一种面向循环优化和非规则代码段的粗粒度半自动并行化方法

刘 松 赵 博 蒋 庆 伍卫国

(西安交通大学电信学院计算机系 西安 710047)

摘 要 多核架构已成为当今的主流,而大量传统的串行程序和遗留软件无法充分利用多核处理器的并行计算性能.人工改写这些遗留软件工作量繁重、成本高昂,自动实现程序并行化的技术成为学术和工业界研究的热点.该文提出了一种新颖的面向一般程序的for循环优化和非规则代码段的粗粒度半自动并行化方法.该方法通过程序动态分析,根据程序的控制流和数据依赖信息将源程序代码映射成可计算单元(CU)图,从中提取出可并行执行的非规则代码段.同时针对程序中for循环部分,提出了一种基于局部性分析的分块收益模型,有效地选择具有收益的循环代码实施循环分块优化;提出了一种基于cache均匀映射的最优分块因子大小选择算法UMC-TSS,以生成优化的分块代码,充分利用cache性能并实现分块的粗粒度并行.该文实现了一个基于LLVM编译架构的C/C++源码到Intel TBB并行源码转换的半自动化工具,它在AST上进行深度代码重构,只需少量的人工干预即可生成高效的并行代码.为了验证该文方法的有效性,从4组不同的基准测试集上选取18个具有代表性的测试程序在一台Intel Xeon多核服务器上进行了一系列实验,在循环级和任务级并行性能上分别获得平均10.95和4.45的加速比.和目前最先进的一种最优分块大小算法相比,UMC-TSS算法平均提升了4%的分块代码性能.实验结果还表明由源到源代码转换工具生成的Intel TBB并行代码具有良好的并行性和可扩展性.

关键词 半自动并行化;循环分块;局部性分析;最优分块大小;源到源代码转换

中图法分类号 TP311 **DOI号** 10.11897/SP.J.1016.2017.02127

A Semi-Automatic Coarse-Grained Parallelization Approach for Loop Optimization And Irregular Code Sections

LIU Song ZHAO Bo JIANG Qing WU Wei-Guo

(*Department of Computer Science and Technology, School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an* 710049)

Abstract Although the multi-core processors have become the mainstream processor architectures of the time, it is still hard to take advantage of the parallel computing power for many serial programs and software due to the lack of efficient parallelization. Manually re-engineering and refactoring of these legacy software is time consuming and costly. Therefore, the automatic parallelization techniques become the focus of attention in academia and industry. In this article, a novel semi-automatic parallelization approach is proposed targeting on optimization for regular for-loops and coarse-grained parallelism for irregular code sections in general programs. This approach employs a dynamic program analyzer to obtain the control- and data-dependences of programs. The gathered dependences information is used to form the Computational Unit (CU) graphs, and then the task graphs are further created, from which coarse-grained task parallelism

收稿日期:2016-01-27;在线出版日期:2017-03-04.本课题得到国家自然科学基金(91630206,91330117)、国家重点研发计划(2016YFB0201800)、陕西省社会发展科技攻关项目(2016SF-428)资助.**刘 松**,男,1987年生,博士研究生,主要研究方向为程序优化、高性能计算.E-mail:lsong28@stu.xjtu.edu.cn.**赵 博**,男,1990年生,硕士,主要研究方向为多核架构、代码转换技术.**蒋 庆**,男,1991年生,硕士,主要研究方向为编译优化、机器学习.**伍卫国**,男,1963年生,博士,教授,博士生导师,中国计算机学会(CCF)高级会员,主要研究领域为高性能计算机体系结构、云计算与嵌入式系统.

of code sections can be extracted. Meanwhile, for the for-loop codes, a series of optimizations are adopted for code transformations. A profitable tiling model is proposed to effectively choose suitable loop codes for further optimization. The model is based on a mass of statistical data on locality analysis of loop iterations and it can determine whether the loop codes should perform loop tiling by invoking a loop transformation optimizer. The tile size selection (TSS) has an important impact on the performance of tiled codes and a uniform-mapping-in-cache-based tile size selection (UMC-TSS) is proposed to generate optimal tiled codes and achieve better performance during tiling. The UMC-TSS improves the method of a state-of-the-art TSS to exploit better cache utilization and loop parallelism. Eventually, a source-to-source transformation frame based on the LLVM frontend Clang is developed to transform sequential C/C++ codes to Intel TBB parallel codes. The frame is integrated with dynamic program analysis, coarse-grained parallelism extraction, loop optimizations (including the proposed profitable tiling model and UMC-TSS) and code transformations. It performs high-level code restructuring on the program abstract syntax tree. According to the task graphs, the Intel TBB parallel_for and flow graph templates are used to package the for-loops and irregular code sections into parallel codes respectively. The code transformation is semi-automatic that only a little manual effort and intervention is involved. A series of experiments have been conducted to evaluate the performance of the transformed parallel codes over 18 representative benchmarks selected from 4 different kinds of benchmark suits. The experiment results show that the parallel codes generated by the semi-automatic approach can achieve good parallelism when compared to the parallel codes written by experts, especially the codes with optimized for-loops. The average speedups of for-loops parallelization and task parallelization are 10.95 and 4.45 respectively on an Intel Xeon multi-core server. The correctness of the profitable tiling model is validated as well in the evaluation. The experiment results also show that the UMC-TSS improves the performance of 4% on average in the tiled loop codes in comparison with a state-of-the-art tile size selection algorithm. The experiment results also show that the generated Intel TBB parallel codes have good scalability when the thread number varies, which demonstrates the effectiveness of the parallelization approach and the source-to-source transformation frame presented in this paper.

Keywords semi-automatic parallelization; loop tiling; locality analysis; optimal tile size selection; source-to-source code transformation

1 引 言

多核架构已经成为目前市场的主流处理器架构.要充分利用多核处理器的计算性能,需要应用程序具有良好的并行性.大量早期串行程序编写的遗留软件无法有效地开发和利用多核架构的并行性能.编写高效的并行化代码对程序员有着较高的门槛,需要程序员能够充分理解、发掘程序的潜在并行性,并对运行环境的底层硬件架构十分了解.人工重构这些遗留软件的并行代码十分困难,并且会消耗大量的时间、人力和金钱,因此,程序的自动并行化方法吸引了学术界和工业界的极大关注,程序自动并行化技术和并行代码开发工具得到广泛的研究和应用.

不少研究从规则的循环结构中提取并行度[1-2].这些工作主要采用静态分析的方法确定循环迭代的依赖关系,通过循环变换技术,如循环偏斜(loop skewing)、循环展开(loop unrolling)、循环分块(loop tiling)[3-4]等,实现循环迭代的并行执行.循环变换技术通常只适用于某些特定的循环结构,往往需要人为地分析循环结构特征.循环变换后的并行代码性能还受变换因子(如偏斜因子、展开因子、分块因子等)的影响,而最优的变换因子选择一直是一个开放式难题.另一类动态分析的方法[5-6]则通过程序插桩来追踪程序运行时的依赖关系,挖掘程序中

非规则代码段的潜在并行性.这类方法往往需要较高的时间和空间成本,而且自动并行时缺乏高层次的代码重构和转换能力.基于投机并行的推测多线程技术(SpMT)[7-8]可以同时开发程序规则和非规则代码的线程级并行性,因而得到了大量的关注.然而,这些方法多数只能实现细粒度并行,不利于多核架构并行资源的充分利用.

传统的并行编程模型和编程库,如 Pthread、MPI 等,主要面向领域专家和资深程序员,通常适用具有规则循环代码的应用.近年来涌现出一批新型的高级并行编程模型,如 OpenMP4.0①、Cilk[9]、Cilk++[10]、Intel Concurrent Collections(CnC)[11]和 Intel Threading Building Blocks(TBB)[12]等.这些编程模型旨在实现任务级并行,提供了高层抽象的并行原语,只需使用合适的并行逻辑任务划分和同步机制,把复杂繁琐的任务调度、同步、负载均衡等统统交给运行时系统,就具有更好的易编程性、可扩展性、可移植性和可维护性,更加适合多核架构上程序并行性能的开发.实现这些高级并行模型的自动代码生成,将具有重要的实用价值.

本文实现了一种面向一般程序的规则嵌套循环自动优化、非规则代码段的任务级粗粒度并行和源到源并行代码转换的半自动化方法.利用程序动态分析工具 DiscoPoP[13-15],将程序的控制流和数据依赖映射成一个并行计算单元(Computational Unit,CU)图,根据 CU 图分别提取程序中规则循环和不规则代码段的粗粒度并行任务.针对规则的 for 循环代码实施分块优化策略,提出一种基于局部性统计分析的分块收益模型,并在此基础上提出一种基于 cache 均匀映射的分块大小选择方法,从而实现优化的分块并行.而对不规则代码段,则根据程序依赖分析所提取的 CU 图进行变换,发掘合适的粗粒度并行任务,构造可并行的 Task 图.最终设计并实现了一个基于 LLVM 编译前端 Clang 的源到源代码转换工具.该工具在抽象语法树(Abstract Syntax Tree,AST)层次上利用新型的高级并行编程模型进行深度的代码封装和重构,可将 C/C++代码以半自动化的方式转换成 Intel TBB 并行代码,只需要少量的人工操作即可生成高效的并行代码.本文从 NAS Parallel Benchmark 3.3[16]、PolyBench 3.2②、Intel CnC[11]和 PARSEC 3.0[17]等基准测试集中选择了 18 个具有代表性的测试程序,在一台多核服务器上进行了一系列的实验,实验结果验证了本文方法的有效性.本文做出的主要贡献有以下 3 点:

(1)对程序中规则的 for 循环,首次建立了一个循环分块收益模型,该模型通过统计、分析循环迭代的访存局部性信息,能有效地自动确认具有分块收益的循环代码段,为实施循环优化提供了保障.

(2)针对循环分块的最优分块因子,提出一种基于 cache 均匀映射的分块因子选择方法,该方法通过对具有分块收益的循环代码进行优化,实现了循环分块并行代码的数据局部性提升、cache 失效率降低和分块数据在 cache 中均匀映射等目标.

(3)实现了一个 C/C++源码到 Intel TBB 并行代码的转换工具,该工具有效地整合了流行的程序分析工具、编译优化框架、本文提出的循环优化方法和任务级粗粒度并行化方法,利用该工具的代码转换过程是半自动化的,只需少量简单的人工干预,且不需要特殊编译器的支持.

本文第 2 节将介绍相关工作;第 3 节将详细描述粗粒度任务的提取、针对嵌套循环优化的自动分块收益模型和最优分块因子选择方法;第 4 节介绍源到源代码转换工具的实现过程;第 5 节给出实验和性能分析;第 6 节进行全文总结.

2　相关工作

静态自动并行化方法已被成功应用于科学计算类程序的循环代码的并行化过程中.基于静态分析的一批优秀并行化编译器和工具,如 SUIF[18]、Polaris[19]、Open64③、PLuTo[20]和 PrimeTile[21]等,通过对循环迭代空间抽象成数学模型,如多面体模型[22],静态分析循环迭代的依赖关系并从中提取并行度.这类方法由于缺乏程序运行时产生的依赖信息,采用保守的数据依赖分析策略,忽略了很多程序中潜在的并行机会.而动态程序分析的方法,通过程序插桩来跟踪获取运行时的信息,提供了更精准详细的依赖关系.然而,典型的动态分析器,如 SD3[23]和 Kremlin[24]等,由于需要记录程序每次访存的详细信息,导致了较大的时间和空间开销.本文采用一种高效的程序动态分析工具 DiscoPoP,可以有效地降低动态数据依赖分析的成本,并利用该工具生成的 CU 图提取出程序的可并行代码段,实现粗粒度

① OpenMP API version 4.0. http://openmp.org/wp/2015,11,13

② PolyBench: The polyhedral benchmark suite. http://web.cse.ohio-state.edu/~pouchet/software/polybench/2015,11,13

③ Open64CompilerandTools. http://sourceforge.net/projects/open64/2015,11,13

并行性.

本文对规则的 for 循环代码采用循环分块进行优化和并行化.分块因子的选择往往决定了分块代码的性能.分块因子选择方法(Tile Size Selection,TSS)的研究主要分为静态分析、经验搜索和机器学习 3 大类.后 2 类方法往往需要高昂的时间成本代价,不适合自动并行代码转换的应用,本文研究基于静态分析的最优 TSS 方法.基于静态分析的 TSS 方法通过对硬件参数(如 cache 容量、cache 行大小等)和代码特征进行建模,选择出具有最少 cache 冲突的分块因子.Lam 等人[25]和 Ghosh 等人[26]通过建立分块因子、问题规模、cache 参数之间的关联方程来降低数据引用的自干扰失效.Coleman 和 McKinely[27]则以减少交叉干扰失效为目的建立分块因子选择模型.这些方法很难考虑全面的影响因素,因此和实际最优分块大小有较大的性能差距.最近 Mehta 等人[28]提出一种 TSS 方法,该方法考虑到之前方法所忽略、但对性能影响十分重要的因素:cache 组相联度和向量化单元的影响,因此获得了显著的性能提升和近似最优的分块大小,成为目前最为先进的一种静态 TSS 方法.

