
21.     {  
           __MYPID 在基于线程的系统中=0  
  
22.     /* Check for conformance to user requirements */  
23.     if (ort->icv.nthreads == -1) /* Let the eelib set the default */  
24.         ort->icv.nthreads = ort->eecaps.default_numthreads+1;  
25.     else                         /* user asked explicitly */  
26.     {  
27.         if (ort->eecaps.max_threads_supported > -1 &&  
28.             ort->icv.nthreads < ort->eecaps.max_threads_supported)  
29.             if (!ort->icv.dynamic || !ort->eecaps.supports_dynamic)  
30.                 ort_error(1, "the library cannot support the requested number (%d)"  
           "of threads.\n", ort->icv.nthreads + 1);  
31.         ort->icv.nthreads++;      /* Restore value */  
32.     }  
34.     /* Fix discrepancies */ 对于 ORT 能力的不一致的地方按最弱的配置来设置  
35.     if (ort->icv.dynamic && !ort->eecaps.supports_dynamic) ort->icv.dynamic = 0;  
36.     if (ort->icv.nested && !ort->eecaps.supports_nested) ort->icv.nested = 0;  
  
37.     ort_check_nd_support();    /* Make sure all are ok with the ee lib */  
           检查 nested 特性是否正确
```

下面对 ORT 的两个锁进行初始化。

```
38.     /* Initialize the 2 locks we need */  
39.     ee_init_lock((ee_lock_t *) &ort->atomic_lock, ORT_LOCK_SPIN);  
40.     ee_init_lock((ee_lock_t *) &ort->preparation_lock, ORT_LOCK_NORMAL);
```

下面首先分配 ort_master 所需的 EECB 内存空间，然后对 master 主线程 EECB 进行初始设置。其中 ort_master 是全局变量。

```
41.     /* The "master" */  
42.     ort_master = (ort_eecb_t *) ort_malloc_aligned(sizeof(ort_eecb_t), NULL);  
43.     ort_master->parent          = NULL;          无父节点（树根节点）  
44.     ort_master->sdn            = ort_master;
```

```

45.     ort_master->num_children      = 0;           当前无子节点
46.     ort_master->num_siblings      = 1;           /* We are just 1 thread! */
47.                                         只有一个兄弟（自己）
48.     ort_master->thread_num        = 0;           自己编号为 0
49.     ort_master->level            = 0;           /* The only one in level 0 */
50.                                         在第一层嵌套层次
51.                                         共享数据暂时为空
52.                                         /* VVD */ 暂时无任务分担
53.                                         未创建线程组
54.                                         /* Must* init to NULL */
55.                                         记账信息为空
56.                                         /* In this key we store a pointer to the eecb */
57.                                         ee_key_create(&eecb_key, 0);       eecb_key 用于“线程专有”变量
58.                                         __SETMYCB(ort_master);         此时的 eecb 应该指向 ort_master
59.                                         if (ORT_DEBUG) ort_debug_thread("<this is the master thread>");
60.                                         /* Upon exit .. */
61.                                         atexit(_at_exit_call);
62.                                         异常退出时将执行_at_exit_call 函数（实际上就是 ee_finalize(-1);）
63.                                         }
64.                                         return ( ort_initialized = 1 );
65.                                         }

对于基于进程的实现

```

其中初始的主线程需要给自己建立 EECB 线程控制块。

10.2.2 EELIB 初始化

EELIB 的初始化函数 ee_initialize 由应用程序的新的主函数里面的 ort_initialize()调用的。Ort_initialize()在调用 ee_initialize()时传入全局变量 ort->icv 和 ort->eecaps 以便 EELIB 的初始化函数对它们进行设置。othr.c 文件中的 othr_initialize()通过 ort_prive.h 中宏定义映射为 ee_initialize()。

该函数除了对 ort->icv 和 ort->eecaps 进行设置外还将创建多个线程，这些线程执行 threadjob()函数，然后在 threadjob()里面形成线程池。

```

1.     /* Library initialization
2.     */
3.     int othr_initialize(int *argc, char ***argv, ort_icvs_t *icv, ort_caps_t *caps)
4.     {
5.         int          nthr, i, e;
6.         pthread_t    thr;           线程标识符指针，此处没有实际用途
7.         void        *threadjob(void *);   新建线程首先执行的函数
8.         othr_pool_t *block;        othr 线程结构，它们构建成线程池链表

```

nthr 记录线程数目，如果通过 icv 传入了线程数目则按这个数据设定，否则按照处理器数目的大小来设定。

```
9.         nthr = (icv->nthreads >= 0) ? /* Explicitely requested population */
10.            icv->nthreads :
11.            icv->ncpus - 1; /* we don't handle the initial thread */
```

下面对应于线程库的能力的信息。

```
12.         caps->supports_nested      = 1;      pthreads 支持嵌套
13.         caps->supports_dynamic     = 1;      pthreads 支持动态生成线程
14.         caps->supports_nested_nondynamic = 0;    不支持嵌套+动态
15.         caps->max_levels_supported   = -1;    /* No limit */ 嵌套层次不限
16.         caps->default_numthreads    = nthr;    缺省线程数为 nthr
17.         caps->max_threads_supported = nthr;    /* Can't do more */
                                         最大线程数不超过 nthr

18.         if (pthread_lib_initied) return (0);    如果已经初始化过了，直接返回
                                         (pthread_lib_initied 是全局变量)
19.         if (nthr > 0)
20.         {
21.             #ifdef HAVE_PTHREAD_SETCONCURRENCY
22.                 pthread_setconcurrency (nthr+1);    4通知 pthreads 库所需的并行度
23.             #endif
24.         }
```

创建线程池（线程池结构参见 6.4.3），首先分配空间，共有 nthr 个线程。

```
24.         /* Create the pool */
25.         if ((block = (othr_pool_t *)malloc((nthr)*sizeof(othr_pool_t))) == NULL)
26.             ort_error(5, "othr_init() failed; could not create the thread pool\n");
                                         为 nthr 个线程的线程池分配空间
27.         othr_init_lock(&block, ORT_LOCK_SPIN); 为线程池创建互斥访问自旋锁
```

然后创建线程池内各个线程。

```
28.         for (i = 0; i < nthr; i++)
                                         创建 nthr 个线程，构成线程池
29.         {
30.             block->spin = 1;                此时应该是阻塞的，所以 spin=1
31.             block->next = H;              以下两步，是将新线程从链表
32.             H = block;                  头部插入，H 始终指向链表头
33.             FENCE;
```

⁴在 linux 下，如果忽略了这个函数的使用，那么能够并发的线程数目由实现者来控制，对于系统调度的效率而言往往不是什么好的事情，因为默认的设置往往不是最佳的。更为糟糕的是，如果在某些系统中，如果你不调用 pthread_setconcurrency() 函数，那么系统中的运行的线程仅仅是第一个被创建的线程，其他线程根本不会被运行。比如在 solaris 2.6 中就有这些情况。为了在 unix 或者是 linux 系统上使移植更加的容易，请不要忘记在适当的地方调用此函数，清晰的告诉系统希望使用的线程个数。虽然在某些系统上，这个调用是徒劳的，但是它的使用增强的移植性！请看一下此函数的定义：

int pthread_setconcurrency(int new_level); new_level 一般是当前进程中线程的使用个数，在我们的系统中，往往线程的个数是可以确定下来的，所以一般在初始化函数中显示的设置对应的数值。

创建线程，`thr` 为线程标识符指针，线程标志为 `NULL` 表示系统缺省类型，`threadjob` 是新线程的执行起点，`block` 是线程的运行参数（实际上就是线程池中自己的那一项）。其中 `threadjob()` 函数见后面 10.3.1 分析。

```
34.         if ( (e = pthread_create(&thr, NULL, threadjob, (void *)block)) )
35.             ort_error(5, "pthread_create() failed with %d\n", e);

36.         block++;                                移动到指向线程池的下一项
37.     }
38. }

39.     plen = nthr;                            线程池数量为 nthr
40.     pthread_lib_initied = 1;                设置初始化完成标志
41.     return (0);
42. }
```

当 ORT 的初始化工作完成后，运行环境已创建好了线程池，也已经为主线程建立 EECB。除了主线程外其他线程都在空转（非阻塞）。此时如果主线程执行并行域创建函数，那么将把线程池中空转的函数取下来并发执行相应的任务函数。

10.3 并行支撑函数

并行支撑函数较多，大部分与各个 OpenMP 制导指令紧密相关，但是互相之间的联系却并不紧密，因此分成多个小节来讨论。具体包括线程状态管理、并行域管理、任务分担、同步和变量数据环境 5 个问题。

10.3.1 线程状态管理

线程状态管理是并行域管理的基础。不同的线程状态切换管理将会影响并行域管理的实现和效率。OMPI 基于 `pthreads` 的 EELIB 对线程的状态管理并没有采用线程阻塞的方式，而是采用线程空闲时自旋的办法，减少线程的阻塞和唤醒开销。下面分析基于 `pthreads` 的 EELIB 的空闲状态和工作状态以及它们之间的切换过程。

空闲状态

在进入并行域之前，除了主线程外线程池内的线程都处于空闲状态，因为 EELIB 初始化函数 `ee_initialize()`/`othr_initialize()` 给线程的初始任务是 `threadjob()`。该函数只是空闲自旋，直到外部清除 `spin` 标记，然后调用 `ort_get_thread_work()` 获取任务并执行之，任务完成后将自己放回到线程池中，并且对线程组内正在运行的线程数目减 1。其代码如下：

```
43. /* The function executed by each thread */
44. void *threadjob(void *env)
45. {
46.     void *arg;
47.     othr_info *myinfo;
```

```

48.     othr_pool_t *env_t = (othr_pool_t *)env;

49.     while (1) {                                无限循环
50.         /* Wait for work */
51.         WAIT WHILE(env_t->spin);               空闲则自旋
52.         /* We have to do the following, before the actual execution */
53.         env_t->spin = 1;                         /* Prepare me for next round */
54.         ort_get_thread_work(env_t->id, env_t->arg, NULL, &arg);           第二个参数是组长的 EECB，而 arg 返回的是自己的 EECB
55.         if (ORT_DEBUG) ort_debug_thread("about to call workfunc()");
56.         (*(env_t->workfunc))(arg);             /* Execute requested code */
57.         myinfo = env_t->info;                   /* Moved this up - thanks to M-CC */
58.         othr_set_lock(&plock);
59.         env_t->next = H;                      /* Re-enter the pool */       挂回到线程池
60.         H = env_t;
61.         plen++;
62.         othr_unset_lock(&plock);
63.         FENCE;

64.         /* Update the "running" field of my parent's node */
65.         othr_set_lock(&myinfo->rlock);
66.         myinfo->running--;                  /* Running threads are one less in my team */
67.         othr_unset_lock(&myinfo->rlock);
68.         FENCE;
69.     }

```

如果没有外力改变 spin 标记，那么它将一直在 while(1) 的循环下执行 WAIT WHILE()。, 此时为空闲的自旋状态。

工作状态

当并行域的组长执行 ort_execute_parallel()进而调用 ee_create ()/othr_create ()创建线程组时，将线程任务函数传入。当 ee_create ()/othr_create ()摘取一个线程，并且为它准备好 EECB、任务函数和参数，最终通过解除自旋而激活，该线程将参与到一个并行域的任务中，转换成工作状态。ee_create ()/othr_create ()创建并激活线程的代码如下：

```

1.     void othr_create(int numthr, int level,
2.                       void *(*workfunc)(void*), void *arg, void **info)
3.     {
4.         othr_pool_t *p;
5.         othr_info    *t;
6.         int i;
7.         t = (*info);

```

```

8.     if (t == NULL) /* Thread becomes a parent for the first time */
9.     {
10.         t = (othr_info *) malloc(sizeof(othr_info));
11.         othr_init_lock(&t->rlock, ORT_LOCK_SPIN);
12.     }
13.     t->running = numthr;

```

下面将从线程库中取下 numthr 个线程，并激活它们。

```

14.     /* Wake up "nthr" threads and give them work to do */
15.     for ( i = 1; i <= numthr; i++ )
16.     {
17.         othr_set_lock(&plock);
18.         p = H;                                从池中取得一个线程
19.         H = H->next;
20.         othr_unset_lock(&plock);

21.         p->workfunc = workfunc;            给该线程分配任务函数
22.         p->arg    = arg;                  指向组长的 ECB
23.         p->info   = t;
24.         p->id     = i;
25.         FENCE;
26.         p->spin = 0; /* Release thread i */      解除自旋状态，激活该线程
27.     }
28.     *info = t;
29. }

```

可见激活过程在于将线程池中的线程的自旋标记的解除。此时的被激活的线程仍在 `threadjob()` 函数中，但是跳出了 `WAIT WHILE()` 自旋操作，进而通过 `ort_get_thread_work()` 获得任务并执行 “`(*env_t->workfunc))(arg);`” 语句转到任务函数中去。其中 `ort_get_thread_work()` 还将为本线程建立 ECB 控制块。

10.3.2 并行域管理

在第 6 章已经讨论过并行域管理需要能和多种线程库接口，因此这些功能的实现就有运行库来完成。首先来看看线程无关接口 EELIB 的实现、OpenMP 执行体 EE 的管理。

EECB 接口

EECB 提供了 ORT 线程到具体线程库线程的映射，如图 10.2 所示。

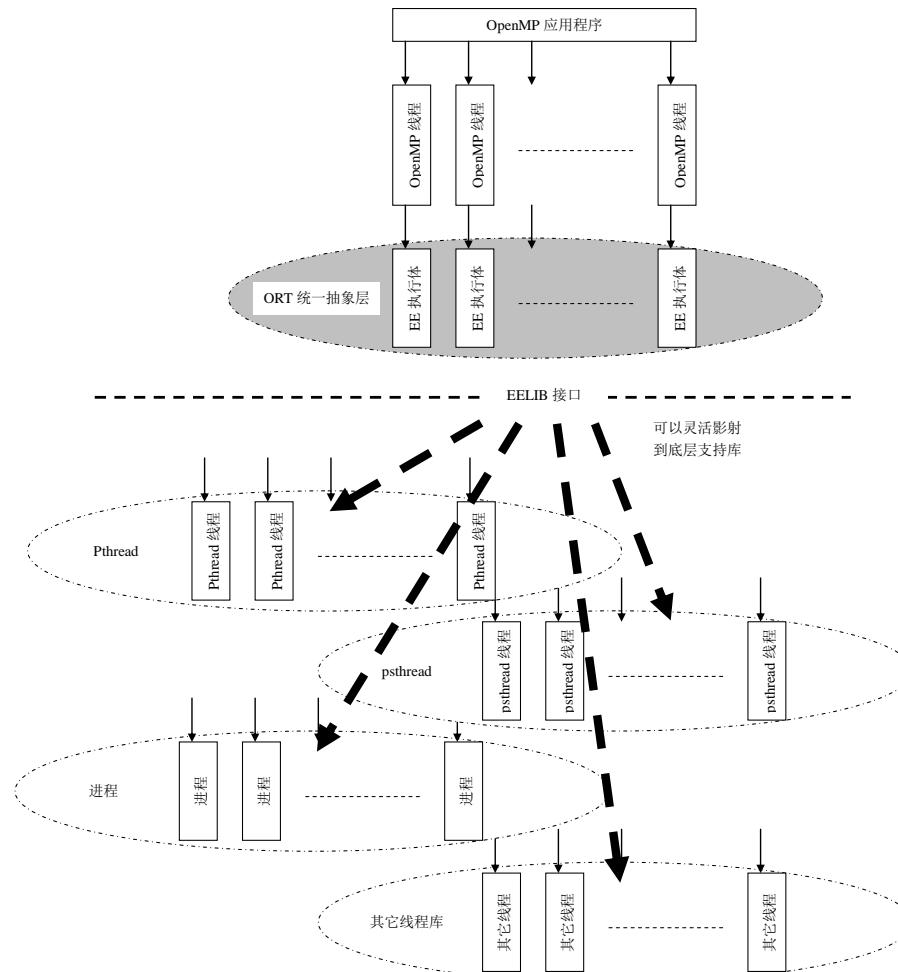


图 10.2 EE 抽象接口

`ort_prive.h` 文件一个很重要的功能就是实现与底层线程库无关的统一的 EELIB 接口，它通过大量的宏定义，将 `ee_xxx` 函数映射到 `othr_xxx` 函数或者 `oprc_xxx` 函数上，从而实现既可以基于线程也可以基于进程来实现执行体（OpenMP 线程），而且由于编译时选用不同的 `lib/ee_yyyy` 目录，可以支持多种线程库和进程库。

并行域产生与撤销

所有 OpenMP/C 源代码中的 `parallel` 制导指令指出的 OpenMP 并行域代码，都被替换成为以下形式的代码：

1. `if(前面第 2 步记录的 if 子句的条件表达式 ifexpr)`
2. `ort_execute_parallel(...);`
3. `else`
4. `ort_execute_serial(...);`

如果条件表达式成立，OpenMP 应用程序进入到并行域会执行 `ort.c` 中的 `ort_execute_parallel()` 函数，它首先通过 `ee_request()`（在 `ort_prive.h` 中的宏定义可以映射到具体的 `lib/ee_pthreads/othr.c` 中的 `othr_request()`）线程池请求一定的执行体/线程，然后准备任务函数及参数并调用 `init_workshare_regions()` 为任务分担作准备，再通过 `ee_create()` 创建多个

执行体线程并发执行指定的任务函数(父节点自己也参与任务函数的执行)。最后用 `ee_waitall()` 等待所有子节点完成任务，从而结束一个并行域的执行。

`ort_execute_parallel()` 是并行域的入口，因此需要给这个函数传入若干相关参数。首先是这个并行域的线程数量，然后是各个线程应该执行的任务（函数），最后是共享变量地址指针。

