

基于区域平均执行时间和数据依赖信息的 可能并行区域识别

张 超^{1),2)} 王 蕾^{1),2)} 向晓娅^{1),2)} 冯晓兵¹⁾

¹⁾(中国科学院计算技术研究所计算机系统结构重点实验室 北京 100190)

²⁾(中国科学院研究生院 北京 100039)

摘 要 随着多核处理器逐渐成为处理器发展的新趋势,为了持续提高程序性能,必须并行执行应用程序.传统的自动并行技术能够很好地并行科学计算应用中的规则循环,但对于含有大量函数调用和指针引用的不规则程序,目前还不能有效地对其实施并行.针对这一现状,文中提出了基于区域平均执行时间和数据依赖信息的可能并行区域识别方法来对一些不规则程序实施高效并行,主要贡献如下:(1)自动识别程序中的多种并行性,不仅包括传统并行性分析中的循环迭代间的细粒度并行性,而且也包括传统并行性分析尚不能有效处理的循环体和函数调用点间的粗粒度并行性.对于程序中蕴含的众多并行性,文中基于区域平均执行时间实施收益分析来选择合适的并行区域实施并行;(2)自动识别可能并行区域间数据依赖关系的数量、类型以及导致数据依赖关系的程序变量.基于文中的分析结果,作者使用面向行为的投机并行系统(behavior oriented parallelism)对 SPEC2006 中的 4 个测试用例实现了并行化.并行化后的程序在 Intel 和 AMD 多核处理器上分别得到了 300%和 260%的平均性能加速.

关键词 可能并行区域;区域平均执行时间;数据依赖信息;投机并行

中图法分类号 TP311

Identifying Possibly Parallel regions Using Average Execution Time of Regions and Data Dependence Profiling

ZHANG Chao^{1),2)} WANG Lei^{1),2)} XIANG Xiao-Ya^{1),2)} FENG Xiao-Bing¹⁾

¹⁾(Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

²⁾(Graduate School of Chinese Academy of Sciences, Beijing 100039)

Abstract The current trend in processor architecture is a move toward multi-core processors. Parallel execution will be required to improve program performance continuously. Traditional automatic program parallelization typically works for regular loops in codes of scientific applications, but in general can not find enough parallelism from irregular programs, especially those that have many pointer references and function calls. This paper presents a method to identifying possibly parallel regions using average execution time of regions and data dependence profiling. The main contributions are as follows: (1) automatically identify possibly parallel regions (PPR) at various levels of granularity. The parallel regions are not only traditional fine-grained parallel regions (inter loop iterations), but also coarse-grained parallel regions (inter loop bodies and function call sites). It selects a set of potentially beneficial regions from all regions of a program using average execution time of regions; (2) automatically identify number and types of inter-region dependences, and find out program variables that cause these inter-region dependences. In

this paper, the authors use Behavior Oriented Parallelism (BOP) to verify the correctness of program transformation. According to the analyses results, the authors parallelize four SPEC2006 test cases. And the parallelized programs show 300% and 260% speedup on Intel and AMD multi-core machines respectively.

Keywords possibly parallel regions; average execution time of regions; data dependence profiling; speculative parallelism

1 引 言

在多核处理器中,一方面,为了充分利用处理器资源,我们必须并行执行多个应用程序;另一方面,由于多核处理器中每个核的处理能力不会高于传统的单核处理器,所以单个程序也必须并行执行才能持续不断地提高程序性能.然而,大多数应用程序都是基于串行模式编写的,不能在多核处理器上并行执行,因此我们需要对程序实施并行化.

理想的并行实施方法是自动并行,它基于静态编译器技术或者动态运行时技术发掘程序的并行性.前者使用编译器中的静态数据依赖分析技术分析程序源代码并抽取可并行循环体^[1],这种方法在分析科学计算应用中的规则循环上非常成功.但是,对于含有大量函数调用和指针引用的不规则程序,这种方法的有效性受到了一定的制约^[2].基于动态运行时技术的自动并行是目前研究的热点,以事务内存^[3-4]和线程级投机并行^[5]为主要代表.基于特殊硬件或者系统软件支持,线程级投机并行能够从不规则程序中发掘出一些并行性.但是对于 SPEC2006 中的测试用例,在只关注循环迭代间的细粒度并行性的前提下,由线程级投机并行独立得到的性能提升所占比例非常小^[6].针对这一现状,我们引入循环体和函数调用点间的粗粒度并行性来扩充原有的并行性分析,并使用区域平均执行时间以及数据依赖关系等动态运行时信息来对程序实施并行分析.分析中,我们必须解决以下两个问题:(1)识别程序中并行执行能够带来收益的区域;(2)识别程序中阻碍程序并行的因素,对程序实施并行化.

