软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(4):981−1001 [doi: 10.13328/j.cnki.jos.005370]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种面向异构众核处理器的并行编译框架*

李雁冰, 赵荣彩, 韩　林, 赵　捷, 徐金龙, 李颖颖

(数学工程与先进计算国家重点实验室,河南 郑州　450001)

通讯作者: 李雁冰, E-mail: li.yanbing@outlook.com

摘　要: 异构众核处理器是面向高性能计算领域处理器发展的重要趋势,但其更为复杂的体系结构使得编程难的问题更加突出.针对这一问题,基于开源编译器 Open64,提出了一种面向异构众核处理器的并行编译框架,将程序自动转换为异构并行程序.该框架主要包括 4 个模块:任务划分模块用来识别适合进行加速计算的程序段,实现了嵌套循环的多维并行识别方法;数据布局模块完成数据在主存和 SPM 之间的布局,实现了数组边界分析和指针范围分析;传输优化模块实现了数据传输合并、传输外提、打包传输、数组转置等多种数据传输优化方法;收益评估模块在构建代价模型的基础上实现了一种动静结合的收益评估方法.并且,基于 SW26010 处理器,对该编译框架进行了实现,测试结果表明,该编译框架能够实现一些程序以面向异构众核结构的并行变换,且获得较好的加速效果.

关键词: 异构众核处理器;SW26010;并行编译;数据传输优化;OpenACC

中图法分类号: TP314

中文引用格式: 李雁冰,赵荣彩,韩林,赵捷,徐金龙,李颖颖.一种面向异构众核处理器的并行编译框架.软件学报,2019, 30(4):981−1001. http://www.jos.org.cn/1000-9825/5370.htm

英文引用格式: Li YB, Zhao RC, Han L, Zhao J, Xu JL, Li YY. Parallelizing compilation framework for heterogeneous many-core processors. Ruan Jian Xue Bao/Journal of Software, 2019,30(4):981−1001 (in Chinese). http://www.jos.org.cn/1000-9825/5370.htm

Parallelizing Compilation Framework for Heterogeneous Many-core Processors

LI Yan-Bing,　ZHAO Rong-Cai,　HAN Lin,　ZHAO Jie,　XU Jin-Long,　LI Ying-Ying

(State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China)

Abstract: Heterogeneous many-core processors become an important trend in high-performance computing area, but the issue that the sophisticated architecture complicates the programming is more significantly. To solve this problem, this study proposes a parallelizing compilation framework for heterogeneous many-core processors based on the open source Open64 compiler, automating the transformation from a sequential program to heterogeneous parallel code. The framework mainly comprises a work scheduling module that identifies the parallelizable regions and achieves a multi-dimensional parallelization recognition for nested loops; a data mapping module that maps data between the main memory and SPM and realizes array boundary analysis and pointer range analysis; a transmission optimizing module that implements optimizations by merging, hoisting and packaging data transmission, and transposing array; and a performance estimation module that proposes a dynamic-static hybrid method to analyze benefit based on the cost model for SW26010. The compilation framework is implemented on top of Sunway SW26010 processors, and experimental evaluations are conducted on numerous benchmarks. The experimental results show that the proposed framework can parallelize these applications and obtain a promising performance improvement on heterogeneous many-core platforms.

Key words: heterogeneous many-core processor; SW26010; parallelizing compilation; data transmission optimizing; OpenACC

∗基金项目: 国家自然科学基金(61702546); 国家高技术研究发展计划(863)(2014AA01A300)

Foundation item: National Natural Science Foundation of China (61702546); National High Technology Research and Development Program of China (863)(2014AA01A300)

收稿时间: 2016-12-13; 修改时间: 2017-01-23, 2017-04-28, 2017-05-29, 2017-07-14; 采用时间: 2017-08-17

© 中国科学院软件研究所　http://www.jos.org.cn

虽然处理器一直在遵循着摩尔定律快速发展,但是目前计算机系统的计算能力仍然不能满足人类社会的需求,在气象预报、石油勘探、材料合成、药物研究、地球模拟等众多领域仍然亟需更快的计算速度来进行科学研究.高性能计算已经成为继理论研究、科学实验之后的第三大科学手段,并且发挥的作用越来越重要[1].传统的单核处理器主要依靠提高主频来获得更快的计算速度,但是由于主频提高后带来的功耗和散热等问题的限制,主频已很难继续提高.因此,增加计算资源成为提高计算速度的主要方式,在单个芯片上集成更多的处理器核心已经成为当前处理器设计的主流.目前处理器核心的数目增长很快,尤其是面向高性能计算领域的处理器已经进入众核时代[2].相比通用的多核处理器,众核处理器在片上集成了超出数量级的简化核心,可以提供更高的计算能力、计算密度和性能功耗比.众核已被业界视为继单核、多核之后处理器发展的第三个时代,它为在今后的一段时间内继续维持摩尔定律提供了可能.2016 年 6 月,我国自主研制的"神威太湖之光"计算机发布,其峰值性能、持续性能、性能功耗比 3 项关键指标均居世界第一[3],"太湖之光"的核心器件之一便是自主设计的 SW26010 众核处理器.

按照集成处理器核种类的不同,众核处理器主要分为两类:同构众核处理器和异构众核处理器.同构众核处理器是在一个芯片内集成了多个结构相同、地位对等的处理器核.异构众核处理器则是在一个芯片内集成了多个不同结构的处理器核.目前,主流的商用众核处理器多为同构众核结构,如 Intel 的至强 Phi 协处理器[4]和 NVIDIA 公司的 GPGPU 系列的产品[5].同构众核主要作为加速器,与通用 CPU 一起构成高性能计算系统,对应用的计算密集部分进行加速.但这种方式存在一些不足:CPU 与众核处理器通过 PCI-E 接口连接,二者之间的数据传输有较大的延迟,并且数据传输带来了额外的功耗,松耦合的系统架构降低了高性能计算系统的组装密度.异构众核处理器则在同一芯片内集成了具有控制管理功能的通用处理器核心和大量用于加速计算的精简计算核心,能够实现较高的性能功耗比和计算密度,弥补了上述同构众核的不足[2].首先在处理器内部采用异构结构的是索尼、东芝和 IBM 等公司联合开发的 Cell 处理器[6].目前各主要处理器厂商已经开始异构众核处理器的开发或研究工作,如 AMD 公司的 APU[7]、NVIDIA 公司牵头的 Echolen[8]项目、Intel 公司牵头的 Runnemede 项目[9]等,采用异构众核处理器是未来构建高性能计算系统的重要趋势.

异构众核架构虽然具有诸多优势,但在单个计算节点内集成大量不同类型的处理器核心,使得体系结构更加复杂,这给程序设计和编译优化都带来了更大的挑战.从程序设计和编译优化的角度看,异构众核处理器结构主要有以下特点:一是异构众核处理器内集成了更多的处理器核心,能够提供更多的计算资源,需要发掘出应用程序中更多的并行性;二是异构众核处理器为了控制功耗,大多使用了软件管理的 SPM(scratch pad memory),编程时需要用户程序管理 SPM 空间,显式地控制数据在通用核心和加速计算核心之间的传输;三是异构众核处理器的 SPM 由软件管理,不同于以往的硬件 Cache,传统的面向 Cache 结构的编译优化技术不再适用.这种结构上的复杂性使得在异构众核处理器上的并行程序开发面临着巨大的挑战.

解决并行编程问题有两种基本方法:一种是程序员手工编写并行程序;另一种是使用并行化编译系统.程序员手工编写的并行程序在经过精心优化后往往有很高的运行效率,但手工编写并行程序需要程序员熟悉相关领域,精通并行编程模型,了解硬件的体系结构和指令集特点等,对程序员要求很高.而且,手工编写并行程序的效率很低,对于过去数十年来积累的大量优秀的应用程序,将其进行手工改写需要花费大量的时间和人力.并行化编译系统是一个自动发掘程序中潜在并行性并将其转换为并行程序的编译工具,相比手工编写并行程序,能够快速地完成串行程序到并行程序的转换,效率更高.基于高性能计算领域对并行程序的迫切需求以及并行化编译的巨大应用前景,大量的研究人员开始对并行化编译技术进行研究,并行化编译成为当前高性能计算领域的一个研究热点.但是由于并行化编译技术本身的复杂性,目前并行化编译得到的并行程序与手工编写的相比,运行效率仍有较大差距.

目前,在面向众核结构的并行化编译领域,大多数的研究工作是面向 GPU 平台的.比如,Lee 等人在文献[10]中实现了一个"源-源"的把面向共享存储结构的 OpenMP 程序转换成 GPGPU 程序的自动转换和优化框架,能够将已有的 OpenMP 程序移植到 GPU 众核平台.在文献[11]中又将其扩展为 OpenMPC 框架,通过添加指导语句和使用环境变量等方式实现了更多的性能优化.Han 等人[12]提出了一种基于指导语句的高级语言 hi CUDA,并利

© 中国科学院软件研究所 http://www.jos.org.cn

用“源-源”编译技术为用户提供了一种完成一般应用程序到 CUDA 程序转换的方式.Baskaran 等人[13]实现了一种通用串行 C 程序直接转换为 CUDA 程序的自动转换系统,与之类似,本文实现的也是一个将串行的 C 和 Fortran 程序直接转换为异构众核程序的编译框架.针对 Intel Phi 协处理器典型的研究工作是 Ravi 等人提出的优化编译器 Apricot[14],能够将已有的 OpenMP 程序和串行程序转换为适合 MIC 结构的 LEO(language extensions for offload)程序,并使用了多种访存及数据传输优化方法以及一个运行时代价模型来进一步优化性能.目前已有的针对异构处理器的研究大多是基于 Cell 处理器的.Eichenberger 等人[15]为 Cell 处理器设计的 Octopiler 编译器实现了 Cell 处理器代码的自动生成,并包括线程级并行、自动向量化等多种优化.王淼等人[16]针对 Cell 处理器提出了一个异构多核编译框架,该框架通过执行数据对齐、数据分布、计算分布等操作,将数据并行程序直接映射到 Cell 处理器上.

为了缓解在我国自主设计的异构众核处理器 SW26010 上遇到的编程难的问题,本文设计实现了一个面向异构众核处理器体系结构的并行化编译框架.该并行化框架基于开源的 Open64[17]编译器开发,采用“源-源”的方式将 C 和 Fortran 程序转换为 SW26010 上运行的 Sunway OpenACC 异构并行程序.该框架主要包括 4 个模块:任务划分模块用来识别适合进行加速计算的程序段,在该模块中首次提出了嵌套循环的多维并行的识别方法;数据布局模块完成数据在主存和 SPM 之间的布局,该模块将指针范围分析的方法应用到异构众核处理器的数据布局中,并提出了一种需求驱动的指针范围分析方法;传输优化模块针对 SW26010 处理器的结构特征实现了数据传输合并、传输外提、打包传输、数组转置等多种数据传输优化方法;收益评估模块通过构建面向 SW26010 处理器的代价模型,实现了一种将编译时静态获取的信息与程序运行时得到的信息相结合的动静结合的收益评估方法.本文的主要贡献包括:(1) 设计和实现了一个面向 SW26010 处理器的并行化编译框架,该框架的创新性工作有:提出了嵌套循环的多维并行识别方法、面向异构众核结构数据布局的需求驱动的指针范围分析方法和一种动静结合的收益评估方法,以及多种针对 SW26010 特性的数据传输优化方法;(2) 为 Open64 编译器提供了一个 OpenACC 代码生成模块,使其能够生成适于异构系统上的并行代码;(3) 为设计实现面向异构众核处理器的并行编译框架提供了一种可行的参考方案.