```
1.   /* This is called upon entry in a parallel region.
2.   *
3.   *   (1) I inquire OTHR for num_threads threads
4.   *   (2) I set up my thrino fields for my children to use
5.   *   (3) I create the team
6.   *   (4) I participate, having acquired a new eecb
7.   *   (5) I wait for my children to finish and resume my old eecb
8.   *
9.   * If num_threads = -1, the team will have ort->icv.nthreads threads.
9.   */
10.  void ort_execute_parallel(int num_threads, void *(*func)(void *), void *shared)
11.  {
12.      ort_eecb_t p, *me = __MYCB;           定义自身的 ECB 变量 me
13.      int          nthr = 0;                线程池提供的执行体数量
14.      /*
15.         * First determine how many threads will be created
16.         */
17.      if (num_threads <= 0)                  /* No num_threads() clause */
18.          num_threads = ort->icv.nthreads;    如果没有指定执行体数量，则用默认值
19.      if (num_threads > 1)                  如果指定了执行体数量，则分为是否嵌套两种情况
20.      {
21.          if (me->level == 0)                /* 1st level of parallelism */
22.          {
23.              nthr = ee_request(num_threads-1, 1);    请求层次为 1 的 num_threads-1 个执行体（父节点
自己也算一个）。
24.              if (nthr != num_threads-1 && !ort->icv.dynamic)
25.              {
26.                  TEAM_FAILURE:
27.                  ort_error(3, "failed to create the requested number (%d) of threads.\n"
28.                             " Try enabling dynamic adjustment using either of:\n"
29.                             " >> OMP_DYNAMIC environmental variable, or\n"
30.                             " >> omp_set_dynamic() call.\n", num_threads);
31.                  给出错误提示
32.              }
33.          }
34.      else                                /* Nested level */
35.      {
36.          if (ort->icv.nested) /* Check if nested parallelism is enabled */
```

```

37.      {
38.          nthr = ee_request(num_threads-1, me->level + 1); 请求 num_threads 个执行体
39.          /* Now here we have an interpretation problem wrt the OpenMP API,
40.             * in the case nthr != num_threads-1.
41.             * Should we breakdown or can we ditch "ort->icv.nested" and create a
42.             * 1 thread team?
43.             * Well, we do a bit of both:
44.             * - if the ee library returned 0, we do the latter with a warning.
45.             * - otherwise, we do the former; it is easier since otherwise
46.                 * we must re-contact the othr library to explain to her that
47.                 * after all we won't need the threads we requested.
48.             */
49.         if (nthr == 0)           线程池没有提供执行体, 给出警告并串行执行
50.             ort_warning("parallelism at level %d disabled due "
51.                         "to lack of threads.\n    >> Using a team of 1 thread.\n",
52.                         me->level + 1); /* GF */
53.         else
54.             if (nthr != num_threads-1 && !ort->icv.dynamic)
55.                 goto TEAM_FAILURE;       数量不够且 ORT 不支持动态, 则出错处理
56.     }
57.     else
58.         nthr = 0;           /* Only me will execute it */
59.         ORT 没有嵌套能力或线程池没有提供执行体, 只用父节点自身来执行任务
60.     }

```

不管是获得线程池的支持还是只有父节点自己来执行任务函数, 都统一往下执行。

```

61.     /*
62.      * Next, initialize everything needed and create the threads
63.      */
64.      me->num_children = nthr + 1;           子节点数量为 nthr+1(nthr 可能为 0)
65.      me->shared_data  = (shared == NULL) ? me->sdn->shared_data : shared;
66.      me->workfunc     = func;            如有共享变量则给出
67.      me->workfunc     = func;            为本线程(父节点 eecb)分配任务函数
68.      me->workfunc     = func;            外部传入, 实际上就是 “_thrFuncXX”
69.      if (nthr != 0)                     如果确实是并发执行的
70.      {
71.          if (!me->have_created_team)      还没有创建子进程组
72.          {
73.              ee_barrier_init(&me->barrier, nthr+1);      为执行 barrier 操作而设定组内线程数目
74.              ee_init_lock(&me->copyprivate.lock, ORT_LOCK_SPIN);
75.              me->have_created_team = 1;          标记为已创建子线程组
76.              me->workshare.blocking.initiated = 0;

```

```

75.      }
76.      init_workshare_regions((ort_workshare_t *)&me->workshare);
    该函数对任务分担区“记账信息”进行初始化

77.      FENCE; /* Just to make sure */

78.      /* Start the threads (except myself - I am the first child, too).
79.         * We do not initialize the info node's fields because it incurs
80.         * quite an overhead (especially if "nthr" is not small.
81.         * Instead we require that ee_create() does it by itself,
82.         * by calling ort_thread_arg_init().
83.         */
84.      ee_create(nthr, me->level, func, me, &me->ee_info);
    创建线程组，此时才真正启动各个执行体（解除空转状态）。其中 ee_info 是
    和线程库相关的，对于 pthreads 库而言是 othr_info 类型的结构体指针。me 是组长 ECB。
85.  }

```

由于自己也做为子线程组的成员，作为 0 号子线程也要执行相关的任务函数。

```

86.  /*
87.   * Participate: I am thread # 0, and I will also call the function
88.   */
89.  p.ee_info = NULL;           /* Sly bug!!!! */0 号子线程不需要 ee_info
90.  __SETMYCB(&p);          /* Change my key */
    此时通过变换 eecb，使得父进程变成自己的 0 号子进程的
91.  ort_get_thread_work(0, me, NULL, NULL);       取得 0 号子线程的任务
92.  if (ORT_DEBUG)
93.      ort_debug_thread("just created a team and about to participate");
94.  (*func)((void *) &p);           执行自己的任务（外部传入的）

```

此时 nthr 个执行体并发执行任务函数，父节点需要等待所有子节点完成任务。

```

95.  /*
96.   * All done; destroy the team (me now points to my "parent")
97.   */
98.  if (nthr > 0)
99.      ee_waitall(&me->ee_info); /* Wait till all children finish */

100. me->num_children = 0;
101. __SETMYCB(me);           /* assume my parent node */恢复原来的控制块
102.
103. }

```

当并行域的 IF 子句说明需要串行执行时，任务函数将由本线程串行执行来完成。它应该在 OpenMP 编译后产生的代码中调用——IF 子句的条件不成立分支。

但是对于并行域的层次管理仍然是相同的，进入并行域后层次增一，退出并行域后层次减一。在嵌套并行域中，执行任务的线程可以不是 master 线程。

如果并行域的 if 子句的条件不成立，那么只能执行 `ort_execute_serial()` 来创建并行域，此时即使并行域内有任务分担域也只能由一个线程来完成里面的任务。

当 `ort_execute_parallel()` 函数退出时，并行域结束，各线程也退出任务函数返回到线程池中去。

EECB 关系维护

在线程从池中取下并激活时，仍未有分配任务也没有建立起 ECB 控制块。直到该线程执行 `ort_get_thread_work()` 才建立起 ECB 并分配任务。并行域组长和成员调用该函数使用的参数略有不同。如果传入了有效的 func 和 arg 指针，则填写为组长 ECB 中的 workfunc 和当前线程的 ECB。

```
1. void ort_get_thread_work(int thrid, void *parent_info,
2.                           void **(***func)(void *), void **arg)
3. {
4.     ort_eccb_t *t = __MYCB, *parent = (ort_eccb_t *) parent_info;
5.
6.     if (t == NULL)          /* 1st time around */    首次使用需要分配 ECB 空间
7.     {
8.         __SETMYCB(t = ort_calloc_aligned(sizeof(ort_eccb_t), NULL));
9.         t->ee_info = NULL;   /* not needed actually due to calloc */
10.    }
11.
12.    t->parent      = parent;                      父结点
13.    t->num_siblings = parent->num_children;       兄弟节点数目
14.    t->level        = parent->level + 1;           /* 1 level deeper */ 嵌套层次
15.    t->thread_num   = thrid;                      /* Thread id within the team */ 组内 ID
16.    t->num_children = 0;                          子节点数目
17.    t->shared_data  = NULL;                       共享数据暂时为空
18.    t->sdb          = parent;                     共享数据所在节点
19.    t->mynextNWregion = 0;                      暂时无任务分担数据
20.    t->nowaitregion = 0;                         /* VVD--actually we don't need to do this */
21. }
```

此时线程分配了 ECB 的内存空间，并填写父结点信息、兄弟节点数目、并行嵌套层次、线程组内 ID 等等信息，共享变量的访问时通过将 `t->sdb` 指向父结点的 ECB 来传递共享变量地址指针的。

10.3.3 任务分担

ORT 中的任务分担支撑函数包括任务分担数据的初始化、任务分担入口函数、任务分担出口函数、`for/sections/single` 的专有任务划分函数几种操作。

初始化准备

每当进入一个并行域时 `ort_execute_parallel()` 将会调用 `init_workshare_regions()` 对组长 EECB 中的 `workshare` 任务分担结构进行初始化，完成并行域执行的相关准备工作（互斥锁的初始化和标记变量的设置）。

由于组长 `eecb` 的工作共享变量 `workshare` 里面包含有带路障的阻塞和不带路障的非阻塞两类情况，所以它们的初始化略有不同。但是无论哪种情况，EECB 的成员 `workshare` 是任务分担域的记账信息，它里面的单个记账信息的关键成员包括所剩的 `section` 任务个数 (`sectionsleft`)，或者 `for` 循环的未完成任务情况 (`forloop`)。下面来看看任务分担数据结构的初始化。

```
1.  /* Called when a new team is created, so as to initialize
2.   * the workshare regions support (in the parent of the team).
3.   * For speed, we only initialize the 1st nowait region;
4.   * the next region is initialized by the first thread to enter the
5.   * previous one.
6.   * It is completely useless if the team has 1 thread.
7.   */
8. void init_workshare_regions(ort_workshare_t *ws)
9. {
10.    if (!ws->blocking.init)           非阻塞的任务分担数据
11.                               如果未初始化，则进行初始化
12.    {
13.        ee_init_lock(&ws->blocking.reglock, ORT_LOCK_SPIN);    创建访问该区的锁
14.        ee_init_lock(&ws->blocking.forloop.ordering.lock, ORT_LOCK_SPIN);    创建访问该区 forloop 的锁
15.        ws->blocking.init = 1;          标记为已完成初始化
16.    }
17.    ws->blocking.empty = 1;         标记为：还没有线程进入
18.    if (!ws->REGION(0).init)        然后是数组中的第 0 项共享区
19.    {
20.        ee_init_lock(&ws->REGION(0).reglock, ORT_LOCK_SPIN);    REGION(X)宏定义为 active[X]
21.        ee_init_lock(&ws->REGION(0).forloop.ordering.lock, ORT_LOCK_SPIN);    创建访问该区 forloop 的锁
22.        ws->REGION(0).init = 1;      标记为已完成初始化
23.    }
24.    ws->REGION(0).empty = 1;       还没有线程进入该域
25.    ws->headregion = ws->tailregion = 0;    没有任务分担
26. }
```

任务分担入口

任务分担入口处主要是为组内线程准备好任务分担数据，无论是 `for`、`sections` 还是 `single` 语句，它们的入口处都是调用的 `entering_workshare_region()` 函数，它们用类型标记 `_OMP_SECTIONS`、`_OMP_FOR` 和 `_OMP_SINGLE` 作为区分。下面是 `sections`、`for` 和 `single` 各自的入口函数。

```
1. /* This is called in the parser-generated code *only* if we are in a
2.  * parallel region AND the number of threads is > 1.
3.  */
4. void ort_entering_sections(int nowait, int numberofsections)      sections 的入口
5. {
6.     enter_workshare_region(__MYCB,_OMP_SECTIONS,nowait,0,numberofsections,0);
7. }

1. /* This is called by the parser-generated code, even if we are
2.  * not in a parallel region.
3.  */
4. void ort_entering_for(int nowait, int hasordered, int lb, int incr,
5.                      ort_gdopt_t *t)      for 的入口。在代码变换时每个 for 任务分担域都
6. 会定义一个变量 t，用于动态和指导调度。
7. {
8.     ort_eecb_t *me = __MYCB;
9.
10.    /* Check if we are not in a parallel region or the team has only 1 thread */
11.    if (me->num_siblings == 1)          只有一个线程可以直接返回
12.    {
13.        t->me = (void *) me;
14.        t->nth = 1;
15.        return;
16.    }
17.    enter_workshare_region(me, _OMP_FOR, nowait, hasordered, lb, incr);
18.    下面代码用于取得任务分担数据结构的内容，分成 nowait 和非 nowait 两种情况
19.    if (me->nowaitregion)
20.    {
21.        /* 1 less since mynextNWregion has been increased upon entrance */
22.        t->data = &(me->parent->workshare.
23.                    REGION(me->mynextNWregion - 1).forloop.lb);
24.        t->lock = (void *) &(me->parent->workshare.
25.                    REGION(me->mynextNWregion - 1).reglock);
26.    }
27.    else
28.    {
29.        t->data = &(me->parent->workshare.blocking.forloop.lb);
30.        t->lock = (void *) &(me->parent->workshare.blocking.reglock);
```

```

29.     }
30.     t->nth = me->num_siblings;
31.     t->me = me;
32. }
33.

1.  /* Returns 1 if the current thread should execute the SINGLE block
2. */
3. int ort_my single(int nowait)           single 的入口
4. {
5.     ort_eecb_t *me = __MYCB;
6.     if (me->num_siblings == 1)          只有一个线程可以直接返回
7.         return (1);
8.     else
9.         return ( enter_workshare_region(me, _OMP_SINGLE, nowait, 0, 0, 0) );
10. }

```

也就是说这三个任务分担最终将会调用 ORT 中相同的入口，见图 10.3 所示。

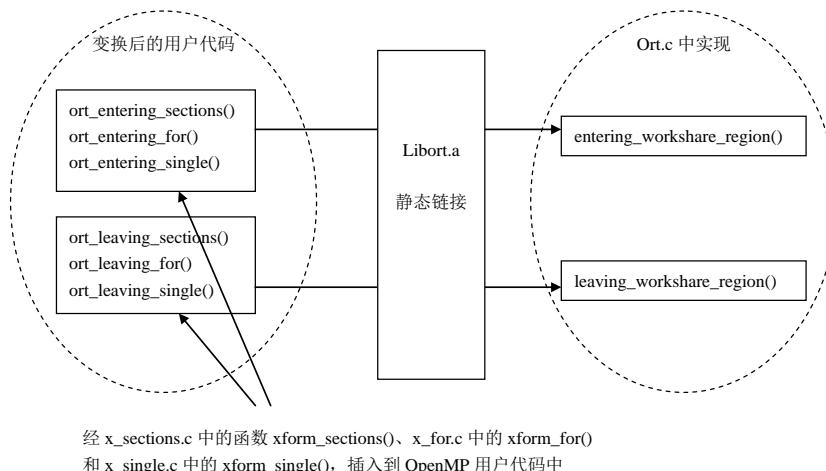


图 10.3 任务分担接口函数

从编译后的代码中插入的函数 `ort_entering_sections()`、`ort_entering_for()` 或 `ort_entering_single()` 里面调用该函数，用于进入到工作分担区域。此时往往有多于 1 个线程在并行执行。第一个进入并行域的线程负责更新相应的计数器数值并返回 1，其他线程则返回 0。但是不论是 `ort_entering_sections()` 还是 `ort_entering_for()` 都不用检查这个返回值，该返回值是给 `ort_my single()` 函数使用的，以判别是否应该执行 `single` 修饰的语句或语句块。

传入的参数中 `wstype` 是任务分担的类型（sections、for 或 single），如果是 sections 类型，那么 `arg1` 是 section 的个数；如果是 for 类型，那么 `arg1` 是 for 循环的 lower bound，而 `arg2` 是循环增量，为将来的 `ort_entering_for()`、`ort_entering_sections()` 和 `ort_engering_single()` 做好“记账”（因不涉及具体任务，所以称为记账）准备。

关于工作分担的细节，还需要在 `for` 语句、`sections` 语句和 `single` 语句的处理函数里面完成。下面是该入口函数的代码。

1. /* Enter a workshare region, update the corresponding counters
2. * and return 1 if i am the first thread to enter the region.

```

3.      * It is guaranteed (by the parser and the runtime routines)
4.      * that we are within a parallel region with > 1 threads.
5.      */
6. static
7. int enter_workshare_region(ort_eecb_t *me,
8.                           int wstype, int nowait, int hasordered,
9.                           int arg1, /* = # sections if wstype is SECTIONS
10.                            = for loop's lower bound if FOR */
11.                           int arg2) /* = the loop's step increment if FOR */
12. {
13.     ort_workshare_t *ws;
14.     wsregion_t      *r;
15.     int             myreg, notfirst = 1;

```

获取自己的工作共享区。如果是带有 nowait 子句，那么取 active，否则取 blocking。REGION 通过宏定义为 active。

```

16.     ws = &(me->parent->workshare);          获取组长 EECB 任务分担域数据指针
17.     myreg = me->mynextNWregion;           mynextNWregion 初始值是 0
18.     r = (nowait) ? &( ws->REGION(myreg) ) : &( ws->blocking ); 根据是否阻塞 r 将指向不同结构

19.     me->nowaitregion = nowait; /* Remember the status of this region */

20.     if (nowait)       对于出现超过一定数量的活动 NOWAIT 的代码区时，阻塞等待
21.     {
22.         /* If too many workshare regions are active, we must block here
23.          * until the tail has advanced a bit.
24.         */
25.         OMPI_WAIT_WHILE(ws->headregion - ws->tailregion == MAXACTIVEREGIONS &&
26.                           myreg == ws->headregion, YIELD_FREQUENTLY); 等待
27.         (me->mynextNWregion)++; /* Make me ready for the next region */
28.     }

29.     if (!r->empty) return (0);           /* We are not the first for sure */
30.                               如果有其他线程进来处理过，则直接返回 0

```

如果还没有线程处理过这个任务分担域，那么根据类型对 for 或 sections 做好准备。因为有可能由多个线程成功检测到前面的 (`!r->empty`) 条件造成并发执行下面的代码，需要先对该结构进行加锁。

```

31.     ee_set_lock(&r->reglock);        /* We'll be short */
32.     if (r->empty)                  /* I am the first to enter */
33.     {
34.         if (wstype == _OMP_SECTIONS) /* Initialize */      对 sections 的处理
35.             r->sectionsleft = arg1;    arg1 是 section 个数的初始值
36.         else

```

```

37.         if (wstype == _OMP_FOR)                      对 for 的处理
38.             /* first_to_arrive_at_for(&r->forloop, hasordered,arg1,arg2); */
39.             {
40.                 r->forloop.lb = arg1;                  确定下界
41.                 if (hasordered)                     确定 ordered 执行顺序
42.                 {
43.                     r->forloop.ordering.next_iteration = arg1;
44.                     r->forloop.ordering.incr = arg2;
45.                 }
46.             }

47.         r->notleft = me->num_siblings;      /* None left yet */全部线程都未离开

```

下面开始对下一区域进行初始化准备（如果还没有初始化）。这样做可以在线程组创建后并行地完成共享区的初始化工作。