本文的解决方法是:(1)使用区域平均执行时间从程序中识别出可能带来并行收益的区域,并把这些区域组织成可能并行区域.可能并行区域中不仅包括传统并行性分析中的循环迭代间的细粒度并行性,也包括循环体和函数调用点间的粗粒度并行

性.可能并行区域构造方案如表 1 所示;(2)利用基于插桩的动态数据依赖分析工具来识别可能并行区域中的各个区域之间的数据依赖关系的数量、类型以及导致数据依赖关系的变量;(3)使用上述信息指导程序变换并实施并行化,如:以导致数据依赖关系的变量为切片标准实施程序切片等.在最后一步中,我们使用面向行为的投机并行系统^[7]来保证程序变换的正确性.

表 1 可能并行区域构造方案

| | 可能并行区域 | 描述 |
|---|-----------|-----------------------------|
| 1 | 循环迭代 | 循环中各个迭代构成可能并行区域 |
| 2 | 循环体和函数调用点 | 同一个嵌套层次的多个循环体和函数调用点构成可能并行区域 |

本文的主要贡献如下:

(1)突破传统并行性分析中的循环迭代间的细粒度并行性,从实际应用程序中发掘出了循环体和函数调用点间的粗粒度并行性.

(2)以区域平均执行时间为标准进行并行收益分析,避免了过于激进的投机并行导致的程序性能下降.

(3)提出并实现了基于动态二进制插桩工具^[8]的高效的程序数据依赖分析工具,能够收集区域间数据依赖的详细信息,包括数据依赖关系的数量、类型以及导致数据依赖关系的程序变量等.这些信息用来分析投机失败的可能性,避免过多的投机失败导致程序性能下降.

(4)基于分析结果,利用面向行为的投机并行系统来保证程序变换的正确性.本文的方法对 SPEC2006 中的 4 个测试用例有 300%和 260%的平均性能加速.

本文第 2 节介绍基于区域平均执行时间的并行收益分析;第 3 节介绍如何识别程序中阻碍并行的变量以及如何提高识别过程的精度和效率;第 4,5 节分别是实验分析和相关工作.最后给出结论和进一步工作的展望.

2 基于区域平均执行时间的并行收益分析

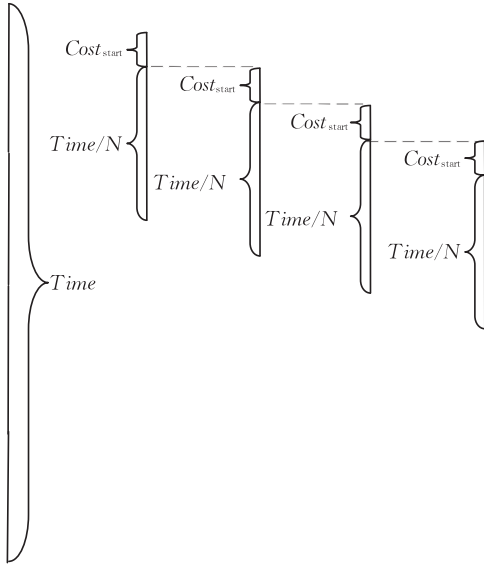
2.1 分析标准

基于区域平均执行时间的并行收益分析的主要目的是:在不考虑投机失败开销的前提下,从程序中识别出并行执行能够带来收益的区域.对于投机失败的情况,我们将在第3节详细分析.当投机并行成功时,投机并行的开销主要包括任务启动的开销、冲突检查和结果提交的开销.任务启动的开销主要取决于操作系统的实现和机器性能.对同一个系统来说,不同任务的启动开销差别不大.冲突检查和结果提交的开销主要取决于硬件实现方式和任务读写变