另一方面,是否实施循环分块通常由人为分析来决定,例如,循环优化框架 PLuTo 在对程序进行优化前需要人为地在待优化循环结构的前后进行标记.而对于一般程序的规则循环代码,则缺乏对其进行分块前的收益分析.相关的代价模型[29-31]往往通过提取软硬件参数,并根据代码结构和语义进行人工建模分析,实现具有收益的并行性、cache 失效、优化组合等代价模型.本文首次提出一种循环分块的收益模型,该模型根据原始循环迭代的访存信息自动判断实施分块优化是否可以带来性能收益,为并行代码自动转化中的有效优化提供保障.

之前的自动化并行技术主要用在实现细粒度的指令级并行方面,而 SpMT 技术可以实现线程级并行性.由于 SpMT 以激进的方式将程序划分为多个推测线程,在推测线程执行过程中,大量的控制和数据等依赖会导致线程撤销和重启,以保障程序的正确性.如果推测不准确,频繁的线程撤销和重启所带来的开销会严重损害程序的并行性能.而且利用 SpMT 实现程序的线程级并行通常还需要特殊的硬件或软件支持.Pei 等人[32]通过增加额外的硬件单元,针对波标量系统结构特点开发其潜在的推测多线程并行性.Li 等人[33]提出一种基于模糊聚类的方法进行线程划分,但是该方法的实现需要特殊并行编译器的支持.Ding 等人[34]设计实现了一个面向行为并行化(BOP)系统,对程序中可能的并行区域(PPR)进行标注并投机地并行执行.Ding 等人[35]和 Ke 等人[36]通过提供一系列依赖提示(dependence hints)作为接口,为用户指定 PPR 间的依赖并实现其通信、同步,利用进程来实现粗粒度的并行,但该方法仍然需要投机编译器的支持.此外,Thies 等人[5]和 Rul 等人[6]提出了一种用动态分析得到数据依赖信息的方法,基于该方法能提取出粗粒度流水线并行性,但仅适用于 C 语言编写的流处理和循环程序.Tournavitis 等人[37]结合静态和动态分析来寻找程序潜在的并行性,再用机器学习算法将这些并行性映射到不同架构的计算环境中.他们的方法也主要针对循环级的并行,采用 OpenMP 编译制导语句来标记可被并行的代码段,没有实现高层次的代码转换功能.本文方法则利用高级并行编程模型在程序的 AST 上进行并行代码的深度封装和重构,实现了任务级粗粒度并行,同时无需特殊编译器和硬件的支持.

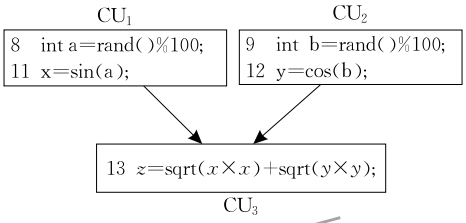
本文方法中的并行代码转换依赖于程序分析器 DiscoPoP 生成的 CU.粗粒度并行任务是通过对程序不同区域(region)[38]内的代码片段构建 CU 来提取的.区域是程序控制流图的子图,每个区域只有一个入口和一个出口,区域之间可以互相嵌套包含.在分析 CU 的方法中,程序定义的函数是顶层区域,区域定义了构建 CU 过程的边界.CU 是一个遵循"读取-计算-写回"模式的指令集合.这个过程几乎不可被并行,使得 CU 可以作为并行执行的逻辑单元.

图 1 是一个 CU 示例.源程序的第 8 行用一个随机数对整型变量 a 进行初始化,第 11 行表示读取变量 a 并执行 sin()函数的计算,最后将计算结果写回给变量 x.这一组指令构成一个 CU,记为 $CU_1=8,11$.同理,第 9 和 12 行也构成一个 CU,记为 $CU_2=9,12$.第 13 行读取变量 x 和 y 的值,计算后写回给变量 z,也构成一个 CU,记为 $CU_3=13$.其中,CU_1 和 CU_2 可以完全独立的并行执行,但是 CU_3 依赖于 CU_1 和 CU_2,如图 1(b).为了保证串行程序和并行程序运行结果一致,假设变量 a 和 b 的初始化值在串行程序和并行程序中相同.

程序分析器 DiscoPoP 将动态分析获取的控制流和数据依赖信息映射到 CU 中,生成 CU 图.控制流为树形结构,其中根节点表示整个程序,中间节点表示控制结构,如循环等,叶子节点表示基本块,它由一个区域内的 CU 构成.对具有包含关系的 CU 进行

```
1 #include <cmath>
2 #include <cstdlib>
3 using namespace std;
4
5 int main()
6 {
7       double x, y, z;
8       int a=rand()%100;
9       int b=rand()%100;
10
11      x=sin(a);
12      y=cos(b);
13      z=sqrt(x*x)+sqrt(y*y);
14
15      return 0;
16 }
```

(a) CU示例源码



(b) CU图

图 1　CU 示例

合并，并且在生成 CU 图时只考虑写后读(Read After Write，RAW)依赖. 因为读后写(Write After Read，WAR)和写后写(Write After Write，WAW)依赖可以通过对相关变量采用 privatizing 等技术消除，而 RAW 依赖无法消除，会严重阻碍程序的并行化.

3　粗粒度任务和自动循环优化方法

在进行粗粒度任务提取之前，DiscoPoP 会在动态分析时记录程序的热点，并根据程序热点的执行时间长短进行排序. 由于程序具有访存局部性，若对执行时间占程序执行总时间比重较小的代码段也进行并行化处理，很可能导致并行开销远大于性能提升. 通过大量实验发现，前 30% 的程序热点的执行时间占程序执行总时间的 80% 以上. 因此，本文只对前 30% 的热点代码段进行粗粒度任务提取和并行优化处理，从而确保有效地提高整体并行化代码的性能.

3.1　程序代码段的粗粒度任务提取

DiscoPoP 通过一种基于 signature 的内存数据结构来检测数据依赖关系，对重复执行的控制区域，如循环结构，采用融合大量冗余依赖信息的方式减小程序分析文件的空间开销，同时采用一种基于 lock-free 的并行化程序分析策略，以降低动态分析的时间开销，最终实现一种高效的程序动态分析方法[15]. 根据程序依赖分析的结果，DiscoPoP 将为程序代码段构建 CU，并根据控制流和依赖信息生成程序的 CU 图. 在源程序中提取合适的粗粒度并行任务，往往更有益于并行代码的性能. 并行粒度过大，将导致并行度降低，无法有效地充分利用多核架构的计算性能、发挥并行程序性能；并行粒度过小，则会产生较大的同步开销，不利于程序性能的提升. 本文通过对 CU 图进行变换，来提取合适的粗粒度并行任务.

首先，将 CU 图中强连通分量(Strongly Connected Component，SCC)包含的所有 CU 合并为一个粗粒度任务[39]. 这是因为 SCC 中所有的 CU 都互相依赖，难以提取并行性. 若要将其分解成并行任务也需要消除许多复杂的数据依赖，导致频繁的线程同步，引入的开销很可能会淹没并行化带来的性能提升. 把 SCC 中的 CU 提取为一个粗粒度并行任务，可以隐藏内部复杂的数据依赖关系，将潜在的并行性暴露在 SCC 外部. 不同的 SCC 之间只有简单少量的依赖关系，没有依赖的 SCC 则可以并行执行. 本文采用著名的 Tarjan 算法[40]在 CU 图中确定 SCC. 其次，将 CU 图中顺序连接并且没有分支或跳转等控制流信息的 CU 合并为一个 CU 链，将其作为一个粗粒度并行任务. 这是因为 CU 链中的 CU 虽然没有复杂的数据依赖关系，但是每一个 CU 都依赖于前一个 CU，它们也无法并行执行，将其当作不同的任务只会带来同步开销而没有任何性能收益. 这样，通过对 CU 图的合并变换成本文并行代码转换所需的 Task 图. Task 图中没有依赖的结点，如 SCC 和 CU 链，可以作为并行任务执行. 另一方面，过于庞大的 SCC 节点内部有可能存在 CU 间依赖关系薄弱的地方，而程序中不是 SCC 的 CU 群也需要找到依赖关系最薄弱的地方进行分割，以提高程序整体的并发度. 对于这种情况，采用图论中的最小割(minimum cut)方法对其进行任务划分.

DiscoPoP 的设计、实现和粗粒度任务提取的工作主要由本文作者曾经的合作单位德国亚琛工业大学研究团队完成，相关具体实现可参考文献[13-15]. 本文的工作主要是循环代码的自动优化方法、高级并行代码封装与重构和源到源并行代码转换工具.

3.2　循环代码的自动识别和优化

循环代码往往是程序中最耗时的热点，尤其是嵌套循环，对其采用合适的优化将会有效地提高程序性能. 循环分块(loop tiling)[3-4]技术是一种有效的优化手段. 一方面，它利用仿射变换改变循环迭代的访存顺序，提高数据局部性，充分利用分块数据重用，减少数据移动和 cache 失效. 另一方面，利用合

适的循环变换技术,通过消除数据依赖,循环分块技术可以实现分块之间和块内数据的并行[41].

for 循环代码可以分为不携带跨迭代依赖的 DOALL 循环、携带跨迭代依赖的 DOACROSS 循环和只能串行执行的循环[42]. DOALL 循环可以直接实现各迭代的完全并行,DOACROSS 循环则需要对迭代间的依赖关系进行处理后实现流水并行或波阵面并行[43],而第 3 种循环无法并行因而不在本文讨论范畴.本文采用一个高层的面向循环优化的架构 PLuTo[20]对 for 循环代码进行分块和并行化处理,它用抽象的数学表达形式(多面体模型)来实现符合循环变量依赖关系的循环变换和访存优化,从而为 DOALL 和 DOACROSS 循环提供粗粒度并行和优化的条件.

DiscoPoP 可以有效地分析识别出 for 循环代码.本文实现的源到源代码转换工具会对循环代码进行标记,并调用 PLuTo 进行相应的循环变换,改变迭代变量的依赖向量方向,保证后续并行化的合法性,最终生成优化后的分块循环代码.由于分块循环代码经过 strip-mining 后原始循环代码行数信息会发生变化(通常情况下变为 2 倍),程序的 AST 信息也随之发生改变,因此需要重新获取最新的 AST 信息,以确保代码转换时的正确性.

3.3 基于局部性的分块收益模型

循环分块方法在提供更好的访存局部性和粗粒度并行性的同时,也会增加程序的控制流和代码量,带来一定的开销.待转换的 for 循环代码是否实施分块优化策略,需要对循环分块的收益进行分析,否则容易发生分块引起的开销大于其带来的性能收益的情况,导致分块后的并行代码性能低于直接并行化代码的性能.

首先,对影响循环分块收益的因素进行分析. DOALL 循环由于不存在迭代间依赖,可以完全的并行执行,因此循环分块对 DOALL 循环的收益主要来自访存局部性的提升.当 DOALL 循环迭代访问的数据具有重用性时,分块可以有效地解决因 cache 容量限制而导致频繁的数据重用失效问题.否则,分块只会带来不必要的开销,降低循环代码的执行效率. DOACROSS 循环由于迭代间存在依赖关系,需要额外的处理机制才能实现流水并行或波阵面并行.本文利用 PLuTo 内核对 DOACROSS 循环进行并行化,对提取的迭代依赖多面体进行偏斜(skewing)变换以保证依赖方向的统一,而后进行分块处理,实现有序地并行执行. DOACROSS 循环通常具有数据重用性,如典型的 stencil 计算.分块的并行开销除了控制流和代码量的增加,还有额外的同步开销.如果迭代次数较小,数据重用的次数和重用距离也相应较小,此时 DOACROSS 循环的并行化将会产生大量的开销,难以实现性能收益.因此,这种情况不应对其进行并行化处理,而是直接串行执行.

由以上分析可知,循环分块的收益主要取决于两个因素:原始循环的访存数据是否存在重用;重用距离是否大于 cache 容量.对于 DOALL 循环,如果访存的数据不存在重用,或者可重用的数据很少,那么循环分块无益于并行性能提升.例如,对 NPB 测试程序 BT 和 FT 的 DOALL 循环代码分别用直接并行化和分块并行化进行实验,均采用 16 线程时,两者直接并行的代码加速比为 3.67 和 9.55,而分块后并行的代码加速比为 3.58 和 9.04.对这两组程序中并行处理的循环代码分析可知,它们的访存数据缺乏重用性.另一方面,当数据重用存在时,需要考虑最大重用距离.考虑到 cache 的数据替换策略,如果最大重用距离小于 L1 cache 容量,那么所有的数据都能在 cache 中得到充分重用.此时,对 DOALL 循环无需分块,可以直接并行化处理;对 DOACROSS 循环而言,说明迭代次数较小(特别是最外层循环),由前文分析可知这种情况下并行化只会产生负收益,因此采用串行执行.