```

48.         /* Now, prepare/initialize the next region, if not already initied.
49.          * This is done so as to avoid serially initializing all regions when
50.          * the team was created
51.          */
52.         if (nowait && myreg+1 < MAXACTIVEREGIONS)
53.         {
54.             if (!ws->REGION(myreg+1).initied)           如果未初始化
55.             {
56.                 ee_init_lock(&ws->REGION(myreg+1).reglock, ORT_LOCK_SPIN);
57.                 ee_init_lock(&ws->REGION(myreg+1).forloop.ordering.lock, ORT_LOCK_SPIN);
58.                 ws->REGION(myreg+1).initied = 1;
59.             }
60.             ws->REGION(myreg+1).empty = 1;
61.         }

62.         /* Do this last to avoid races with threads that test without locking */
63.         if (nowait) (ws->headregion)++;      /* Advance the head */    可用结构数量减少 1 个
64.         FENCE;
65.         r->empty = notfirst = 0;
66.     }
67.     ee_unset_lock(&r->reglock);

68.     return ( !notfirst );
69. }

```

退出任务分担

退出任务分担域时执行的 `ort_leaving_sections()` 或 `ort_leaving_for()` 里面将会进一步调用 `leave_workshare_region()`，它的返回值是仍然留在任务分担域内的线程数量，如果返回值为 0 说明是最后一个结束的线程，全部组内线程都已执行完毕，具体代码如下：

```
1.     /* Returns the # of threads that have not yet left the region.
```

```

2.      * A zero means that I am the last thread to leave.
3.      */
4.      static
5.      int leave_workshare_region(ort_eecb_t *me)
6.      {
7.          ort_workshare_t *ws;
8.          wsregion_t      *r;
9.          int             myreg, remain;

10.         ws = &(me->parent->workshare);           取得当前任务分担结构
11.         myreg = me->mynextNWregion - 1;    /* It is ahead by 1 */
12.         r = (me->nowaitregion) ? &( ws->REGION(myreg) ) : &( ws->blocking );

13.         /* Well, it would be better if we used a seperate lock for leaving */
14.         ee_set_lock(&r->reglock);
15.         remain = --(r->notleft);           /* Increment the # threads that left */
16.         if (!remain)                      本线程要离开, 计数值减 1
17.         {                                /* If I am the last to leave */
18.             r->empty = 1;                /* The region is now empty */
19.             if (me->nowaitregion)        对于 NOWAIT 任务分担域需处理头尾指针
20.                 (ws->tailregion)++;   /* Advance the tail */ 可用的结构数量增加 1 个
21.         }
22.         ee_unset_lock(&r->reglock);

23.         return (remain);
24.     }

```

主要工作就是对任务分担数据进行清理, 将 `empty` 设置为 1 表明没有线程留在任务分担域中, 如果是 `nowait` 的任务分担则将 `ws-tailregion` 指针往前移一步指向结尾处。

for 任务划分

对于 `for` 制导指令, 每个线程在执行 `ort_entering_for()` 进入 `enter_workshare_region()` 设置好任务分担数据结构后, 就要进行任务划分。根据调度类型的不同, 可能是 `ort_get_static_default_chunk()`、`ort_init_static_chunksize()`、`ort_get_dynamic_chunk()`、`ort_get_guided_chunk()` 函数之一, 它们分别对应于缺省调度、静态调度、动态调度和指导调度, `runtime` 调度最终也是映射到上面的四种调度之一。这些函数都将根据 7.1.5 小结介绍的算法进行循环变量的划分, 返回被调用者 `form_` 和 `to_` 两个参数用于界定调用者线程当前循环变量的起止范围。在执行这些任务划分函数之前的 `ort_entering_for()` 中已经获得了当前的任务分担数据结构 (对于静态调度而言是不用担心 `nowait` 问题的, 因为它们的任务划分不依靠于其他线程)。

对于动态和指导调度而言, 需要用到 `ort_gdopt_t` 类型的结构体, 用于辅助带有 `nowait` 子句时的任务划分:

```
1.  typedef struct _ort_gdopt_
```

```

2.      {
3.          volatile int *data; /* Denotes the current lb of the loop */ 指向当前任务分担数据结构
    的循环下界
4.          volatile void *lock; /* Lock to access *data */ 指向当前任务分担数据结构的互斥锁
5.          int         nth;   /* # siblings */
6.          void        *me;    /* my info node */
7.      } ort_gdopt_t;

```

该数据结构的变量在变换后的 for 代码中出现并命名为 gdopt_，并在调用 ort_entering_for() 时传入，然后再 ort_entering_for() 中记录了当前线程组长的 workshare 变量。而后可以调用任务划分函数，如果是动态调度调用的是 ort_get_dynamic_chunk():

```

1.  int ort_get_dynamic_chunk(int lb, int ub, int step, int chunksize,
2.                             int *from, int *to, int *ignored, ort_gdopt_t *t)
3.  {
4.      int newlb;
5.
6.      if (chunksize <= 0) ort_error(1, "fatal: dynamic chunksize <= 0 requested!\n");
7.
8.      if (((ort_eecb_t *) t->me)->num_siblings == 1)
9.      {
10.         /* Here we are based on the fact that the parser-generated code, checks
11.            * for < or > to, not <= or >=.
12.         */
13.         if (*from == lb && *to == ub) /* We did our sole chunk */
14.             return (0);
15.         *from = lb;                  /* Get just 1 chunk: all iterations */
16.         *to   = ub;
17.         return (1);
18.     }
19.
20.     ee_set_lock((ee_lock_t *) t->lock);
21.     newlb = *(t->data);           当前循环变量下界移动 step*chunksize
22.     (*t->data) += step*chunksize;
23.     ee_unset_lock((ee_lock_t *) t->lock);
24.
25.     if (step >= 0)               对于 step>0 的情况
26.     {
27.         if (newlb < ub) {
28.             *from = newlb;           本线程执行的循环变量起点
29.             *to = newlb + chunksize*step; 本线程执行的循环变量终点
30.             if (*to > ub)
31.                 *to = ub;
32.             return 1;
33.         }
34.         else return 0;
35.     }
36.     else {                       对于 step<0 的情况
37.         if (newlb > ub) {
38.             *from = ub;
39.             *to = newlb - chunksize*step;
40.             if (*to < ub)
41.                 *to = ub;
42.             return 1;
43.         }
44.         else return 0;
45.     }
46. }

```

```

32.     if (newlb > ub) {
33.         *from = newlb;                                本线程执行的循环变量起点
34.         *to = newlb + chunksize*step;                本线程执行的循环变量终点
35.         if (*to < ub)
36.             *to = ub;
37.         return 1;
38.     }
39.     else return 0;
40. }
41. }
```

由上面代码可知，对于动态调度机制下，`for` 循环变量的范围需要从任务共享数据结构中计算得到。此处并不在乎有无 `nowait` 子句问题，因为前面的 `ort_entering_for()` 已经选择了正确的任务分担数据。

section 任务划分

对于 `sections` 的任务划分是通过 `ort_get_section()` 函数来完成的，该函数代码如下：

```

1.  int ort_get_section()
2.  {
3.      wsregion_t *r;
4.      int      s;
5.      ort_eecb_t *me = __MYCB;
6.                                              下面代码用于找到自己的任务分担数据结构，由 r 指针指出
7.      s = me->mynextNWregion - 1; /* My region ('cause it is 1 ahead) */
8.      r = (me->nowaitregion) ?
9.          &(me->parent->workshare.REGION(s)) :
10.         &(me->parent->workshare.blocking);

11.     if (r->sectionsleft < 0) return (-1);           如果没有剩余未完成的 section，直接返回
12.     ee_set_lock(&r->reglock);
13.     s = -(r->sectionsleft);                      找到一个未完成的 section 的编号
14.     ee_unset_lock(&r->reglock);
15.     return (s);                                    返回即将要执行的 section 编号
}
```

首先当前线程要获取任务分担数据结构，然后判断是否有剩余 `section` 未完成，如果有的话则返回一个 `section` 编号用于后续的执行。

single 任务划分

`single` 语句的任务分担很简单，调用 `ort_mysingle()` 函数判断一下自己的线程号是否为 0 以决定是否执行相应的任务，代码如下：

```

1.  int ort_mysingle(int nowait)
2.  {
3.      ort_eecb_t *me = __MYCB;
4.      if (me->num_siblings == 1)
5.          return (1);
```

```

6.     else
7.         return ( enter_workshare_region(me, _OMP_SINGLE, nowait, 0, 0, 0) );
8. }

```

10.3.4 同步

ORT 库函数对同步的支持比较简单，只是提供相应的锁机制和其他一些辅助信息。

对于 atomic 而言，将在 `ort_atomic_begin/end()` 函数中对运行环境的 `ort->atomic_lock` 进行加锁和解锁操作，即可实现原子操作。

`critical` 制导指令将调用 `ort_critical_begin/end()` 函数，通过对一个类型为 `ORT_LOCK_SPIN` 的自旋锁的加锁和解锁操作来实现。OMPI 将该锁声明在应用程序中而不是在 ORT 运行环境中，在编译后的源代码中被命名为 `_ompi_crity` 的变量。

`ordered` 制导指令需要调用 ORT 运行环境的 `ort_ordered_begin/end()` 函数，在 `ort_ordered_begin()` 处如果发现前面的循环还没有完成而需要等待时，可以通过 `sched_yield()` 让出 CPU 将自己挂到就绪队列的末尾。如果不希望因阻塞和调度增加的开销，可以在编译时有 `NO_SCHED_YIELD` 宏定义，则反复测试而不让出 CPU。

`barrier` 制导指令调用的是 `ort_barrier_me()` 函数，该函数只是保证线程组所有成员到达该点之前不能越过该函数。这个操作可以不需要用锁，ORT 利用下面的结构体来统计到达该点的线程数目：

```

1.     typedef struct
2.     {
3.         aligned_int arrived[MAX_BAR_THREADS];
4.         int sense, nthr;
5.     } ort_defbar_t;

```

线程组的组长的 EECB 里面的 `barrier` 就是这个结构体，线程组长在执行 `ort_execute_parallel()` 里设置好组内线程数目 `barrier->nthr`, `arrived[]` 用于记录到达该点的线程，`sense` 表示组内线程是否可以通过该点的标志。在执行 `barrier` 的时候，线程组长逐个检查 `arrived[]` 数组确定所有子线程都已到达，然后将改变 `sense` 标记让所有线程通过该点。EELIB 也可以自己实现 `barrier` 操作，这个可以通过在 `ee.h` 里面添加 `AVOID_OMPI_DEFAULT_BARRIER` 宏定义来实现。

`master` 制导指令严格来说不是同步操作，它只是简单地判断自己是否为 0 号线程以决定是否执行相关代码。ORT 只需要提供 OpenMP API 函数 `omp_get_thread_num()` 即可以支持此行为。

另外 `reduction` 制导指令实际上也包含有同步关系，在执行归约操作之前调用的 `ort_reduction_begin()` 实际上是对 `&_paredlockXX` (`XX` 是数字编号) 的加锁操作，该锁在编译后代码中定义并且逐个编号。

10.3.5 变量的数据环境

ORT 运行环境对数据环境的支持也比较简单，因为大多数工作在编译时的代码变换中已经完成，没有多少工作留给运行环境去做。各种共享变量和私有变量的声明、初始化等赋值操作在编译后的源代码中已经准备好，OMPi 留给 ORT 的主要是两个功能函数：`ort_get_shared_vars()` 和 `ort_get_thrpriv()`。

第一个函数非常简单，以至于它可以通过代码变换直接插入到源代码中：

```
1. void *ort_get_shared_vars(void *me)
2. {
3.     return ((ort_eecb_t *) me)->sdn->shared_data;
4. }
```

此处 `me->sdn` 指向父结点的 EECB，父结点 EECB 中的 `shared_data` 保存了共享变量地址指针列表，具体参见 10.1.4 的图 10.1。

第二个函数主要用于获取线程独有数据，其实主要就是各个线程访问自己的 `threadprivate` 变量时使用的。它的也不复杂，其代码如下：

```
1. void *ort_get_thrpriv(void **key, int size, void *origvar)
2. {
3.     void *var;
4.
5.     if (*key == NULL) /* Uninitialized key */
6.     {
7.         /* Non-initialized key; be careful */
8.         /* we use the ort->preparation_lock, so as not to define 1 more lock */
9.         ee_set_lock((ee_lock_t *) &ort->preparation_lock);
10.
11.        if (*key == NULL)
12.        {
13.            ee_key_create((ee_key_t *) key, 0);
14.            FENCE;
15.        }
16.
17.        if ((var = ee_getspecific((ee_key_t *) key)) == NULL)    实际上调用的 pthreads 的对应函数
18.        { /* 1st reference for this thread */
19.            if (__MYCB == ort_master)      /* Master thread has the origvar */
20.                var = origvar;
21.
22.            if ((var = (void *) malloc(size)) == NULL)
23.                ort_error(1, "[ort_get_thrpriv]: out of memory\n");
24.            ee_setspecific((ee_key_t *) key, var);
25.
26.        }
27.    }
28.    return (var);
29. }
```

其关键是利用 `key` 获取 `pthreads` 库中的线程独有数据，如果还没有创建的话先创建该变量并确定 `key` 值。

10.4 OpenMP 的 API

无论哪种 OpenMP 编译器的实现，都必须给可执行文件提供 OpenMP 的 API 函数，这些函数的功能和使用情况在已经第 2 章 OpenMP 编程基础里面讨论过，参见表 2.1。为了提供这些 API，使得应用程序的编写中能够引用这些函数，需要做到以下几点：

- 1) 提供 `omp.h` 头文件，以供 OpenMP/C 源代码编写时使用；
- 2) 提供实现了 OpenMP API 的库；
- 3) 编译可执行程序时可以访问到这个 API 库。

10.4.1 API 函数

头文件 `omp.h` 声明了用户可以使用的 OpenMP API 函数，其形式应该是比较统一的，如果严格按照 OpenMP 规范来实现的 OpenMP 编译器，所有的 OpenMP 编译器其实可以共用一个 `omp.h` 头文件。这个头文件的内容就是对 API 函数以及一些数据结构类型的声明，但是许多机构自己提供了 OpenMP 的一些增强版本，因此它们的编译器自带 `omp.h` 头文件。下面给出 OMPI 的 `omp.h` 头文件内容，它与 OpenMP v2.5 规范中推荐的头文件内容很相似，GCC 的 `omp.h` 头文件也很几乎一样。

```
1. /* execution environment functions */  
2. int omp_in_parallel(void);  
3. int omp_get_thread_num(void);  
4. void omp_set_num_threads(int num_threads);  
5. int omp_get_num_threads(void);  
6. int omp_get_max_threads(void);  
7. int omp_get_num_procs(void);  
8. void omp_set_dynamic(int dynamic_threads);  
9. int omp_get_dynamic(void);  
10. void omp_set_nested(int nested);  
11. int omp_get_nested(void);  
  
12. /* lock functions */  
13. typedef void * omp_lock_t;  
  
14. void omp_init_lock(omp_lock_t *lock);  
15. void omp_destroy_lock(omp_lock_t *lock);  
16. void omp_set_lock(omp_lock_t *lock);  
17. void omp_unset_lock(omp_lock_t *lock);  
18. int omp_test_lock(omp_lock_t *lock);  
  
19. /* nestable lock fuctions */  
20. typedef void * omp_nest_lock_t;
```

```

21. void omp_init_nest_lock(omp_nest_lock_t *lock);
22. void omp_destroy_nest_lock(omp_nest_lock_t *lock);
23. void omp_set_nest_lock(omp_nest_lock_t *lock);
24. void omp_unset_nest_lock(omp_nest_lock_t *lock);
25. int  omp_test_nest_lock(omp_nest_lock_t *lock);

26. /* timing routines */
27. double omp_get_wtime(void);
28. double omp_get_wtick(void);

```

这个头文件是符合 OpenMP 2.5 版本的 `omp.h`。前 10 个函数正是 OpenMP v2.5 标准中规定的运行环境函数（Runtime Library Routines）。接着对于普通锁和嵌套锁各有 5 个符合 OpenMP 2.5 规范的锁函数（Lock Routines），同时还有各自的锁类型定义。最后是两个时间函数（Timing Routines）。图 6.7 将这些 API 作简单的分类。

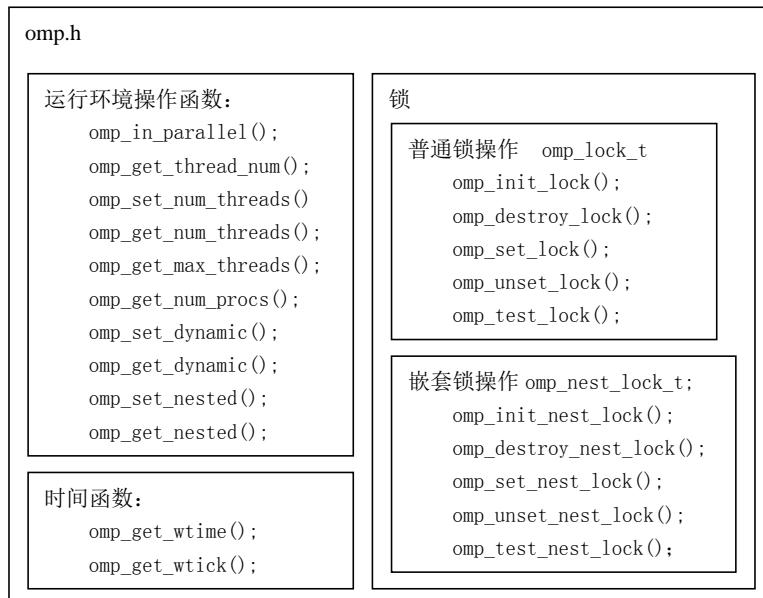


图 10.4 `omi.h` 函数分类关系

它们的实现比较简单，除了锁函数以外，基本上是对某个变量的读取或设置行为。OMPI 的部分相关代码实现如下：