本文第 1 节介绍我们的研究的目标平台 SW26010 处理器及其使用的编程模型.第 2 节总体说明本文提出的并行编译框架.第 3 节~第 6 节详细介绍任务划分、数据布局、传输优化和收益评估模块的实现.第 7 节为实验测试与结果分析.第 8 节对本文进行总结.

1 研究基础

1.1 目标平台

本文的研究基于国产 SW26010 处理器,SW26010 是面向高性能计算领域开发的处理器,其结构如图 1 所示,芯片主要由 4 个核组(group)、片上互联网络(network on chip,简称 NoC)和系统接口(system interface,简称 SI)组成[18].单个核组内采用异构众核结构,包括 1 个管理核心 MPE(management processing element)、1 个运算核心簇 CPE cluster(computing processing elements cluster)和存储控制器 MC(memory controller).运算核心簇则包含 64 个运算核心,采用拓扑为 8×8 的簇通信网络进行连接.管理核心是功能完整的 64 位 RISC 核心,支持 32/64 位整数运算、单/双精度浮点运算和原子操作,可以高效处理程序的串行段,并完成计算资源和功耗的管理,提供监控、消息、文件、调试等服务.运算核心也是 64 位 RISC 结构,其指令集在管理核心指令集的基础上进行了精简,并针对运算核心结构特征增加了通道操作、寄存器通信、行列同步等特殊指令.

SW26010 的核组是进行计算资源管理的基本单位.单个核组的存储系统结构如图 2 所示[18],管理核心拥有 L1 和 L2 两级硬件 Cache,运算核心则支持软件管理的 SPM.同时,管理核心和运算核心还共享片外主存.运算核心的 SPM 与主存统一编址,管理核心和运算核心都可以通过访存指令访问 SPM 和主存,还可以通过 DMA 命令实现 SPM 和主存之间的批量数据传输.但是,处理器核访问不同存储器的延迟有很大差别,运算核心访问 SPM 的延迟远低于访问主存的延迟,而由于芯片面积及功耗的限制导致 SPM 的空间有限.在实际应用中,充分利用访问时延短的 SPM 才能充分发挥 SW26010 处理器的性能优势.

© 中国科学院软件研究所 http://www.jos.org.cn

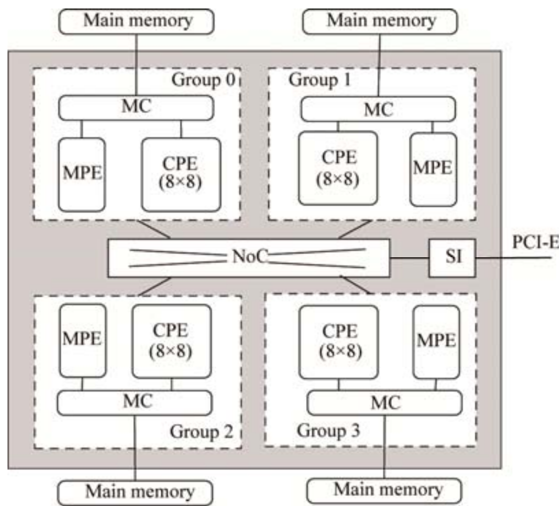

Fig.1 SW26010 processor structure diagram

图 1 SW26010 处理器结构图

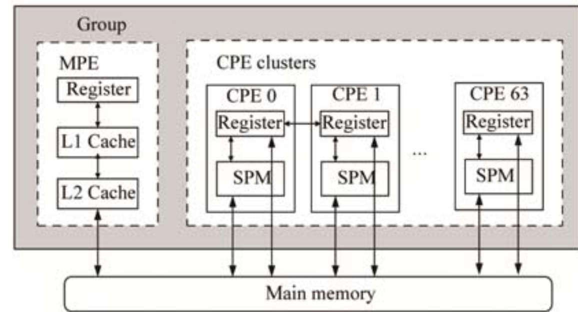

Fig.2 SW26010 memory hierarchy of each group

图 2 SW26010 单核组存储系统结构图

1.2 编程模型

SW26010 处理器 4 个核组之间的并行,既可以使用 OpenMP,也可以使用 MPI 实现;单个核组内管理核心和运算核心簇之间的并行则通过 Sunway OpenACC 实现[18].Sunway OpenACC 是在 OpenACC 标准[19]的基础上针对 SW26010 处理器的结构特性作了一些扩展.OpenACC 是一个针对加速设备的异构编程模型标准,通过编译指示将指定的程序段加载到加速设备上执行,程序的其他部分仍然在主处理上执行.Sunway OpenACC 对 OpenACC 的扩展主要有:针对零散数据传输的数据打包拷贝、针对不连续访存的数据转置拷贝等.

在“太湖之光”计算机上开发和移植程序,一个主要的工作就是添加 Sunway OpenACC 指示,实现程序在运算核心上的并行执行.Sunway OpenACC 程序要想获得好的性能,需要仔细分析哪些循环适合在运算核心上运行,还需要根据程序特征和处理器特性对程序的数据传输等作深入优化.本文的研究工作即是在编译器中通过程序分析,自动添加 Sunway OpenACC 指示.本文编译框架通过“源-源”方式生成的 Sunway OpenACC 程序,还需要经过 Sunway OpenACC 编译工具的二次编译.本文的编译框架判断哪些代码适合加载到运算核心进行加速计算,以及哪些数据是运算核心加速计算时需要的,而 Sunway OpenACC 编译工具则完成异构代码的生成,以及数据在存储系统中的分布.

2 并行编译框架

本文的并行编译框架用来开发应用程序在 SW26010 处理器单个核组内运行的并行性,通过在原始程序中自动插入 Sunway OpenACC 编译指示,将串行程序转换为异构并行程序.该框架基于开源编译器 Open64 实现,以 C 和 Fortran 程序作为输入,以 Sunway OpenACC 并行程序作为输出,主要包括任务划分、数据布局、传输优化、收益分析模块,如图 3 所示.

并行化编译之前首先需要对程序进行预优化,预优化包括常量传播、归纳变量替换、死代码消除、循环正规化等简单优化,为后续的分析和优化作准备,本框架使用了 Open64 中原有的预优化方法.任务划分模块用来识别适合在运算核心上进行加速的程序段,通常是计算量足够且可以并行执行的循环.数据布局模块根据数据流分析的结果,分析在运算核心上进行加速计算时哪些数据是运算核心需要的.传输优化模块则对数据在主存和 SPM 之间的传输过程进行优化,尽可能地减少不必要的数据传输和提高传输效率.在运算核心上并行执行程序需要一定的启动和数据传输等开销,并不是所有的循环并行执行都能获得收益,所以需要收益评估模块进行

© 中国科学院软件研究所 http://www.jos.org.cn

收益分析,避免计算量太小无法获得收益的循环被加载到运算核心上执行."源-源"翻译过程则将编译器中间语言表示形式转换为高级语言源程序,Open64 中有一个实验性的"源-源"翻译模块,本文框架在其基础上开发了用于生成 Sunway OpenACC 并行程序的"源-源"翻译模块,并做了大量的优化和 BUG 修复工作,使其能够正确翻译大型程序.目前"源-源"翻译模块仅支持 C 和 Fortran77 程序.

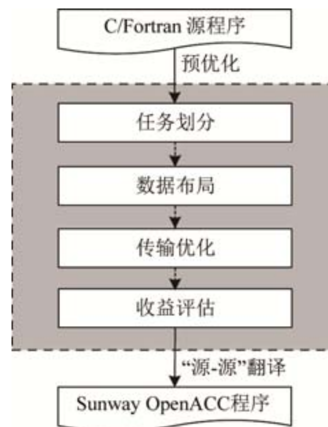

Fig.3 Parallel compilation framework

图 3 并行编译框架图

本文是一个"源-源"的编译框架,输出的是添加了 Sunway OpenACC 编译指示的 C 和 Fortran 程序,本文框架并不直接完成计算划分和数据分布,而是在程序中加入指导计算划分和数据分布的编译指示,指导 Sunway OpenACC 编译工具在二次编译时生成计算划分和数据分布代码.

3 任务划分

SW26010 处理器的性能优势主要来自于数量众多的精简运算核心.任务划分模块用来识别适合在运算核心上进行加速的程序段,并在中间表示中插入相应的指示.适合在运算核心上运行的程序段,通常是计算量较大且可以并行执行的循环.Open64 编译器能够自动生成 OpenMP 并行程序,在 LNO(loop nest optimizer)阶段已有一个能力较强的并行循环识别模块[20].Open64 的并行识别方法是一种一维并行识别方法,用于众核处理器时存在一定不足,当选择的并行维迭代数较少时可能导致严重的负载不均衡.图 4(a)所示的矩阵乘法核心代码通过 Open64 原有的并行识别阶段后生成的并行代码如图 4(b)所示,这里,矩阵的规模为 32,并行层 i 层只有 32 次迭代,在拥有 64 个运算核心的 SW26010 处理器上运行时无法做到负载均衡,会使部分运算核心处于空闲状态.

```
do i=1,32
   do j=1,32
      do k=1,32
         C(i,j )=C(i,j)+A(i,k)* B(k,j)
      enddo
   enddo
eNddo
```

(a) 矩阵乘核心

```
!$ acc parallel loop
 do i=1,32
    do j=1,32
       do k=1,32
          C(i,j )=C(i,j)+A(i,k)* B(k,j)
       enddo
    enddo
enddo
!$ acc end parallel
```

(b) 单维并行

```
!$ acc parallel loop collapse(2)
 do i=1,32
    do j=1,32
       do k=1,32
          C(i,j )=C(i,j)+A(i,k)* B(k,j)
       enddo
    enddo
enddo
!$ acc end parallel
```

(c) 多维并行

Fig.4 Multi dimension parallelization of nested loop example

图 4 嵌套循环多维并行示例

针对这一问题,本文在 Open64 已有并行识别方法的基础上提出了一种多维并行识别方法,在现有并行识别方法无法做到较好的负载均衡时,选择嵌套循环的多个维进行并行,将多个并行维的迭代空间合并后再做任务划分,减少负载不均衡对程序并行效率的影响.现有的解决这一问题的方法主要有两种:一种是通过循环交换把

迭代量大的可以并行的维交换到外层,这种方法的缺点是循环交换可能影响数据局部性,而且使用的范围受限,可能嵌套循环的各个维迭代数都比较少,这时循环交换无法解决问题;另一种是直接对循环的多个维进行合并以便有足够的迭代数,这种方法与本文提出的多维并行在本质上是类似的,但是需要对循环进行合并变换,较为激进,可能对后续优化产生较大影响.本文是目前最早在并行识别阶段直接将循环的多个维识别为并行的方法,有别于以往通过循环交换或者合并后仍然进行单层并行识别的方法.