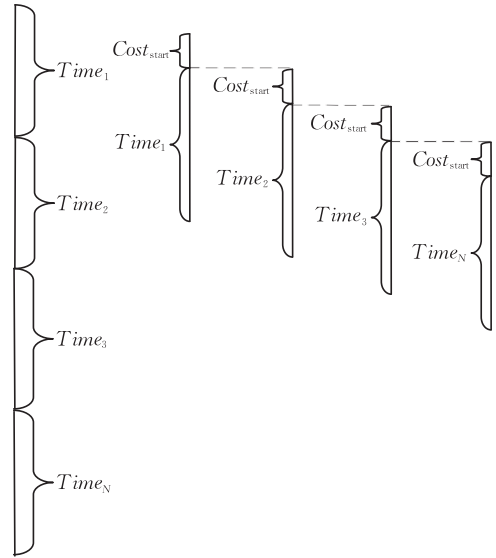
量的数量.对同一个系统来说,由于不同任务读写变量的数量之间存在非常大的差别,因此,不同任务间的冲突检查和结果提交的开销也会有显著不同.在本文中,我们假设投机并行的开销只包括任务启动的开销,这是真实投机并行开销的一个下界.这样,经过该分析得到的可能并行区域集合将是能够带来性能收益的并行区域的一个上界.

假设循环体或者函数调用点的执行时间为 $Time$,任务启动的开销为 $Cost_{start}$,系统中处理器的数量为 N .图1(a)给出了可能并行区域构造方案1对应的并行执行状态.从图中可以看出,循环体的执行时间必须满足如下公式,并行执行才能带来性能收益.

$$Time > N^2 \times Cost_{start} / (N - 1) \quad (1)$$



(a) 可能并行区域构造方案 1 的执行状态



(b) 可能并行区域构造方案 2 的执行状态

图 1

图1(b)给出了可能并行区域构造方案2对应的并行执行状态,其中 $Time_x$ 表示第 x 个区域的执行时间.如果相同嵌套层次中的循环体和函数调用点的总数量大于 N ,则把所有区域按照程序顺序组织成并行度为 N 的多个并行组;如果相同嵌套层次中的区域总数量小于 N ,则以执行时间为 0 的虚拟区域补齐.在图1(b)中,假设 $Time_m \geq Time_{m-1}$,那么并行后的最长执行时间为 $N \times Cost_{start} + Time_N$.要并行执行能够带来性能收益,则必须满足 $Time_1 + Time_2 + \dots + Time_N > N \times Cost_{start} + Time_N$,从而 $Time_N > N \times Cost_{start} / (n - 1)$.在本文中,我们保守地假设所有区域的执行时间都必须满足以下公式:

$$Time > N \times Cost_{start} / (N - 1) \quad (2)$$

理想状况下,上述循环体或者函数调用点的执行时

间 $Time$ 是每次进入该区域的执行时间.由于准确记录该值的代价非常高,所以本文以区域平均执行时间来代替.区域平均执行时间等于程序执行过程中区域的总执行时间除以进入区域的次数.这样,并行收益分析就转化为以下两个问题:(1)区域总执行时间的衡量;(2)区域总执行次数的记录.通常情况下,我们认为程序的热点区域即总执行时间长的区域都适合并行执行,但事实并非如此.在对 SPEC2006 milc 中的最热的 7 个区域实施并行后,程序性能下降了 139 倍.进一步分析发现,并行后程序性能下降的原因是:虽然各个区域的总执行时间很长,但是区域平均执行时间太短,因此并行收益较小;同时每次进行区域都会有并行的开销,这样并行的开销大于并行的收益,从而导致并行后的程序性

能急剧下降. 下面,我们将要具体介绍如何得到区域平均执行时间信息.

2.2 具体实现

本文使用程序插桩的方法收集区域总执行时间和执行次数的信息,最后对这两者求商得到区域平均执行时间. 具体的,我们在编译程序时对循环体和函数调用点进行插桩,然后执行可执行程序. 算法如下:

fb_file 文件保存反馈信息,初始时为 空.

```
//fb_file为保存profile信息的文件, 第一次迭代编译时空
void compile_test_case() {
    for (each循环和函数调用点loop_fun) //暂时不考虑通过函数指针调用的情况
        if (!(loop_fun in fb_file))
            在loop_fun的入口和出口处插入函数调用do_instrument;
}
void do_instrument (loop_fun) {
    enter_times++; //进入loop_fun的次数加1
    if (loop_fun的入口)
        start_time=clock(); //start_time为本次进入loop_fun的时刻
    else if (loop_fun的出口) {
        total_time += clock() - start_time; //total_time为loop_fun的总执行时间
        if (enter_times > 1000000 && enter_times > total_time * 10000)
            {把loop_fun写入fb_file文件的末尾; exit(0); }
    }
}
```