```

1. int  omp_in_parallel(void)      { return ( __MYCB != ort_master ); }
    判断是否在并行域中是通过检查自己的 EECB 是否为 ort_master 来实现的。
2. int  omp_get_thread_num(void)   { return ( __MYCB->thread_num ); }
    返回 EECB 中记录的线程编号
3. int  omp_get_num_threads(void) { return ( __MYCB->num_siblings ); }
    返回 EECB 中记录的线程数量
4. int  omp_get_max_threads(void) { return ( ort->icv.nthreads ); }
    返回 ICV 中设定的最大线程数目
5. int  omp_get_num_procs(void)   { return ( ort->icv.ncpus ); }
    返回 ICV 中记录的处理器个数
6. void omp_set_dynamic(int dyn)  { if (ort->eecaps.supports_dynamic)
                                    ort->icv.dynamic = dyn;
                                    ort_check_nd_support(); }

```

```

设置 ICV 中记录是否能动态调整线程数目的标志
9. int omp_get_dynamic(void) { return (ort->icv.dynamic); }

.....
10. void omp_init_lock(omp_lock_t *lock)
11. { *lock = NULL; ort_prepare_omp_lock(lock, ORT_LOCK_NORMAL); }
    按指定类型初始化一个锁
12. void omp_set_lock(omp_lock_t *lock)
13. { ee_set_lock((ee_lock_t *) *lock); }
    对指定的锁执行加锁操作
14. void omp_unset_lock(omp_lock_t *lock)
15. { ee_unset_lock((ee_lock_t *) *lock); }
    对指定的锁执行解锁操作
16. int omp_test_lock(omp_lock_t *lock)
17. { return (ee_test_lock((ee_lock_t *) *lock)); }
    对指定的锁执行测试操作
.....
18. }

```

可以看出 OpenMP API 函数的实现是比较简单的。

10.4.2 ICV 变量

在 OpenMP 的规范中有关于内部控制变量（Internal Control Variables）的说明，这些变量虽然出现在规范中，但实际上如何实现它们是由编译器自己决定的。在编译器中看起来像是有这么一些变量，可以控制并行域内的线程数目、任务分担中的调度策略等等。OpenMP 2.5 规范中 ICV 及其作用如表 10.1 所示。

表 10.1 ICV 变量

类型	名称	作用
并行域控制	nthreads-var	并行域线程数
	dyn-var	是否允许动态调整线程数
	nest-var	是否允许嵌套
调度控制	run-sched-var	保存 runtime 调度时的调度信息
	def-sched-var	缺省调度

规范中使用的 ICV 变量的名称仅仅是为了便于表示，并不强制编译器使用相同的变量名。编译器在实现这些变量的时候可以用整型、布尔型或其他数据类型，甚至可以将这些变量形成一个结构体来实现，也可以用整数的各个位来表示。对应这些变量，有一些操作控制的 API 函数，它们的关系如表 10.2。

表 10.2 ICV 变量的操作函数

变量	设置	读取	初始值
nthreads-var	omp_set_num_threads()	omp_get_max_threads()	未定义
dyn-var	omp_set_dynamic()	omp_get_dynamic()	未定义
nest-var	omp_set_nested()	omp_get_nested()	false
run-sched-var	无	无	未定义
def-sched-var	无	无	未定义

10.4.3 引用与链接

编译系统必须提供上述 API 的库，GCC 的 OpenMP 库位于类似这样的目录下：`/usr/lib/gcc/x86_64-redhat-linux/4.4.4/`，动态库命名为 `libgomp.so`，静态库命名为 `libgomp.a`。而 OMPI 的 OpenMP 库安装后保存在类似这样的目录中：`/usr/local/lib/ompi/default/`，命名为 `libort.a`。

源程序编写时通过包含 `omp.h` 头文件，就可以对相应的 API 进行引用。在编译时需要指出相应的 OpenMP 运行库的位置和名称，如果是 GCC 那么只需要在命令行加入 `-fopenmp` 选项即可，而使用 OMPI 的库则需要用类似这样的选项：“`-L/usr/local/lib/ompi/default`” 和 “`-lort`” 指出库的位置和库名。

10.5 环境变量

在第 2 章 OpenMP 编程基础中已经讨论过 OpenMP 环境变量及其作用，这些环境变量与编译过程没有关系，但是和运行环境有关。用户通过环境变量能够控制程序运行的行为，因此 OpenMP 的运行库需要具备根据这些环境变量的设置而作调整的能力。

OpenMP v2.5 只定义了 4 个环境变量，它们是 `OMP_SCHEDULE`、`OMP_NUM_THREADS`、`OMP_DYNAMIC` 和 `OMP_NESTED`。

这些环境变量由操作系统负责维护，ORT 只需要在初始化时读入、判断并作相应的设置操作。在初始化时调用了 `ort_get_environment()`，该函数最终调用的是 linux 系统函数 `getenv()` 来读入环境变量的。

```
1. void ort_get_environment()
2. {
3.     char *s, *t;
4.     int n;
5.
6.     if ((s = getenv("OMP_NUM_THREADS")) != NULL)           获取 OMP_NUM_THREAD 环境变量
7.         if (sscanf(s, "%d", &n) == 1 && n > 0)
8.             ort->icv.nthreads = n;                           设置相应的 ICV 变量
9.
10.    if ((s = getenv("OMP_DYNAMIC")) != NULL)            获取 OMP_DYNAMIC 环境变量
11. ...
12.
13.    if ((s = getenv("OMP_NESTED")) != NULL)           获取 OMP_NESTED 环境变量
14. ...
15.
16.    if ((s = getenv("OMP_STACKSIZE")) != NULL)          获取 OMP_STACKSIZE 环境变量
17.        if (sscanf(s, "%d", &n) == 1 && n > 0)
18.            ort->icv.stacksize = n;
19. }
```

读入环境变量后，按其取值情况设置运行环境中相应的 `ort->icv.xxxx` 变量，从而影响应用程序的运行行为。

10.6 小结

本章讨论了 **OMPI** 编译系统上的 OpenMP 运行环境，内容包括 **ORT** 运行库、OpenMP API 和环境变量三方面的内容。

对于 **ORT** 运行环境的分析是从关键数据结构开始的，讲述了描述运行环境的结构体、线程池和线程控制块 **EECB**、任务分担结构体和共享变量结构体。然后通过 **ORT** 初始化过程讲述 **OMPI** 基于线程池和 **EECB** 的并行线程环境的建立。然后分成 5 个专题分别讲述所提供的支撑功能：线程状态转换管理、并行域管理、任务分担支撑功能、线程同步支持功能以及变量共享属性支持功能。

OpenMP 的 API 实现是比较简单和直接的，主要是设置和返回相应的 **ICV** 内部控制变量。**OpenMP** 环境变量的支持也比较简单，通过系统的环境变量值来修改 **ICV** 内部控制变量。API 及外部环境变量对运行环境的控制和交互主要是通过 **ICV** 变量来实现的。

第三篇 实践篇

第三篇是实践部分。首先介绍了三种 OpenMP 编译器的基本情况、OpenMP 编译器性能测评工具。然后是关于 OMPI 编译器的框架流程的分析，主要是整体框架和代码变换部分的基本步骤分析。因此读者可以根据这部分内容，对 OMPI 这个小型的编译器软件进行修改和增强，并能使用相应的测评工具对所做的修改和增强进行分析。

第 11 章 编译器及测试工具

本章将介绍几种开源的 OpenMP 编译系统，读者可以下载代码进行阅读分析，以了解其运行机理，甚至可以进行增强与修改以优化其性能。然后介绍几款 OpenMP 的性能测试工具，可以对读者设计和优化后的编译器进行量化的性能分析，以验证新机制或算法的有效性。其中对 Microbenchmark 给出了测试原理的简介，读者可以自行扩展其测试能力（比如 task 的时间开销）。

11.1 常见 OpenMP 编译器

本章将简要介绍三种常用的支持 OpenMP 的编译器/系统，它们分别是：GCC，Ompi 和 Omni。通过对它们下载安装和使用，使我们能快速的了解 OpenMP 源代码的编译流程。

11.1.2 OMPi 编译器

OMPi 简介

OMPi 是一个轻量级的，开源的 C 语言 OpenMP 编译器和运行时系统，当前最新版本 1.2.0 兼容 OpenMP 3.0 标准。它是由 Ioannina 大学的并行处理小组开发的一个项目。OMPi 编译器将带有 OpenMP #pragma omp 标记的 C 语言代码转换成多线程的 C 语言代码，然后用系统的本地编译器生成可执行文件。因此 OMPi 编译器是一个源代码到源代码的编译器，将带有 OpenMP 编译制导指令的 C 源文件编译成标准的 C 语言代码。

编译输出的可执行文件可以支持多种线程库，也可以基于进程来并发执行。OMPi 编译器的一个特色是对并行嵌套的良好支持。OMPi 编译器还可以利用各种 SVM (Shared Virtual Memory) 库作为支撑运行于 Cluster 系统上，SVM 的另一个更常见的名字是 sDSM (software Distributed Shared Memory)。

OMPi 1.0.0 版本支持整个 OpenMP 2.5 标准，1.1.0 版本支持 OpenMP 3.0 版本（除了“collapse”子句），并且重写了运行时系统，在写成本书时 OMPi 的最新版本是 1.2.0，支持整个 OpenMP 3.0 版本，增强了运行时系统，修复了一些 bug 并且删除了一些线程库。更多信息可以查看 OMPi 官方网站：<http://www.cs.uoi.gr/~ompi/>

OMPi 安装

要使用 OMPi 编译器来编译带有 OpenMP 制导指令的源文件，我们必须会正确的安装 OMPi 编译器，下面我们一起来学习 OMPi 编译器的安装。

- 首先在其官方网站上找到下载地址：<http://www.cs.uoi.gr/~ompi/download.html>，点击下载 `ompi-1.2.0.tar.gz`（本书主要分析 1.0.0 源码）。
- 下载完成后，使用命令：

```
# tar -zxf ompi-1.2.0.tar.gz
```

解压文件后会在当前目录下生成一个 `ompi-1.2.0` 的文件夹。

- 然后使用命令：

```
# cd ompi-1.2.0
```

- 进入该文件夹后，需要使用命令：

```
# ./configure --prefix=<安装目录>
```

进行配置（如果没有指定安装目录就默认安装到 `/usr/local/bin` 中）。

- 如果配置没有出错就可以编译安装 OMPI 编译器了，使用命令：

```
# make |make install
```

编译安装 OMPI 编译器。或者先使用命令：

```
# make
```

编译源文件，然后使用命令：

```
# make install
```

来安装 OMPI 编译器。

- 最后我们还需要检查一下环境变量设置是否正确。使用命令：

```
# echo $PATH |grep <安装目录>
```

来查看环境变量中是否有 OMPI 编译器的安装目录。如果存在的话就表明你的 OMPI 编译器安装完成了。

下面介绍在配置环境 `configure` 时候的一些可选参数：

- 如果你希望能调试，则需要在配置环境 `./configure` 后面加上 `--enable-debug` 选项，即是：

```
# ./configure --enable-debug ...
```

- OMPI 编译器在编译时默认仅仅使用 `-O3` 优化选项，如果你想更改默认的设置，则在配置环境时使用命令：

```
# ./configure CFLAGS=<your_flags> CPPFLAGS=<preprocessor_flags> ...
```

- 在 `configure` 的时候 OMPI 编译器就指定了默认的编译器（一般是 `GCC`），如果你想使用另外一个编译器（比如说 Intel 的 `icc` 编译器）的话可以使用命令：

```
# ./configure CC=icc ...
```

- OMPI 提供了四个线程库，分别是：`pthreads`（默认，基于 `POSIX` 线程）、`pthreads1`、`solthr` 和 `solthr1`，并且还支持更多。你可以这样选择你所想要的线程库使用：

```
# ./configure --with-ortlib=<name> ...
```

OMPI 使用

首先让我们看看 OMPI 的五个环境变量：`OMPI_CPP`，`OMPI_CC`，`OMPI_CPPFLAGS`，`OMPI_CFLAGS`，`OMPI_LDFLAGS`。前两个环境变量是用来设置系统的预处理和编译器的参数，后三个用来指定特殊的一些参数。

接下来看看 OMPI 编译器可以使用的一些参数：

GCC 编译器所有的参数，OMPI 编译器都可以使用。

OMPI 编译器还可以使用的一些参数：

-k

可以保留源代码编译过程中的中间结果。

-v

可以查看编译过程的所有步骤。

--ort=<libname>

可以指定使用的运行时库。

最后我们通过上一小节中的 helloworld 的例子来具体看看 OMPI 编译器的使用：

使用命令：# *ompicc -o ompihelloworld helloworld.c*

编译 helloworld.c 文件，指定输出文件名为：ompihelloworld，然后执行这个文件，结果如下：

```
Hello World! Thread number is 0  
Hello World! Thread number is 1  
Hello World! Thread number is 3  
Hello World! Thread number is 6  
Hello World! Thread number is 7  
Hello World! Thread number is 5  
Hello World! Thread number is 4  
Hello World! Thread number is 2
```

11.3 小结

本章内容分成两个部分。首先是 3 种常用的开源 OpenMP 编译系统：GCC、OMPI 和 Omni。分别给出了获取源代码的下载方式、配置、编译安装以及基本使用方法，各自都有详细的操作命令和说明。然后介绍了几种性能测试工具，主要是测量 OpenMP 各种制导指令的时间开销的 EPCC microbenchmark 和测量核心程序计算时间的 NSA Parallel Benchemark，结合这两个测试可以大体上反映出一个系统 OpenMP 相关的性能。对这两种工具的下载、配置、编译安装以及使用方法都有比较消息的介绍，特别是 EPCC mircobenchmark 还给出了测量的基本原理，用于读者根据需要编写其他感兴趣参数的测量。

以上的编译系统和相关测量工具都给出了相应的官方网站，读者可以从网站上进一步获取所需信息。

第 12 章 OMPI 框架分析

本章是对 OMPI 的框架性的分析，也是对 OpenMP 编译过程及相关技术的一个概览，基本上将本书各章的内容都有所涉及。熟悉 OpenMP 编程和编译基本概念且对其编译器实现技术感兴趣的读者可以从本章开始阅读，对 OpenMP 编译器设计的问题有一个初步认识，然后再阅读第二篇的内容。

对 OMPI 的框架分析首先是从它的工作流程开始，然后将编译过程中的各个步骤和相应的输出做一个简单介绍；接着对代码翻译变换给出一个简单的例子，以便读者形成源代码转换的直观认识；然后对运行环境的能力作初步分析；最后给出 OMPI 的源代码文件目录结构。

12.1 工作流程

OMPI 的编译本质上是一个源代码转换器，它将带有 OpenMP 编译制导指令的源代码通过编译转换输出为基于线程库的并行源代码，所以它的源程序和目标程序都是 C 语言程序，并不像一般编译器那样直接输出可执行的目标程序。OMPI 的工作流程如图 12.1 所示。

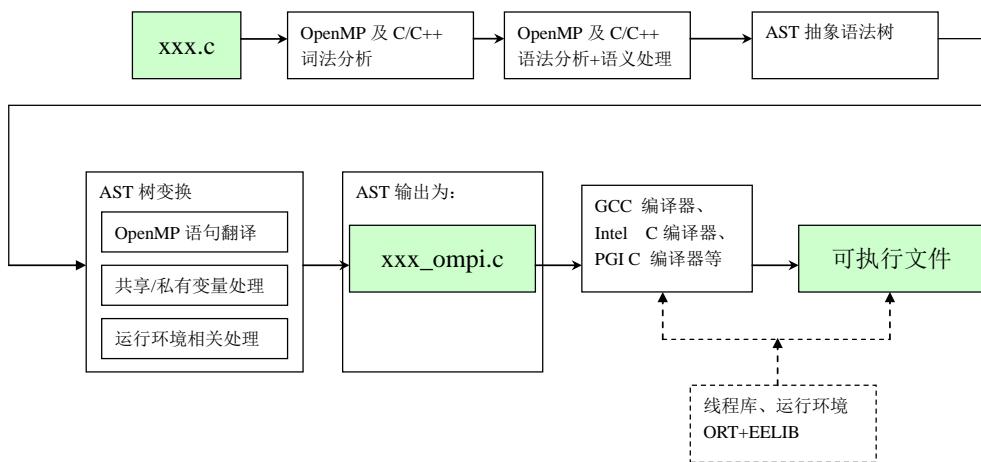


图 12.1 OMPI 工作流程

可执行文件 `ompicc` 是 OMPI 编译器的外壳，当我们运行 `ompicc` 命令对源文件 `xxx.c` 进行编译时，实际上综合调用了 OMPI 和 Linux 的多个应用程序的功能。它首先通过词法分析和语法分析的扫描，并在语法分析的过程中执行了相应的语义动作，相应的语义动作将建立起源代码的 AST 抽象语法树，这是第一阶段。

然后需要对所生成的 AST 树进行分析变换，翻译成另一种形式的源代码——其主要工作包括对 OpenMP 编译制导指令到具体的线程库操作的转换、共享变量和线程私有变量的处理、及其他比如增加与运行环境有关的代码等等，然后输出增强后（augmented）的源代码

`xxx_ompi.c`, 这是第二阶段, 第二阶段的工作是通过 OpenMP 编译的核心引擎——可执行文件 `ompi` 来完成的。

最后, 外壳程序 `ompicc` 利用某种 C 编译器, 例如 GCC 编译器、Intel C 编译器或 PGI C 编译器等, 对 `xxx_ompi.c` 进行编译输出可执行文件, 最后一步的链接过程还要链接所用的底层支撑线程库, 此时的编译工作最终完成。

作为源代码转换器, 其中的第二步是 OMPI 编译过程的核心所在, 而编译后的可执行文件需要运行环境支持(包括 ORT 和 EELIB 两个部分)。外壳 `ompicc` 和核心 `ompi` 的源代码将分别在第 13 章和第 14 章中进行分析。

12.2 OMPI 的处理步骤

为了对 OMPI 的工作流程有一个直观的宏观上的认识, 以便在后续阅读源代码的细节之前在读者头脑中建立起提纲性的概念, 我们将 OMPI 的各个工作环节快速扫描一遍。

我们还是以 `omp-hello.c` 作为例子, 看看 OMPI 的 `ompicc` 是如何编译源代码的, 变换前的代码如下:

```
1. ....
2. int main(int argc, char* argv[])
3. {
4.     int nthreads, tid;
5.     int nprocs;
6.     char buf[32];
7.     omp_set_num_threads(8);
8. #pragma omp parallel private(nthreads, tid)
9. {
10.     tid = omp_get_thread_num();
11.     printf("Hello World from OMP thread %d\n", tid);
12.     if (tid==0) {
13.         nthreads = omp_get_num_threads();
14.         printf("Number of threads %d\n", nthreads);
15.     }
16. }
17. return 0;
18. }
```

我们用 `-k` 和 `-v` 选项来运行编译命令 “`ompicc -o myhello omp-hello.c -k -v`”, 可以看到共有 4 个步骤来, 如图 12.2 所示。

