

引用格式:刘文琦,王竹平.一种面向多核 DSP 并行编译的设计与实现[J].微电子学与计算机,2019,36(1):51-55. [LIU W Q, WANG Z P. Design and Implimentation of Parallel Compiler for Multicore DSP[J]. Microelectronics & Computer, 2019, 36 (1):51-55.]

一种面向多核 DSP 并行编译的设计与实现

刘文琦,王竹平

(西安微电子技术研究所, 陕西 西安 710065)

摘 要: OpenMP 支持并行递增开发,已成为目前用于共享内存系统上的一套事实主流并行编程标准.为了在多核 DSP 上实现对 OpenMP 的支持,设计了一个对 OpenMP 程序面向多核 DSP 的并行编译器,主要工作成果包括翻译器和运行时两部分,翻译器将源文件中的 OpenMP 指令转换为运行时中的函数调用,运行时提供具体实现.如何设计并行策略将计算任务分配在各个核上是并行编译的核心问题,对应在 OpenMP 标准中为并行域的概念.编译器通过对并行指令的变换及运行时的支持,实现了主从核的并行执行,对并行编译器的设计具有指导意义.

关键词: 多核 DSP;并行编程;OpenMP;并行编译器

中图分类号: TP314

文献标识码: A

文章编号: 1000—7180(2019)01—0051—05

Design and Implimentation of Parallel Compiler for Multicore DSP

LIU Wen-qi , WANG Zhu-ping

(Xi'an Microelectronics Technology Institute, Xi'an 710065, China)

Abstract: Supporting incremental parallelization, OpenMP has become the de facto mainstream standard of parallel programming on shared memory processors. To enable parallel programming with OpenMP standard on multicore DSP, a compiler is designed for OpenMP programs running on multicore DSP . Our main work consists of a translator and runtime. The translator part converts the OpenMP directives into the function interface in runtime where the concrete implementation is provided. How to design the tactic to distribute computing task among the cores is the main concern of parallel compilation, which corresponds to the concept of parallel region in OpenMP. By means of the translation of parallel directives and support provided by runtime, parallel executing of master and slave cores is achieved, which offers instructive meaning to the design of parallel compilation.

Key words: multicore DSP; parallel programming; openMP; parallel compiler

1 引言

面对计算需求日益膨胀的现状,对单核处理器的时钟频率的提升已经达到了瓶颈,处理器生产商则通过增加处理器的内核数来补偿为减少功耗而降低时钟频率带来的性能影响,使得多核处理器显现出相较于单核明显的计算优势,随着更多应用需要在多核平台上开发,并行编程模式逐渐受到程序员的青睐.并行的概念已经在操作系统中以线程的方式体现,然而

以 Pthread 为代表的线程库更倾向于任务并行,并且在使用时需要程序员在与线程相关的底层操作上花费较多的精力,为了使程序员更专注于业务的流程逻辑上,目前已经提出了一些易于使用的并行编程规范,比如基于共享内存的 OpenMP、基于消息传递模式的 MPI 等等^[1].为了在多核 DSP 上提高并行编程的效率,本文详细介绍了编译器对多核在并行域中的调度和任务分配,说明如何在多核 DSP 平台上以 fork-join 模式实现并行计算.

2 OpenMP 标准及并行编程模型

2.1 OpenMP 标准简介

OpenMP 标准提供一套与硬件平台无关的基于共享存储模型的编译指令和应用编程接口,可添加在 C/C++ 和 Fortran 语言编写的程序中显示地说明一个串程序的并行方式. 它应用了多线程技术,采用 fork-join 模式实现并行,由于其可移植性良好、易于使用、支持并行的增量开发,目前已成为共享存储模型中的主流标准^[2].

OpenMP 中包括四类指令和子句:并行域创建、任务分担、同步与互斥、数据环境控制,应用程序接口用于设置或返回并行域中的相关信息^[3]. 并行域创建指令声明一个并行域,其下的代码块中的语句将由多线程并行完成;任务分担指令指明多核共同完成并行域中的任务,可以是迭代任务 for 循环,或者非迭代任务块;同步与互斥指令用来协调多核在执行过程中的时序制约关系,以及确保对共享资源的读写正确;数据环境控制子句用来指明在并行域中变量的共享属性.