在手工编写的并行程序中,多维并行的思想已有很多的应用.典型的例子是 OpenMP、OpenACC 等基于编译指示的并行编程模型中使用的 collapse 指示.OpenMP 和 OpenACC 中的 collapse 指示在语法和功能上基本相同,都是作为 loop 指示的子句,用来指定有多少层紧嵌套循环与 loop 指示相关联,这些循环的迭代空间将被合并后做任务划分,本质上即是多维并行.如图 4(c)所示,代码即表示将 i、j 两层循环合并后并行执行,此时进行并行任务划分的迭代空间变成了 32×32,在 SW26010 上运行时则可以有较好的负载均衡.

任务划分模块的主要贡献是在 Open64 单维并行识别算法的基础上首次实现了嵌套循环的多维并行识别.在进行并行识别前,会对循环次序按照程序局部性最优的原则进行排序,即访存跨步小的层位于内层.在实现嵌套循环多维并行时需要解决的两个关键问题是:第一,什么情况需要进行多维并行;第二,满足什么条件的多个循环层能够进行多维并行.针对第 1 个问题,需要进行多维并行识别的条件是当前识别的并行循环的迭代次数小于运算核心数目的 4 倍(即小于 256),并且迭代次数不能被 64 整除,这通常意味着没有好的负载均衡.这里遵循了一个基本假设,即不同迭代的计算量是相当的,这个假设在科学计算类程序中大多数时候是满足的.在这个假设下,当迭代数能被核心数整除时,不会有负载不均衡的情况发生;当迭代数目较大时,这里设定为大于运算核心数的 4 倍,负载不均衡现象常也表现得不那么明显.针对第 2 个问题,多维并行的多个循环层需要满足每一维都可并行,且相互之间可置换.该嵌套循环多维并行识别算法具体描述如下.

算法. Parallel_LoopNest_Recognition(L,m).

输入:L={L1,L2,…,Lm}是要进行并行识别的嵌套循环,且循环次序按局部性最优原则进行排序;

输出:带有并行区指示的嵌套循环 L.

```
do i:=1,m   //遍历嵌套循环的每一个位置
  do j:=1,m   //遍历嵌套循环中的每一个循环
    if (Lj 可以移动到第 i 层且是可并行的) then
      if (Lj 的迭代次数<256 且 Lj 的迭代次数%64 != 0) then
        Lj 加入多维并行候选集;
        do k:=j+1,m   //遍历 Lj 之后的循环,选择其他多维并行的层
          if (Lk 可以移动到第 i 层且是可并行的)
            Lk 加入多维并行候选集;
            if (当前候选内循环次数乘积>256 或%64==0) then
              将当前多维并行候选集中的循环按加入的顺序移动到第 i 层及后续层,
              在第 i 层生成并行区,并根据候选集里的循环数生成 collapse 子句的参数;
              算法结束;
            endif
          endif
        enddo
      else
        将循环 Lj 移动到第 i 层,选择第 i 层并行,并在第 i 层生成并行区;算法结束;
      endif
    endif
  enddo
```

© 中国科学院软件研究所 http://www.jos.org.cn

enddo

该并行识别算法中第 1 层循环用来遍历循环的位置,优先识别外层的并行性,通常,外层并行有更大的并行粒度,效果更好;第 2 层循环遍历循环的各个层,在当前位置优先并行访存跨步大的层,以便内层循环的访存有更好的数据局部性;第 3 层在需要进行多维并行时,选定能够多维并行的维.该算法的时间复杂度为 $o(n^3)$,n 为嵌套循环层数.在编译系统中实现多维并行的自动识别,优势在于编译工具能够结合循环交换选择更有利于数据局部性的层进行并行,以及能够通过多种程序变换将部分非完美嵌套循环转换为完美嵌套循环,增加程序中多维并行的比例.

4 数据布局

运算核心访问 SPM 的延迟远小于访问主存的延迟,所以在利用运算核心进行加速计算时,将所需数据在计算开始前使用 DMA 方式批量传输到 SPM 中能够显著提高程序的运行效率.对每一段要加载到运算核心上并行执行的代码,数据布局模块根据数据流分析的结果,判断哪些数据在计算开始前需要从主存复制到 SPM,哪些数据在计算结束后需要从 SPM 复制回主存,并在生成相应的数据传输指示.

Open64 的 LNO 阶段会对每个循环进行局部数据流分析,将循环中引用的数据根据定义-使用关系划分为 def、use、pri 等集合.def 集合表示在循环内重新定义变量的集合,在循环内的定义消除了变量的原有定义.use 集合表示在循环中使用变量的集合,这些变量是向上暴露的.pri 集合表示在循环内定义并且只在循环内使用的变量的集合.需要说明的是,这些集合的交集并非总是为空,即一些变量可能在多个集合中出现.Open64 中的上述信息封装在 ARA_LOOP_INFO 类中,其 OpenMP 自动并行化阶段即是根据这些信息判断变量的属性为 shared 还是 private.本框架同样也是根据这些信息判断哪些数据是运算核心需要的,并生成 copy、copyin、copyout 和 create 等数据子句.本框架判断的方法如下:属于 pri 集合的变量使用 create 子句;属于 use 集合中的变量使用 copyin 子句;属于 def 但是不属于 pri 集合的变量使用 copyout 子句;同时,将在 copyin 和 copyout 子句中的变量替换为 copy 子句.

Sunway OpenACC 编译工具根据数据复制子句决定数据在存储系统的分布以及生成传输数据的 DMA 命令.但数据传输过程需要较大的开销,而且运算核心的 SPM 空间有限,所以在生成数据子句时需要精确计算传输数据的范围,避免冗余数据的传输.数据管理模块使用了数组边界分析和指针范围分析来确定数组和指针数据的起始和结束位置.

4.1 数组边界分析

计算数组传输范围最简单的方法是根据数组的定义获得整个数组的大小,将其全部传输到 SPM.但是这样可能导致冗余数据的传输,增加数据传输的开销和降低 SPM 的利用率.因此,对在并行区使用的数组进行边界分析能够有效避免冗余数据的传输.Open64 在 LNO 阶段使用了一种基于区域分析的数组访问分析方法,将数组元素划分为不同的区域,每个区域是一个用凸多边形表示的访问集合,凸多边形则是由从数组元素边界信息得到的简化线性方程组表示的.这些信息保存在 Open64 的 ARA_REF 结构中.在 ARA_REF 结构中可以方便地取到数组访问的下标线性表达式,将循环的上下界信息带入下标表达式即可得到数组访问的边界信息.

图 5 所示为数组边界分析示例.

图 5(a)所示代码为原始代码;

图 5(b)所示为根据 ARA_REF 中信息得到的 A 数组下标线性表达式,以及计算得到的 A 数组下标边界;

图 5(c)所示为根据数组边界信息生成的并行代码(B 数组的分析过程与 A 数组相同).

目前的数组边界分析方法仅能分析线性数组下标.

© 中国科学院软件研究所 http://www.jos.org.cn

```
int a[128][128], b[128][128];
void foo()
{
  int i, j;
  for(i=0; i<64; i++ )
     for(j=0; j<64; j++)
        a[i+1][2*j] = b[i][j];
}
```

(a) 原始代码

```
数组A下标线性表达式:
Linex[0] = loop_index[0]*1
Linex[1] = loop_index[1]*1+1

数组A边界:
Dimension 0:   [0, 128]
Dimension 1:   [1, 65]
```

(b) 数组边界分析过程

```
int a[128][128], b[128][128];
void foo()
{
  int i, j;
  #pragma acc parallel loop  copyin(b[0:64][0:64]) opyout(a[1:64][0:128])
  for(i=0; i<64; i++ )
    for(j=0; j<64; j++)
       a[i+1][2*j] = b[i][j];
}
```

(c) 并行代码

Fig.5 Array boundary analysis example

图 5 数组边界分析示例

4.2 指针范围分析

OpenACC 中数据复制子句的参数也可以是用指针声明的动态数组.对循环引用的指针变量的范围进行分析,而不是简单地使用动态申请的空间大小,也是减少冗余通信和节省 SPM 空间的重要手段.指针范围分析多用于安全分析、BUG 检测和硬件综合等方向[21],本框架将指针范围分析的方法用于异构众核处理器的数据布局,并提出了一种需求驱动的指针范围分析方法.

图 6 所示为指针范围分析的代码示例,图 6(a)所示代码为原始代码;图 6(b)所示为指针范围分析过程;图 6(c)所示为根据指针范围信息生成的并行代码.图 6(b)所示分析过程采用范围描述符域(descriptor-offset domain)[22]进行符号范围表示,在每条语句后,给出了整型变量 i、k 和指针变量 p、q 范围分析的结果,分析过程使用了前向数据流分析.其中,范围描述符域 $p\rightarrow\langle x:\tau[\sigma],[\min,\max]\rangle$表示指针 p 的指向为数组 x,x 为元素类型为τ、大小为σ的数组,偏移范围为[min,max],即:指针 p 指向内存空间的范围从 $x+\min\times sizeof(\tau)$到 $x+\max\times sizeof(\tau)$.特别地,如果 $x:\tau[\sigma]$为 *null*,则表示 p 为整数,取值范围为[min,max].在 if-else 分支交汇处,需要对符号范围进行合并,取两者范围的并集.

```
void foo()
{
  int k, i, flag, *p, *q;
  int *a=malloc(sizeof(int)*2048);
  int *b=malloc(sizeof(int)*2048);
  for(i=0; i<32; i++ )
  {
    if(flag)
      k = i*i;
    else
      k = i;

     p = a+k;
     q = b+i;
     *q = *p;
  }
  ......
}
```

(a) 原始代码

```
void foo()
{
  int k, i, flag,*p, *q;
  int *a=malloc(sizeof(int)*2048);
  int *b=malloc(sizeof(int)*2048);
  for(i=0; i<32; i++ )
  {                     i→<null,[0,32]>
    if(flag)
      k = i*i;          k→<null, [0,1024]>
    else
      k=i;              k→<null, [0,32]>
                        k→<null, [0,1024]>
    p = a+k;            p→<a:int[2048],[0,1024]>
    q = b+i;            q→<b:int[2048],[0,32]>
    *q = *p;
  }
  ......
}
```

(b) 指针范围分析过程

```
void foo()
{
  int k, i, flag, *p, *q;
  int *a=malloc(sizeof(int)*2048);
  int *b=malloc(sizeof(int)*2048);
  #pragma acc parallel loop
  copyin(a[0:1024]) copyout(b[0:32])
  for(i=0; i<32; i++ )
  {
    if(flag)
      k = i*i;
    else
      k=i;
    p = a+k;
    q = b+i;
    *q = *p;
  }
  ......
}
```

(c) 并行代码

Fig.6 Pointer range analysis example

图 6 指针范围分析代码示例

© 中国科学院软件研究所 http://www.jos.org.cn

传统的数据流分析技术通常是在控制流图上进行迭代,更新每个程序点的信息直至到达定点.指针分析是数据流分析的一种,本框架的指针范围分析也采用经典的数据流分析方式,并针对面向异构众核处理器并行化编译的需求,在精度和效率之间加以折中,是一种需求驱动的方法.本框架提出的需求驱动的指针范围分析方法的主要步骤如下.