```
[lqm@10 OMPI-Book-test]$ ompicc -o myhello omp-hello.c -k -v
This is OMPI compiler 1.0.0 using
  >> system compiler: gcc -std=gnu99
  >> runtime library: pthreads library (with nested parallelism)
====> Preprocessing file (omp-hello.c)
[ gcc -std=gnu99 -E -U_GNUC_ -D_OPENMP=200505 -D_POMP=200203 -D_REENTRANT -D_REENTRANT -I/usr/
local/include/ompi  omp-hello.c > omp-hello.pc ]
====> Transforming file (omp-hello.c)
[ ompi omp-hello.pc __ompi__ > omp-hello_ompi.c ]
====> Compiling file (./omp-hello_ompi.c):
[ gcc -std=gnu99 ./omp-hello_ompi.c -c -O3 -I/usr/local/include/ompi   ]
====> Linking:
[ gcc -std=gnu99 omp-hello.o -O3 -I/usr/local/include/ompi   -o myhello -L/usr/local/lib/ompi
-L/usr/local/lib/ompi/default -lort -lrt -lpthread -lort ]
[lqm@10 OMPI-Book-test]$
```

图 12.2 用-k-v 参数执行 ompicc

由于使用了-v 参数，所以 ompicc 给出了具体的处理步骤信息。

- 1) 前三行的提示信息告诉我们，它的版本是 1.0.0，然后可以知道它在代码变换后还是采用 gcc 来进行编译产生可执行文件的，还可以知道它当前是使用 POSIX 的 pthreads 线程库作为支撑的。
- 2) 它使用 gcc 的预处理能力对源代码进行预处理，通过形如 “gcc -E omp-hello.c > omp-hello.pc”的命令输出一个预处理结果文件 omp-hello.pc 文件。
- 3) 对预处理后的 omp-hello.pc 进行源代码转换，此时调用了 ompi 这个可执行文件来完成这项工作，具体命令形如“ompi omp-hello.pc __ompi__ > omp-hello_ompi.c”，转换后的输出文件就是“omp-hello_ompi.c”，此时已经加上了 OpenMP 的多线程的控制代码等内容。
- 4) 然后再调用 gcc 的编译功能生成目标代码，此时命令行形如 “gcc/omp-hello_ompi.c -c -O3 ”，此处说明只对转换后的 omp-hello_ompi.c 进行编译 (-c 选项)，此时的优化选项使用 O3（如果在配置时给出其他选项，这时会有体现），输出的目标代码为 omp-hello.o 文件。
- 5) 最终用 gcc 进行连接，输出可执行文件，所用的命令形如 “gcc omp-hello.o.....-l...-L...-l...”。最中输出可执行文件 myhello（此处用-o myhello 指出输出文件名，否则输出默认的 a.out 可执行文件）。

为了更细致的查看各个步骤到底发生了什么，我们将 omp-hello.c 在处理过程中的各个环节的处理结果都打开看看。虽然-k 选项只保留了 omp-hello_ompi.c 这个结果，但是我们还是想办法重现 4 个步骤的第一步，如图 12.3 所示的命令我们得到预处理的结果。

```
[lqm@10 OMPI-Book-test]$ gcc -std=gnu99 -E -U_GNUC_ -D_OPENMP=200505 -D_POMP=200203 -D_REENTRANT
-T -D_REENTRANT -I/usr/local/include/ompi  omp-hello.c > omp-hello.pc
[lqm@10 OMPI-Book-test]$
```

图 12.3 获得预处理结果

这时 omp-hello.pc 输出文件比原来的 omp-hello.c 要显得非常庞大，因为预处理将头文件包含进了源代码里面。所以我们可以先将<stdio.h>头文件去掉，再执行图 12.3 所示的命令，这样减少干扰来看看所包含的 ompi.h 里面有些什么，此时的输出结果如所示。

```

# 1 "omp-hello.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "omp-hello.c"
# 1 "/usr/local/include/ompi/omp.h" 1
# 35 "/usr/local/include/ompi/omp.h"
int omp_in_parallel(void);
int omp_get_thread_num(void);
void omp_set_num_threads(int num_threads);
int omp_get_num_threads(void);
int omp_get_max_threads(void);
int omp_get_num_procs(void);
void omp_set_dynamic(int dynamic_threads);
int omp_get_dynamic(void);
void omp_set_nested(int nested);
int omp_get_nested(void);

typedef void * omp_lock_t;

void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);

typedef void * omp_nest_lock_t;

void omp_init_nest_lock(omp_nest_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);

double omp_get_wtime(void);
double omp_get_wtick(void);
# 2 "omp-hello.c" 2

int main(int argc, char* argv[])
{
.....
}

```

图 12.4 预处理结果文件 `omp-hello.pc`

此时只是将 OpenMP 的运行时的例程库函数 `omp_xxx_yyy` 的声明放进来了，并没有进行实质性的代码转换，而真正关键的是将 OpenMP 编译制导指令到使用具体线程库的代码的转换，这个将在通过可执行文件 `ompi` 来完成。

下面再查看刚才使用 `-k` 参数调用 `ompicc` 命令时保留下来的经过转换后的源代码 `omp-hello_ompi.c` 文件。首先会发现增加了很多运行库的支撑例程 `ort_xxx_yyy` 形式的函数声明，如图 12.5 所示。

```

# 1 "omp-hello.pc"
# 1 "ort.defs"
# 1 "scanner_string_buffer"
Void ort_execute_serial(void * (* func)(void *), void * shared);
void ort_execute_parallel(int numthreads, void * (* func)(void *), void * shared);
void * ort_get_shared_vars(void * );
void * ort_get_thrpriv(void ** key, int size, void * origvar);
void ort_sglvar_allocate(void ** dataptr, int size, void * initer);
void ort_fence();
void ort_atomic_hesin().

```

图 12.5 增加的运行库支撑例程

同时，我们也发现新增了一个名为`_thrFunc0`的函数，如图 12.6 所示。

```

static void *_thrFunc0_(void * __me)
# 1 "scanner_string_buffer"
{
    int tid;
    int nthreads;

    /* (19) #pragma omp parallel private(nthreads, tid) -- body moved below */
# 9 "omp-hello.c"
# 10 "omp-hello.c"
{
    tid = omp_get_thread_num();
    printf("Hello_World from OMP thread %d\n", tid);
    if (tid == 0)
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads %d\n", nthreads);
    }
}
return (void *) 0;
}

```

图 12.6 新增函数`_thrFunc0`

`main` 函数被修改成了`__original_main`，如图 12.7 所示。

```

int __original_main(int argc, char * (* argv))
{
    int nthreads, tid;
    int nprocs;
    char buf[ 32];

    omp_set_num_threads(8);
    {
        /* (19) #pragma omp parallel private(nthreads, tid) */
        # 1 "scanner_string_buffer"
        ort_execute_parallel(-1, _thrFunc0_, (void *) 0);
    }
    # 18 "omp-hello.c"
    return (0);
}

```

图 12.7 新增函数`__original_main`

最后，还给出了一个与原来的 `main` 函数不同的实现，如图 12.8 所示。

```

/* OMPI-generated main() */
int main(int argc, char **argv)
{
    int _xval = 0;

    ort_initialize(&argc, &argv);
    _xval = (int) __original_main(argc, argv);
    ort_finalize(_xval);
    return (_xval);
}

```

图 12.8 被更换后的 `main` 函数

原来，`ompi` 将程序入口 `main()` 函数改名为普通函数 `__original_main()`，但是为了不改变它原来的行为，`__original_main()` 的调用方式仍然和主入口函数的方式相同，也会传入 `argc`、`argv` 命令行变量。`__original_main()` 将在新的 `main()` 函数中以普通函数的身份被调用执行。

而主函数里面的并行部分代码，被提取到外边面形成新的一个单独函数 `_thrFunc0_`，可以看出该函数编号为 0。实际上所有的并行域内的代码都被封装成函数并以 `_thrFunX_` 的形式命名，其中 X 是递增的序号。这个并行域的代码函数在主函数中被 `ort_execute_parallel()` 函数所调用，因此该并行域代码会被 `ort_execute_parallel()` 所产生的多个线程所并行执行——实现了 OpenMP 的并行化这一终极目标。

12.3 代码转换

OMPI 的编译工作就是要将带有 OpenMP 编译制导指令的 C 语言源代码转换成不带 OpenMP 编译制导指令、插入了调用 OMPI 运行库函数的 C 语言代码。所以，OMPI 首先需要将原来的源代码生成 AST 抽象语法树，然后再扫描这个语法树，发现 OpenMP 编译制导指令就进行变换：将该节点对应的子树摘下来，基本保留原来的代码不做太大变化，在合适的地方插入新增的 OMPI 运行库函数的调用，然后将该变换后的子树再插入到原来的地方。最后，OMPI 需要遍历这整个 AST 树，并还原输出成 C 语言源代码文件。至此，代码转换工作就完成了。

对带有 OpenMP 编译制导指令的子树进行变换时，有的是很简单的有的却比较复杂。比较复杂的有 parallel 结构。下面以一个 OpenMP/C 代码来说明这个转变过程。

```
1. void f()
2. {
3.     #pragma omp parallel
4.     {
5.         printf("hello world from thread %d \n", omp_get_thread_num());
6.     }
7. }
```

并行域 3~4 行的内容，将被移到一个新的函数中（名为 `_thrFunc0_`）。这个函数将被并发执行的多个线程所调用，我们称之为线程“任务函数”。

```
1. Static void * _thrFunc0_(void *_arg)
2. {
3.     /* #pragma omp parallel - body moved below */
4.     {
5.         printf("hello world from thread %d \n", omp_get_thread_num());
6.     }
7. }
```

而原来的 `f()` 函数中，一个 `ort_execute_parallel()` 的函数调用取代了原来的并行域中的代码。当主线程执行 `f()` 中的 `ort_execute_parallel()` 的函数调用时，将产生一组工作线程。第一个调用参数是创建的线程数目，-1 表示采用默认值，第二个参数是各个工作线程所要执行任务——线程任务函数（`_thrFunc0_`），第三个参数是线程间的共享变量。

```
1. void f()
2. {
3.     /* #pragma omp parallel */
4.     ort_execute_parallel(-1, _thrFunc0_, (void *)0);
5. }
6. g();
```

```
7. }
```

所有并发执行的线程，包括主线程，在执行完 `_thrFunc0` 之后经过同步操作都返回到 `f()` 函数中，然后主线程继续执行后续代码 `g()`。

代码转换中另一个大的问题是变量的数据环境问题。对于采用 `private` 修饰的私有变量，`OMPI` 编译器只是简单的将该变量的声明语句拷贝到线程任务函数中，根据 C 语言的变量作用域规则，这些变量都将是私有变量（堆栈变量）。而对于共享的全局变量，则根本不需要什么工作，默认这些变量都是线程可见的。主要的难点在于堆栈变量需要共享的情况，其解决方法是利用指针。来看看下面的 OpenMP/C 例子程序。

```
1. int a;
2. void f()
3. {
4.     int b,c,d;
5.     #pragma omp parallel private(d)
6.     a=b+c+d;
7. }
```

OpenMP 中缺省的是共享变量，因此 `a`、`b`、`c` 都是共享变量，`d` 被 `private` 子句修饰为私有变量。由于 `a` 是全局可见的变量，因此在线程中也是可见的，不需要做什么处理。但是 `b`、`c` 两个变量是堆栈变量，需要利用指针来处理。所以变换后的代码如下。

```
1. int a;
2. void f()
3. {
4.     int b,c,d;
5.     struct { int (*b); int (*c); } _shvars = {&b,&c};
6.     ort_execute_parallel(-1, _thrFunc0_, &_shvars);
7. }
```

变量 `b`、`c` 通过指针保存在一个结构体 `_shvars` 中然后传递到线程任务函数里。变量 `d` 是私有变量，将在线程任务函数中重新声明。线程任务函数里面，将调用 `ort_get_shared_vars()` 获得 `_shvars` 中的各变量指针，从而访问共享变量。线程任务函数的代码如下所示。

```
1. static void *_thrFunc0_(void * _arg)
2. {
3.     struct { int (*b); int (*c); } * _shvars = ort_get_shared_vars();
4.     int *b=_shvars->b; /* shared non-global */
5.     int *c=_shvars->c; /* shared non-global */
6.     int d; /* private */
7.     a=(*b)+(*c)+d; /* transformation due to pointers */
8.     return (void *)0;
9. }
```

由于各个线程在同一个进程空间中，因此只要使用相同的地址指针，必定指向相同的物理位置，从而可以实现变量的共享。

12.4 进程问题

如果要在集群系统上编写 OpenMP 程序，那么原来基于线程技术的编译器和运行环境就不再适用。此时不同计算节点（各自有独立的操作系统）上的进程不再拥有相同的虚存空间，

从而也没有共享变量。编译器需要不改变 OpenMP 程序原来的基于共享内存的编程方式，而使得这些程序能在集群系统上正确运行。

原来用于线程环境代码变换思想仍然可以用在这里，但是关于全局变量和 OpenMP 的数据变量的修饰（`private`、`shared` 及 `threadprivate` 等）子句需要不同的处理。由于没有共享的内存空间，所以即使像前面那样使用指针的办法也不能实现数据共享。下面就对全局变量和非全局共享变量的特殊处理进行分析。

12.4.1 全局变量

在集群计算机上利用 SVM 提供的共享内存区，就可以实现进程间的变量共享问题。问题是重新分配整个全局地址空间到 SVM 的共享内存中去。方法就是编译器在识别出所有的全局变量后，将它们转变为相同类型的指针。然后再调用 `main()` 函数之前，调用 `ort_sglvar_allocate()` 函数为所有的全局变量分配共享内存。各个全局变量都映射到这个共享内存区内，如果需要则将有些变量进行初始化。下面以一段代码为例来说明。

```
1. int a =1, b=2, c;
2. void f()
3. {
4.     #pragma omp parallel
5.     c=a+b;
6. }
```

此处变量 `a`、`b`、`c` 都是全局变量，所以都必须分配在共享内存区内。而且 `a`、`b` 变量还需要初始化。经过变换后的代码如下。

```
1. int _sglini_a =1, (*a), _sglini_b=1, (*b), (*c);
2. static void *_thrFunc0_(void *arg)
3. {
4.     /*#pragma omp parallel - body moved below */
5.     (*c)=(*a)+(*b);
6.     return (void *)0;
7. }
```

变量 `a`、`b`、`c` 都被转换成相应的同类型指针（见第 1 行）。代码中所有引用这些变量的地方也都被修改成指针访问（见第 4 行）。对于有初始化的变量，新增了带有前缀 `_sglini_` 的变量用来保存相应的初始值（见第 1 行）。而原来的函数则变成如下代码所示。

```
1. void f()
2. {
3.     ort_execute_parallel(-1, _thrFunc0_, (void *)0);
4. }
```

同时还提供了在 `main()` 函数之前就需要调用的其他函数(`constructor` 构造函数)。