图 2 区域平均执行时间分析中的编译插桩算法

在第一次迭代过程中,fb_file 文件为 空,所以程序中的所有循环体和函数调用点都被插桩. 每次迭代结束时,识别出的调用次数多但平均执行时间不满足式(1)或式(2)的循环体和函数调用点的信息被添加到 fb_file 文件末尾,再次编译的时候该区域就不会被插桩,因此被插桩区域的数量不断减少. 当 fb_file 文件不再增加的时候,整个迭代过程就结束了. 此时,我们发现只有少量循环体和函数调用点被插桩,且插桩库函数的总执行时间被控制在较小的范围内,这样插桩库函数的执行对分析精度的影响就可以忽略不计. 根据实验平台信息,区域平均执行时间的下限为 1/1000s(式(2)). 如果区域是循环体,我们会进一步判断其是否满足式(1)). 另外,为了降低异常情况对分析精度的影响,只有进入区域的次数 (enter_times) 达到一定数量 (此处我们使用 1000000) 的时候才会判断区域的平均执行时间是否大于该下限.

迭代结束后,该算法返回符合式(1)或式(2)的区域的入口和出口对应的源文件名字及行号信息. 这些信息在插入 BOP 系统(面向行为的投机并行系统)的接口函数时使用.

1. 读文件对源程序实施编译插桩. 图 2 给出了插桩算法,包括插桩动作和插桩库函数,其中插桩动作 compile_test_case,选择不在 fb_file 中的循环体和函数调用点实施插桩,而插桩库函数 do_instrument 则负责记录区域的总执行时间 total_time 和执行次数 enter_times.

2. 运行可执行程序,将运行中调用次数多但平均执行时间不满足式(1)或者式(2)的循环体和函数调用点的信息添加到 fb_file 文件末尾. 如果有新的信息写入 fb_file 文件,则跳转到步 1,否则结束.

3 区域间数据依赖分析

3.1 区域间的数据依赖关系的定义

区域间数据依赖分析用于识别程序执行过程中各个可能并行区域之间的数据依赖关系,以此来分析投机并行失败的情况. 与传统的以指令为单位的循环迭代间的数据依赖关系不同,本文定义区域间的数据依赖关系如下:

对于任意被两个区域都访问的变量 A,首先识别出各个区域中对 A 的首次读操作和首次写操作,

(1) 如果一个区域中存在对 A 的写操作,而另一个区域对 A 的首次操作为读操作,那么变量 A 导致这两个区域之间产生一个真实(real)数据依赖关系.

(2) 如果在两个区域中对 A 的首次操作均为写操作,那么变量 A 导致这两个区域之间产生一个输出(output)数据依赖关系.

对于基于进程的并行系统,各个进程有独立的虚地址空间,因此只需保证共享变量之间的真实数据依赖关系. 对于基于线程的并行系统,各个线程共享虚地址空间,则必须同时保证共享变量之间真实和输出数据依赖关系.

3.2 基于插桩的区域间数据依赖分析

我们基于动态二进制插桩工具 PIN^[8] 构造可能并行区域间的数据依赖关系的分析工具. 在程序执行过程中, 我们对可能并行区域中的所有访存指令进行插桩并收集其读写信息, 用以分析区域间数据依赖关系.

根据 3.1 节中区域间数据依赖关系的定义, 我们只需要记录各个区域中对某个变量的首次读和首次写操作信息, 忽略对该变量的后续访存操作. 我们记录访存操作访问的虚地址和访存操作的类型, 以此来识别区域间数据依赖关系的数量和类型; 记录访存操作指令对应的源代码信息(文件名字和行号), 用于识别导致数据依赖关系的变量, 上述信息组成一个 4 元组 $\langle virtual_add, type, file_name, line_num \rangle$. 本文使用上述 4 元组的一个序列来表示区域的访存信息. 在程序执行过程中, 为区域中的每条访存指令建立相应的 4 元组, 并检查现有 4 元组序列中是否存在与新建组的前两个域相同的组, 如果已经存在则忽略新建组, 否则把新建组添加到 4 元组序列. 这样, 我们就得到了程序执行过程中可能并行区域的访存信息. 然后, 根据区域间数据依赖关系的定义, 我们得到两个区域间数据依赖关系的详细信息, 包括数据依赖关系的数量、类型以及导致数据依赖关系的程序变量等, 这些信息用来指导程序并行化.