(1) 收集待分析循环的指针变量引用点,并构造待分析符号变量集合 RPset;

(2) 根据 RPset 构建循环的初始需求驱动控制流图 DCFG,DCFG 中仅包含对待分析符号变量有副作用的语句;

(3) 在初始 DFCG 上按照拓扑后续依次分析,并不断加入待分析的符号变量存在的控制依赖和数据依赖对应的节点和边,当所有可达节点依次加入到该流图后,针对该循环的 DCFG 构建完毕;

(4) 在构造的需求驱动控制流图上进行迭代数据流分析,当某个节点的变量存在控制范围约束时,对控制范围和数据范围信息进行合并.

5 传输优化

在任务划分和数据布局完成后就可以得到能够在 SW26010 处理器上运行的异构并行程序,但是要想获得更好的性能,还需要对主存和 SPM 之间数据的传输进行优化,尽量减少不必要的数据传输和提高传输效率.在实际应用中,不当的数据传输往往是影响程序性能的关键因素.本节提出的面向 SW26010 处理器结构特征的数据传输优化方法有数据传输外提、数据传输合并、离散传输聚合和数组转置等.

5.1 数据传输合并

在实际程序中,两个相邻的并行区使用相同数据的情况是很常见的.在这种情况下,每个并行区的开始和结尾都进行数据复制的方式往往能够找到优化的机会.如图 7(a)所示为 jacobi 迭代核心的 OpenACC 并行程序,在这段代码中有两个相邻的 parallel 并行区,第 1 个并行区的结尾需要将数组 Anew 从 SPM 复制回主存,而第 2 个并行区的开始处又需要将数组 Anew 复制到 SPM.如果在这两个并行区计算过程中把数组 Anew 一直都存放在 SPM 中,则可以减少数组 Anew 的一次拷出和一次拷入操作.要实现这个目的,一种可行的方法是将这两个并行区的数据传输进行合并,只在第 1 个并行区的开始和第 2 个并行区的结尾传输数据,即如图 7(b)所示的代码.

```
while ( err > tol&& iter< iter_max) {
  err=0.0;
  #pragma acc parallel loop  reduction(max:err)
          copyin(A) copyout(Anew)
  for( j = 1; j < n-1; j++) {
    for(i= 1; i< m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1]
                  + A[j-1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i]-A[j][i]));
    }
  }
  #pragma acc parallel loop
          copyin(Anew) copyout(A)
  for( j = 1; j < n-1; j++) {
    for( i= 1; i< m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

(a) jacobi迭代核心

```
while ( err > tol&& iter< iter_max) {
  err=0.0;
  #pragma acc data copy(A) create(Anew){
    #pragma acc parallel loop  reduction(max:err)
    for( j = 1; j < n-1; j++) {
      for(i= 1; i< m-1; i++) {
        Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1]
                    + A[j-1][i] + A[j+1][i]);
        err = max(err, abs(Anew[j][i]-A[j][i]));
      }
    }
    #pragma acc parallel loop
    for( j = 1; j < n-1; j++) {
      for( i= 1; i< m-1; i++ ) {
        A[j][i] = Anew[j][i];
      }
    }
  }
  iter++;
}
```

(b) 数据传输合并的代码

```
#pragma acc data copy(A) create(Anew)
while ( err > tol&& iter< iter_max) {
  err=0.0;
  #pragma acc parallel loop reduction(max:err)
  for( j = 1; j < n-1; j++) {
    for(i= 1; i< m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1]
                  + A[j-1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i]-A[j][i]));
    }
  }
  #pragma acc parallel loop
  for( j = 1; j < n-1; j++) {
    for( i= 1; i< m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

(c) 数据传输外提后的代码

Fig.7 Example of merging and hoisting transmission

图 7 数据传输合并及外提示例

© 中国科学院软件研究所 http://www.jos.org.cn

数据传输合并面临的第 1 个问题是什么情况下进行合并是有利的?如果两个并行区使用的数据不同,则数据传输的合并会使得传输的数据量增大,占用更多的 SPM 存储空间,这可能影响程序的执行效率.考虑到有利性,本框架只对两种情况进行数据传输合并:一种情况是两个连续并行区使用的数组变量完全相同;另一种情况是两个并行区使用的数据总量不超过 SPM 空间大小.数据传输合并需要解决的第 2 个问题是什么情况下合并是安全的,即不会引起程序运行错误?如果在两个并行区之间,有 MPE 端的代码对两个并行区共同使用数据的引用,则数据传输合并可能导致程序运行错误.所以在进行数据传输合并时,必须保证前一个并行区的 copyout 子句和后一个并行区的 copyin 子句共有的变量,在两个并行区之间的 MPE 端代码中没有对其的引用.在满足有利性和安全性的前提下,数据传输合并的过程则较为简单:用 OpenACC 的 data 指示创建一个数据区,包含需要合并的并行区,并根据待合并并行区的数据子句生成 data 指示的数据子句,然后删除并行区的 parallel 指示的数据子句.

5.2 数据传输外提

图 7(b)所示进行数据传输合并后的代码,数据区位于外层的 while 循环内,while 循环的每次执行都会进行数据传输.如果把数据区移到循环外,如图 7(c)所示,则在整个外层 while 循环的执行过程中只在循环计算开始和结束后进行数据传输,可以极大地减少数据传输的开销,程序执行效率会有大幅提升.

针对外层串行内层并行的嵌套循环,本框架使用数据传输外提将嵌套循环内层被多次执行的数据区代码转移到嵌套循环外,减少数据传输次数.数据传输外提采用自底向上的分析过程,当数据区包含在循环内部,且数据区满足以下两个条件时进行外提:第一,数据区中数据复制子句复制数据的范围在外层循环执行过程中是不变的,即外层循环的每次执行都复制同样的数据;第二,数据区传输的变量在外层循环除数据区以外的代码中没有引用,即这些变量只在数据区内使用,一直保留在 SPM 内不会影响外层循环内其他代码段的执行,也不会受其他代码段的影响.重复执行以上外提过程,直至不满足外提条件或数据区外层没有循环为止.

5.3 数据打包传输

在数据传输过程中,DMA 命令缓冲有限,零散数据的多次传输会占用 DMA 命令缓冲,也会增加 DMA 多次启动的开销,进而降低数据的传输效率.因此,把零散变量打包成一个大的变量进行传输,可以减少数据传输次数,降低开销.Sunway OpenACC 在标准 OpenACC 的基础上增加了数据打包子句 pack/packin/packout,用来实现零散变量打包之后的传输.数组打包子句的格式和使用方式与数据复制子句相同,其含义如下.

(1) pack 子句:先对参数列表中的数据进行打包,然后操作步骤同 copy 子句,之后进行解包;

(2) packin 子句:先对参数列表中的数据进行打包,之后操作步骤同 copyin 子句;

(3) packout 子句:前部分操作同 copyout,之后对数据进行解包.

图 8 所示即为一个数据打包的示例,将标量 x、y、z 以及数组 *coef* 打包为一个大的变量,通过一次数据复制传输到 SPM,减少了数据复制的次数,能够提高传输效率.

```
#pragma acc parallel loop copyin (x,y,z,coef[0:3],A)
for(i=0; i<Natoms; i++)
{
    mx = A[y+z] * coef[0];
    my = A[x+z] * coef[1];
    mz = A[y+x] * coef[2];
    ......
}
```

(a) 原始代码

```
#pragma acc parallel loop packin (x,y,z,coef[0:3])copyin(A)
for(i=0; i<Natoms; i++)
{
    mx = A[y+z] * coef[0];
    my = A[x+z] * coef[1];
    mz = A[y+x] * coef[2];
    ......
}
```

(b) 数据打包代码

Fig.8 Example of packaging transmission

图 8 数据打包传输示例

但在数据打包时需要在主存中申请存放打包后数据的空间,并且打包过程还需要一些时间开销.所以,使用数据打包传输优化的关键问题是确定哪些数据的打包传输能够带来收益?对于标量,其打包过程需要的内存及

© 中国科学院软件研究所 http://www.jos.org.cn

其他开销较小,认为其打包传输总是有收益的.对于数组,根据经验,当数组的数据量小于 DMA 的带宽(DMA 通道一个时钟周期传输的数据量)时,打包传输是有收益的.数据打包传输的分析过程较为简单,依次对数据复制子句中的变量进行分析:对于标量,直接将其放入数据打包子句;对于数组,计算其数据量大小,如果小于 DMA 带宽,则放入数据打包子句.

5.4 数组转置

对访存不连续的数组进行复制,会使编译器生成带跨步的 DMA 传输命令,其性能相对较差.Sunway OpenACC 在标准 OpenACC 的基础上增加了数组转置子句 swap/swapin/swapout.数组转置子句的格式为

swap/swapin/swapout(数组名(dim order:…),…)

数组转置子句的使用方式与数据复制子句相同,其含义如下.

(1) swap 子句:先对数组按照 dim order 进行转置,然后操作步骤同 copy 子句,之后再转置回原数组;

(2) swapin 子句:先对数组按照 dim order 进行转置,然后操作步骤同 copyin 子句;

(3) swapout 子句:先把程序中的数组访问替换成对转置后数组的访问,然后操作同 copyout,之后再对数组按照 dim order 进行转置.