本文首次针对一般程序中的 for 循环代码提出一种基于局部性统计分析的分块收益模型,能自动地确认循环分块对 for 循环并行性能的影响.首先选取一批具有 DOALL 循环和 DOACROSS 循环的程序,根据多级 cache 的容量分别选择不同的迭代次数,以保证这些 for 循环的分块并行执行对性能提升恰好具有积极作用.然后分别对这些 for 循环的原始代码插桩运行,记录每个测试循环的访存地址序列.根据访存地址序列分别计算数据重用次数和相应的重用距离.接着,对计算出的大量数据进行统计和分析.本文发现当具有数据重用的访存地址超过 for 循环的访存地址的 70%,且相应的重用距离大于 L1 cache 容量的地址序列占到这些具有数据重用的地址序列的 15%以上时,循环分块能够获得收益.这个统计分析的过程是离线完成的,不会增加本文提出的并行化和优化方法的运行时开销.

访存地址序列的重用距离计算采用经典的 Bennett-Kruskal 算法[44],通过位向量 B 动态地标记每个内存地址的最近一次访问,计算 B 中当前内存地址和上一次访问该地址之间"1"的位数得到重

用距离.该算法的时间复杂度为 $O(N\log(N))$,优于直接计算整个访存地址序列的时间开销.由于本文提出的并行化和优化方法只实施于程序的前 30% 热点,提取的 for 循环往往具有较大的迭代次数和串行执行时间,因此记录的访存地址序列也十分庞大.对大规模的访存地址序列计算重用次数和相应的重用距离会花费大量的时间,不利于并行代码在线优化和生成.鉴于循环迭代访存的周期性规律,使用采样的方法只需对一小部分连续的访存地址序列进行统计分析,其结果即可近似地反映全部地址序列的局部性收益情况.本文方法随机地从某个访存地址开始统计,采样率为 $2.5/N$,N 为最外层循环迭代次数.这样可以保证至少连续 2 个最外层迭代的访存地址都能进行统计.

本文提出的分块收益模型仅适用于迭代步长为常数的 for 循环.具有重用的访存地址所占比例反映了循环迭代是否具有数据重用性,这是循环代码的固有特征,也是分块具有收益的必要条件,与运行环境的计算机硬件架构无关.重用距离大于 L1 cache 容量的访存地址所占比例则反映了循环迭代是否存在因 cache 容量引起频繁的 cache 失效情况.因为 L1 cache 容量最小,只需记录能够引起 cache 失效的最小重用距离即可,而且本文是对重用距离超过 L1 cache 容量的数据进行统计,所以不用考虑 L2、L3 cache 的失效情况.值得注意的是,当数据重用距离小于 L1 cache 容量时,也有可能因为在重用前数据所映射的 cache 组的有效路数被其他数据占满,从而被替换出去.然而,这种情况相对于 cache 容量造成的数据重用失效还是相对较少,而且本文通过大量实际循环代码的统计分析得到的数据也包含了发生的这种情况.当循环中没有代码行号小的变量依赖于行号大的变量或者依赖自身变量时,该循环为一个 DOALL 循环.DiscoPoP 可以据此识别出 DOALL 循环.无论 DOALL 循环是否具有分块收益,都将自动地进行并行代码转换.而只有具有分块收益的 DOACROSS 循环才会调用 PLuTo 进行循环变换,实现流水并行.

3.4　基于 Cache 均匀映射的分块大小选择算法

分块的大小将严重影响变换后循环代码的性能.最优分块大小和最差分块大小获得的分块代码性能差异可高达数十倍.分块过大,将难以充分发掘 cache 中数据的重用性,循环分块的并行度也有待提高;分块过小,循环分块带来的成本开销和分块并行执行时的通信开销将会淹没因分块带来的性能提升[45].

本文在目前最为先进的一种 TSS 方法[28]的基础上提出了一种基于 cache 均匀映射的最优 TSS 方法.Metha 等人[28]在建立 TSS 模型时加入了 cache 组关联度和向量化单元等因素,通过模拟分块数据在 cache 中的映射行为来避免数据在重用前被替换的情况发生,同时计算出的分块因子还能充分利用多级 cache 资源,从而获得了近似最优的分块大小和良好的分块代码性能.该 TSS 算法以 L1 和 L2 cache 的总失效率 TCM 和数据重用率 RR 为目标函数.以矩阵乘法的 kernel 为例,如图 2 所示,图 2(a)为原始代码,图 2(b)为经过循环分块后的伪代码,其中 I、J 和 K 分别表示相应循环的分块大小.假设对最外层的块内循环(即 i 循环),矩阵的数据引用可以在 L1 cache 中实现重用;对最内层的块间循环(即 kT 循环),矩阵的数据引用可以在 L2 cache 中实现重用.通过分析可得 L1 和 L2 cache 的 TCM 和 RR,如式(1)、(2)所示.其中 CLS 表示一个 cache 行容纳的数据数目.

$$\mathrm{TCM}_{\text{L1 cache}}=\left(\frac{1}{I}+\frac{1}{J}+\frac{1}{K}\right)\times\frac{N^3}{CLS} \tag{1}$$

$$\mathrm{RR}_{\text{L1 cache}}=\frac{K\times J}{K\times J+2\times J+K+1}$$

$$\mathrm{TCM}_{\text{L2 cache}}=\left(\frac{1}{I}+\frac{1}{J}+\frac{1}{N}\right)\times\frac{N^3}{CLS} \tag{2}$$

$$\mathrm{RR}_{\text{L2 cache}}=\frac{I\times J}{K\times(I+1)+2\times K\times J+I\times J}$$

数组 $B[k][j]$ 和 $C[i][j]$ 被认为是在 L1 和 L2 cache 中具有最多的重用次数而占据工作集的主导数组.算法将模拟这两组数组的分块在 cache 中的映射存储行为,并生成不会因 cache 组相联度造成失效的分块大小候选列表,从中选择满足 TCM 最小、RR 最大的分块作为最优分块大小.

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    for(k=0;k<N;k++)
      C[i][j]=C[i][j]+A[i][k]*B[k][j];
```

(a) 原始循环

```
for(iT=0;iT<N/I;iT++)
  for(jT=0;jT<N/J;jT++)                    } 块间循环
    for(kT=0;kT<N/K;kT++)
      for(i=iT*I;i<iT*I+I-1;i++)
        for(k=kT*K;k<kT*K+K-1;k++)         } 块内循环
          for(j=jT*J;j<jT*J+J-1;j++)
            C[i][j]=C[i][j]+A[i][k]*B[k][j];
```

(b) 分块后循环

图 2　矩阵乘法 Matmul 的循环示例

然而经分析发现,当分块大小尚未确定时,数组 C 并不一定是 L2 cache 中工作集的主导数组.根据

式(2),为了保证 L2 cache 的 TCM 最小,RR 最大,需要选择使$\left(\frac{1}{I}+\frac{1}{J}\right)$值最小的 I,J,即 I 和 J 的值尽可能的取大.而根据式(1),则会尽量选择最大的 K 和 J 值.如果最终选择的 K 值较大,对于 L2 cache 而言,将数组 C 作为主导数组的方法有失公平,这是因为 K 值较大时,数组 $A[i][k]$和 $B[k][j]$ 占工作集的比例相对于数组 C 远没有到可以忽略的地步.数组 A 和 B 也会占用部分 L2 cache 的空间,映射到 cache 组中从而造成可能的冲突.例如,对于问题规模为 2000,数据类型为 double 型的矩阵乘法,该算法选出的(I,J)为(168,104),比实际最优分块因子(96,160)偏大.此外,以一个 cache 组首次被占满即将发生数据替换为条件,终止对主导数组分块的模拟映射,还会导致出现不能充分利用 cache 容量的情况.因为一个分块的数据在不同 cache 组中的映射分布可能不均衡,此时部分 cache 组尚存在较多未被占用的有效路数.例如,对于同样规模的矩阵乘法,该算法过于保守的冲突避免策略对 L1 cache 选出的(K,J)值为(32,104),比实际最优分块因子(32,160)偏小.

本文针对文献[28]算法的不足,提出一种基于 cache 均匀映射的最优分块因子选择方法 UMC-TSS(Uniform-Mapping-in-Cache-based Tile Size Selection).UMC-TSS 根据 cache 的 LRU 替换策略分别计算工作集满足 L1 和 L2 cache 容量的分块大小作为候选分块因子序列,如式(3)所示:

$$\begin{cases} K\times J+2\times J+K+1\leq \text{L1\_cache\_size} \\ (I+1)\times K+2\times K\times J+I\times J\leq \text{L2\_cache\_size} \end{cases} \tag{3}$$

对满足式(3)的候选分块大小,分别对其工作集进行 L1 和 L2 cache 中的映射模拟.UMC-TSS 并不以 cache 组冲突发生为映射终止条件,而是允许 cache 组冲突的发生,通过一个 UM 模型来衡量分块数据在 cache 中均匀映射的情况,从而选择最优的分块大小.UM 模型如式(4)所示:

$$UM=\sum_{i=0}^{nSets}(n_e-emu[i])^2+\sum_{i=1}^{overcnt}penalty \tag{4}$$

其中,n_e表示一个 cache 组中的有效路数,即可以使用的 cache 行数目;数组 $emu[i]$用来记录总数为 $nSets$ 的 cache 组中第 i 个 cache 组已经使用的 cache 行数目;$overcnt$ 表示 cache 冲突的次数;$penalty$ 为惩罚因子,其值为n_e^2,表示冲突发生时需要额外加上 $penalty$ 值.UM 模型的前半部分为均匀映射项,其值越小表示数据在 cache 中映射越均匀;后半部分为冲突惩罚项,其值越小表示 cache 冲突次数越少.UMC-TSS 将首先对 L1 cache 模拟分块工作集的映射,计算每对候选分块因子的 UM 值,接着选择具有较小 UM 值的候选分块因子进行 L2 cache 的工作集映射,并找到具有最小 UM 值的分块,作为最优分块大小.以图 2 的矩阵乘法为例,UMC-TSS 算法计算最优分块大小的过程示意图如图 3 所示.

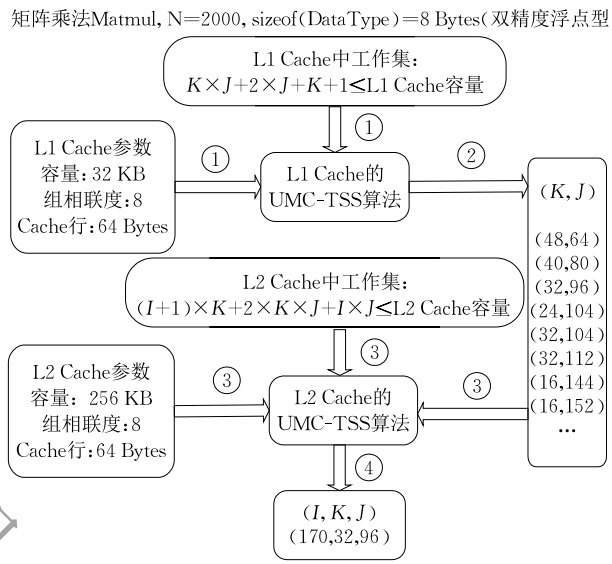


图 3 UMC-TSS 算法计算最优分块大小的过程

对 L1 cache 来说,数组 $B[k][j]$对应的数据占据了总工作集的绝大部分.由于数据在 cache 中的映射是一个复杂的过程,为了更好的模拟该过程,对其采取如下简化:令 $n_e=n-1$,n 是 cache 组相联度,让数组 B 的 $K\times J$ 个数据占用 cache 组中的有效路数,即 $n-1$ 路;余下的一路 cache 行则留给工作集中的其他数据.针对 L1 cache 的 UMC-TSS 实现如算法 1 所示.根据式(3),K,J 的值应该尽量取大一些,为了避免较小的 K,J 值计算,降低算法执行时间,引入一个控制变量 cnt 来限制工作集的下限.为了提高 L2 cache 中算法的执行效率,引入 $limitval$ 和 $limitcnt$ 两个阈值,其使用见算法 1 的第 26 行中前 2 个判断条件.本文中两个阈值的具体取值为最小 UM 值和最少冲突次数的 1.3 倍,这样可以保证具有较小的 UM 值和冲突次数的优秀候选分块因子才会进行下一步 L2 cache 中的计算,提高算法效率.同时,最后一个判断条件保证了最内层循环的分块因子具有较大的取值,有利于充分利用 SIMD单元的并行能力.此外,为保证空间局部性的

最大化，数组的列下标所对应循环层的分块大小应尽量选取为 CLS 的整数倍，如算法 1 的第 27～31 行.

算法 1.　L1 cache 的 UMC-TSS 算法.

输入：L1 cache 容量 $cSize1$；L1 cache 组相联度 n_1；cache 行大小 $lSize$；L1 cache 包含的 cache 组数目 $nSets1(=cSize1/(lSize\times n_1))$；一个 cache 行容纳的数据数目 $CLS(=lSize/sizeof(\text{DataType}))$；矩阵规模 N

输出：候选分块因子序列 $L(K,J)$