2.2 OpenMP 并行编程模型

下面展示了一个在标准 C 语言基础上添加 OpenMP 指令编写的并程序,其意图是为一个数组赋值,for 语句中的 100 次循环将分配由不同的线程执行. #pragma omp parallel 指令下的语句块为并行域,由多线程执行,其中调用了 OpenMP 标准的 API 设置线程数为 8, #pragma omp for 指令意味着 8 个线程共同完成其后的 for 循环,数组 a 由于是线程共享数据,所以每个线程对数组成员的赋值是互相可见的. 在并行域之外,整个程序均只有单线程执行^[4].

```
void main( ){\n    int i, a[100];\n    #pragma omp parallel\n    {\n        omp_set_num_threads(8);\n        #pragma omp for\n        for( i=0; i<100; i++)\n            a[i] = i;\n    }\n}
```

OpenMP 基于 fork-join 模型实现多线程的并行,图 1 为 fork-join 的模型图,主线程遇到 OpenMP 标准中的并行域创建指令后,会在并行域中的代码执行前生成多个从线程(即 fork 操作),主

从线程由系统分配在不同的处理器上,并行执行并行域中的代码,并行域结束时执行 join 操作,进入串行区,程序中恢复至只有主线程执行的状态. 当再次产生并行域时,重复上述步骤.

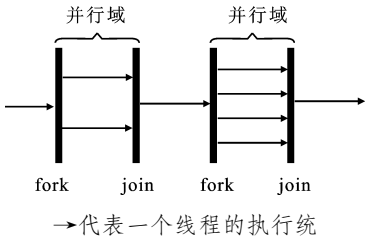


图 1 fork-join 模型图

3 并行编译方案设计及翻译变换

3.1 硬件平台简介

本文实现的并行编译器基于 TI 的 TMS320C6678 DSP(简称 C6678)硬件平台,这款芯片基于 KeyStone I 架构集成了 8 个结构相同的处理核,每个核可独立地读取并执行指令,主频最高可达 1.25 GHz,各自均有 32 kB 的 L1P cache,32 kB 的 L1D cache 和 512 kB 的本地 L2 cache,此外所有核可共同使用 4 MB 的多核共享内存,可存放程序或数据.

3.2 多核并行方案

为了在多核平台上实现 fork-join 模型,我们设计了如图 2 所示的主从核运行方案,程序一旦开始运行,主从核均启动,但执行不同的流程,主核执行 OpenMP 程序中的串行代码,从核则等待主核发送任务. 当主核进入并行域后,类似 fork 操作生成多个任务线程,然后逐一将任务发送给各个从核,从核收到任务后,开始执行,一旦完成任务需要立即给主核反馈. 主核在向从核分配完任务后,执行自己分属的任务,完成后等待从核的消息,直到确认所有从核均已完成任务,才退出此并行域(相当于 join 操作),继续执行串行代码. 从核则在任务完成后再次进入等待. 如果再次进入并行域,则重复执行上述流程.

3.3 编译流程及变换操作

整套编译系统由翻译器、运行时和本地编译器构成,本文的主要工作为翻译器和运行时的设计. 编译系统对 OpenMP 程序的处理流程如图 3 所示,输入为 OpenMP 程序(简称源 1),即含 OpenMP 指令的.c 文件,翻译器将源程序中的 OpenMP 指令翻译为在运行时中实现的库函数,生成含运行时接口的.c 文件(简称源 2),本地编译器采用 TI 提供的编译器 cl6x 对源 2 进行编译,结合运行时链接,最终生成以 C6678 DSP 为运行平台的可执行文件.

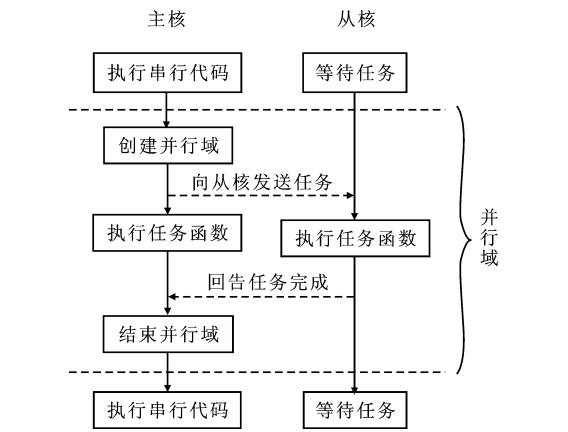


图 2 主从核并行模式



图 3 并行编译器对 OpenMP 程序的处理流程

编译器中包括了分析和变换两个步骤,源 1 经过词法、语法、语义分析后生成抽象语法树 (Abstract Syntax Tree, AST) 形式的中间代码,再对 AST 进行变换,反向输出源 2,这一过程中我们借助 Lex 和 Yacc 工具完成了词法和语法分析,并在执行语义动作的过程中完成对 AST 的构建。