在图 9(a)所示的代码段中,最内层循环 *k* 的下标出现在 FDV 数组的最高维,也就是说,对 FDV 的访问是不连续的,使用 copyin(FDV)子句则会生成带跨步的 DMA 命令,数据传输效率较低.如图 9(b)所示代码,使用 swapin(dim order:2,1,3)替换 copyin(FDV),对数组 FDV 进行转置,转置后数组的维度顺序为(2,1,3),这使得数组存储顺序与访问顺序一致.数组转置后,一方面可以使用连续的 DMA 命令替换带跨步的 DMA 命令传输数据,提高了数据传输的效率;另一方面,在运算核心端连续的访存也会提高访存效率.但需要注意的是,数组转置功能需要在主存中申请存放转置后数组的空间,并且转置过程需要一定的开销.

```
#pragma acc parallel loop  copyin(FDV))creat(FDV1k, FDVl, FDV0T)
for(j = 0; j < jm; j++)
  for(i = 0;  i< im; i++)
  {
    for(k = 0; k < km; k ++)
    {
      FDV1k = FDV[k][j][i];
      FDV1[k] = FDV1k;
      FDV0T[k] = FDV1k;
    }
    ......
  }
```
(a) 原始代码

```
#pragma acc parallel loop swapin(FDV(dim order:2,1,3))
creat(FDV1k,FDV1, FDV0T)
for(j = 0; j < jm; j++)
  for(i = 0; I < im; i++)
  {
    for(k = 0; k < km; k ++)
    {
      FDV1k=FDV[k][j][i];
      FDV1[k]=FDV1k;
      FDV0T[k]=FDV1k;
    }
    ......
  }
```
(b) 数组转置后代码

Fig.9 Exampe of array transpose

图 9 数组转置代码示例

数组转置优化的分析过程为:依次分析数据复制子句中的数组,当循环中数组的下标索引次序和循环嵌套迭代次序不一致时,即要按循环嵌套的迭代顺序对数组进行转置,则生成相应的数组转置子句,将该数组放入数组转置子句中.转置的维度顺序按照如下方法分析:首先对复制子句中的数组的下标索引变量从低维到高维依次存到数组 ref_order 中(访存跨步小的为低维),再将循环嵌套索引变量从外层到内层依次存到数组 loop_order 中,然后从最后一个元素,即最内层循环索引开始遍历,查找其在数组中的编号,存到数组 swap_order 中,若 swap_order 与 ref_order 不一致,说明数组需要转置,并将 swap_order 作为数组转置时的维度顺序.

6 收益评估

将循环加载到运算核心上运行需要一定的启动和数据传输等开销,所以需要通过收益评估来判断循环的并行执行是否能带来收益.目前,收益分析主要有两种方法:一种是通过构建代价模型[23],评估串行执行和并行

© 中国科学院软件研究所 http://www.jos.org.cn

执行的时间开销,确定是否有收益;另一种是使用机器学习的方法[24],基于已有的经验,根据循环的迭代次数、循环体大小等程序信息判断是否有收益.代价模型是一种编译器中普遍使用的用来判断各种程序优化是否有收益的方法,比如 GCC、LLVM、Open64 等优秀编译器都使用了代价模型.用机器学习的方法进行收益评估是一种比较新的方法,但是由于需要构建学习模型,进行大量训练,而且基于经验的预测存在判断失误的情况,目前在产品级编译器中较少使用该方法.本框架选择代价模型进行收益评估,通过比较循环在单个管理核心上的串行执行代价和在运算核心簇上的并行执行代价,决定是否将循环加载到运算核心上运行.

Ravi 等人针对 Intel Phi 协处理器提出的优化编译器 Apricot[14]中使用了一个简单的代价模型来优化性能,该代价模型主要通过循环迭代次数、CPU 操作数、访存操作数以及 CPU 操作和访存操作占总操作数的比例等与经验值进行比较来评估收益.Grosser 等人[25]开发了一个 Polly-ACC 编译框架,将通用程序移植到异构结构上,其中使用了一个简单的代价模型,也是通过一些条件判断来筛选适合在加速器上进行加速计算的代码段.在这些面向众核结构的并行化系统中,使用的代价模型较为简单,在开发产品级编译系统时无法满足实际需求.Open64 编译器中包含了一个层次化代价模型,用于对程序中的循环代价进行评估[23].与本文工作最为接近的是黄品丰等人[26]在 Open64 代价模型的基础上所做的面向 Cell 处理器的代价模型,本文借鉴了其基本思路,将其扩展为面向核数更多的 SW26010 处理器的代价模型.

本节首先构建面向 SW26010 处理器的准确的代价模型,再在代价模型的基础上实现了一种动静结合的并行收益评估方法.对于循环信息在编译时可全部获取的情况,在编译时即可根据并行代价模型静态确定是否有收益;对于循环信息在编译时无法全部获取的情况(比如循环边界根据程序输入确定),生成 if 子句,根据编译时静态收集的信息和运行时动态得到的信息,在运行时确定是否将该循环加载到运算核心并行执行.

6.1 代价模型的建立

代价模型通常由一组底层模型组成,反映了应用程序在计算机系统上的具体执行细节,并且使用执行开销(一般是时间)对程序的执行效率进行评估.许多编译器和运行库中都包含有代价模型,用于评估编译器中的程序变换,指导编译优化的过程.Open64 编译器的 LNO 中包含了一组代价模型[27],用于对程序中的 SNL(singly nested loop)循环代价进行评估,如图 10 所示.Open64 中的循环代价模型包括了串行模型和并行模型两个子模型,分别用来评估程序在单个处理器上串行执行的开销和在多个处理器上使用 OpenMP 的 fork-join 模式多线程并行执行的开销.每个子模型的开销可以进一步分为粒度更细的开销类型,并且分别建模.其中,串行执行开销依赖于与处理器、存储相关的开销;并行执行开销则在串行开销的基础上还要考虑并行所带来的额外开销,主要是并行启动开销和线程同步开销.

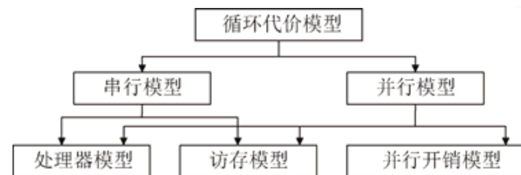

Fig.10 The structure of cost model in Open64
图 10 Open64 中的代价模型框图

构建面向 SW26010 处理器的代价模型也需要构建两个子代价模型,分别是程序在单个管理核心上运行时的串行模型和在运算核心簇上运行时的并行模型.管理核心的结构,以及程序在管理核心上的运行方式与通用 X86 处理器类似,所以串行代价模型可以借鉴 Open64 已有的串行模型.而程序在运算核心簇上并行执行的方式与在多核上的 OpenMP 并行执行方式有较大不同,而且运算核心没有传统的硬件 Cache,所以并行模型需要重新构建.

6.1.1 串行模型

循环串行执行时间是评估循环并行收益的基准.在 SW26010 处理器上,循环串行执行开销是指循环在管理

© 中国科学院软件研究所 http://www.jos.org.cn

核心上的运行的时钟周期数,主要包含处理器开销、存储访问开销,可用公式(1)表示.

$$T_{seq}=T_{processor}+T_{memory} \tag{1}$$

(1) 处理器开销

处理器开销 $T_{processor}$ 是指在忽略 Cache 和 TLB 不命中等访存开销的情况下,循环指令在处理器上执行的开销.在编译过程中,可以通过分析循环的中间表示语法树获得循环语句(包括循环头和循环体)中各类指令的数目,并根据各类指令的执行周期数,通过累加得到循环一次迭代的运行时间 $T_{machine_per_iter}$,然后通过与循环迭代次数 N_{loop_iter} 相乘就能得到循环的处理器开销,如公式(2)所示.

$$T_{processor}=T_{machine_per_iter}\times N_{loop_iter} \tag{2}$$

指令的执行周期可以通过查阅处理器资料获得.循环迭代次数 N_{loop_iter} 的获取则需要根据程序的具体情况来定,有些循环的上下界为常数或常数表达式,可以在编译时就静态分析得到;而有些循环的循环上下界则取决于程序输入,只能在运行时得到,这时将循环迭代次数用变量代替,将处理器开销表示为循环迭代次数的函数.

(2) 访存开销

访存开销 T_{memory} 是指一级数据 Cache 不命中情况下的数据访问开销,与循环的数据局部性以及 Cache 行大小及 Cache 失效开销紧密关联.编译过程可以通过分析循环的数据访问特征以及 Cache 行大小和 Cache 配置策略来预测 Cache 失效次数.在计算 Cache 失效次数时,考虑到空间局部性,对相邻数组元素的访问只计算 1 次失效.SW26010 处理器上管理核心拥有两级数据 Cache,因此存储访问开销为数据在一级 Cache 失效二级 Cache 命中,以及两级 Cache 都失效的情况下的访存开销.如公式(3)所示,T_{L2} 为一级 Cache 失效二级 Cache 命中情况下的访问开销,T_{main_mem} 为两级 Cache 都失效情况下直接访问主存的开销,N_{L2} 为二级 Cache 命中次数,N_{main_mem} 为主存访问次数,两者之和为一级 Cache 失效次数.T_{L2} 和 T_{main_mem} 可通过简单测试得到.其中,T_{L2} 和 T_{main_mem} 可通过简单测试得到,N_{L2} 和 N_{main_mem} 可通过程序分析进行预测.

$$T_{memory}=T_{L2}\times N_{L2}+T_{main_mem}\times N_{main_mem} \tag{3}$$

6.1.2 并行模型

循环在 SW26010 处理器上的并行执行过程为:首先,加载程序段和传输数据到运算核心;然后,由运算核心完成计算;最后,把计算结果返回主存.根据程序在 SW26010 上的运行方式,可以将并行代价分解为并行控制开销、数据传输开销和并行计算开销,用公式表示为

$$T_{para}=T_{para_overhead}+T_{data_trans}+T_{compute} \tag{4}$$

(1) 并行控制开销

并行控制开销主要是并行执行时的启动和管理开销,包括初始化并行计算环境、创建加速线程组、加载工作任务到运算核心并启动任务执行等的开销.并行控制开销 $T_{para_overhead}$ 可以表示为公式(5),是启动的运算核心数 p 的线性函数.其中,T_c 为并行启动开销,T_p 为单个运算核心的并行管理开销.T_c、T_p 可通过运行简单循环进行测定.

$$T_{para_overhead}=T_c+p\times T_p \tag{5}$$

(2) 数据传输开销

数据传输开销主要取决于两个方面:传输的数据量;DMA 的数据传输带宽.在处理器总线带宽一定的情况下,传输的数据量决定了传输的开销.通过分析数据传输子句,就能得到传输的数据量.数据传输开销 T_{data_trans} 可用公式(6)表示.其中,T_s 为 DMA 操作的初始化时间,T_o 为数据的数据传输过程中所做的数据打包和转置的开销,*in* 和 *out* 分别为计算开始前从主存拷入 SPM 和计算结束后从 SPM 拷回主存的数据集合,x、y 分别为 *in* 和 *out* 集合中的元素,*x.size* 和 *y.size* 为它们占用的存储空间大小,w 为 DMA 通道数据传输带宽,其中,T_s 可以通过构造测试程序测试得到.数据打包和转置实际上是管理核心上执行的操作,所以,T_o 可以通过调用串行模型得到.

$$T_{data_trans}=T_s+T_o+\frac{\sum_{x\in in}x.size+\sum_{y\in out}y.size}{w} \tag{6}$$

© 中国科学院软件研究所 http://www.jos.org.cn

(3) 并行计算开销

并行计算开销实际上是循环在多个运算核心上运行时,从第 1 个运算核心开始执行到最后一个运算核心执行结束的时间.由于运行时的不确定性,这个值无法在编译时准确预测,这里使用循环在单个运算核心上的运行时间除以计算所用核心数的值近似表示.单个运算核心上运行开销的计算过程与管理核心类似,但是由于运算核心的结构和性能与管理核心不同,各部分消耗的计算方法有所不同.并行计算开销 $T_{compute}$ 可以用公式(7)表示.$T'_{processor}$、T'_{memory} 分别为运算核心的处理器开销和访存开销.

$$T_{compute}=\frac{T'_{processor}+T'_{memory}}{p} \tag{7}$$

$T'_{processor}$ 的计算过程和在管理核心上 $T_{processor}$ 的计算过程相同,只需将各类指令的执行周期替换为管理核心的指令周期,管理核心指令周期数也可以通过查阅系统手册得到.因为运算核心没有硬件 Cache,只有软件管理的 SPM,所以 T'_{memory} 的计算方式有所不同,计算过程如公式(8)所示.

$$T'_{memory}=T_{spm}\times N_{spm} \tag{8}$$

其中,T_{spm} 为 SPM 访问延迟,N_{spm} 为 SPM 访问次数,T_{mem} 为运算核心访问主存延迟,N_{mem} 为访问主存次数.N_{spm} 和 N_{mem} 可以通过程序分析得到,加速循环中访问的数据如果是在数据复制子句 copy/copyin/copyout 和 create 子句中,则在访问时数据在 SPM 中,否则,加速循环访问的数据则需要从主存中取得.