```
1.  n_e=n_1-1; //有效 cache 路数
2.  nLine=nSets1×n_e;//L1 cache 所有有效 cache 行数目
3.  Jbound=min(nLine/CLS,N/CLS); //J 的最大值
4.  for J=1 to Jbound do
5.   cnt=15; //控制变量取值 15
6.   Kbound=max(min(nLine/i,N),15+CLS);
7.   repeat
8.    overcnt=0;
9.    temp=0.0;
10.   for i=0 to nSets1 do
11.    emu[i]=0;
12.   endfor
13.   for row=0 to (Kbound-cnt) do
14.    for col=0 to J do
15.     emu[(row×N/CLS+col)%nSets1]++;
            //模拟 K×J 工作集在 cache 中的映射
16.    endfor
17.   endfor
18.   for i=0 to nSets1 do
19.    if emu[i]<=n_e then //计算 UM 值
20.     temp+=(n_e-emu[i])×(n_e-emu[i]);
21.    else //发生冲突则增加惩罚项
22.     temp+=(emu[i]-n_e)×(emu[i]-n_e)+n_e*n_e;
23.     covercnt++;
24.    endif
25.   endfor
26.   if temp<=limitval && overcnt<=limitcnt
        &&(J×CLS)>(Kbound-cnt) then
27.    if (Kbound-cnt)<CLS then
28.     Add (Kbound-cnt, J×CLS) into L;
29.    else
30.     Add((Kbound-cnt)/CLS×CLS,J×CLS) into L;
31.    endif
32.   endif
33.   cnt--;
34.  until cnt=0
35. endfor
```

对 L2 cache 来说，数组 $A[i][k]$，$B[k][j]$ 和 $C[i][j]$ 的数据共同构成了总工作集 $(I+1)\times K+2\times K\times J+I\times J$，cache 组的有效路数就等于 cache 组相联度. 由于算法 1 已经确定了较好的候选分块因子序列 $L(K,J)$，扫描序列 L，对每对 K 和 J 的值以 I 为变量模拟整个工作集在 L2 cache 中的映射，同时计算相应的 UM 值. 扫描完整个序列 L 就可以计算出最优的分块因子，即具有最小 UM 值的 (I,K,J). 针对 L2 cache 的 UMC-TSS 实现如算法 2 所示.

算法 2.　L2 cache 的 UMC-TSS 算法.

输入：候选分块因子序列 $L(K,J)$；L2 cache 容量 $cSize2$；L2 cache 组相联度 n_2；L2 cache 包含的 cache 组数目 $nSets2(=cSize2/(lSize\times n_2))$；一个 cache 行容纳的数据数目 $CLS(=lSize/sizeof(\text{DataType}))$；矩阵规模 N

输出：最优分块因子大小 (I,K,J)

```
1.  pointer=L;
2.  while L is not empty && pointer is not at tail of L
3.   K=L.K;
4.   J=L.J;
5.   Ibound=min(N,(cSize2/sizeof(DataType)-
              K-2×K×J)/(K+J)); //I 的最大值
6.   for I=8 to Ibound do //I 的值应该大于一个 CLS
7.    overcnt=0;
8.    temp=0.0;
9.    for i=0 to nSets2 do
10.    emu[i]=0;
11.   endfor
12.   for row=0 to (I+1) do
13.    for col=0 to K/CLS do
14.     emu[(row×N/CLS+col)%nSets2]++;
            //模拟数组 A 的工作集在 cache 中的映射
15.    endfor
16.   endfor
17.   for row=0 to I do
18.    for col=0 to J/CLS do
19.     emu[(row×N/CLS+col)%nSets2]++;
            //模拟数组 C 的工作集在 cache 中的映射
20.    endfor
21.   endfor
22.   for row=0 to 2×K/CLS do
23.   for col=0 to J/CLS do
24.    emu[(row×N/CLS+col)%nSets2]++;
            //模拟数组 B 的工作集在 cache 中的映射
25.   endfor
26.  endfor
27.  for i=0 to nSets2 do
28.   if emu[i]<=n_2 then //计算 UM 值
29.    temp+=(n_2-emu[i])×(n_2-emu[i]);
30.   else //发生冲突则增加惩罚项
```

```
31.      temp+=(emu[i]-n2)×(emu[i]-n2)+n2*n2;
32.      covercnt++;
33.    endif
34.  endfor
35.  if temp<tuple.val then //记录最小 UM 值
                            //对应的分块大小
36.    tuple.val=temp;
37.    tuple.I=I/CLS×CLS; //空间局部性优化
38.    tuple.K=K;
39.    tuple.J=J;
40.  endif
41. endfor
42. pointer++;
43. endwhile
44. Output (tuple.I,tuple.K,tuple.J);
```

PLuTo 不提供 TSS 算法，每层循环默认采用 32 作为分块因子. 将 UMC-TSS 算法集成到本文实现的源到源并行代码转换工具中，在调用 PLuTo 对 for 循环进行分块优化时启动，将计算出的最优分块大小写到 tile. sizes 文件中供 PLuTo 生成相应的优化代码. 需要注意的是，尽管该算法计算最优分块大小时所需的软硬件参数均可自动获得，但式(3)和算法 1、2 仅适用于包含类似矩阵乘法循环结构的科学计算 kernel，如 BLAS 3 库中的 Matmul、Dsyrk、LU 等以及 Jacobi 迭代和 Seidel 方程等. 对于一般程序中的 for 循环语句，用户可以根据表 1 和表 2 所总结的数组数据在 L1 和 L2 cache 中的工作集大小对式(3)中左侧工作集和 UMC-TSS 算法中工作集的模拟映射代码进行修改. 表中第 1 列和第 1 行分别表示数组的行和列下标. 表 1 中工作集大小的计算前提假设是块内循环顺序为 i,j,k，表 2 的前提假设是最内层块间循环为 jT，块内循环顺序为 i,j,k. 当循环变量顺序改变时，用户只需按新的循环顺序改变表中的下标和相应的分块大小变量即可.

表 1 L1 cache 中数组的工作集

下标	i	j	k
i	1	$J+1$	$2\times K$
j	$J+1$	J	$J\times K$
k	$2\times K$	$J\times K$	K

表 2 L2 cache 中数组的工作集

下标	i	j	k
i	I	$(I+1)\times J$	$I\times K$
j	$(I+1)\times J$	J	$2\times J\times K$
k	$J\times K$	$2\times J\times K$	K

4 半自动化并行代码的生成

本文在作者之前工作[46]的基础上进行了深度扩展，有效地集成了高效程序依赖分析、粗粒度并行任务提取和自动循环优化技术，并实现了一个源到源并行代码转化工具，利用 C++11 标准的新特性在 AST 层面上进行深度代码重构，最终提出了一种面向一般程序的粗粒度半自动并行化模型. 它的工作流程如图 4 所示，分为 3 个阶段. 第 1 阶段，程序被送入 LLVM 编译前端 Clang，通过语法分析生成 AST，同时 DiscoPoP 对源程序的 LLVM 中间代码进行一系列的静态和动态分析，生成 CU 图并识别出 for 循环. 第 2 阶段，对 CU 图进行合并以获取合适的粗粒度并行任务，并将其映射为 Task 图. 同时，对 for 循环代码进行分块收益分析，通过循环分块收益模型的 for 循环将调用循环优化框架 PLuTo，根据 UMC-TSS 算法获得的最优分块因子以生成优化后的循环分块代码. 由于此时原始代码发生变化，需要重新获取当前代码的 AST. 而没有通过收益模型的 for 循环则进入下一阶段进行代码封装(DOALL 循环)或者终止并行代码转换(DOACROSS 循环).

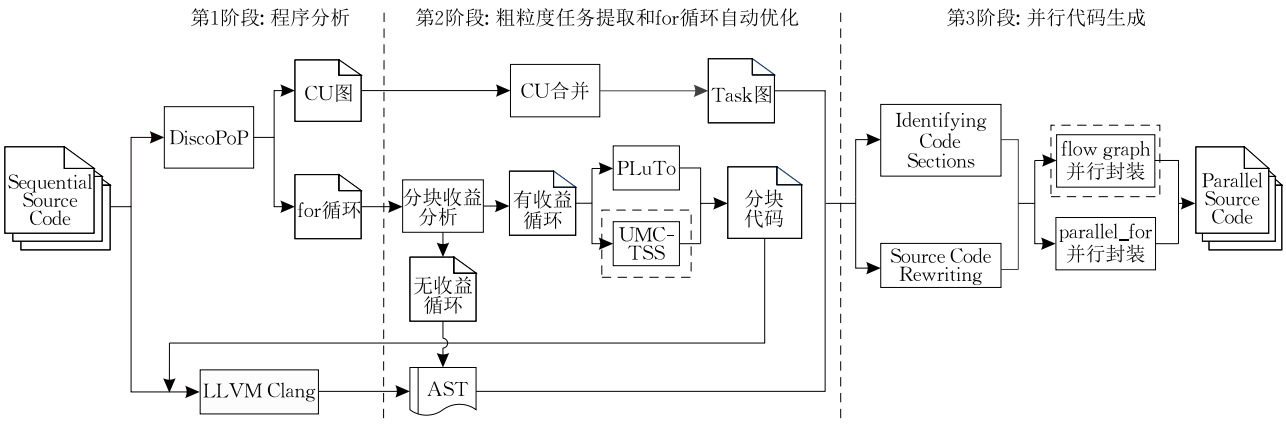


图 4 粗粒度半自动并行化模型的工作流程

第 3 阶段，根据 Task 图遍历 AST，由 Identifying Code Sections 模块定位 for 循环和 Task 图中每个 Task 所对应代码段的位置，然后由 Source Code Rewriting 模块调用 Intel TBB 的 parallel_for 和 flow graph 并行模板在相应的位置进行代码封装和重构，最终生成具有粗粒度并行性的 Intel TBB 并行代码.

图 4 中 2 个虚线框内的部分是整个并行化过程中需要用户干预和操作的地方. 计算最优分块因子时可能需要用户根据循环语句更改工作集大小，此外，在进行 flow graph 并行模板重构时，需要用户根据程序语义为同步结点指定缓冲策略，而其他部分都可以自动完成.

4.1　parallel_for 并行代码生成

可并行的 for 循环代码将由 Intel TBB 提供的 parallel_for 模板进行并行代码转换. 目标 for 循环必须遵循 *for*(*initialization*; *condition*; *increase*) *loop_body* 的形式. 其中 *condition* 子表达式必须是一个二元操作符. 循环边界可以是常量也可以是变量，但是在循环的执行过程中，循环边界不能改变. LLVM Clang[①] 提供了一系列的方法来提取 *initialization*、*condition* 和 *increase* 子表达式以及 *loop_body* 子表达式. 由于 parallel_for 中的迭代空间是左闭右开区间[*begin*, *end*)，因此需要考虑待转换的 for 循环的迭代空间形式. 若不符合左闭右开区间的形式，则需要对其做特殊处理并映射到等价的左闭右开区间中. 需要注意的是，这只是形式上的变换，其实际的物理迭代空间并没有发生改变.

源到源转换工具用递归的方式遍历程序的 Clang AST，当遍历到目标循环时，代码转换就被激活并开始执行. 整个代码转换过程分为 4 步. 第 1 步，利用 Clang 提供的 *getCond*()方法提取循环中的 *condition* 子表达式，以确定循环控制变量和循环上界；第 2 步，利用 *getInit*()方法提取 *initialization* 子表达式，以确定循环的下界；第 3 步，利用 *getInc*()方法提取 *increase* 子表达式，以确定循环的步长；第 4 步，利用 *getBody*()方法提取 *loop_body*，将其作为字符串存储，源到源代码转换工具中的 Source Code Rewriting 模块将代码中原来的 for 循环删除，在该位置插入 parallel_for 模板，并用 C++11 中的 *lambda* 函数重构循环体 *loop_body*. 具体的实现可参考文献[46].

图 5 展示了图 2 矩阵乘法 Matmul 的循环代码转换成 parallel_for 并行循环代码的示例. 图 5(a)是经过 PLuTo 变换后的循环分块代码. 图 5(b)为源到源代码转换工具生成的 Intel TBB 并行代码，其中每层循环迭代的次数为 2000，分块大小为 170×32×96.