如图 4 为一个简单 OpenMP 程序对应的 AST,每一个方框均代表一个 AST 结点,根据程序中不同类型的语句(如变量声明、函数定义、表达式等),设计了不同种类的 AST 结点,各类结点中记录的信息不同,比如在图 4 中,函数定义的 AST 结点中记录的是函数名(decl 成员)和函数体(body 成员),函数调用属于表达式中的一类,对应的 AST 结点中记录表达式的左值(成员 left)和右值(right 成员)。OpenMP 指令对应一种类型的结点,在变换的步骤中会重点处理。这些结点互相连接组织成树的形式,所以整个程序可用最顶端的一个 AST 结点来表示。

在根据源 1 建立起 AST 后,开始进行变换。变换在对整棵 AST 遍历的过程中完成,但大多数对应标准 C 语言语句的 AST 结点不需要变换,只有对 OpenMP 结点才需要处理,具体操作为将 OpenMP 结点对应的子树进行裁剪,替换为预期变化后代码的 AST 子树。对应图 4 是将虚线框中的 OpenMP 结点从原 AST 中拆下来,替换为新的子树,同时要保证在不改变原 AST 中其他结构的前提下,将新的子树完全嵌入。在构建出新的 AST 后,再次遍历输出字符形式的代码程序,即产生源 2。

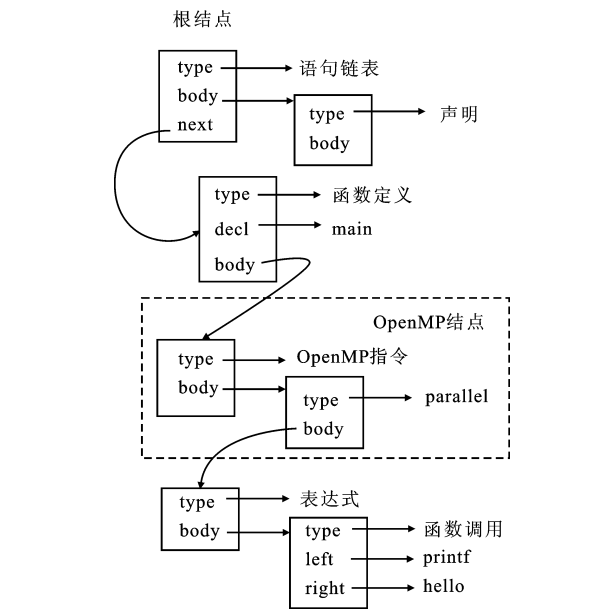


图 4 程序 AST 示意图

3.4 变换后的代码形式

本节将介绍源 1 中的 OpenMP 指令经过编译器变换后的代码形式。首先,在源 2 中 # pragma omp parallel 指令替换为并行域创建函数接口,其次该指令下的代码块(即并行语句)被抽离出来封装成新的任务函数,最后依据源 1 中声明的并行域中变量的共享属性构建数据环境。

结合一个 OpenMP 程序的翻译结果进一步阐释,在下面的源 1 程序中,并行域外定义了一个数组变量 z,并行域内输出 z 中以执行线程的线程号为下标的元素的值,根据 OpenMP 标准,z 是并行域中的共享数据。源 1 中的 main 函数经过翻译在源 2 程序中对应两个部分:原函数和并行域的任务函数,源 2 的 main 函数中包括源 1 中的串行语句、共享变量的信息封装、并行域创建函数接口。新生成的任务函数 _thrFunc0_ 中包括了并行语句,以及并行语句前的数据环境准备。本例中的并行语句较为简单,如果源 1 中的并行域代码里有其他 OpenMP 指令,则编译器会在源 2 中将它们替换为运行时中相应的函数。

```
/* * * * * * * * * * * 源 1 程序 * * * * * * * * * * */
int main( )
{
    int z[5] = {1,2,3,4,5}; } 串行语句
    printf("serial\n");
    #pragma omp parallel num_threads(5)
    {
        int j = omp_get_thread_num( );  printf("%d",z[j]); } 并行语句
}
```

```
}
/* * * * * * * * * * * 源 1 程序 * * * * * * * */
/* * * * * * * * * * * 源 2 程序 * * * * * * * */
int main( )
{
    int z[5] = {1,2,3,4,5};
    printf("serial\n");
    /* 在 shvars 中记录共享变量的信息; */
    struct _shvt {
        int (* z)[5]
    } shvars = { &z };
    execute_parallel(5, _thrFunc0_, shvars);
}

void _thrFunc0_(void * me)
{
    /* 从 shvars 获得共享变量的信息; */
    struct _shvt {
        int (* z)[5];
    } shvars ;
    shvars = get_shared_vars(me);
    int (* z)[5] = shvars->z;
    {
        int j = omp_get_thread_num();
        printf("%d\n", (* z)[j]);
    }
    return ;
}

/* * * * * * * * * * * 源 2 程序 * * * * * * * */
```