目前,本文给出的并行编译框架只能处理器迭代间无依赖的 DOALL 循环,无需考虑进行核间通信开销.

6.2 收益评估方法

在代价模型的基础上,就可以通过比较程序串行执行的代价和并行执行的代价来判断循环的并行执行是否能够带来收益,公式(9)即为判断循环并行是否有收益的标准.

$$T_{para_overhead}+T_{data_trans}+T_{compute}<T_{processor}+T_{memory} \tag{9}$$

对于循环信息在编译时可全部获取的情况,在编译时即可根据公式(9)静态确定是否有收益.然而,对于循环信息在编译时无法全部获取的情况,比如循环边界根据程序输入确定,编译时无法判断循环并行是否有收益,本框架提出了一种将编译时静态得到的信息与运行时动态获得的信息相结合的动静结合的收益评估方法.如图 11(a)所示为 SPEC CPU2006[28]中程序 462.libquantum 中的一个热点循环,该循环的上界 $reg\rightarrow size$ 是通过函数参数 reg 传入的,无法在编译时确定其值.在一个极端的情况下,循环被调用多次,而不同调用的循环边界有所不同,这样可能会有一部分调用并行执行有收益,而另一部分调用并行执行无收益甚至严重影响程序执行效率.对于这种情况,无论编译时保守地选择不并行,还是激进地选择并行,都可能无法获得好的效果.本框架对编译时循环边界未知的情况生成 if 子句,根据编译时静态收集的信息和运行时动态得到的信息,在运行时确定是否将该循环并行执行.以图 11(a)中所示代码为例,对该循环分别调用串行模型和并行模型,得到串行代价和并行代价分别为 $f(reg\rightarrow x)$ 和 $g(reg\rightarrow x)$.利用公式(9)进行收益评估,当 $f(reg\rightarrow x)>g(reg\rightarrow x)$时,并行是有收益的,此时可以求得 $reg\rightarrow x$ 的临界值 X.于是我们生成 if 子句,如图 11(b)所示,在运行时根据 $re\rightarrow x$ 的值决定是否将该循环加载到运算核心.

```
for(i = 0;i < reg->size; i++) {
  if(reg->node[i].state & 1 << control1) {
    if(reg->node[i].state & 1 << control2) {
      reg->node[i].state ^= 1 << target;
    }
  }
}
```
(a) 原始代码

```
#pragma acc parallel loop copyin (control1, control2, target)
  copy(reg->node[0:reg->size]) if(reg->size > X)
for(i = 0; i < reg->size; i++) {
  if(reg->node[i].state & 1 << control1) {
    if(reg->node[i].state & 1 << control2) {
      reg->node[i].state ^= 1 << target;
    }
  }
}
```
(b) 使用if子句代码

Fig.11 Example of benifit estimate

图 11 收益评估代码示例

本文使用的编译时静态建模的方法虽然无法做到足够精确,但在应用于并行化编译系统时,只要考虑到精

度上的不足,就能在实际应用时提升并行代码的性能,所以,Open64、GCC、LLVM 等很多优秀的编译器中也都使用了代价模型作为收益评估的手段.

7 实验测试

实验测试部分包括优化效果测试和整体性能测试.优化效果测试通过选取一些典型的测试程序,对本文提出的多维并行、传输优化、收益评估等方法进行对比测试;整体性能测试则使用基准测试集和一些典型的科学计算程序,验证本框架的整体效果.实验平台为"神威太湖之光"计算机系统,其计算节点为SW26010异构众核处理器,并配备有完整的编译工具链和性能分析工具.

7.1 优化效果测试

本文提出的多维并行、传输优化、收益评估等优化方法只对具备一定特征的程序有效,本节所选测试程序为符合相应特征的程序,比如多维并行测试部分选择了嵌套层数较多的循环,传输优化测试选择了数据传输较多的程序,收益评估测试则选择了有大量较小并行循环的程序.这里对每种优化方法选择了 4 个典型的程序进行测试分析,测试程序主要来自于 SPEC CPU 2006 测试集、NPB3.3.1 测试集、PGI OpenACC SDK 和一些科学计算程序.

7.1.1 多维并行测试

本节对任务划分模块实现的嵌套循环多维并行的有效性进行测试,测试选择了科学计算常用的矩阵乘法(规模为 100×100)、PGI OpenACC SDK 中的三维空间时域有限差分程序 FDTD3d 以及 NPB3.3.1 中的 LU(B 规模)和 BT(B 规模)4 个程序.这 4 个程序中都存在多层嵌套循环,且外层循环迭代次数较少,容易导致负载不均衡,因此适合用本文提出的多维并行识别方法进行优化.测试过程对比了这 4 个程序在使用单维并行以及本文实现的多维并行后程序的加速比,测试结果如图 12 所示.

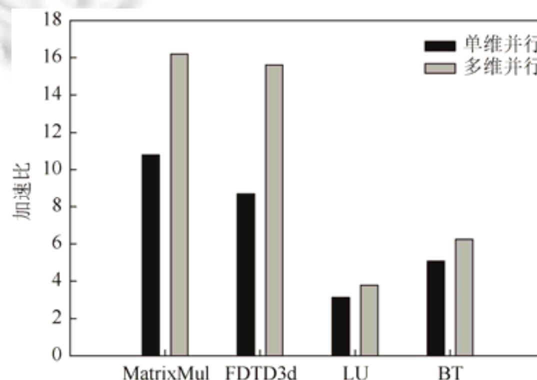

Fig.12 Comparison of speedup before and after multi dimension parallelization

图 12 多维并行前后程序加速比对比图

从图 12 可以看出,本文实现的多维并行识别方法,能够有效解决嵌套循环单维并行时并行层迭代次数较少导致的负载不均衡问题,有效地提升程序性能,这 4 个程序的加速比平均提升了 1.49 倍.

7.1.2 传输优化测试

本节对传输优化模块提出的传输合并、传输外提、数据打包和数组转置优化方法进行了测试分析.测试程序选择了 Jacobi 迭代、一个科学计算程序飞行器绕流应用 gkufs[29]、NPB3.3.1 中的 BT(B 规模)以及 SPEC CPU 2006 中的 410.bwaves(ref 规模).这 4 个程序中有较多的并行区,且并行区的数据传输量较大,有较多数据传输优化的机会.通过测试每种优化方法之后程序运行时间的变化,验证优化方法的有效性.图 13(a)~图 13(d)分别为 Jacobi 迭代、gkufs、BT 和 410.bwaves 的测试结果,图中给出了程序优化前的原始运行时间、每种优化之后的

© 中国科学院软件研究所 http://www.jos.org.cn

运行时间以及 4 种优化同时使用的整体时间.

在图 13 所示的测试结果中,由于程序特征不同,不同的优化方法对不同的程序优化效果也有所不同.比如传输合并和传输外提能够较大地提升 Jacobi 迭代程序的性能;传输合并、数据打包和数组转置 3 种方法对 gkufs 都有一定效果;而传输合并、传输外提和数组转置对 410.bwaves 有一定效果;传输合并和传输外提则对 BT 能够取得较好的效果.本文提出的各种优化方法只适用于某一类程序特征,故测试结果中有些方法对某些程序没有效果,而效果的好坏则主要取决于程序特征.数据传输优化后,Jacobi 迭代整体上性能提升 3.91 倍,gkufs 整体上提升 1.64 倍,410.bwaves 整体上提升 1.15 倍,BT 整体上提升 1.27 倍.