```
for (int t1=0;t1<=floord(M-1,170);t1++)
  for (int t2=0;t2<=floord(N-1,96);t2++)
    for (int t3=0;t3<=floord(K-1,32);t3++)
      for (int t4=170*t1;t4<=min(M-1,170*t1+169);t4++)
        for (int t5=32*t3;t5<=min(K-1,32*t3+31);t5++)
          for (int t6=96*t2;t6<=min(N-1,96*t2+95);t6++)
            C[t4][t6]=C[t4][t6]+A[t4][t5]*B[t5][t6];
```

(a) PLuTo分块后的循环代码

```
parallel_for(blocked_range<int>(0,floor(((double)(2000-1))/((double)(170)))+1),
[&](blocked_range<int>&_range){
for(int t1=_range.begin();t1!=_range.end();++t1)
{
  for (int t2=0;t2<=floor(((double)(2000-1))/((double)(96)));t2++){
    for (int t3=0;t3<=floor(((double)(2000-1))/((double)(32)));t3++){
      for (int t4=170*t1;t4<=((2000-1)<(170*t1+169)?(2000-1):(170*t1+169));t4++){
        for (int t5=32*t3;t5<=((2000-1)<(32*t3+31)?(2000-1):(32*t3+31));t5++){
          for (int t6=96*t2;t6<=((2000-1)<(96*t2+95)?(2000-1):(96*t2+95));t6++){
            C[t4][t6]=C[t4][t6]+A[t4][t5]*B[t5][t6];
          }
        }
      }
    }
  }
}});
```

(b) parallel_for并行封装后的循环代码

图 5　parallel_for 并行代码转换示例

① Clang: A C language family frontend for LLVM. http://clang.llvm.org/2015,11,13

4.2 flow graph 并行代码生成

Intel TBB 的 flow graph 模板允许用户表达任意任务形式的并行，只需指明任务以及任务之间的依赖关系. Intel TBB 的运行时系统会负责同步、负载均衡等复杂的任务调度问题. flow graph 包含 flow graph 对象、结点和边 3 个要素. flow graph 对象是为整个 flow graph 创建任务的持有者，它拥有着资源；flow graph 结点用来执行计算，边表示结点之间的依赖关系. flow graph 在逻辑拓扑上与 Task 图非常相似. 源到源并行代码转换工具根据粗粒度任务的 Task 图进行相应的 flow graph 代码重构和封装.

以图 1 中 CU 图为例，经过合并转换后生成的 Task 图如图 6 所示. 其中，$Task_0$ 是一个虚拟任务，不进行任何计算，只负责传递消息给后继节点. Task 图生成后，直接调用源到源代码转换工具生成 flow graph 并行代码，其具体实现可参考文献[46].

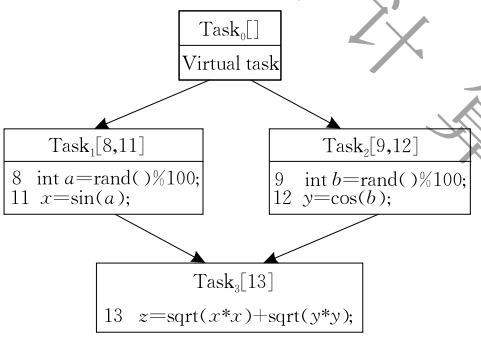


图 6 Task 图示例

粗粒度任务的 flow graph 代码转换的过程主要分为 3 步.

第 1 步：确定每个 Task 对应的代码段. 对 Task 图中每个任务 $Task_i$（包括 $Task_0$），遍历 Clang AST. 当 AST 结点的源代码行号存在于 $Task_i$ 中时，Identifying Code Sections 模块将其作为字符串存储在任务对应的对象中.

第 2 步：生成 flow graph 结点的源代码. Source Code Rewriting 模块将根据 3 种不同的情况，分别生成 flow graph 结点的源代码.

(1) 当前任务 $Task_i$ 有一条或者零条入边，多条出边. 如果它的后继结点都接受相同的数据（即依赖于同一变量），那么 Source Code Rewriting 模块插入一个 Intel TBB 的 broadcast_node 结点，否则插入一个 Intel TBB 的 split_node 结点. split_node 结点能够将不同类型的数据传递给对应的后继结点. 如图 6 中的 $Task_0$，将插入 broadcast_node 结点相应的代码.

(2) 当前任务 $Task_i$ 有一条入边，一条或者零条出边. Source Code Rewriting 模块直接用 lambda 表达式将其转换为 Intel TBB 的 function_node 结点. 图 6 中的 $Task_1$ 和 $Task_2$ 就是这种情况.

(3) 当前任务 $Task_i$ 有多条入边，一条出边. 这意味着至少有两个变量需要同步后再传入 $Task_i$. 因此，首先插入一个 Intel TBB 的 join_node 结点进行同步，然后再插入 function_node 结点完成 $Task_i$ 的实际计算. 图 6 的 $Task_3$ 就属于这种情况. join_node 结点具有多个入口，每个入口的数据缓存到一个多元组中作为单一的输出.

其中，join_node 结点的输入端口支持 3 种缓冲策略：

① queueing 缓冲策略. join_node 的每个入口都有一个先进先出队列，只要每个队列中至少存在一个元素时，join_node 结点就贪婪地消费所有元素，生成一个多元组广播给该结点的所有后继结点. 当至少有一个后继结点成功接收到多元组后，join_node 将所有队列中的队首元素删除. 否则，维持所有队列不变.

② reserving 缓冲策略. 只有当 join_node 结点能够成功保留每个入口的一个元素时，它才会生成一个多元组传递给所有的后继结点. 如果不能保留所有入口的元素，就释放已经保留的元素，直到在上次不能成功保留元素的入口收到消息时，才重新试图获取保留每个入口的元素.

③ tag_matching 缓冲策略. 由用户提供的函数为每个入口元素计算相应的 *key* 值. join_node 结点用哈希表缓冲每个入口的元素. 只有当每个入口的元素具有相同的 *key* 时，join_node 结点才将入口元素生成一个多元组传递给所有的后继结点，并删除相应的入口元素.

通常情况，queueing 缓冲策略适用于大多数的同步问题，例如生产者消费者问题，因此 join_node 结点的输入端口默认采用 queueing 缓冲策略. 对于诸如哲学家就餐问题这类资源竞争激烈、容易产生死锁和资源耗尽的情况，建议采用 reserving 缓冲策略. 需要注意，采用 reserving 缓冲策略的 join_node 的所有前驱结点必须是可保留类型的结点，其它 2 种缓冲策略则没有这个要求. 当 join_node 结点的前驱结点可能产生多种数据时，则采用 tag_matching 缓冲策略. 或者，当前 2 种缓冲策略都难以有效实现多资源分配问题时，也可以采用该种缓冲策略，避免死锁发生. 缓冲策略的选择是一个语义问题，

程序员需要根据程序代码段的语义手动指定合适的缓冲策略，自动并行化方法目前无法解决该问题. 源到源代码转换工具会向用户报告在何处加入 join_node 结点，并且给出已经分析得到的数据依赖和控制流信息，需要用户自己选择合适的缓冲策略作为参数输入到源到源转换工具. 3 种采用不同缓冲策略的 join_node 结点代码如图 7 所示.

```
tbb::flow::join_node<tbb::flow::tuple<type_1,type_2,...,type_n>,
                     tbb::flow::queueing>joinNodeID(g);
```

(a) 采用queueing缓冲策略的join_node代码

```
tbb::flow::join_node<tbb::flow::tuple<type_1,type_2,...,type_n>,
                     tbb::flow::reserving>joinNodeID(g);
```

(b) 采用reserving缓冲策略的join_node代码

```
tbb::flow::join_node<tbb::flow::tuple<type_1,type_2,...,type_n>,
tbb::flow::tag_matching>joinNodeID(g,
   [](const pair<type_1,tag_type>&p)->size_t
   {return (size_t)p.first; },
   [](const pair<type_2,tag_type>&p)->size_t
   {return (size_t)p.first; },
   ...
   [](const pair<type_3,tag_type>&p)->size_t
   {return(size_t)p.first; }
 );
```

(c) 采用tag_matching缓冲策略的join_node代码

图 7　采用不同缓冲策略的 join_node 代码

图 6 的 $Task_3$ 选择 queueing 缓冲策略. Source Code Rewriting 模块从 AST Context 中获取 $Task_3$ 中每个输入变量的类型，然后将采用 queueing 缓冲策略的 join_node 代码插入到 $Task_3$ 源代码的最后一行. 同时，将 join_node 结点作为当前任务 $Task_3$ 的前驱结点加入到 Task 图中，然后插入 function_node 结点.

对于有多条入边和出边的结点，可以由多入边单出边、单入边单出边和单入边多出边的结点共同实现. 首先加入一个 join_node 结点，然后加入一个 function_node 结点，最后加入一个 split_node 结点.

第 3 步：生成 flow graph 边的源代码. 当所有 flow graph 结点都在源代码中定义后，接下来要根据 Task 图生成边的源代码. 需要注意，从 broadcast_node 结点到 function_node 结点的边定义与从 function_node 结点到 join_node 结点的边定义形式不同. 图 6 所示的 Task 图中，从 $Task_0$ 到 $Task_1$、$Task_2$ 的 2 条边与从 $Task_1$、$Task_2$ 到 $Task_3$ 的 2 条边就采用不同的定义.

当 Task 图中所有的结点和边都在源代码中定义后，源到源代码转换就此终止，随之生成并行源代码. 图 1(a) 所示的代码最终映射的 flow graph 图和转换后的并行代码分别如图 8 和图 9 所示. 其

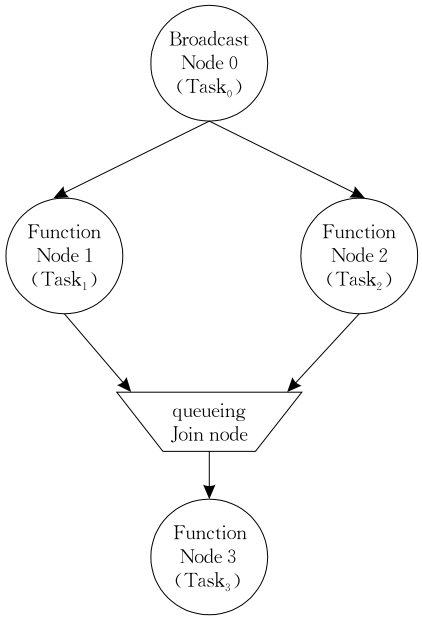


图 8　示例 Task 图对应的 flow graph 图

```
#include "tbb/flow_graph.h"
#include <cmath>
#include <cstdlib>
using namespace std;

int main()
{
   tbb::flow::graph g;
   double x,y,z;

   tbb::flow::broadcast_node<tbb::flow::continue_msg>Node0(g);
   tbb::flow::function_node<tbb::flow::continue_msg,double>
      Node1(g,tbb::flow::unlimited,[&](tbb::flow::continue_msg _msg)
      {
      int a=rand()%100;
      x=sin(a);
      return x;
      });
   tbb::flow::function_node<tbb::flow::continue_msg,double>
      Node2(g,tbb::flow::unlimited,[&](tbb::flow::continue_msg _msg)
      {
      int b=rand()%100;
      y=cos(b);
      return y;
      });

   tbb::flow::join_node<tbb::flow::tuple<double,double>,
                        tbb::flow::queueing>joinNode1(g);
   tbb::flow::function_node<tbb::flow::tuple<double,double>,double>
      Node3(g,tbb::flow::serial,[&](tbb::flow::tuple<double,double>
      nodeTuple)
      {
      z=sqrt(tbb::flow::get<0>(nodeTuple)*
         tbb::flow::get<0>(nodeTuple))+
         sqrt(tbb::flow::get<1>(nodeTuple)*
         tbb::flow::get<1>(nodeTuple));
      return z;
      });

   tbb::flow::make_edge(Node0,Node1);
   tbb::flow::make_edge(Node0,Node2);
   tbb::flow::make_edge(Node1,get<0>(joinNode1.input_ports()));
   tbb::flow::make_edge(Node2,get<1>(joinNode1.input_ports()));
   tbb::flow::make_edge(joinNode1,Node3);