3.3 提高区域间数据依赖分析的精度

为了提高区域间数据依赖分析的精度, 本文对程序执行过程中访问的不同类型的数据进行详细分析. 程序执行过程中访问的数据分为 3 种: 全局数据、栈数据和堆数据.

(1) 如果访问的是全局数据, 2 元组 $\langle virtual_add, type \rangle$ 足以区别各个访存操作.

(2) 如果访问的是栈数据, 2 元组 $\langle virtual_add, type \rangle$ 不能完全区别各个访存操作. 如图 3 所示, 如果函数调用点 Q 和 R 都满足式(2), 那么在区域数据依赖分析中就需要分析 Q 和 R 的数据依赖关系. 对于函数调用点 Q , 我们得到的 2 元组序列为 $\langle a, W \rangle, \langle a, R \rangle, \langle b, W \rangle$. 对于函数调用点 R , 我们得到的 2 元组序列为 $\langle a, R \rangle, \langle c, W \rangle$. 这样, Q 中的 $\langle a, W \rangle$ 和 R 中的 $\langle a, R \rangle$ 导致两个区域之间产生一个真实数据依赖. 假设变量 b 的虚地址和变量 c 的虚地址是相同的, 那么 Q 中的 $\langle b, W \rangle$ 和 R 中的 $\langle c, W \rangle$ 导致两个区域之间产生一个输出数据依赖. 但事实上, 当函数调用点 Q 和 R 并行执行的时候, 它们都

有自己的私有栈空间, 变量 b 和 c 不会对 Q 和 R 的并行产生任何影响. 为了解决这个问题, 对于图 3 中的情况, 本文只记录虚地址不大于函数 P 的栈顶且不小于栈底的栈数据的访问情况, 即图 4(a) 所示的阴影区域.

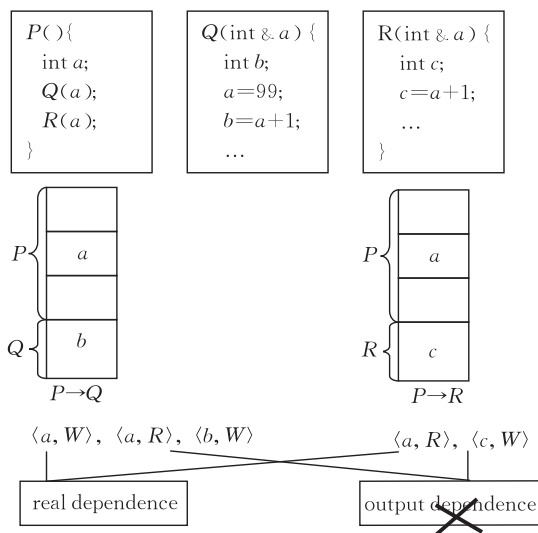
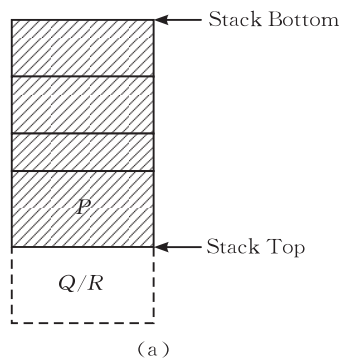
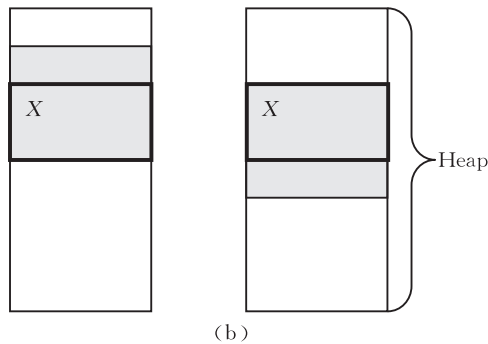


图 3 访问栈数据导致的假依赖关系



(a)