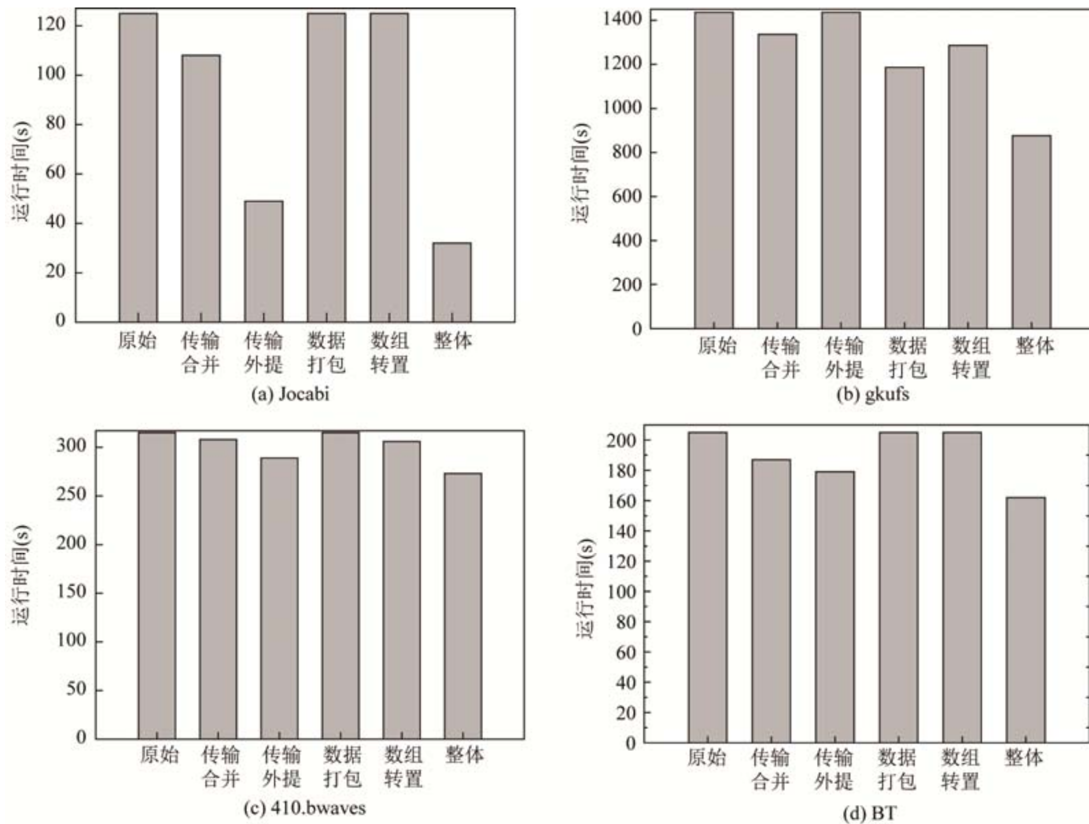

Fig.13 Effect of transmission optimization method

图 13 传输优化方法效果图

7.1.3 收益评估测试

本节对收益评估模块的有效性进行测试,测试用例选择了 SPEC CPU 2006 中的 462.libquantum 和 436.cacatusADM,以及 NPB3.3.1 中的 BT(B 规模)和 SP(B 规模)4 个程序,分别测试了这 4 个程序在进行收益评估前后并行循环的数量和程序获得的加速比.这 4 个程序有大量无收益的规模较小的并行循环,需要收益评估.表 1 中的数据给出了进行收益评估前的并行循环数目以及收益评估后的并行循环数目.图 14 给出了收益分析前后程序获得加速比的对比情况.从表 1 和图 14 的结果可知,本框架的收益评估模块能够有效地过滤计算量不足的循环,避免其加载到运算核心上导致程序性能下降,对程序性能提升效果明显.其中,程序 462.libquantum 由于存在大量计算量较小的内层并行循环导致程序加载到运算核心上的运行时性能下降明显,收益分析后能够将计算量不足的循环过滤掉,程序加速比大幅提升 27.27 倍.

优化效果测试部分并没有对数据布局模块使用的数组边界分析以及指针范围分析方法进行测试.这里没有对其进行测试主要是因为在标准测试集或者经过精心优化的成熟科学计算程序里,数组大小的定义以及指

针变量内存空间的申请都是根据程序需要进行的,通常没有进一步优化的空间.然而,在编译框架的实际使用场景中,编译的经常是没有经过充分优化的程序,可能是一个新手程序员编写的程序,也可能是一个程序初步的版本,在这样的程序中申请了过大的内存空间是一种常见的现象.所以,这两种优化方法在编译框架的实际应用中是非常有用的,本文给出的编译框架会在数组边界分析和指针范围分析后,将实际使用空间小于定义或申请空间大小的变量在输出信息中进行提示,方便程序员检查优化程序.

Table1 Comparison of the number of parallel loops before and after benifit estimate

表 1 收益评估前后并行循环数目对比结果

程序	无收益评估	收益评估	减少比例
462.libquantum	71	14	80.28%
436.cacatusADM	156	28	82.05%
BT	39	23	41.02%
SP	60	31	48.33

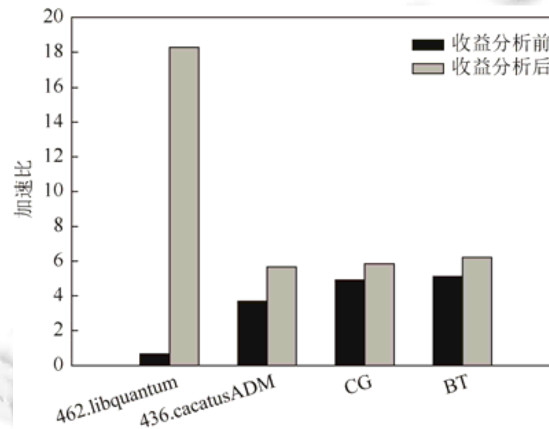

Fig.14 Comparison of speedup before and after benifit estimate

图 14 收益分析前后程序加速比对比图

7.2 整体性能测试

整体性能测试选择了 NPB3.3.1 测试集和 3 个典型的科学计算程序进行测试与分析.NPB 是用于测评大规模并行机和超级计算机的标准测试程序集.所选 3 个科学计算程序分别为计算流体动力学程序 openCFD[30]、地震波模拟和反演程序 3DWING[31]、飞行器绕流应用 gkufs.

7.2.1 NPB 测试分析

本节使用 NPB3.3.1 测试集(B 规模)对本文框架的并行化效果进行测试与分析.图 15 展示了 NPB3.3.1 测试集各个程序通过本文框架并行化后的运行时间相比于原始串行程序运行时间的加速比.NPB3.3.1 测试集中的全部 10 个程序通过本文框架的并行化编译在 SW26010 单个核组上运行时平均能够获得 5.19 的加速,其中加速比超过 10 的有 CG 和 SP,而 DC、EP 和 IS 这 3 个程序则没有获得明显的加速.这 3 个程序没有获得有效加速的原因分别为:DC 的程序热点中没有循环,无法实现并行化;EP 热点循环中有较多函数调用,影响了依赖分析以及并行识别;IS 热点循环内有较多复杂的间接数组访问,无法进行有效的依赖分析和并行识别.

各个异构众核平台结构差异较大,软件系统不兼容,很难与其他平台的研究成果直接进行对比.NPB3.3.1 测试集提供了串行、OpenMP 并行和 MPI 并行等多个版本,OpenMP 与 OpenACC 有很多相似性,用 OpenMP 编译指示标注的并行循环经过改写一般也能成为 OpenACC 程序,所以我们将 NPB3.3.1 OpenMP 版本中标注的并行循环认为是手工并行的循环,作为与本文框架识别情况对比分析的对象.图 16 则给出了 NPB3.3.1 测试集中各个程序手工并行的循环数目与本文框架自动识别出的并行循环数目的对比情况,这里所提自动并行的循环数是

© 中国科学院软件研究所 http://www.jos.org.cn

在关掉收益评估模块后得到的.SW26010 处理器的结构特殊,收益评估模块过滤掉了较多计算量较小的循环,为真实反映本文框架的并行识别能力,测试自动识别的并行循环数时暂时关掉了收益分析模块.可以看到,除了EP、IS 和 UA 这 3 个程序,其他程序自动并行化识别出的并行循环数都多于手工并行.

NPB3.3.1 中的程序根据并行识别情况可以将其分为 4 类:第 1 类,CG 和 SP 两个程序自动识别的并行循环数多于手工并行的,而且自动并行识别的并行循环基本上全部覆盖手工并行,这两个程序的自动并行的识别情况与手工并行基本一致;第 2 类,BT、LU 和 MG 这 3 个程序自动识别的并行循环数目虽然多于手工,但原因是手工并行选择了外层的循环,而自动并行由于程序主要循环过于复杂未能识别出外层的并行,而是识别出了大量内层循环,这 3 个程序自动并行的识别情况弱于手工并行;第 3 类,EP、IS 和 UA 这 3 个程序,自动识别时未能将手工并行的循环识别出来,原因分别是 EP 主要循环有较多函数调用,IS 循环内有复杂的间接数下标,UA 的热点循环程序复杂且有函数调用,自动并行目前无法很好地处理函数调用和间接数组下标等问题;第 4 类,DC 和 FT 两个程序的情况较为特别,DC 的程序热点中没有循环,而 FT 手工并行的版本对程序结构改动较大,不便于进行对比.整体来看,本文框架对 NPB3.3.1 的自动并行识别与手工并行相比有较大差距,主要原因是编译系统不能很好地处理函数调用引起的过程间问题和间接数组下标等非规则访存问题,这也是目前编译领域面临的最为困难的问题中的两个.

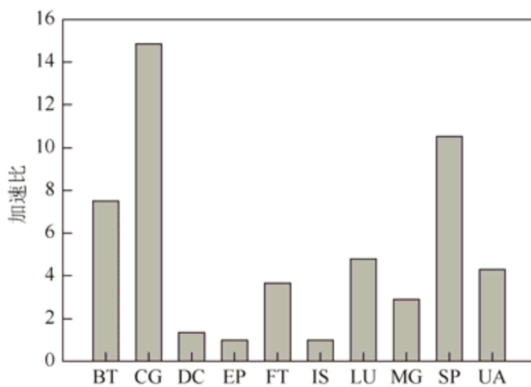

Fig.15 Parallization speedup of NPB3.3.1

图 15 NPB3.3.1 并行化加速比

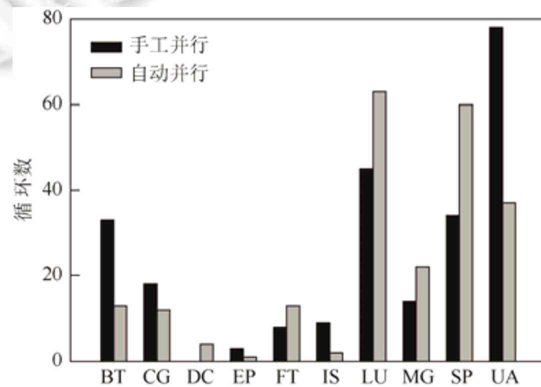

Fig.16 Comparison of the number of parallel loops NPB3.3.1

图 16 NPB3.3.1 并行循环数目对比图

7.2.2 科学计算程序测试分析

本节的测试选择了 3 个实际应用的科学计算程序作为测试用例,计算其加速比,并与手工编写的并行程序进行对比,通过并行化效率反映编译框架的能力.并行化效率则是指并行化编译得到的自动并行程序获得的加速比与程序员手工编写的并行程序获得的加速比的比值.表 2 给出了这 3 个大型应用程序的测试结果.这 3 个程序都是 MPI 程序,节点数是运行程序使用的 SW26010 处理器计算节点(SW26010 处理器一个核组为一个计算节点)的数目.为了方便测试和分析,程序测试时使用了较小的输入规模.

Table 2 Test results of scientific computing program

表 2 科学计算程序并行化测试结果

程序	节点数	串行时间(s)	手工并行时间(s)	自动并行时间(s)	手工加速比	自动加速比	并行化效率(%)
gkufs	128	2 402.87	110.12	876.96	21.82	2.74	12.55
3DWING	8	342.10	7.22	10.58	47.38	32.33	68.24
openCFD	128	129.05	17.72	22.28	7.28	5.79	79.53

由表 2 可知,本文编译能够对大型的科学计算程序进行一定的加速,而加速效果取决于程序本身的并行性,其中,3DWING 加速比达到 32.33.这 3 个科学计算程序代码量大、程序结构复杂,通过对这 3 个程序的测试,说明本文的并行编译框架具有较好的健壮性,因而具有一定的实际应用价值.然而,由并行化效率可以看到,目前,

© 中国科学院软件研究所 http://www.jos.org.cn

本文给出的编译框架的并行化效率仍然较低,这 3 个程序获得的加速比平均仅为手工并行程序的 53.44%.通过与手工并行代码对比可以看到,自动并行效率较低的主要原因有两个:一是手工并行主要是外层的循环,而自动并行的循环很多是内层的计算量较小的循环,这主要是因为函数调用等影响了自动并行发掘更粗粒度的并行;二是手工并行时程序员知道更多的信息,可以对代码做更大的改动,而自动并行使用的程序静态分析和变换都有一定的局限性.