   Node0.try_put(tbb::flow::continue_msg());
   g.wait_for_all();

   return 0;
}
```

图 9　示例代码转换的 flow graph 并行代码

中 Node0 是虚拟任务 Task0.Node0.try_put(tbb::flow::continue_msg())语句负责启动 flow graph. flow graph 中所有任务对应的物理线程在语句 g.wait_for_all()处同步,即 g.wait_for_all()语句之后的代码必须在 flow graph 中的所有计算都已完成之后才能执行.

5 实验结果与分析

5.1 实验环境

为了验证本文提出的半自动并行化方法的有效性,从 NAS Parallel Benchmarks 3.3、PolyBench 3.2、PARSEC 3.0 和 Intel CnC 等基准测试集中选取了 18 个具有代表性的基准测试程序进行并行代码转换,并对生成的并行代码性能进行了实验分析. 基准测试程序的详细信息如表 3 所示. 除了 PolyBench 测试集中的 10 个测试程序,其他测试集中的程序都分别提供了串行版本代码和官方并行版本代码. PolyBench 测试集只提供了串行版本代码,本文用 PLuTo 为其生成 OpenMP 并行版本代码作为官方并行版本代码. 由于 PLuTo 不提供分块因子选择策略,本文采用其默认的分块大小 32 作为官方并行代码的分块因子.

表 3 测试程序信息

测试程序	测试集	串/并行代码语言	输入数据集和数据大小
BT	NPB 3.3	C/OpenMP	data set B
SP	NPB 3.3	C/OpenMP	data set B
CG	NPB 3.3	C/OpenMP	data set B
Matmul	PolyBench 3.2	C/OpenMP	(2000,2000,2000)
Strm	PolyBench 3.2	C/OpenMP	(2000,2000,2000)
LU	PolyBench 3.2	C/OpenMP	(2000,2000,2000)
Tmm	PolyBench 3.2	C/OpenMP	(2000,2000,2000)
Trisolv	PolyBench 3.2	C/OpenMP	(2000,2000,2000)
Dsyr2k	PolyBench 3.2	C/OpenMP	(2000,2000,2000)
Corcol	PolyBench 3.2	C/OpenMP	(2000,2000,2000)
Covcol	PolyBench 3.2	C/OpenMP	(2000,2000,2000)
Jacobi-2d	PolyBench 3.2	C/OpenMP	(128,2000)
Seidel-2d	PolyBench 3.2	C/OpenMP	(128,2000)
Mandelbrot	Intel CnC	C++/IntelCnC	(2000,2000,10000)
Floyd_warshall	Intel CnC	C++/IntelCnC	2000
FaceDetection	Intel CnC	C++/IntelCnC	20000 images
Blackscholes	PARSEC 3.0	C/OpenMP	65536 options
Fluidanimate	PARSEC 3.0	C++/OpenMP	5 frames, 300000 particles

实验在一台具有 4 个 8 核、2 GHz 主频 Intel Xeon E7-4820 处理器的服务器上进行. 每个核拥有 32 KB 的 L1 private cache 和 256 KB 的 L2 private cache,每个处理器拥有 18 MB 的 L3 shared cache. 服务器总共配有 128 GB 内存,运行 64 位服务器版的 CentOS 7.0 操作系统. 实验中使用了 Clang 3.3 和 GCC 4.8.1 两套编译器,前者用于程序动态分析和并行代码生成,后者用于对串行代码和并行代码进行加速比测试. 其中采用 GCC 的 -O3 -std=c++11 编译选项,生成并行代码使用的是 Intel TBB 4.3. 所有的实验结果是测试程序执行 5 次的平均值.

由于 for 循环和 flow graph 并行代码的生成发生在程序的不同层面上,因此两者可以共存并且可以嵌套. 这意味着 flow graph 的结点中可能包含着一个或多个 parallel_for 循环,即任务级并行中包含循环级并行. 默认情况下同时进行两种并行代码的生成. 为了更好的测量两者各自的性能,本文分别对 parallel_for 和 flow graph 并行代码进行实验和分析.

5.2 parallel_for 并行代码性能测试

本文实现的源到源并行代码转换工具生成的 Intel TBB 并行代码与官方 OpenMP 和 Intel CnC 并行代码进行性能比较,以测试程序的串行版本执行时间作为基准,实验获得的加速比结果如图 10 所示. 图中标注为"Official"的柱状条表示官方提供并行版本程序的加速比;"Semi-Auto"表示本文工具生成的 Intel TBB 并行版本代码的加速比. 横坐标测试程序名称中带 * 号的表示未通过本文实现的分块收益模型直接并行封装的加速比,这些程序均使用了 32 线程进行测试. 而不带 * 号的测试程序表示分块具有收益,采用本文提出 UMC-TSS 方法实施循环优化后的并行代码加速比. 由于分块后的并行代码对最外层循环以分块为单位并行执行,即迭代次数由 N 变为 N/TileSize,根据 UMC-TSS 计算的分块大小使得这些程序最外层循环分块数目只有 8~16,为了避免资源浪费和线程轮空,这些程序均使用 8 线程进行测试.

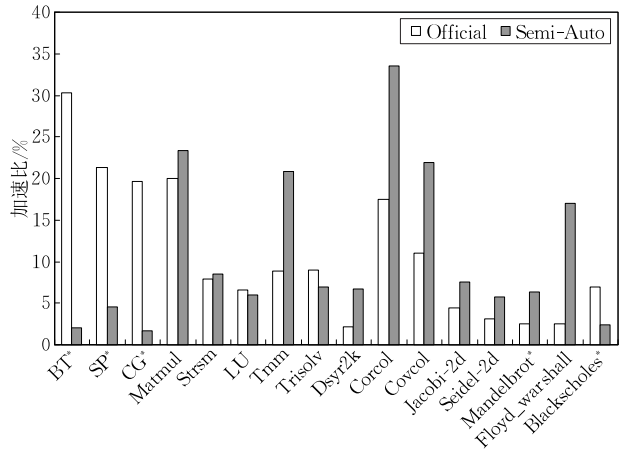


图 10 parallel_for 并行代码与官方并行代码的加速比

本文从 NPB 测试集选取了 3 个测试程序 BT、SP 和 CG，分别获得了 1.98、4.58 和 1.71 的加速比，而官方提供的 OpenMP 专家手动并行版本程序的加速比分别为 30.31、21.33 和 19.63. 这是因为官方并行版本不仅并行了 for 循环，还使用了其他方式对非循环的代码段进行并行化处理，例如用 #progma omp parallel sections 编译制导语句进行任务级并行，而本文仅对 for 循环采用了 Intel TBB 并行执行. 通过对 NPB 测试集的 for 循环分析，这类循环迭代并不适合分块优化.

从 PolyBench 测试集中选择了 10 个测试程序，其中 Corcol 和 Covcol 是数据挖掘中的核心程序，Jacobi-2d 和 Seidel-2d 是 Stencil 计算程序，剩余 6 个是基本线性代数方程子程序. 这类测试程序包含了典型的 DOALL 和 DOACROSS 循环，具有良好的数据重用性，采用分块优化后的并行代码能够有效地降低 cache 失效率，实现良好的性能提升. 这 10 个测试程序的并行代码平均获得 14.11 的加速比，而官方版本并行代码的平均加速比为 9.06. PolyBench 测试程序的 Intel TBB 并行代码性能优于官方 OpenMP 并行代码的原因在于前者采用了本文提出的 UMC-TSS 算法，选择出了最优分块大小来生成循环分块代码，充分利用了两级 cache 中的数据重用，而后者采用 PLuTo 默认的分块大小，没有充分利用 cache 资源，也不能较好地解决 cache 失效问题.

来自 Intel CnC 示例程序的 Mandelbrot 和 Floyd_warshall 分别获得 6.41 和 17.03 的加速比，远高于官方并行版本实现的加速比 2.47 和 2.46. 这是因为官方版本代码忽略了对 Mandelbrot 中一个二维嵌套循环的并行，而该嵌套循环占用了程序执行的主要时间；Floyd_warshall 中具有一个三维嵌套循环，源到源并行代码转换工具对它进行了分块优化和并行处理，更好的开发和利用了程序的局部性和粗粒度并行性，而官方版本忽略了对该嵌套循环的优化. 另一个来自 PARSEC 的测试程序 Blackscholes 获得的加速比仅有 2.39，而官方版本达到了 6.96. 这是因为源到源并行代码转换工具无法对该程序的循环进一步优化，官方并行版本代码则将数据集根据线程数进行划分，优化了线程的负载均衡问题，而根据程序语义对数据集进行划分超出了本文研究范畴.

16 个测试程序的 parallel_for 并行代码总共获得了平均 10.95 倍的加速比. 这得益于本文选择的大部分测试程序中对 for 循环的优化获得了显著的性能提升，parallel_for 并行代码的平均加速比甚至比官方并行版本的平均加速比提高了 0.8%. 而对 NPB、PARSEC 和 Intel CNC 中其他测试程序的 for 循环并行，也能获得和本文从中选择的 6 个测试程序类似的并行性能，但是加速比远没有 PolyBench 这类能够进一步循环优化的测试程序显著.

实验中并行代码的执行全部使用 GCC 编译器，为避免编译器对 OpenMP 和 Intel TBB 运行时库的差异影响实验结果，本文将 10 个 PolyBench 测试程序的官方版本代码修改成 Intel TBB 版本再次进行测试. 官方版本代码仍然采用每层循环分块大小 32，8 线程并行执行. 原始的 OpenMP 版本官方并行代码的平均加速比为 9.06，TBB 版本官方并行代码的平均加速比为 6.25，而本文源到源代码转换工具生成的 TBB 并行代码的平均加速比为 14.11. 实验结果验证了本文方法在 for 循环的分块优化和并行上的性能优势.

5.3　UMC-TSS 算法的性能评估

对 UMC-TSS 算法计算的最优分块大小进行性能评估，与目前最先进的一种静态分析算法 TSS[28] 进行实验比较. 因为循环分块只对具有数据重用的一类循环代码有效，所以只选择表 3 中 PolyBench 的 10 个测试程序进行实验，结果如表 4 所示. 其中，UMC-TSS 算法得到的最优分块大小也是 5.2 节实验中"Semi-Auto"并行代码所采用的分块大小. 性能对比实验表明，本文提出的UMC-TSS算法提供

表 4　UMC-TSS 与 TSS 的性能比较

测试程序	算法	分块大小	运行时间/s	加速比
Matmul	TSS	168×32×104	6.37	3.63
	UMC-TSS	170×32×96	6.27	3.68
Strsm	TSS	168×32×104	7.15	1.30
	UMC-TSS	170×32×96	7.15	1.30
LU	TSS	168×32×104	2.79	1.53
	UMC-TSS	170×32×96	2.70	1.57
Tmm	TSS	160×16×208	1.45	3.43
	UMC-TSS	160×16×224	1.43	3.45
Trisolv	TSS	168×32×104	0.63	1.46
	UMC-TSS	170×32×96	0.60	1.51
Dsyr2k	TSS	120×8×88	6.83	1.04
	UMC-TSS	120×8×136	6.27	1.13
Corcol	TSS	160×16×208	2.27	6.17
	UMC-TSS	128×16×288	2.09	6.71
Covcol	TSS	160×16×208	1.41	3.60
	UMC-TSS	128×16×288	1.43	3.57
Jacobi-2d	TSS	128×16×560	1.87	1.61
	UMC-TSS	16×11×176	1.90	1.59
Seidel-2d	TSS	128×104×16	8.10	1.09
	UMC-TSS	128×200×8	6.65	1.33

的最优分块大小比 TSS 算法平均提高了近 4%的性能.这是因为相比于 TSS 算法,UMC-TSS 算法计算出的最优分块大小能够更准确地模拟分块在 L2 cache 中的映射,而且对 cache 资源和分块数据的利用更加充分.根据文献[28]所述,TSS 算法相比之前的传统方法具有明显的性能优势,这也验证了本文提出的 UMC-TSS 算法具有良好的竞争力.

同时,本文还对 UMC-TSS 算法的可扩展性进行了实验.同样与 TSS 算法进行比较.分别采用 2、4、8 和 16 线程对 Matmul、LU 和 Jacobi-2d 测试程序进行了测试,结果如图 11 所示.UMC-TSS 和 TSS 算法对 Matmul 和 LU 获得了近似的加速比.矩阵乘法 Matmul 具有典型的数据重用特点,当问题规模为 2000 时,分块优化能够有效地降低 cache 失效率,因而表现出了超线性的加速比.下三角矩阵分解 LU 的核心计算 for 循环中包含条件分支,具有更复杂的循环体结构,因此分块因子无法像矩阵乘法那样获得超线性的加速比.而且根据最外层循环的分块大小,最外层循环总共有 12 个分块并行执行,因此当线程数超过 12 时性能提升不再明显,甚至会因为线程轮空和竞争导致性能的下降.这一点在二维雅各比迭代 Jacobi-2d 上体现得更为明显,不过 Jacobi-2d 最外层循环总共有 8 个分块并行,当线程数超过 8 时出现加速比明显降低的趋势.TSS 算法对 Jacobi-2d 最外层的时间维循环不做分块处理,使其并行性能低于本文提出的 UMC-TSS 算法,同时它的加速比随线程数增加没有明显上升.由实验结果可知,本文 UMC-TSS 算法具有良好的可扩展性.

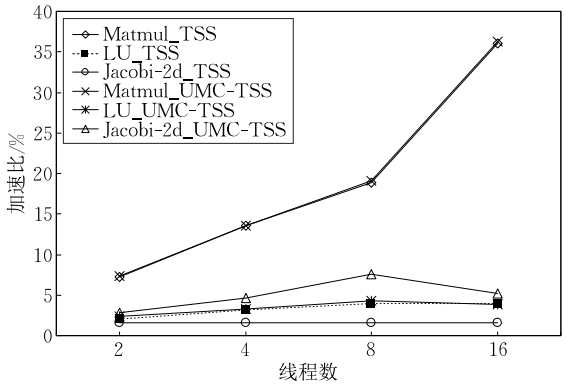


图 11 UMC-TSS 算法的可扩展性实验

5.4 分块收益模型验证

由于循环分块方法主要适用于特定的一部分循环,通常需要通过人为地分析来确定是否实施分块.PLuTo 根据静态编译分析,对于因为依赖关系无法实施分块的循环代码不做处理,串行执行.而对于可分块的循环代码,据作者所知,之前并没有建立针对原始循环代码是否具有分块收益的评估模型.

本节实验对 5.2 节中没有通过分块收益模型的 5 个测试程序手工地进行循环分块,测试其并行加速比,以验证本文提出的分块收益模型的有效性.实验依然采用 UMC-TSS 算法分别为这 5 个测试程序计算最优分块因子大小,保证分块代码的性能.实验结果表明,分块后的并行代码性能相比未分块直接并行代码的性能平均下降了 3.4%.如 3.3 节所述,这是因为这些循环代码不具有较好的数据重用性,分块为这类循环带来了较高的成本开销.实验结果验证了本文提出的基于局部性的分块收益模型的正确性和有效性.

5.5 Flow graph 并行代码性能测试

为了验证本文 flow graph 并行化方法的有效性,本节实验采用源到源代码转换工具为 2 个程序生成了 flow graph 并行代码,并分别测试和分析它们的性能.2 个测试程序是 Intel CnC 示例程序 FaceDetection 和 PARSEC 3.0 测试程序集中的流体力学计算程序 Fluidanimate.根据测试程序的 Task 图进行并行代码的生成,将 Task 图映射为 Intel TBB 的 flow graph.下面将分别分析这 2 个程序的测试结果.

(1)FaceDetection 是一个面部识别程序,广泛应用于计算机视觉领域,其面部识别模块由不同的过滤器 filter 组成.如图 12(a)所示,每个过滤器会识别一个或若干个面部特征,一幅图片含有该过滤器定义的面部特征才能通过,并被传递给下一个过滤器进行处理.只有当一幅图片通过所有的过滤器才被判定是人脸图片.本文的并行化方法将每个过滤器划分为一个任务.当图片流进入面部识别器时,每个过滤器并行执行,同时将本次检测的结果作为一个布尔值返回.这时插入一个 join_node 结点来缓存所有的返回值.为了正确地判定一幅图片中是否包含人脸,需要每个过滤器返回的布尔值,因此 join_node 结点采用 tag_matching 缓冲策略.对 join_node 输出的多元组中所有布尔值进行逻辑与运算来判定图片中是否含有人脸特征.FaceDetection 的 flow graph 逻辑图如图 12(b).

为了测试生成的并行代码的可扩展性,实验分别用不同数量的线程去测试官方提供的 Intel CnC 并行版本代码和本文生成的并行代码的加速比,测试结果如图 13 所示.面部识别器中有 3 个过滤器,它们占整个程序执行时间的 99.9%以上.实验输入

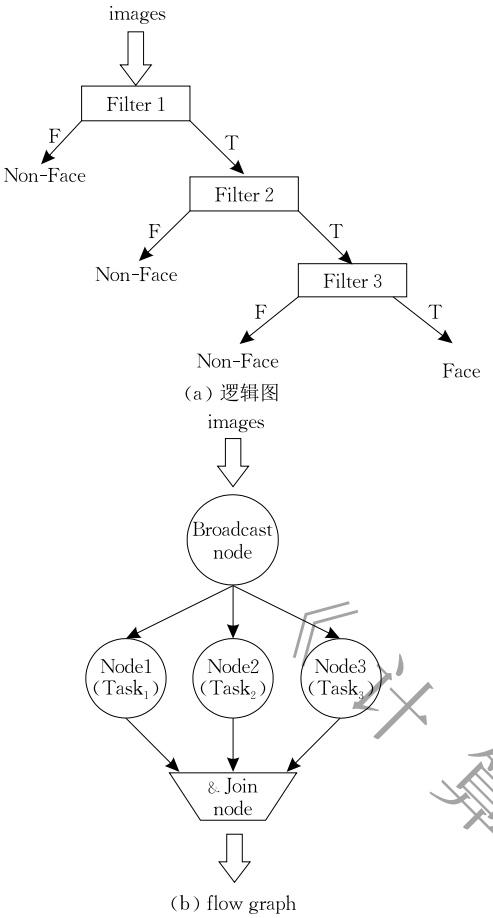


图 12　FaceDetection 的逻辑图和 flow graph