并行域创建函数 `execute_parallel` 只由主核调用,其具体实现由运行时提供,该函数有三个参数,分别为并行域中参与执行的线程数(本例中由 `num_threads` 子句声明)、任务函数名(本例中为 `_thrFunc0_`)和记录共享变量信息的结构体(本例中为 `shvars`)。由于并行域中的共享变量随程序的定义有较大的不确定性,所以为了便于以参数的形式传递,编译器将一个并行域中的共享变量全部封装在一个结构体中。

一个并行域对应一个任务函数,每一个进入并行域的核都会调用此任务函数,任务函数中在并行语句前需要重构源 1 中声明的数据环境,通过重新定义、初始化同名共享和私有变量实现。从上面的例子中可以看到,调用 `execute_parallel` 函数前记录共享变量信息的结构体与任务函数中构建数据环境采用的结构体完全一致,实际上通过 `get_shared_vars` 函数获得的结构体即为之前 `execute_parallel` 中传入的参数变量。

之所以要在并行域创建和任务函数间传递共享变量的信息,是因为每个核的执行上下文均存放在各自本地的内存中。如果一个变量声明为并行域中的私有变量,则不需要对其进行额外的处理,然而对

共享变量而言,为了使并行域中的各个核均能对称地访问,该变量需声明在共享区中并由主核记录其地址发送给从核,在任务函数的并行语句中通过地址间接访问。

4 运行时实现

本章介绍运行时对上述并行方案的支持,包括其中关键的数据结构,以及对 `parallel` 指令变换后在源 2 中生成的并行域创建函数的实现,还有从核在并行域中的执行流程。

4.1 运行时中模拟并行域的数据结构

运行时中对并行域的处理围绕两个数据结构展开,分别为用于记录每个核在并行域中状态的执行体结构以及主核向从核分发任务的线程结构。

• 执行体结构

运行时中,每个核在进入并行域后都会创建自己独立的执行体,其中记录了各自在并行域中的信息,比如自己在并行域中的编号、进入同一并行域中的总核数,等等。OpenMP 标准中的 API 可返回当前并行域中的总线程数、当前线程的线程号等等,均通过查询每个核的执行体中相关成员的值来实现。

执行体结构维护了进入同一并行域中的主从核之间的父子关系,主核创建并行域后,其执行体将成为此并行域中从核执行体结点的父结点。父执行体结点中记录了并行域中的共享信息,如任务函数、共享变量等,子结点可通过其父结点获取这些信息。

• 线程结构

线程是主核向从核发送任务的实际承载结构,其中封装了指向记录并行域信息的父执行体的指针及各自的线程编号。执行体与线程的关系如图 5 所示,线程 1、2、3 为主核发送给 3 个从核的任务,同一并行域内的线程中的指针指向同一个父执行体,所以接收到线程的从核将访问到相同的任务函数,线程之间通过编号区分。

线程的生命周期仅存在于在并行域中,为了避免频繁地创建和销毁线程带来的计算开销,在运行时中以链表的形式创建一组线程,组成线程池。每当创建并行域时,由主核从线程池中申请相应数目的线程发给从核,当从核执行完任务后,将各自从主核接收的线程重新放回至线程池中以备下一个并行域中使用。

主核对并行域中从核的任务完成状态进行监控也是通过执行体和线程结构共同实现。如图 5 所示,父执行体中记录了一个关键变量——“并行域中尚未完成任务的从核数”。当主核创建并行域时,会将

该变量的初始值设置为并行域中的线程数. 当主核将线程发送给从核后,从核执行完任务,则会通过父执行体访问到该变量,并在其当前值的基础上减 1. 因此该变量是多核共享的,且其值会随着从核的执行改变,主核通过该变量的值就可判断当前并行域中是否还有从核未执行完任务.