```
1. static void __attribute__ ((constructor)) _init_shvars_0(void)
2. static void _init_shvars_0(void)
3. {
4.     ort_sglvar_allocate((void **)&c, sizeof (int), (void*)0);
5.     ort_sglvar_allocate((void **)&b, sizeof (int), (void*)&_sglini_b);
6.     ort_sglvar_allocate((void **)&a, sizeof (int), (void*)&_sglini_a);
7. }
```

构造函数 `_init_shvars_0` 在 `main()` 函数调用之前执行，它包含了 3 个 `ort_sglvar_allocate()` 函数的调用，每个调用对应一个共享全局变量。使用构造函数的原因是考虑到当一个程序有多个独立的 C 代码源文件编译成的模块经链接构成的可执行文件时，是无法在编译时知道全部的全局共享变量。但是通过在每个 C 代码源文件中定义一个构造函数，就可以保证在 `main()` 函数调用之前被调用执行。还需要注意的就是每个 C 代码源文件中的构造函数名字不能相同，因此 OMPI 的编译器将用带有 “`_init_shvars_`” 前缀不同的编号来命名这些构造函数。Omni 编译器也采用了相似的方法来运行于集群系统上，而另一些编译器如 NanosCompiler 则采用不同的方法，Nanos 中的进程利用底层的 SVM 系统共享全部进程内存空间，此时与普通线程模式的处理完全一样，不需做额外的努力，但是这样一来性能将非常差。

12.4.2 非全局共享变量

非全局变量是保存在堆栈中的，但是多个线程之间虽然共享代码段、数据段，堆栈却是各自分开的。这种情况下以堆栈方式来操作则无法实现共享，所以在基于线程的技术时需要借助于指针（线程间的存储空间是共享的）而不是堆栈操作来获得这些共享变量。但是如果基于进程的技术来实现运行环境，一个进程即使使用指针也无法访问到其他进程的存储空间。OMPI 的解决方法是同时结合 SVM 提供的共享内存和指针技术，主进程 master 的堆栈放置在共享内存区，所有对这些非全局的共享变量的访问都修改成指针访问，这样一来其他进程借助于指针可以访问到这个共享内存区的变量。

另外一些编译器则使用下面的解决方法。每次遇到一个并行域的时候，首先开辟一个 SVM 共享内存区，然后将这些需要共享的堆栈变量拷贝到这个共享区，所有进程都使用指针来访问这些共享数据，最后在退出并行域的时候将这些共享取得数据拷贝回到堆栈中。主要的问题是需要增加数据拷贝的开销。

12.5 运行环境

OpenMP 是基于 fork-join 模型的，因此在并行域内是由多个经 `fork` 操作产生的实体来并发执行任务，这些执行任务的实体我们称为“OpenMP 线程”，而这些 OpenMP 线程具体是由操作系统负责的进程与线程来实现，还是由用户级线程来实现都是可以的。在 OMPI 中将 OpenMP 线程抽象为“执行体”（EEs, Execution Entities）。由 ORT (OMPI Runtime) 模块来统一管理这些执行体 EE 并通过这些 EE 来并发执行任务，而 EE 的具体实现由相应的 EELIB(Execution Entities Library) 模块负责，系统中可以有多种 EELIB 模块，但是每次只有一种生效。

下面对运行库的初始化、任务分担、同步、线程私有变量以及通用线程接口等问题作简单介绍。

12.5.1 初始化

OpenMP 程序运行时，ORT 先调用 `ort_initialize()` 函数进行整个运行时系统的初始化，编译器将这个函数调用插入到 `main()` 中，负责下面的工作：

1. 处理 OpenMP 的环境变量；
2. 初始化 EELIB；
3. 构造主线程的 EE 的控制块 EECB (EE Control Block)，EECB 类似于操作系统中的 PCB；
4. 如果 EELIB 是基于进程技术的，则还将处理全局变量。

12.5.2 并行域的处理

在进入并行域代码的时候，实际上是调用了运行库中的 `ort_execute_parallel()` 函数（参见 12.3 的代码转换部分），此时 ORT 将会与 EELIB 进行协商，请求并获得一定数量的 EE（取决于用户的要求、是否启动了并行嵌套特性和动态调整特性），这些 EE 将首先调用 `ort_get_ee_work()` 确定自己所要执行的任务。EE 获取一个以函数指针指向的任务函数，同时也将初始化自己的 EECB 控制快（包含组的大小、组内 id、并行嵌套级数、父结点 EE 的指针）。

在并行嵌套的情况下，所有的 EE 都通过父结点 EE 指针构成一棵动态变化的树，每当新进入到一层（级）的并行域，就会新增一个有若干 EE 构成的子树，而每退出某一级的并行域则会有一个子树消失。

12.5.3 任务分担

OpenMP 提供了三种任务分担的结构，分别是 `for`、`sections` 和 `single` 用于将任务合理的分配在多个 OpenMP 线程上。这些结构对应的代码区通常都是在结束处隐含有同步操作的（阻塞），除非使用了 `nowait` 子句显式地说明不需要同步，这使得 OMPI 需要跟踪记录这些额外的状态（某个 EE 组内的 EE 可能分别处于不同的并行域中）。每当一个并行域中有一个 EE 在执行任务，我们就称这个并行域是活跃的。由于在每个需要任务分担的任务中我们需要记录哪些子任务已经完成，哪些还没有完成，如果因为 `nowait` 子句使得有多个负载分担结构同时活跃，那就需要为每一个负载分担结构都提供相应的独立记录信息。考虑到在一个循环中的 `single` 语句或 `sections` 语句，上面的方法只为负载分担结构保留一个记录信息并不够用。此时可以采用动态生成这些记录信息，也可以像 Omni 编译器那样禁止超过一个非阻塞的负载分担结构。OMPI 的方法是在某一组 EE 的父结点的 EECB 上为各个负载分担结构预生成一组记录信息的队列，当前最大的活跃负载分担结构数量为 `MAXWS`。每个记录里面记录有结构相关的信息（`sections` 中剩余的 `section` 数目，`for` 结构中的下一个需要调度的任务和步长增量，用于 EE 并发访问的锁等等）、以及队列相关的信息（退出此负载分担结构的 EE 数量等），一旦活跃的负载分担结构数量达到 `MAXWS` 时，将阻止 EE 进入新的负载分担结构，直到分配新的空间来满足这些记录。OMPI 的 ORT 对这些负载分担队列的访问是经过仔细优化的。如果可以，尽量采用无锁的方式来访问这些队列信息、尽量采用原子操作、只在必要时使用非嵌套的单层锁 `plain lock`、避免对整个队列作完整的初始化。只有当一个 EE 遇到为初始化的记录项时，才进行完整的初始化。

在 ORT 看来，每次遇到负载分担结构，就需要调用两个函数，一个是进入时的 `ort_enter_workshare_region()`，一个是退出时的 `ort_leave_workshare_region()`。这两个函数都要修改前面提到的队列里的信息。第一个进入负载分担结构的 EE 将初始化当前的记录信息并

为下一个记录作准备，其他后续进入的 EE 则无需做什么。最后一个退出的 EE 在 `ort_leave_workshare_region()`，需要清空里面的记录，而前面离开的 EE 只需要将 `notleft` 计数器（未退出该区线程数目）进行减 1 操作。一旦清空，就可以回收利用于其他的负载分担结构的记录。

下面用一个例子来展示相关的过程。

```
1. void f()
2. {
3.     int i;
4.     #pragma omp parallel
5.     #pragma omp for private(i) schedule(static)
6.     for( i=0; i< 100; i++)
7.         do_some_calculations(i);
}
```

经 OMPI 的代码变换后的代码如下。

```
1. static void *_thrFunc0_(void *arg)
2. {
3.     {    int i;
4.         Int from_=0,to_=0,step_;
5.         strcut _ort_getopt_gdopt_;

6.         step_=1;
7.         ort_entering_for(1,0,0,step_, &gdopt_);
8.         if(ort_get_static_default_chunk(0,100,step_,&from_ ,&to_ ))
9.             {    for(i=from_; i< to_ ; i=i+1)
10.                 do_some_calculations(i);
11.             }
12.             ort_leaving_for();
13.     }
14.     return(void *)0;
15. }
```

第 7 行的 `ort_entering_for()` 函数的内部会调用 `ort_enter_workshare_region()` 函数，它通知 ORT 这个负载分担的类型、是否为阻塞方式（1 表示非阻塞）。但是编译器可以识别出没有必要为 `parallel` 结构和 `for` 结构各自启用一个同步路障，因此它忽略了 `for` 要求的路障并告知 ORT 这是一个 `nowait` 的负载分担结构。第二个参数是告诉 ORT 是否 `for` 结合了 `ordered` 子句，第三和第四个参数指出循环的下界和步长。最后一个参数用于 `dynamic` 和 `guided` 的调度方式。`ort_get_static_default_chunk()` 用于确定各个 EE 所需要负责的 `for` 循环中那一部分。最后 EE 退出该负载分担结构时调用 `ort_leaving_for()` 函数，该函数在进而调用 `ort_leave_workshared_region()` 告诉 ORT 本 EE 已经退出了该负载分担结构。

12.5.4 同步

ORT 将源程序中的 `barrier` 语句用 `ort_barrier_me()` 函数调用来取代。当 EE 执行该函数时将会把自己阻塞（利用一个共享的数组），然后等待父结点 EE 唤醒。等待可以使用一个标志上

的自旋锁，但是为了节约 CPU 资源，自旋锁并不一直自旋下去，而是在一段时间后让出 CPU。父结点 EE 检查那个共享数组，如果发现所有 EE 都已经到达同步点，那么它将通过设置标志从而释放所有阻塞的 EE。但是 ORT 允许用户在必要时自己提供路障的实现代码。

ORT 对另外的两个相关语句——`critical` 和 `atomic` 的处理方式也是相同的。EELIB 的锁操作可以提供必要的互斥。编译器会在需要保护的代码前面调用加锁、解除保护时调用解锁操作。语句 `atomic` 的处理方式就是这样的，这些锁在 ORT 内部声明和初始化。但是 `critical` 有所不同，ORT 允许 `critical` 有不同的名字，编译器会为每一个 `critical` 声明全局锁并传递给 ORT，第一个到达的 EE 负责锁的初始化。

12.5.5 线程专有变量

ORT 提供了处理 OpenMP 线程专有（`threadprivate`）变量的必要机制。这与普通的线程模型是不一致的，因为线程中的全局变量自然而然地是共享变量。被 `threadprivate` 语句所限定的全局变量需要拷贝到每个线程中，各自独立处理。反过来在基于进程的实现中，所有变量都是进程私有的，需要设法让它们能共享。编译器和运行环境共同努力才能实现 `threadprivate` 的特性。我们以下面的代码片段为例来说明（变量 `a`、`b` 都是 `threadprivate` 变量）。

```
1. int a, b = 1;
2. #pragma omp threadprivate(a, b)
3. void f()
4. {
    #pragma omp parallel copyin(b)
5.     a = omp_get_thread_num() + b;
6. }
```

变量 `a` 和 `b` 被编译器分别更名为 `tp_a` 和 `tp_b`，同时编译器也为每一个线程私有变量分配了一个线程相关的数据键值——`tp_a_key` 用于变量 `a`、`tp_b_key` 用于变量 `b`。根据 POSIX 的线程标准，所有线程都是用相同的数据键值，但是它们可以和不同的数值相关联。原来的代码经过编译器变换后形成新的代码。

```
1. int tp_a , tp_b = 1;
2. static void *tp_a_key;
3. static void *tp_b_key;
4. static void *_thrFunc0_(void *arg)
5. {
    int (*a)=ort_get_thrpriv(&tp_a_key, sizeof(tp_a), &tp_a);
6.     int (*b)=ort_get_thrpriv(&tp_b_key, sizeof(tp_b), &tp_b);
7.     /* copyin initialization(s) */
8.     *b= tp_b;
9.     ort_barrier_me();
10.    (*a)=omp_get_thread_num() + (*b);
11.    return (void *) 0;
12. }
```

每个线程在进入到 `_thrFunc_` 函数的时候，必须初始化自己的线程专有变量的拷贝。在第 10 行进行赋值的时候必须使用各自的私有变量。这是通过调用 ORT 的 `ort_get_thrpriv()` 函数来实现的，调用时需要键值、变量的 `size` 及指针三个参数。首先每个线程将会分配一个内存空

间并将初始值拷贝进去，此时线程将会把编译器给出的键值和相应的存储空间对应起来。此时每个线程都能通过数据的键值找到它们自己的私有拷贝。这些分配的存储空间在程序退出前都不会释放，后续的时间里线程在不同的并行域中继续维护它们自己的私有变量。

这里的 `copyin` 的实现比较简单。所有在 `copyin` 里面说明的变量，在线程并发执行之前都必须将主线程中的对应数值拷贝到专有变量中。还是以上面的例子来说，主线程是唯一将变量 `tp_a` 和 `tp_b` 作为私有变量的拷贝的（这个通过 `ort_get_thrpriv()` 函数中将指针指向这两个变量来实现的）。作为全局变量，这两个变量被包括主线程的所有线程所共享。主线程在第 1 行对 `b` 变量进行初始化，而其他所有变量在第 8 行都可以访问到它。第 9 行的同步路障是必须的，否则在其他线程没有完成对 `b` 的初始化时主线程将 `tp_b` 的值作了修改将会引起错误。

ORT 对于 `copyprivate` 语句的处理要复杂一些。这个语句只出现在 `single` 编译制导指令里面。它提供了一种用于将一个线程的专有变量广播到所有线程的机制。广播操作是通过调用 ORT 的 `ort_broadcast_private()` 函数来实现的。这个函数将传入的参数作为需要广播的线程专有变量的指针，它在父结点的 EECB 里面维护一个可以被所有线程所访问的动态构建的指针数组。其他线程只需要调用 `ort_copy_thrpriv()` 就可以将新的变量值从父结点的 EECB 里拷贝到它们自己的专有变量中。

12.5.6 与 EELIB 的接口

EELIB 负责提供主执行体之外的所有执行体，外加三种类型的锁：普通锁、嵌套锁和自旋锁。前两种锁通过 OpenMP 的运行库接口被用户所使用，而自旋锁只是由 ORT 内部所使用。如果是采用线程来实现执行体，那么 EELIB 没有太多事可做，此时主要由 ORT 处理几乎所有的事情。但是如果执行体是由进程来担当，EELIB 需要作少量的扩展以支持新的运行环境，此时 EELIB 需要提供共享内存分配的函数。同时 ORT 的通信子系统也需要访问一些有 EELIB 管理的特定数据结构。

在初始化的时候，EELIB 向 ORT 声明自己的能力特性，包括是否支持嵌套并行、是否支持动态调整执行体的数目、最大的执行体数目和最高的并行嵌套层次等。在执行体方面，EELIB 提供了三个函数共 ORT 调用，它们是 `ee_request()`、`ee_create()`、`ee_waitall()`。前两个函数用于新创建一个组的时候。父结点可以通过 `request()` 请求一定数目的执行体，EELIB 则告知它所能提供的执行体数目。在不支持并行嵌套的 EELIB 里，当前套层次大于 1 层的时候总是返回 0。如果 EELIB 提供不了足够数量的执行体，同时系统又不允许动态调整执行体数目，那么将会使程序提前终止。反之，如果可以动态调整，程序只是给出警告然后继续运行。因此当 ORT 调用 `ee_create()` 的时候指定的数目和 `ee_request()` 的时候地数目可能是不一样的。ORT 需要向 EELIB 提供必要信息来创建执行体并引导执行体完成任务，包括执行体数目、所有执行的任务函数等。当一个执行体开始执行时，首先就是调用 `ort_get_ee_work()` 函数，填写好执行并行域里的任务所需的 EECB 信息。在并行域结束时主执行体调用 `ee_waitall()`，阻塞直到所有其他线程都完成任务并到达这个同步点。

12.6 源代码文档结构

OMPI 的源代码解压后在 `ompi-1.0.0` 目录中，最外层是一些 `configure` 和 `make` 文件，另有三个目录 `doc`、`compiler` 和 `lib`。目录 `doc` 里面是一些 OMPi 设计的说明文档，目录 `compiler` 里面主要是编译器的源代码，而 `lib` 里面主要是运行库的源代码。在 `compiler` 目录中主要是 `ompicc.c` 文件和一个 `ompi` 目录。`Ompicc` 是编译器的外壳，它通过调用 C 预处理器、OpenMP 编译器（`ompi`）、C 编译器以及链接器来完成整个编译工作，生成可执行文件。其中的 `ompi` 可执行文件是 OpenMP 代码翻译的核心引擎，所以 `ompi` 目录下有许多 C 代码源文件：

1. `ompi.c/ompi.h`。OpenMP 编译器核心，它将调用语法分析器的功能完成 AST 建立，另外还调用翻译变换函数以及 AST 输出函数等。
2. `scanner.l/scanner.h/parser.y`。这两个文件在编译时执行 `make` 的时候将由 Flex 和 Bison 处理，生成词法扫描器和语法分析器文件，执行 `make` 编译后起作用的是：`scanner.c`、`scanner.h`、`parser.c` 和 `parser.h`。
3. `ast.c/ast.h/symtab.c`。包含了语法分析中为了建立 AST 而执行语义动作时所调用的函数，主要是 AST 节点的创建和符号管理等操作。
4. `ast_show.c/ast_print.c/ast_free.c/ast_copy.c`。AST 节点的基本操作函数，包括显示输出到 `stdout` 或字符串中、释放 AST 节点、拷贝 AST 节点等。
5. `ast_xform.c`。这是实现代码变换的上层代码，它还将调用所有其他 OpenMP 变换代码（函数），里面也实现了一些简单的变换函数。
6. `x_parallel.c/x_single.c/x_sections.c/x_for.c,x_thrpriv.c`。这 5 个文件实现了比较复杂的变换，这五种构造的变换要远比 `ast_xform()` 里面实现的简单变换的难度要高得多。
7. `ast_vars.c/x_clauses.c`。用于对代码中和 OpenMP 子句中变量变换。
8. `x_types.c`。处理用户自定义类型和声明，包括 `struct/union/enum entities`。

外壳 `ompicc` 的主文件是`.../ompi-1.0.0/compiler/ompicc.c`，编译核心程序 `ompi` 的主程序是`.../ompi-1.0.0/ompi/ompi.c`。`ompi` 相对比较庞大，不仅有 OpenMP 源代码翻译变换代码，而且 Lex 和 Yacc 生成的扫描器也包含在其中。

运行库的代码在`.../ompi-1.0.0/lib` 目录中。抽象的线程管理功能全部在 `ort.c` 中实现，另外有多种线程库的 EECB 接口实现，各自在一个独立的子目录中。每种 EELIB 里面主要是两个文件：`ee.h` 和 `othr.c`，它们将具体的线程库封装成抽象的执行体 EE。

12.7 后续阅读建议

由于本书后续章节与源代码结合紧密，阅读中容易陷于细节而进入只见树木不见森林的状态。因此建议读者以“快速浏览”的方式将后续章节大致阅读一下，了解各章节都涉及什么领域，然后有重点地结合自己需要和兴趣有选择的阅读，对于有疑问的地方还需要参考第二篇中

的内容。中途涉及 C 语言编程、Linux 系统编程（线程、进程、同步、共享内存等）、编译原理（词法分析、语法分析、AST 树和符号表等）和工具的时候，如果读者有疑惑可以结合 google 搜索来查找相应的函数或工具的说明和使用方法。由于 OMPI 的程序规模并不算太大，只要理解各部分功能和相互关系、抓住关键的数据结构和处理流程，经过由粗入细的过程是可以逐步将它完全掌握的。理解之后，读者就可以对 OMPI 进行修改和增强了。

12.8 小结

本章对 OMPI 作了整体上的介绍和分析。首先说明了 OMPI 的工作流程，指出其核心工作是由 `ompi` 完成的代码变换，接着通过还原 `ompicc` 对源代码处理过程中的各个中间环节，给读者一个非常简单和直观的概念。然后简单扼要地将代码变换、进程问题和运行环境中的若干问题作了概念性的介绍，以此快速建立起工作原理和流程的认识，并大致了解到 OpenMP 编译中所涉及的问题。最后对 OMPI 代码树的结构作了简单的分析和归类。如果熟悉 OpenMP 编程且对编译实现比较感兴趣的，可以将本章作为阅读的起点，然后根据需要阅读书中其他章节。

第 13 章 `ompicc.c` 源码分析

OMPI 编译器的可执行文件是一个外壳程序：`ompicc`，它由 `ompicc.c` 编译而来。`ompicc.c` 在 `main` 函数里面调用 `ompicc_compile()` 函数进行编译变换（源代码到源代码）输出`***_ompi.c` 文件并用 C 编译器编译（源代码到机器码目标文件）输出`*.o` 文件，如果需要的话再进行链接生成可执行文件。其中源代码到源代码的编译是通过调用外部 `ompi` 可执行文件来进行的。由于一个程序可能有多个源程序文件组成，所以 `ompicc` 可以接受多个源文件作为输入，并且逐个调用 `ompi` 来进行编译（源代码变换），然后调用 C 编译器来编译这些经过变换的代码以生成目标文件，最终再将这些目标文件进行链接。因此 `ompicc` 这个外壳程序的一个重要功能就是从 `ompicc` 的命令行参数选项中提取、管理和处理这些源文件列表。

由于是源码分析，本章内容的安排是按照 `ompicc.c` 文件的文本顺序来组织的，因此有功能可能分散在不同的小节中。如果想从整体到细节来掌握，可以先阅读 13.5 从 `main()` 函数入手。由于整个过程涉及命令行参数、环境变量和配置文件信息等多方面的变量，因此在第一遍阅读时可不用太追究这些变量之间的关系（不影响代码的阅读），如果有需要再仔细阅读 13.7 小节的内容。

13.1 `ompicc` 工作流程

外壳程序 `ompicc` 的功能包括三个部分：1) 需要编译的源文件名列表以及各种参数选项的管理；2) 调用 `ompi` 对上述文件进行变换；3) 调用 C 编译器将前面翻译变换好的源程序生成目标文件，如有必要则进行链接生成可执行文件。整个程序的流程如图 13.1 所示，第一部分有三个主要函数 `ompicc_get_envvars()`、`parse_args()` 和 `get_ort_flags()`，第二部分主要是 `ompicc_compile()`，第三部分主要是 `ompicc_link()` 函数。

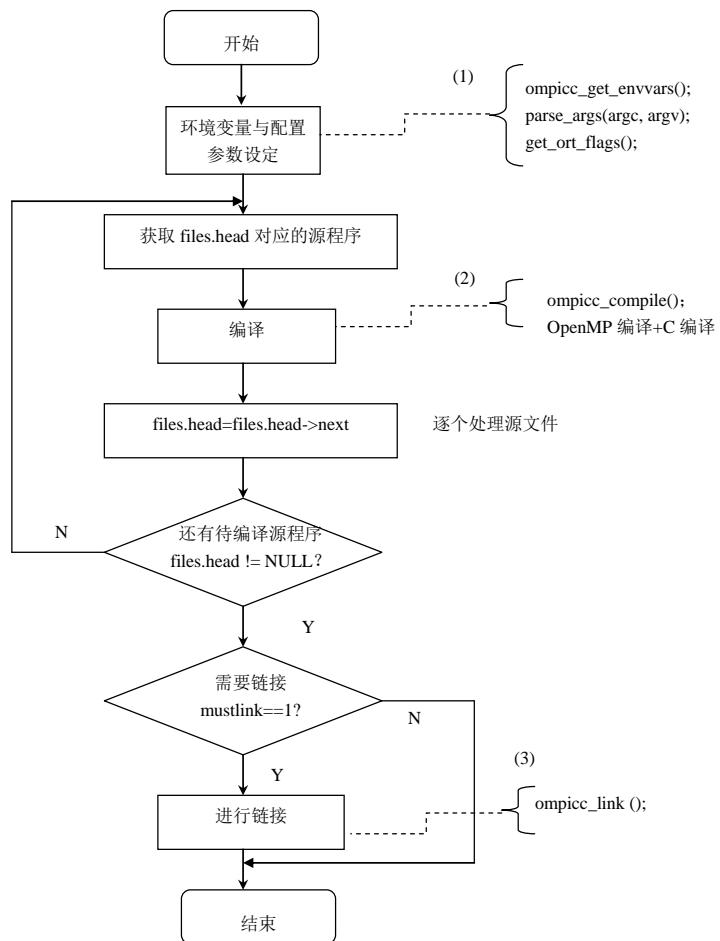


图 13.1 ompicc 工作流程

13.2 变量声明及参数处理

外壳程序源代码 `ompicc.c` 的前面是一些变量声明，然后是关于参数项的处理函数。

`ompicc` 中多处用到参数列表或文件名列表，因此它们统一用 `arg_t` 类型的变量来表示，并且这些项还用 `next` 指针形成链表，并用 `arglist_t` 类型的变量来指出链表的首尾节点。并配上相应的操作函数进行处理，其中 `new_arg()` 函数用于创建一个指定类型和内容的 `arg_t` 节点，`arglist_init()` 用于初始化某一参数项的上层 `arglist_t` 类型的变量，`arglist_add()` 用于插入某个项到指定的链表的末尾处，`append_arg()` 用于将多个参数（用 `argc` 和 `argv` 表示）分成多个项逐个插入到指定的链表中，`strarglist()` 用于将某个链表中的多个项输出到指定的字符串中。