的图片数目为 20 000 幅. 当使用 32 线程时，生成的并行化代码加速比为 9.92. 本例中 flow graph 的所有对象只需创建一次，但是执行了 20 000 次，使得初始化 flow graph 的开销相对很小. 当使用 2 线程和 4 线程时，TBB 并行代码与手动版本性能相当. 当线程数大于 8 时，官方版本的性能要明显优于本文生成的 Intel TBB 并行代码. 这是因为官方提供的 Intel CnC 并行代码对串行代码进行了深度优化和重构，这些优化显然超出了本文方法的能力范围. 实际上，如图 13 所示，由于官方版本代码的深度优化，当仅使用一个线程时，官方版本的加速比就已经达到了 2.00. 当使用 32 线程时，官方版本加速比为 19.36.

需要特别注意，只有当原程序中存在 flow graph 并行模式，即具有大规模的数据并行时，生成的 flow graph 并行代码才能获得良好的加速比. 否则，进行 flow graph 并行模式转换不会取得显著的性能提升. 为了进一步说明这个问题，下面对 PARSEC 基准测试集中的 1 个程序进行测试.

(2) Fluidanimate 是一种用扩展的光滑粒子法

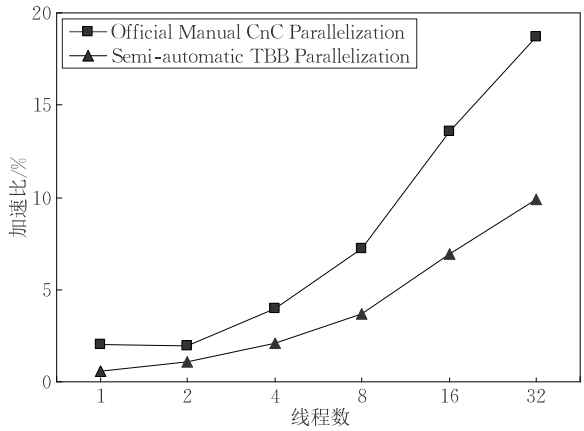


图 13　采用不同线程数时 FaceDetection 并行代码的加速比

(SPH)模拟不可压缩流体间相互受力影响情况的程序. 本文的方法在其中的两个函数 RebuildGrid 和 ProcessCollisions 中检测到 flow graph 的任务级并行度.

函数 RebuildGrid 中进行 Courant-Friedrichs-Lewy(CFL)条件检查的代码可以和其余代码段并行执行. 图 14 展示了该函数被检测出的任务代码和对应的 flow graph，其中 $Task_1$ 进行 CFL 条件检查. 由于 flow graph 定义在一个深度嵌套循环中，每次迭代都要先创建 flow graph 中的所有对象然后再执行，因此带来了巨大的开销. 并行化的 RebuildGrid 函数使用 16 线程时只获得了 1.63 的加速比.

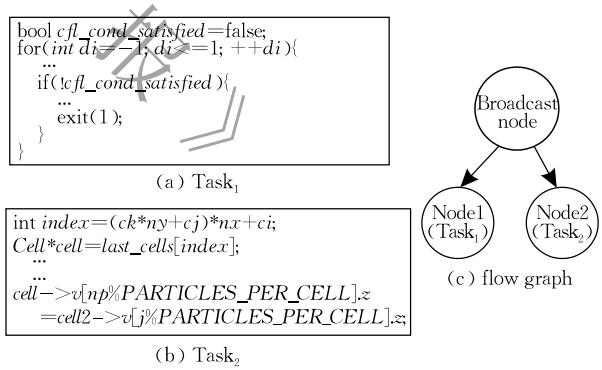


图 14　Fluidanimate 中 RebuildGrid 函数的并行任务

函数 ProcessCollisions 也被检测出一个 flow graph. 它由 6 个嵌套循环组成，用于分别检查流体中粒子是否撞击到 3D 立方网格的 6 个面上. 程序分析结果显示这 6 个嵌套循环之间没有任何数据和控制依赖关系，可以并行执行. 令每个嵌套循环为一个并行任务，图 15 展示了 6 个嵌套循环组成的任务和对应的 flow graph. 和 RebuildGrid 函数一样，ProcessCollisions 函数每次被调用时，flow graph 中所有对象都会被创建和初始化，但是只执行一次.

使用 16 个线程时该函数的并行化代码获得的加速比为 1.81.

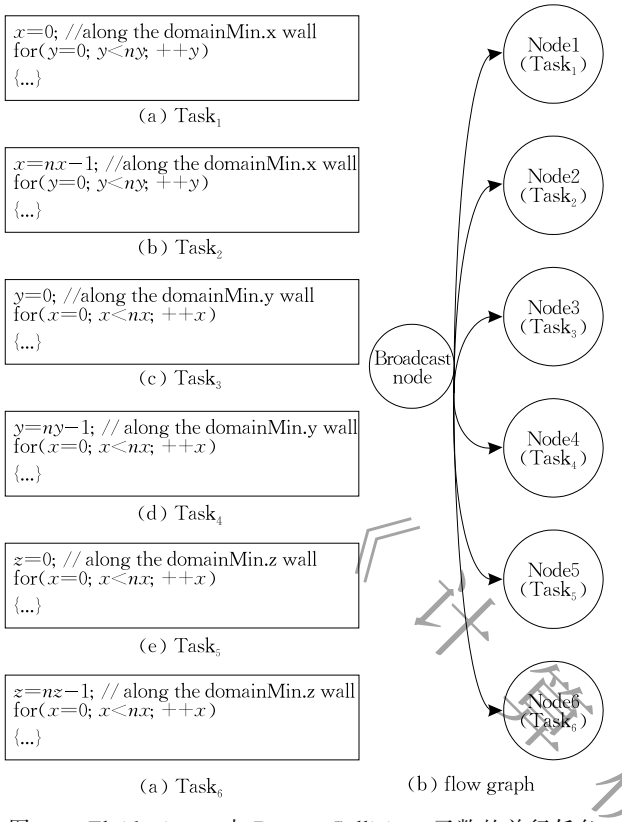


图 15 Fluidanimate 中 ProcessCollisions 函数的并行任务

基于推测多线程的投机并行化方法[7-8]也可以对一般非规则代码进行并行,但是这些方法需要特殊的硬件或者编译器支持.当推测不准确时,因为控制和数据依赖等原因会带来频繁的线程撤销和重启等开销,使得并行效果不佳.最近,Ding 等人[34-35]利用进程来实施粗粒度投机并行.为了保证并行的安全执行,他们采用 lock 机制来保障不确定的依赖关系下的数据一致性,这会导致进程的竞争并带来一定开销.本文的方法不同于投机并行方法,采用动态程序分析预先提取程序的控制和依赖信息,在确定的依赖关系下发掘程序的潜在粗粒度并行性.本文的方法相比于投机并行的方法一般具有更好的效率,但也带来了更大的开销.下一小节将对本文半自动并行化方法的开销进行分析.

5.6 半自动并行化方法的开销分析

本文提出的半自动并行化方法和实现的源到源并行代码转换工具的主要开销来自两方面:程序的动态分析和 for 循环代码分块收益分析.本文基于 DiscoPoP[13-15]实现程序的动态分析,以获得运行时的控制流和数据依赖关系.根据文献[15],DiscoPoP 在分析时间上稍慢于著名的 SD3[23],因为 SD3 只对热点循环代码进行分析,而 DiscoPoP 则是面向整体程序的分析.相比于程序分析器 Alchemist[47]和 Parwiz[48],DiscoPoP 具有明显的优势.在程序分析文件所占用的空间开销方面,DiscoPoP 和 SD3 旗鼓相当,但是优于其他分析器,如 Parwiz 等.另外,本文提出的分块收益模型需要记录 for 循环迭代过程中的访存地址序列,并对其计算重用次数和重用距离.由于循环的访存地址数量往往达到千万级,本文根据循环迭代的周期特性采用采样的方式进行统计.因此,当选择合适的采样率时,相比完整的地址序列计算,本文方法能够在保证准确率的情况下极大的缩减计算时间.由于半自动和自动并行化方法对大量的遗留软件的并行化处理具有重要的实用价值,这些开销相对于手动并行化方法仍然是可以接受的.

通过实验和分析可知,本文的半自动并行化方法适用于具有数据重用的循环代码和大规模数据并行的一般程序代码段.对这些类型的应用程序,本文方法能够获得较理想的并行性能.

6 结束语

针对目前自动化并行技术存在保守的依赖分析、并行粒度过细、代码生成简单等限制,本文提出一种全新的半自动并行化方法,该方法集成了高效的动态程序分析、粗粒度并行任务提取、规则 for 循环的自动优化和高层次代码生成技术,用户只需少量的手动操作,就可以为一般串行应用程序生成高效的 Intel TBB 并行代码.本文采用动态程序分析工具 DiscoPoP 进行数据依赖分析,根据程序分析结果生成的 CU 图来提取程序中非规则代码段的粗粒度并行任务.对热点的 for 循环代码,本文提出一种基于局部性的分块收益模型,通过对访存地址序列的重用次数和重用距离进行计算分析,选择出具有循环分块收益的 for 循环代码进行进一步的优化.根据当前最先进的一种最优分块因子大小选择算法存在的不足,本文提出一种基于 cache 均匀映射的最优分块大小选择算法 UMC-TSS,能有效地利用 cache 资源和分块数据的重用性,同时实现了分块的粗粒度并行.最终设计并实现了一个半自动化源到源并行代码转换工具,该工具可以有效地为 C/C++ 程序生成 Intel TBB 并行代码.从 NPB、PolyBench、PARSEC 和 Intel CnC 等基准测试集中选择了 18 个典型的测试程序,利用本文实现的源到源代码转换

工具为这些测试程序生成并行化代码，并对其进行了一系列的性能实验.实验结果表明本文提出的半自动并行化方法生成的粗粒度并行代码在循环级和任务级上分别获得了平均 10.95 和 4.45 倍的加速比.此外，优化后的循环分块代码采用本文提出的 UMC-TSS 算法比采用当前最先进的一种 TSS 算法可以提高平均 4%的性能，同时具有更好的可扩展性.本文的工作对串行程序和遗留软件的自动并行化转换提供了一种有效的解决方法，具有良好的实用价值，对自动并行化方法的研究和发展具有积极的意义.

下一步的工作将考虑从 Task 图中提取更多的并行模式映射到并行编程模型中.尽管 flow graph 具有很强的通用性，为了提高并行代码效率，并行代码转换工具需要支持更多的并行模板转换，如 pipeline.而根据程序语义分析，自动地实施相应的优化策略，如数据划分、线程负载均衡等，并将其集成到本文实现的工具中，也是以后工作将要深入研究的一个方向.此外，未来工作还将考虑自动地对生成的并行代码的正确性进行检查，并提供反馈和更正的机制.

参　考　文　献

[1] Burke M, Cytron R. Interprocedural dependence analysis and parallelization//Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction. New York, USA, 1986: 162-175

[2] Lim A W, Lam M S. Maximizing parallelism and minimizing synchronization with affine partitions. Parallel Computing, 1998, 24(3-4): 445-475

[3] Lam M S, Wolf M E. A data locality optimizing algorithm//Proceedings of the 12th ACM SIGPLAN Conference on Programming Language Design and Implementation. Toronto, Canada, 1991: 30-44

[4] Pouchet L N, Bondhugula U, Bastoul C, et al. Loop transformations: Convexity, pruning and optimization//Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Austin, USA, 2011: 549-562

[5] Thies W, Chandrasekhar V, Amarasinghe S. A practical approach to exploiting coarse-grained pipeline parallelism in C programs//Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. Chicago, USA, 2007: 356-369

[6] Rul S, Vandierendonck H, Bosschere K D. A profile-based tool for finding pipeline parallelism in sequential programs. Parallel Computing, 2010, 36(9): 531-551

[7] Sohi G S, Breach S E, Vijaykumar T N. Multiscalar processors//Proceedings of the International Symposium on Computer Architectures. Margherita Ligure, Italy, 1995: 414-425

[8] Krishnan V, Torrellas J. A chip-multiprocessor architecture with speculative multithreading. IEEE Transactions on Computers, 1999, 48(9): 866-880

[9] Blumofe R D, Joerg C F, Kuszmaul B C, et al. Cilk: An efficient multithreaded runtime system. SIGPLAN Not, 1995, 30(8): 207-216

[10] Leiserson C E. The Cilk++ concurrency platform. The Journal of Supercomputing, 2010, 51(3): 244-257

[11] Budimlić Z, Burke M, Cavé V, et al. Concurrent collections. Scientific Programming, 2010, 18(3): 203-217

[12] Reinders J. Intel threading building blocks: Outfitting C++ for multi-core processor parallelism. Sebastopol, USA: O'Reilly Media, 2007

[13] Li Z, Jannesari A, Wolf F. Discovery of potential parallelism in sequential programs//Proceedings of the 42nd International Conference on Parallel Processing Workshops. Lyon, France, 2013: 1004-1013

[14] Li Z, Atre R, Ul-Huda Z, et al. DiscoPoP: A profiling tool to identify parallelization opportunities//Proceedings of the 8th International Workshop on Parallel Tools for High Performance Computing. Stuttgart, Germany, 2014: 37-54

[15] Li Z, Jannesari A, Wolf F, et al. An efficient data-dependence profiler for sequential and parallel programs//Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium. Hyderabad, India, 2015: 484-493

[16] Bailey D H, Barszcz E, Barton J T, et al. The NAS parallel benchmarks. International Journal of Supercomputer Applications, 1991, 5(3): 63-73

[17] Bienia C, Kumar S, Singh J P, et al. The PARSEC benchmark suite: Characterization and architectural implications//Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. Toronto, Canada, 2008: 72-81

[18] Hall M W, Anderson J M, Amarasinghe S P, et al. Maximizing multiprocessor performance with the SUIF compiler. Computer, 1996, 29(12): 84-89

[19] Padua D A, Eigenmann R, Hoeflinger J, et al. Polaris: A new-generation parallelizing compiler for mpps. Urbana-Champaign, Illinois, USA: University of Illinois at Urbana-Champaign, Technical Report: No. 1306, 1993

[20] Bondhugula U, Ramanujam J, Sadayappan P. A practical and fully automatic polyhedral program optimization system. ACM SIGPLAN Notices: PLDI, 2008, 43(6): 101-113

[21] Hartono A, Baskaran M M, Bastoul C, et al. PrimeTile: A parametric multi-level tiler for imperfect loop nests//Proceedings of the 23rd International Conference on Supercomputing. Yorktown Heights, USA, 2009: 147-157

[22] Irigoin F, Triolet R. Supernode partinioning//Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Diego, USA, 1988: 319-328

[23] Kim M, Kim H, Luk C. SD3: A scalable approach to dynamic data-dependence profiling//Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. Atlanta, USA, 2010: 535-546