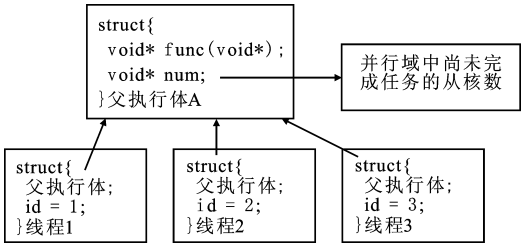


图 5 执行体与线程的关系图

4.2 并行域创建函数的实现

并行域创建函数由主核调用,实现向从核分发任务、自己执行任务以及并行域结束前的同步. 它的执行流程如下:

- (1)根据源程序中指定的并行域中的核数,从运行时的线程池中申请相同数目的线程,如果可提供的线程数不够,则终止,并返回提示信息.
- (2)设置整个并行域的信息,填写代表并行域的父执行体结点中的参数,包括任务函数、共享变量、进入此并行域的核数等等.
- (3)向从核发送任务:在申请到的线程中填写要发送的信息,主要包括图 5 中展示的指向父执行体结点的指针和线程编号,设置完毕后,如图 6 所示,将线程依次发送给相应的从核.

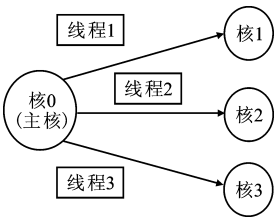


图 6 主核向从核发送任务

- (4)执行任务函数完成自己在并行域中分属的任务.
- (5)同步退出,等待并行域中的所有从核均执行完任务后,退出并行域. 具体实现为主核进入循环,直到执行体中“并行域中尚未完成任务的从核数”变量的值为 0,才跳出循环.

4.3 从核在并行域中的执行流程

主核在并行域创建函数中以线程的结构向从核发送任务,从核收到任务后执行进入并行域、调用任

务函数、退出并行域三项步骤,这三个环节均与接收到的线程中封装的父执行体有关.

- (1)进入并行域:建立与主核执行体结点间的父子关系,根据父执行体填写自己的执行体结构中的信息.
- (2)调用任务函数:通过访问父执行体中记录的并行域的任务函数指针,执行翻译器生成的任务函数.
- (3)退出并行域:将主核发送的线程放回线程池中,将父执行体中的变量“并行域中尚未完成任务的从核数”减 1,相当于向主核反馈已经完成任务的消息.

5 结束语

本文设计并实现了一种针对 OpenMP 程序在多核 DSP 平台上运行的并行编译器,通过将源程序中的编译制导指令翻译为运行时中的接口,用本地编译器结合运行时对翻译后的 C 程序进行编译链接,生成的可执行文件运行在多核 DSP 上可达到主从核并行执行并行域中代码的效果. 在充分利用多核 DSP 计算资源的同时,提高了程序员开发并行程序的编程效率. 后续工作可针对编译器的性能优化展开进一步的研究,比如对并行域中每个核分配的计算任务进行量化分析,加入更合理的任务分配策略,以达到计算核的负载均衡.

参考文献:

[1] 罗秋明, 明仲, 刘刚, 等. OpenMP 编译原理及实现技术[M]. 北京:清华大学出版社, 2012.

[2] Leonardo Dagum, Ramesh Menon. OpenMP: An industry-standard API for shared memory programming [J]. IEEE Computational Science & Engineering, 1998, 5(1): 46-55.

[3] Open MP Architecture Review Board. OpenMP4. 5 Specifications[EB/OL]. (2015-01-16). <http://www.openmp.org>.

[4] 王子聪. 基于 OpenMP 的多核 DSP 并行优化方法的研究及应用[D]. 湖南:国防科学技术大学, 2014.

作者简介:

刘文琦 女, (1993-), 硕士研究生. 研究方向为多核并行计算. Email: liuwenqi365@163.com.

王竹平 男, (1963-), 硕士, 研究员, 研究生导师. 研究方向为嵌入式计算机应用技术.