```

1. /* OMPICC
2. * A driver for the OMPI compiler.
3. */
4. /* Aug 2007:
5. *   Major simplifications due to new threadprivate implementation.
6. *   Removed POMP support.
7. * May 2007:
8. *   Fixed some definitions for compiling/linking/preprocessing.

```

```

9.      * -lrt now determined at configuration time.
10.     * Some less important bug fixes.
11.    */

12.   #include <stdio.h>
13.   #include <stdlib.h>
14.   #include <string.h>
15.   #include <ctype.h>
16.   #include <stdarg.h>
17.   #include <sys/types.h>
18.   #include <sys/wait.h>
19.   #include <sys/stat.h>
20.   #include <unistd.h>

21.   #include "config.h"                                配置信息

22.   #define LEN 4096
23.   #define SLEN 256

```

由 compiler/Makefile.am 外部定义了下列的宏，然后通过编译时 GCC 的-D 命令而生效并在 parse_arg() 时分析 LinkFlags 等参数的时候再保存到 link_args 等链表中。

```

24.  /* Definitions (macros) provided externally (see Makefile.am)
25.  *
26.  * OmpiName
27.  * CPPcmd
28.  * CCcmd
29.  * PreprocFlags
30.  * CompileFlags
31.  * LinkFlags
32.  * IncludeDir
33.  * LibDir
34.  */

```

下面这些预处理器名和编译器名，以及它们的参数选项可以有多个来源：ompicc 源代码目录中的 configure 文件、运行库的 ortconf(pthread) 文件、环境变量、命令行参数。

```

35.  /* Flags collected from the OMPI_CPP, OMPI_CPPFLAGS,
36.  * OMPI_CC, OMPI_CFLAGS and OMPI_LDFLAGS, from the ./configure info and
37.  * the library-specific configuration file.
38.  */

```

下面确定与处理器、编译器及相关的参数选项。

39. #define PATHSIZE 1024	路径名长度最大为 1024 字节
40. #define FLAGSIZE 2048	编译选项最长 2048 字节

下面两个变量在 `main()` 调用 `ompicc_get_envvars()` 时根据命令行的宏定义或环境变量的值来设定。

```
41. char PREPROCESSOR[PATHSIZE], COMPILER[PATHSIZE],
```

下面四个变量在 `main()` 调用 `ompicc_get_envvars()` 时根据命令行的宏定义或环境变量的值来首次设定，然后可能在 `get_ort_flags()` 调用 `conffile_read()` 并间接调用 `setflag()` 的时候追加新的选项。

```
42.     CPPFLAGS[FLAGSIZE], CFLAGS[FLAGSIZE], LDFLAGS[FLAGSIZE],
```

```
43.     ORTINFO[FLAGSIZE];
```

```
44. char ortlibname[PATHSIZE],           /* The runtime library needed */
```

ort 线程库的名字

```
45. RealOmpiName[PATHSIZE];          真正的 ompi 可执行文件名
```

以下是用于记录 `ompicc` 被执行时的命令行参数/选项/文件名的通用数据结构 `arg_t`, 及其相应的链表 `arglist_t`。

```
46. typedef struct arg_s {
```

```
47.     char opt;                  选项类型 “c/l/L/I/D/U/o/k/v” 之一
```

```
48.     char val[SLEN];          选项内容
```

```
49.     struct arg_s *next;      指向下一选项
```

```
50. } arg_t;                   类型定义
```

```
51. typedef struct {
```

```
52.     arg_t *head, *tail;
```

```
53. } arglist_t;              所有选项构成一个链表
```

`ompi_info()` 宏定义用于打印 OMPI 的相关信息，其中 `PACKAGE_STRING` 在 `config.h` 中定义，剩下三个变量在本文件定义。

```
54. #define ompi_info() \
```

```
55.     fprintf(stderr,\
```

```
56.         "This is %s using\n >> system compiler: %s\n >> runtime library: %s\n",\
```

```
57.         PACKAGE_STRING, COMPILER, *ORTINFO ? ORTINFO : ortlibname)
```

`ompicc_error()` 用于打印出错误信息，注意这里使用参数个数不确定的函数调用方式，需要用到 `va_list`、`va_start()`、`va_end()`，所以文件开头就包含了 `stdarg.h` 的头文件。

```
58. void ompicc_error(int exitcode, char *format, ...)  参数个数不确定的调用方式
```

```
59. {
```

```
60.     va_list ap;             va_list 用于不确定个数的参数
```

```
61.     va_start(ap, format);
```

```
62.     fprintf(stderr, "[ompicc error]: ");
```

```
63.     vfprintf(stderr, format, ap);
```

```
64.     va_end(ap);
```

```
65.     exit(exitcode);
66. }
```

`get_basename()`根据完整路径名获得不带路径的文件名。该函数的前面两个特殊路径名的处理似乎是不必要的。

```
67. char *get_basename(char *path)
68. {
69.     char *s;

70.     if (path == NULL || *path == 0)           如果路径为空（应该不会出现）
71.         return ".";
72.     else if (path[0] == '/' && path[1] == 0)   则按当前路径处理
73.         return path;
74.     else {                                     如果是绝对路径根目录
75.         s = path;                            则直接返回路径
76.         while (*s)                          其他情况（除了当前路径、根目录）
77.             s++;                           s 指向字符串结束符
78.         s--;                           s 指向 path 的最后一个字符
79.         while (*s == '/' && s != path)       查找排在最后面的一个 ‘/’ 字符
79.             (例如: /abc/def///)            (将 ‘/’ 换成字符串结束符)
80.         *s-- = 0;                         While 语句结束时 s 后退到指向 f 字符)
81.         if (s == path)                   如果后退之后就是 path 的开头（实际上不可能出
81.             现，因为绝对路径在前面已经处理过了）
82.             return path;                  返回路径 path
83.             while (*s != '/' && s != path)    如果形如/abc/def/jkl
84.                 s--;                     则 s 指针后退跳过 jkl 三个字符，找到最后一个 “/”
85.             if (*s == '/')                有路径部分，则返回路径后的内容
86.                 return s + 1;              无路径部分，直接返回全部内容
87.             else return path;
88.     }
89. }
```

`get dirname()`根据完整的路径名，获取其中的路径部分，它从完整路径名的末尾开始查找“/”符号，找到后在此位置截断字符串，从而获得一个只有路径的字符串。

```
90. char *get dirname(char *path)
91. {
92.     char *s;
93.     if (path == NULL || *path == 0)           空路径，则返回 “.”
94.         return ".";
95.     else if (path[0] == '/' && path[1] == 0)   根路径，直接返回
96.         return path;
97.     else {                                     其他情况
98.         s = path;
```

```

99.     while (*s)
100.         s++;
101.         s--;
102.         while (*s == '/' && s != path)           指针移到字符串末尾处
103.             *s-- = 0;
104.         if (s == path)                         删掉后如果没有字符
105.             return "/";
106.         while (*s != '/' && s != path)          将结尾处的 "/" 删掉
107.             s--;
108.         if (*s == '/') {                      找到排在最后的 "/"
109.             if (s != path)                     且不是第一个字符
110.                 *s = 0;                      则就此截断
111.             else return "/";                否则返回根目录
112.         }
113.         else return ".";                  没有找到最后面的 "/" 符号，返回当前目录 "."
114.         return path;                    返回截断文件名后的路径
115.     }
116. }

```

`new_arg()`根据指定的类型和字符串生成一个 `arg_t` 参数项。

```

117. arg_t *new_arg(char opt, char *val)
118. {
119.     arg_t *p;
120.     if ((p = (arg_t *) malloc(sizeof(arg_t))) == NULL)    首先分配空间
121.         ompicc_error(-1, "malloc() failed\n");
122.     p->opt = opt;                           设置类型
123.     if (val != NULL)
124.         strcpy(p->val, val);              复制字符串
125.     else p->val[0] = 0;
126.     p->next = NULL;
127.     return p;                            返回参数项
128. }

```

`arglist_init()`参数链表初始化，只是将列表的头和尾指针都置为空。

```

129. void arglist_init(arglist_t *l)
130. {
131.     l->head = l->tail = NULL;
132. }

```

`arglist_add()`将一个参数 `arg` 插入到参数链表 `l` 中，新插入的参数项是插到队尾的。

```

133. void arglist_add (arglist_t *l, arg_t *arg)
134. {
135.     if (l->head == NULL)                  如果参数链表为空

```

```

136.     l->head = l->tail = arg;           则是唯一元素（既是链表头又是尾）
137. else {                                否则链表非空
138.     l->tail->next = arg;            插入到队尾
139.     l->tail = arg;                队尾指向新插入的参数项
140. }
141. }

```

`append_arg()`是参数分析函数 `parse_arg()`的核心函数，用于将某个选项对应的多个参数插入到参数链表中，这些选项用 `argc` 和 `argv` 表示，`proceed` 标志表示处理完选项类型后，后有相应的参数需要处理，例如“`-o mypro`”，在识别出“`-o`”选项后还需要处理“`mypro`”参数。因此对于“`-l/usr/local/include`”形式的命令行参数时按照 155 行进行处理的，而“`-o mypro`”是按照 161 行处理。

其返回值表示额外还处理了多少个命令行参数项。

```

142. int append_arg(arglist_t *l, int argc, char **argv, int proceed)
143. {
144.     char opt, val[SLEN];
145.     arg_t *p;

146.     val[0] = 0;
147.     if (argv[0][0] == 0)           命令行参数项为空
148.         return 0;               不作处理直接返回 0
149.     if (argv[0][0] != '-')       如果选项字符串不是以“-”开头
150.         p = new_arg(0, *argv);   按类型 0 生成参数项
151.         arglist_add(l, p);    插入到 l 链表中
152.         return 0;              返回 0
153.     }
154.     opt = argv[0][1];          选项是以“-”开头，所以第二个字符是类型

155.     if (argv[0][2] != 0) {      如果第三个字符不是结束字符，这种情况对应于
156.         strcpy(val, &argv[0][2]);  “-l/usr/local/include”的情况
157.         p = new_arg(opt, val);  则将后面的字符拷贝到 val 中
158.         arglist_add(l, p);    按类型和内容生成参数项
159.         return 0;              插入到指定的 l 链表中
160.     }
161.     else {                   第三个字符是结束字符
162.         if (proceed && argc > 1) 且还有未处理项
163.            这种情况对应于“-o myproc”
164.             strcpy(val, &argv[1][0]);  则将下一个参数内容拷贝到 val 中
165.             p = new_arg(opt, val);  按类型和内容生成参数项
166.             arglist_add(l, p);    插入到指定的 l 链表中
167.             return proceed && argc > 1;
168.     }
169. }

```

`strarglist()`将参数链表 `l` 的各项保存到 `dest` 的字符串中。

```
170. void strarglist(char *dest, arglist_t *l)
171. {
172.     arg_t *p;
173.     *dest = 0;
174.     for (p = l->head; p != NULL; p = p->next) {           遍历参数链表
175.         if (p->opt != 0) {                                opt 不为 0
176.             sprintf(dest, "-%c ", p->opt);                形成类型字串到 dest
177.             if (p->opt == 'o')
178.                 dest += 3;                                  -o 参数需要空格
179.             else dest += 2;                               其他参数不需要空格
180.         }
181.         sprintf(dest, "%s ", p->val);                  将字符串继续输出到 dest
182.         dest += strlen(p->val) + 1;                      调整 dest 指针位置
183.     }
184. }
```

`fok()`检查名为 `fname` 的文件是否存在。

```
185. int fok(char *fname)
186. {
187.     struct stat buf;
188.     return (stat(fname, &buf) == 0);
189. }
```

`mustlink` 用于标记是否需要链接操作，如果使用了“`-c`”选项，那么不需要链接，此时 `mustlink=0`。

```
190. int mustlink = 1;          /*< Becomes 0 if -c parameter is specified */
```

`keep` 用于标记是否需要保留经过 `ompi` 变换后的代码，如果使用了“`-k`”选项，那么将会保留变换的后的源代码，此时 `keep=1`。

```
191. int keep = 0;            /*< Becomes 1 if -k parameter is specified */
```

`verbose` 模式标记（冗余输出模式），初始化为 0。

```
192. int verbose = 0;
```

```
193. /*< Collection of bits to enable/disable some instrumentation.
```

```
194.     Using POMP by default is annoying, though. So we give an
195.     unofficial extra parameter "--pomp-enable" to
196.     explicitly enable POMP. Now if a user wants to enable
197.     POMP but disable barrier monitoring he should
198.     specify --pomp-enable --pomp-disable=barrier
199. */
```

性能统计的选项可以由“`--pomp-enable`”选项来控制，对应于 `pomp_disable_mask`，实际上 `ompi1.0.0` 并没有实现 POMP 功能。

```
200. int pomp_disable_mask = /* 0; */ -1;
```

下面定义的变量是 ompicc 中类型为 arglist_t 的所有变量。

```
201. arglist_t files;           /**< Files to be compiled/linked */
                                需要被编译的源代码文件列表
202. arglist_t goutfile;        /**< Output, -o XXX */
                                编译的输出文件
203. arglist_t prep_args;      /**< Preprocessor args */
                                预处理的参数
204. arglist_t link_args;      /**< Linker args */
                                链接参数
205. arglist_t scc_flags;       /**< Remaining system compiler args */
                                其他系统编译参数
206. char curdir[SLEN];        当前工作路径,全局变量
```

下面的函数 parse_args() 用于处理 ompicc 的命令行参数。它们包括指定 ort 库的“-ort=...”、参数选项“-c/L/I/D/U/o/k/v”、版本信息“--version”和源文件列表。运行库名保存在 ortlibname 中。“-c/L/I/D/U/o/k/v”的信息保存在 link_args、pre_args、goutfile、scc_flags（前面的变量都是 arglist_t 类型）、keep、verbose 中；源文件列表保存在 arglist_t 类型的 files 中。

```
207. void parse_args(int argc, char **argv)
208. {
209.     int d, ortlib = 0;          ortlib=1 表示用命令行参数 “--ort=xxx” 指定了线程库
210.     char *parameter;         进行参数分析时的这在处理的临时参数字符串
211. #if 0                      #if 0 部分的代码无效
212.     static char *pomp_enable = "--pomp-enable";
213.     static char *pomp_disable = "--pomp-disable=";
214.     static char *pomp_region_name[11] = {
215.         "atomic", "barrier", "critical", "for",
216.         "master", "parallel", "region", "section",
217.         "sections", "single", "sync"
218.     };
219.     static int  pomp_region_mask[11] = {1,2,4,8,16,32,64,128,256,512,-1};
220.     int pomp_enabled = 0;
221. #endif
222.
223.     getcwd(curdir, SLEN);      获取当前工作路径, 保存到 curdir 变量中
```

下面的参数链表初始化只是将链表的头尾指针置为 NULL， arglist_init() 函数参见前面定义，第 129 行。