[24] Garcia S, Jeon D, Louie C M, et al. Kremlin: Rethinking and rebooting gprof for the multicore age//Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. San Jose, USA, 2011: 458-469

[25] Lam M D, Rothberg E E, Wolf M E. The cache performance and optimizations of blocked algorithms//Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems. Santa Clara, USA, 1991: 63-74

[26] Ghosh S, Martonosi M, Malik S. Cache miss equations: An analytical representation of cache misses//Proceedings of the 11th International Conference on Supercomputing. Vienna, Austria, 1997: 317-324

[27] Coleman S, McKinley K S. Tile size selection using cache organization and data layout. ACM SIGPLAN Notices: PLDI, 1995, 30(6): 279-290

[28] Mehta S, Beeraka G, Yew P. Tile size selection revisited. ACM Transactions on Architecture and Code Optimization, 2013, 10(4): 35: 1-35: 27

[29] Zhao M, Childers B R, Soffa M L. A model-based framework: An approach for profit-driven optimization//Proceedings of IEEE/ACM International Conference on Code Generation and Optimization. San Jose, USA, 2005: 317-327

[30] Sarkar V, Megiddo N. An analytical model for loop tiling and its solution//Proceedings of International Symposium on Performance Analysis of Systems and Software. Austin, USA, 2000: 146-153

[31] Whitfield D, Soffa M L. An approach to ordering optimizing transformations. ACM SIGPLAN Notices, 1990, 25(25): 137-146

[32] Pei Song-Wen, Wu Bai-Feng. SpMT WaveCache: Exploiting speculative multithreading for dataflow computer. Chinese Journal of Computers, 2009, 32(7): 1382-1392(in Chinese)
(裴颂文, 吴百锋. SpMT WaveCache: 开发数据流计算机中的推测多线程. 计算机学报, 2009, 32(7): 1382-1392)

[33] Li Yuan-Cheng, Yin Pei-Pei, Zhao Yin-Liang. A FCM-based thread partition algorithm for speculative multithreading. Chinese Journal of Computers, 2014, 37(3): 580-592(in Chinese)
(李远成, 阴培培, 赵银亮. 基于模糊聚类的推测多线程划分算法. 计算机学报, 2014, 37(3): 580-592)

[34] Ding C, Shen X, Kelsey K, et al. Software behavior oriented parallelization. ACM SIGPLAN Notices: PLDI, 2007, 42(6): 223-234

[35] Ding C, Fletcher Z. Parallel programming by hints//Proceedings of the Workshop on ACM Systems, Programming, Languages and Applications: Software for Humanity. Portland, USA, 2011: 101-104

[36] Ke C, Liu L, Zhang C, et al. Safe parallel programming using dynamic dependence hints. ACM SIGPLAN Notices: OOPSLA, 2011, 46(10): 243-258

[37] Tournavitis G, Wang Z, Franke B, et al. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping//Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. Dublin, Ireland, 2009: 177-187

[38] Johnson R, Pearson D, Pingali K. The program structure tree: computing control regions in linear time//Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation. Orlando, USA, 1994: 171-185

[39] Li Z, Zhao B, Jannesari A, et al. Beyond data parallelism: Identifying parallel tasks in sequential programs. Lecture Notes in Computer Science, 2015, 9531: 569-582

[40] Tarjan R. Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1972, 1(2): 146-160

[41] Renganarayana L, Kim D, Rajopadhye S, et al. Parameterized loop tiling. ACM Transactions on Programming Languages and Systems, 2012, 34(1): 1-14

[42] Cytron R. Doacross: Beyond vectorization for multiprocessors//Proceedings of the International Conference on Parallel Processing. St. Charles, IL, USA, 1986: 836-844

[43] Wonnacott D G, Strout M M. On the scalability of loop tiling techniques//Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques. Berlin, Germany, 2013: 3-11

[44] Bennett B T, Kruskal V J. LRU stack processing. IBM Journal of Research and Development, 1975, 19(4): 353-357

[45] Liu Song, Wu Wei-Guo, Zhao Bo, Jiang Qing. Loop tiling for optimization of locality and parallelism. Journal of Computer Research and Development, 2015, 52(5): 1160-1176(in Chinese)
(刘松, 伍卫国, 赵博, 蒋庆. 面向局部性和并行性优化的循环分块技术. 计算机研究与发展, 2015, 52(5): 1160-1176)

[46] Zhao B, Li Z, Jannesari A, et al. Dependence-based code transformation for coarse-grained parallelism//Proceedings of the 2015 International Workshop on Code Optimization for Multi and Many Cores. San Francisco, USA, 2015: 1-10

[47] Zhang X, Navabi A, Jagannathan S. Alchemist: A transparent dependence distance profiling infrastructure//Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization. Seattle, USA, 2009: 47-58

[48] Ketterlin A, Clauss P. Profiling data-dependence to assist parallelization: Framework, scope, and optimization//Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture. Vancouver, Canada, 2012: 437-448

LIU Song, born in 1987, Ph. D. candidate. His research interests include code optimization and high performance computing.

ZHAO Bo, born in 1990, M. S. His research interests include multicore systems and source to source code transformation.

JIANG Qing, born in 1991, M. S. His research interests include compiler optimization and machine learning.

WU Wei-Guo, born in 1963, Ph. D., professor, Ph. D. supervisor. His research interests include high performance computer architecture, cloud computing and embedded system.

Background

Multi-core architecture has been popular as a result of the stagnating single core performance. However, this strong computing power cannot be fully exploited unless the programs have been well parallelized. Unfortunately, many existing software products are mostly written sequentially. Manually rewriting these legacy programs is time consuming and is a big economic challenge. Effective methodologies and tools to parallelize existing software with minimum manual programming effort and user intervention are in great demand for both academic community and industry. A plenty of works corresponding to automatic parallelization has been widely studied to mainly exploit fine-grained parallelism from loops, which may not take full advantage of multi-core architecture. The static program analysis used to identify data dependences is conservative that misses many potential parallelized opportunities. This paper presents a semi-automatic parallelization approach to transform coarse-grained tasks for irregular code sections in general program and optimize the regular for-loops. It employs dynamic program profiling tool to identify data dependences and forms the Computational Unit graph for finding parallel task. The for-loops are analyzed according to the statistic of reuse distances and the profitable loops are further tiled and parallelized for optimization. Additionally, a new tile size selection algorithm is proposed to generate tiled loop codes. Eventually, a source-to-source transformation tool based on the LLVM frontend Clang is developed to generate high-level Intel TBB parallel codes for C/C++ codes with only little human effort. The transformed parallel codes, using TBB parallel_for and flow graph parallel templates, gain good speedups. The work of this paper is mainly supported by the National Natural Science Foundation of China under Grant Nos. 91630206 and 91330117, the National Key Research and Development Plan under Grant No. 2016YFB0201800 and the Social Development and Science and Technology Research Project of Shaanxi Province under Grant No. 2016SF-428.