8 总 结

高性能计算机在现代科学研究中的作用越来越重要且不可替代.异构众核处理器在构建高性能计算机系统时低功耗、高性能的优势使其成为高性能计算领域处理器发展的重要趋势,但其更为复杂的结构也使得原本就存在的编程难的问题更加突出.面向异构系统的自动并行化研究时间较短,因异构系统多种多样,各种面向异构系统并行化研究工作的侧重点也各不相同,其应用效果取决于具体的程序特征和平台特性.本文的研究旨在通过自动化的编译工具减轻编程人员将应用移植到"太湖之光"计算机上的工作负担.本文基于开源编译器 Open64,设计和实现了面向 SW26010 异构众核处理器的一个并行编译框架,能够实现一些程序到异构众核并行程序的转换且获得较好的加速.但该框架目前的实现仍有很多不足,比如只能处理规则的数组及指针访问形式,而且本框架使用的主要是静态程序分析方法,具有一定的局限性,限制了本框架在一些程序中的应用.后续的研究工作:第一,要改进现有方法,弥补不足,比如用机器学习的方法进行收益评估,能够克服静态代价模型的一些不足;第二,异构众核处理器仍在不断发展,面对新的体系结构,需要不断探索新的方法;第三,过程间分析、指针分析、依赖分析等编译领域的通用技术直接影响并行识别的能力,目前这些技术还比较薄弱,需要继续深入研究.虽然本文提出的具体方法大多是针对 SW26010 处理器的,但其基本思路对于其他异构众核架构处理器的并行化编译工作也有一定的参考价值.

References:

[1] Zang DW, Cao Zh, Sun NH. The development of high-performance computing. Science & Technology Review, 2016,34(14):22–28 (in Chinese with English abstract). [doi: 10.3981/j.issn.1000-7857.2016.14.002]

[2] Zheng F, Xu Y, LI HL, Xie XH, Chen ZN. A homegrown many-core processor architecture for high-performance computing. Sciece China (Information Sciences), 2015,45(4):523–534 (in Chinese with English abstract).

[3] Yang GW, Zhao WL, Ding L. "Sunway TaihuLight" and application system. Science (Shanghai), 2017,69(3):12–16 (in Chinese with English abstract).

[4] Sodani A, Gramunt R, Corbal J. Knights landing: Second-generation Intel Xeon Phi Product. IEEE Micro, 2016,36(2):34–46.

[5] Wu G, Greathouse JL, Lyashevsky A. GPGPU performance and power estimation using machine learning. In: Proc. of the IEEE Int'l Symp. on High Performance Computer Architecture. IEEE, 2015. 564–576.

[6] Ju XG, Yang L, Huang ST. An overview of architecture of cell processor. Engineering Journal of Wuhan University, 2010,43(6): 774–779 (in Chinese with English abstract).

[7] Daga M, Aji AM, Feng W. On the efficacy of a fused CPU+ GPU processor (or APU) for parallel computing. In: Proc. of the 2011 IEEE Symp. on Application Accelerators in High-performance Computing. IEEE, 2011. 141–149.

[8] Keckler SW, Dally WJ, Khailany B. GPUs and the future of parallel computing. IEEE Micro, 2011,31:7–17.

[9] Carter NP, Agrawal A, Borkar S. Runnemede: An architecture for ubiquitous high-performance computing. In: Proc. of the IEEE Int'l Symp. on High Performance Computer Architecture (HPCA). Shenzhen: IEEE, 2013. 198–209.

[10] Lee S, Min SJ, Eigenmann R. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. ACM SIGPLAN Notices, 2009,44(4):101–110.

[11] Lee S, Eigenmann R. OpenMPC: Extended open MP programming and tuning for GPUs. In: Proc. of the 2010 ACM/IEEE Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. IEEE, 2010. 1–11.

[12] Han TD, Abdelrahman TS. hi CUDA: High-level GPGPU programming. IEEE Trans. on Parallel and Distributed Systems, 2011, 22(1):78–90.

© 中国科学院软件研究所 http://www.jos.org.cn

[13] Baskaran MM, Ramanujam J, Sadayappan P. Automatic C-to-CUDA code generation for affine programs. Compiler Construction, 2010,6011:244–263.

[14] Nishkam R, Yi Y, Tao B, Srimat C. Apricot: An optimizing compiler and productivity tool for X86-compatible many-core coprocessors. In: Proc. of the ICS 2012. Venice: IEEE, 2012. 1–11.

[15] Eichenberger AE, O'Brien JK, O'Brien KM. Using advanced compiler technology to exploit the performance of the cell broadband Engine™ architecture. IBM Systems Journal, 2006,45(1):59–84.

[16] Wang M, Bodin F, Matz S. Automatic data distribution for improving data locality on the cell BE. In: Proc. of the 22nd Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC 2009). Heidelberg: Springer-Verlag, 2009. 247–262.

[17] Chan SC, Gao GR, Chapman B. Open64 compiler infrastructure for emerging multicore/manycore architecture all symposium tutorial. In: Proc. of the IEEE Int'l Parallel & Distributed Processing Symp. IEEE, 2008. 1.

[18] Fu H, Liao J, Yang J, *et al*. The Sunway TaihuLight supercomputer: System and applications. Sciece China (Information Sciences), 2016,59:1–16.

[19] OpenACC-Standard.org. The openacc application programming interface. v2.5. OpenACC-Standard.org, 2015. 1–118.

[20] Zhao J, Zhao RC, Han L. An MPI backend for Open64 compiler. Chinese Journal of Computers, 2014,37(7):1620–1632 (in Chinese with English abstract).

[21] Liu P, Zhao RC, Pang JM, Yao Y. Prioritizing pointer analysis algorithm based on points-to updating. Ruan Jian Xue Bao/Journal of Software, 2014,25(11):2486−2498 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4596.htm [doi: 10.13328/j.cnki.jos.004596]

[22] Yong SH, Horwitz S. Pointer-range analysis. Lecture Notes in Computer Science, 2004,3148:133–148.

[23] Li YB, Zhao RC, Liu XX, Zhao J. Cost model for automatic OpenMP parallelization. Ruan Jian Xue Bao/Journal of Software, 2014,25(Suppl.(2)):101−110 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/14028.htm

[24] Wang Z, Tournavitis G, Franke B. Towards a holistic approach to auto-parallelization integrating profile-driven parallelism detection and machine-learning-based mapping. ACM Trans. on Architecture and Code Optimization, 2014,11(1):2.

[25] Tobias G, Torsten H. Polly-ACC transparent compilation to heterogeneous hardware. In: Proc. of the 2016 Int'l Conf. on Supercomputing. ACM, 2016.

[26] Huang PF, Zhao RC, Yao Y, Zhao J. Parallel cost model for heterogeneous multi-core processors. Journal of Computer Applications, 2013,33(6):1544–1547 (in Chinese with English abstract).

[27] Chunhua L. A compile-time OpenMP cost model [Ph.D. Thesis]. Houston: University of Houston, 2007.

[28] Henning JL. SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News, 2006,34(4):1–17.

[29] Li ZH, Zhang HX. Unified algorithm for three-dimensional complex problems covering various flow regimes based on Boltzmann model equations. Sciece China, 2009,(3):414–427 (in Chinese with English abstract).

[30] Li XL, Fu DX, Ma YW. Development of high accuracy CFD software Hoam-OpenCFD. e-Science Technology & Application, 2010,(1):53–59 (in Chinese with English abstract).

[31] He X, Zhou ZM, Liu X. Design and implemention of multi-level heterogeneous parallel algorithm of 3D acoustic wave equation forwarded. Computer Applications & Software, 2014,(1):264–267 (in Chinese with English abstract).

附中文参考文献:

[1] 臧大伟,曹政,孙凝晖.高性能计算的发展.科技导报,2016,34(14):22–28. [doi: 10.3981/j.issn.1000-7857.2016.14.002]

[2] 郑方,许勇,李宏亮,等.一种面向高性能计算的自主众核处理器结构.中国科学(信息科学),2015,45(4):523–534.

[3] 杨广文,赵文来,丁楠,等."神威·太湖之光"及其应用系统.科学(上海),2017,69(3):12–16.

[6] 巨新刚,杨靓,黄士坦.Cell 处理器结构概述.武汉大学学报(工学版),2010,43(6):774–779.

[20] 赵捷,赵荣彩,韩林,等.面向 MPI 代码生成的 Open64 编译器后端.计算机学报,2014,37(7):1620–1632.

[21] 刘鹏,赵荣彩,庞建民,姚远.基于指向更新的优先权指针分析算法.软件学报,2014,25(11):2486–2498. http://www.jos.org.cn/1000-9825/4596.htm [doi: 10.13328/j.cnki.jos.004596]

[23] 李雁冰,赵荣彩,刘晓娴,赵捷.面向 OpenMP 自动并行化的代价模型.软件学报,2014,25(Suppl.(2)):101−110. http://www.jos.org.cn/1000-9825/14028.htm

© 中国科学院软件研究所 http://www.jos.org.cn

[26] 黄品丰,赵荣彩,姚远,赵捷.面向异构多核处理器的并行代价模型.计算机应用,2013,33(6):1544–1547.

[29] 李志辉,张涵信.基于 Boltzmann 模型方程各流域三维复杂绕流问题统一算法研究.中国科学,2009,(3):414–427.

[30] 李新亮,傅德薰,马延文,等.高精度计算流体力学软件 Hoam-OpenCFD 开发.科研信息化技术与应用,2010,(1):53–59.

[31] 何香,周明忠,刘鑫.三维声波方程正演多级异构并行算法设计与实现.计算机应用与软件,2014,(1):264–267.

李雁冰(1989－),男,甘肃陇西人,博士生,主要研究领域为高性能计算,并行编译优化.

赵捷(1987－),男,博士,讲师,CCF 专业会员,主要研究领域为高性能计算,并行编译优化.

赵荣彩(1957－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为高性能计算,并行编译,反编译.

徐金龙(1985－),男,博士,讲师,主要研究领域为高性能计算,并行编译优化.

韩林(1978－),男,博士,副教授,CCF 专业会员,主要研究领域为高性能计算,并行编译优化.

李颖颖(1984－),女,讲师,CCF 专业会员,主要研究领域为高性能计算,并行编译优化.

© 中国科学院软件研究所 http://www.jos.org.cn