```
224. arglist_init(&files);      被编译文件列表初始化
225. arglist_init(&goutfile);    输出文件初始化
226. arglist_init(&prep_args);   预处理参数初始化
227. arglist_init(&link_args);   链接参数初始化
228. arglist_init(&scc_flags);   其他编译参数初始化
```

```

229. argv++;                                跳过“ompicc”字串
230. argc--;                                 跳过第一个参数

231. while (argc) {                         逐个参数分析直到全部完成 (argc==0)
232.     d = 0;                               处理过的参数个数计数值
233.     parameter = argv[0];                 获取一个当前参数字符串
234. #if 0                                     以下直到#endif 都是无效代码
235.     if (strncmp(parameter, pomp_enable, strlen(pomp_enable)) == 0)
236.         pomp_enabled = 1;
237.     else
238.         if (strncmp(parameter, pomp_disable, strlen(pomp_disable)) == 0)
239. {
240.     parameter += strlen(pomp_disable); /* pass the "--pomp-disable=" part */
241.     do
242.     {
243.         for (i = 0; i < 11; i++)
244.             if (strncmp(parameter, pomp_region_name[i],
245.                         strlen(pomp_region_name[i])) == 0)
246. {
247.             pomp_disable_mask |= pomp_region_mask[i]; /* set the mask */
248.             parameter += strlen(pomp_region_name[i]);
249.             if (*parameter == ',') /* e.g. --pomp-disable=atomic,region */
250.                 parameter++;
251.             break;                  /* exit for */
252.         }
253.     } while (i < 11 && *parameter != '\0');
254. }
255. else
256. #endif
257.     if (strncmp(parameter, "--ort=", 6) == 0)    参数为“—ort=”
258.     {
259.         strncpy(ortlibname, parameter+6, 511);
260.         ortlib = 1;                            将“--ort”后面的字符串拷贝到 ortlibname 变量中
261.     }
262.     else                                    除了“—ort=”开头的选项,
263.         if (argv[0][0] == '-')                /* option */
264.             switch (argv[0][1])              如果是“-”开头的选项
265.             {
266.                 case 'c': mustlink = 0; break;      则可以是“c/l/L/I/D/U/o/k/v”之一
267.                 case 'l': d = append_arg(&link_args, argc, argv, 1); break;
268.                 case 'L': d = append_arg(&link_args, argc, argv, 1); break;
269.                 case 'I': d = append_arg(&prep_args, argc, argv, 1); break;

```

下面的各种参数都借助于 `append_arg()` 函数来处理，参见前面定义，见 142 行。

```

267.                 case 'l': d = append_arg(&link_args, argc, argv, 1); break;
268.                 case 'L': d = append_arg(&link_args, argc, argv, 1); break;
269.                 case 'I': d = append_arg(&prep_args, argc, argv, 1); break;

```

```

270.         case 'D': d = append_arg(&prep_args, argc, argv, 1); break;
271.         case 'U': d = append_arg(&prep_args, argc, argv, 1); break;
272.         case 'o': d = append_arg(&goutfile, argc, argv, 1); break;
273.         case 'k': keep = 1; d = 0; break;
274.         case 'v': verbose = 1; d = 0; break;
275.         default:
276.             d = append_arg(&scc_flags, argc, argv, 0);

```

其他选项都归入到编译选项

如果带有“--version”选项，则除了前面执行 default 分支将参数添加到 scc_flags 里面外，还在 ompicc 执行到这里的时候输出版本信息。

```

277.         if (strcmp(argv[0], "--version") == 0)
278.         {
279.             printf("%s ", VERSION);
280.             fflush(stdout);
281.             if (strcmp(COMPLIER, "gcc") == 0)
282.                 system("gcc --version");
283.             else printf("\n");
284.             _exit(0);
285.         }
286.     }
287.     else
288.     {
289.         d = append_arg(&files, argc, argv, 0); 将文件名保存到 files 变量
290.         if (!fok(files.tail->val)) 确定文件存在
291.             ompicc_error(1, "file %s does not exist\n", files.tail->val);
292.     }

```

前面变量 d 用于记录本次 while 循环所处理的参数个数（d 被 append_arg() 所修改），下面参数个数要减少 d，参数字符串数组要往后移动 d 个字符串。

```

293.     argc = argc - 1 - d;
294.     argv = argv + 1 + d;
295. }
296. #if 0
297.     if (!omp_enabled) 当前还不支持 POMP
298.        omp_disable_mask = -1;
299. #endif

300.     if (!ortlib) 没有指定线程库，则用缺省的线程库
301.         strcpy(ortlibname, "default"); /* Default lib to use */
302. }

```

fext_fun() 用于获取文件名的扩展名（后缀名），根据文件名中“.”的位置而截取后缀字符串。

```

303. char *fext_fun(char *fname)
304. {

```

```
305.     char *s;
306.     s = strrchr(fname, '.');
307.     if (s == NULL)
308.         return "";
309.     else return s + 1;
310. }
```

`fzapext()`用于将文件的扩展名截断，方法是将文件名中的“.”的位置截断（填写字符串结束符“\0”）。

```
311. void fzapext(char *fname)
312. {
313.     char *s;
314.     s = strrchr(fname, '.');
315.     if (s != NULL) *s = 0;
316. }
```

13.3 编译部分

紧接着变量声明和参数处理函数的是编译函数 `ompicc_compile()`。在 `ompicc` 的 `main()` 函数里会根据变量 `files` 里面记录的文件名，逐个调用 `ompicc_compile()` 函数进行编译处理，所以 `ompicc_compile()` 每次只须处理一个源文件，文件名以 `char*` 类型的 `fname` 变量传入。如果在文件列表中有后缀为“.o”的文件名，说明该文件不需要编译直接返回即可，如果需要的话还将在后续的 `ompicc_link()` 中进行链接。

`ompicc_compile()` 根据传入的文件名进行判断，如果是“.o”文件则不需要处理直接返回即可。如果是源代码则将文件名分成路径、文件名（不含扩展名部分），然后再形成输出文件名等。随后生成命令行字符串“`gcc -E.....`”并用 `system()` 函数来执行这个外部命令完成预处理过程，输出“`****.pc`”文件。再往后则形成命令行字符串“`ompi ****.pc`”并用 `system()` 函数来执行这个外部命令完成 OpenMP 编译制导指令的代码转换工作，输出“`****_ompi.c`”文件。最后形成命令字符串“`gcc ****_ompi.c -o ****`”实现 C 语言编译。

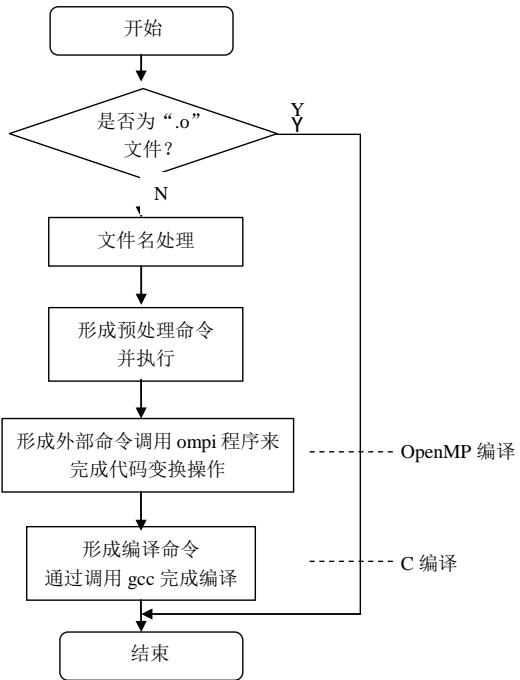


图 13.2 OMPi 编译的完整处理流程

下面是相应的源代码。

```

317. ****
318. *
319. *      COMPILING
320. *
321. ****

322. void ompicc_compile(char *fname)    fname 由 main()根据 files 获取而传入
323. {

```

13.3.1 文件名处理

该函数的开头部分是关于文件名的处理。下面的 `tmp` 为临时文件名，`bfile` 为文件名（不带路径名、不带扩展名），`filedir` 为输入文件的路径名（目录名），`outfile` 保存用于 `ompi` 变换后的输出文件名（.c 文件）。下面的代码将从传入的 `fname` 字符串中产生出上述各种文件名。

```

324. char tmp[SLEN], bfile[SLEN], filedir[SLEN], curdir2[SLEN];
325. char outfile[SLEN];           ompi 变换后的输出文件名
326. char cmd[LEN], strscce_flags[LEN], strgoutfile[LEN];
327. char tmp2[SLEN];
328. int res;                   用于记录调用 system 命令后的返回值

329. if (strcmp(fext_fun(fname), "o") == 0) /* skip .o files */   参见 303 行
330.     return;           源文件名为*.o 则不用编译

331. strcpy(tmp, fname);       拷贝输入文件名到 tmp 中

```

```

332. strcpy(bfile, get_basename(tmp)); 获得文件名（不带路径），参见 67 行
333. fzapext(bfile); 去掉 bfile 的扩展名

334. strcpy(tmp, fname); 拷贝输入文件名到 tmp 中
335. strcpy(filedir, get dirname(tmp)); 将目录名拷贝到 filedir 中，参见 90 行

336. sprintf(outfile, "%s_ompi.c", bfile); 输出文件名，形如 xxxx_ompi.c

337. chdir(filedir); 进入到输入文件所在目录
338. getcwd(curd़ir2, SLEN); 将文件所在目录存放在 curdir2 变量中

```

下面开始对 fname 文件进行预处理。

```

339. /* Preprocess
340. */
341. strarglist(tmp2, &prep_args); 将预处理选项形成字符串 tmp2, 下面马上要用到。
strarglist() 见 170 行

```

13.3.2 预处理

下面的代码用于形成外部命令的字符串。该命令形如 “gcc -std=gnu99 -E -U__GNUC__ -D_OPENMP=200505 -D_POMP=200203 -D_REENTRANT -D_REENTRANT -I/usr/local/include/ompi xxxx.c > xxxx.pc”的命令。其中 PREPROCESSOR 是预处理器，CPPFLAGS 是 C 的预处理参数，IncludeDir 是头文件目录，tmp2 此时是预处理选项（参见 341 行）。

这一小段代码仅用于 cygwin 环境，不做分析。

```

342. #if defined(__SYSOS_cygwin) && defined(__SYSCOMPILER_cygwin)
343. /* Hack for CYGWIN gcc */
344. sprintf(cmd, "%s -U__CYGWIN__ -D__extension__= -U__GNUC__ -D_OPENMP=200505"
345.           " -D_POMP=200203 %s"
346.           "-I%s %s %s.c > %s.pc",
347.           PREPROCESSOR, CPPFLAGS, IncludeDir, tmp2, bfile, bfile);
348. #else

```

下面是 linux 系统下可用的代码（非 cygwin 环境）

```

349. sprintf(cmd, "%s -U__GNUC__ -D_OPENMP=200505 -D_POMP=200203 %s "
350.           "-I%s %s %s.c > %s.pc",
351.           PREPROCESSOR, CPPFLAGS, IncludeDir, tmp2, bfile, bfile);
352. #endif

```

对应于非 cygwin 部分代码

```

353. if (verbose) verbose 模式则要输出上面的 cmd 命令字符串
354. fprintf(stderr, "====> Preprocessing file (%s.c)\n [ %s ]\n", bfile, cmd);

```

下面启动 cmd 命令完成对源文件的预处理。

```

355. if ((res = system(cmd)) > 0) { 执行 cmd 命令，并检查其结果
356.     chdir(curd़ir); 回到启动 ompicc 时的目录

```

```
357.     _exit(res);
358. }
```

13.3.3 代码变换

预处理后，接着进行源代码变换。

```
359. /* Transform
360. */
361. #ifdef DEBUGGING
362.     fprintf(stderr, "pomp_disable_mask = %d\n", pomp_disable_mask);
363. #endif
```

这部分是变换的核心部分

下面的代码用于形成形如：“ompi xxxx.pc __ompi__ > xxxx_ompi.c”的命令，RealOmpiName 是从 OmpiName 拷贝过来的（一般就是“ompi”），而 OmpiName 则是通过 compiler/Makefile.am 中的宏定义确定的。如果在 CFLAGS 中出现了“OMPI_MAIN=LIB”选项，则在调用 ompi 的时候带有“--nomain”表示编译的是库，没有 main() 函数的。如果在 CFLAGS 中出现了“OMPI_MEMMODEL=PROC”选项，则说明底层需要用到基于进程的 ort 库，此时在调用 ompi 时将带有“--procs”命令行参数。

变换完成后还将删除预处理产生的文件。

```
364. sprintf(cmd, "%s %s.pc __ompi__%s%s> %s", RealOmpiName, bfile,
365. strstr(CFLAGS, "OMPI_MAIN=LIB") ? " --nomain " : "",
366. strstr(CFLAGS, "OMPI_MEMMODEL=PROC") ? " --procs " : "", outfile);    生成命令字符串

367. if (verbose)                                verbose==1 则将命令行打印输出
368.     fprintf(stderr, "====> Transforming file (%s.c)\n"
369.                 " [ %s ]\n", bfile, cmd);
```

执行 cmd 命令完成对源代码的变换输出相应目标文件。

```
370. res = system(cmd);                          执行源代码的 openmp 转换
371. res = WEXITSTATUS(res);
372. sprintf(cmd, "rm -f %s.pc", bfile);        生成用于删除前一个步骤生成的 xxxx.pc 源代码的
373.                                         命令行字符串
374.                                         执行删除操作

375. if (pomp_disable_mask == 1) {
376.     printf("Ready to link, press enter...\n");
377.     getchar();
378. }
```

如果 ompi 执行的返回结果是 33，说明该输入文件没有 OpenMP 编译制导指令。对于非 OpenMP 代码，不需要特殊处理，只是加了一个开头的行。这些功能也可以在 ompi 中统一处理。

```
378. if (res == 33) {                           /* no pragma omp directives */
379.     FILE *of = fopen(outfile, "w");
380.     if (of == NULL) {
```

```

381.     fprintf(stderr, "Cannot write to intermediate file.\n");
382.     _exit(1);
383. }
384. fprintf(of, "# 1 \"%s.c\"\n", bfile);      /* Identify the original file */
385.                                         输出文件的第一行是一个标记
386. fclose(of);
387. sprintf(cmd, "cat %s.c >> %s", bfile, outfile);
388.                                         cmd 命令: 将原来的文件紧接着标记行拷贝即可
389.                                         执行 cmd 命令
390. system(cmd);
391. }
392. else if (res > 0) {                      其他情况对应出错
393.     if (!keep)
394.         unlink(outfile);                  不需要保留的话就删除掉
395.     chdir(curdir);
396.     _exit(res);
397. }

```

13.3.4 C 编译

完成 OpenMP 源代码变换后，下面就需要进行 C 编译了。

```

394. /* Compile the xformed file
395. */
396. chdir(curdir);                          进入到当前目录
397. strarglist(strscc_flags, &scc_flags);    将编译选项转变成字符串形式

```

下面的语句用于生成形如：“gcc -std=gnu99 ./xxxx_ompi.c -c -O3 -I/usr/local/include/ompi”的命令字串。其中 **COMPILER** 是编译器名称，**filedir** 是文件目录名，**outfile** 是 ompi 变换后的源文件名，**tmp2** 仍然是预处理选项字符串（参见 341 行，此处应该不再必需），**strscc_flags** 是编译选项字符串。

```

398. sprintf(cmd,
399. "%s %s/%s -c %s -I%s %s %s",
400. COMPILER, filedir, outfile, CFLAGS, IncludeDir, tmp2, strscc_flags);

                                         如果 verbose==1，则打印出对应的命令字符串。
401. if (verbose)
402.     fprintf(stderr, "====> Compiling file (%s/%s):\n [ %s ]\n",
403.               filedir, outfile, cmd);

404. res = system(cmd);                     执行编译命令
405. if (!keep)                            如果 keep==0，则不保留变换后的代码
406. {
407.     chdir(filedir);                   进入到输入文件目录
408.     unlink(outfile);                 回到当前目录（启动 ompi 的目录）
409.     chdir(curdir);
410. }

```

```

411.     if (res > 0)
412.         _exit(res);

413.     strglist(strgoutfile, &goutfile);

414.     sprintf(tmp, "%s_ompi.o", bfile);
415.     if (goutfile.head != NULL && mustlink == 0)
416.         rename(tmp, goutfile.head->val);
417.     else {
418.         sprintf(tmp2, "%s.o", bfile);
419.         rename(tmp, tmp2);
420.     }
421. }
```

13.4 链接部分

再接下来的是链接功能的代码。函数`ompicc_link()`是对`ompicc_compile()`编译输出输出文件（.o文件）以及`ompicc`命令行中的“.o”进行链接，因此`ompicc_link()`需要从`files`变量（`arglist_t`类型）中获知有哪些文件进行了编译。然后形成链接命令字符串再调用`system()`来执行即可。需要链接的文件名列表的处理过程需要区分`files`中给出的“.o”文件和“.c”文件的区别，前者直接记录下来即可，后者需要将“.c”后缀修改为“.o”后缀。

```

422. ****
423. *
424. *      LINKING
425. *
426. ****

427. void ompicc_link()
428. {
429.     arg_t *p;
430.     char cur_obj[SLEN], tmp[SLEN], cmd[LEN];
431.     char objects[LEN], *obj;
432.     char strgccargs[LEN], strlinkargs[LEN], strgoutfile[LEN], strprepargs[LEN];
433.     int len, is_tmp;
434.     char rm_obj[LEN];

435.     obj = objects;
436.     *obj = 0;
437.     strcpy(rm_obj, "rm -f ");           // 删除临时*.o 文件的命令
```

下面开始遍历所有的输入文件名，逐个分析，形成需要链接的“.o”文件列表字符串`objects`（以及相应的指针`obj`），其中的*.c形式的文件名需要改为*.o的文件名。如果是从*.c改名而来的临时*.o，则需要在编译完成后进行删除，它们保存在`rm_obj`字符串中。

```
438.     for (p = files.head; p != NULL; p = p->next)
```