(b)

图 4 区域间数据依赖分析

(3) 如果访问的是堆数据, 2 元组 $\langle virtual_add, type \rangle$ 也不能完全区别各个访存操作. 如图 4(b) 所示, 在程序执行过程中, 首先分配了一块堆数据, 然后释放, 随后又分配了一块堆数据, 两次分配之间有 X 部分是重叠的. 如果仅使用 $\langle virtual_add, type \rangle$ 来区分

访存操作,那么 X 部分将导致很多依赖关系. 为了解决这个问题,我们使用 4 元组 $\langle start_add, size, id, valid \rangle$ 来表示一个堆,使用 3 元组 $\langle virtual_add, type, id \rangle$ 表示对堆中的一个数据的访问,其中 $start_add$ 表示堆的起始地址; $size$ 表示堆的大小; id 是堆的一个标识,每分配一个堆时, id 加 1; $valid$ 表示当前堆是否有效,当堆分配后 $valid$ 为 TRUE,释放后置为 FALSE. 对于一个堆数据的访问操作,首先使用访问的虚地址 ($virtual_add$) 查找对应的有效堆,该堆必须满足以下公式:

$$\begin{aligned} & heap \rightarrow valid == \text{TRUE} \\ & \&\& heap \rightarrow start_add \leq virtual_add \\ & \&\& virtual_add \leq (heap \rightarrow start_add + \\ & \quad heap \rightarrow size) \end{aligned} \tag{3}$$

程序执行过程中有且仅有一个堆满足式(3);然后基于有效堆的 id ,我们构建该访问操作对应的 3 元组 $\langle virtual_add, type, id \rangle$,并检查现有 3 元组序列中是否存在与新建组完全相同的组,如果已经存在则忽略新建组,否则把新建组添加到 3 元组序列. 对于图 4(b)中的情况,前后两次分配的堆的 id 不同,这样 X 部分也就不会导致任何数据依赖关系.

3.4 降低区域间数据依赖分析的开销

区域间数据依赖分析的时间主要消耗在堆数据的分析上. 从 3.3 节我们知道对堆数据的数据依赖分析分为两步:(1) 使用访存虚地址查找对应的有效堆;(2) 基于有效堆的 id ,使用访问虚地址和访存类型查找对应的 3 元组. 一般情况下,程序执行过程中有效堆的数量要远小于堆中数据的数量,这样在堆数据的数据依赖分析中,第二步查找非常耗时. 基于上述情况,本文把堆数据依赖分析分为两步:(1) 执行插桩后的程序,执行过程中只实施第一步查找,这样可以得到被多个可能并行区域都访问且访问操作包含写操作的堆,只有这些堆才会导致区域间数据依赖关系;(2) 执行插桩后的程序,执行过程中实施两步查找,但是只对第 1 步中得到的堆进行查找,忽略其他的堆. 这样可以大幅度减少数据依赖分析的时间.

4 实验分析

4.1 实验平台及测试用例

本文在 PathScale 编译器^①中实现了区域平均执行时间分析算法,采用 SPEC2006 提供的 train 输入集作为本过程的输入.

本文基于 PIN^[8]实现了区域间数据依赖分析工具,以 SPEC2006 提供的 test 输入集作为本过程的输入. 为了避免编译优化改变程序的数据依赖关系并便于提取程序源代码信息,本过程采用 -O0-g 的编译选项,对程序不实施任何优化并添加调试信息到可执行文件.

文献[9]把现有的应用分为 13 类,同时按照应用的性能受限原因把这 13 类应用分为 5 种情况:受限计算能力、受限访存延迟、受限访存带宽、受限原因暂时不明和任何改进都无效. 本文提出的可能并行区域识别方法主要着眼于充分利用多核系统的计算能力,所以本文选择性能不受限于访存系统性能的应用作为本文的测试用例. 在 13 类应用中,有 5 类应用的性能受限原因不是访存系统性能,包括密集线性代数、 N -体问题、回溯和分支定界、图像模型、有限状态自动机. 这样,我们从这 5 类应用中各选出一个测试用例组成本文的测试用例集合,分别为 libquantum、namd、astar、hmmmer 和 bzip2. 测试用例的主要信息见表 2. 测试平台有 Intel Xeon 和 AMD Opteron,测试平台的主要信息见表 3. 基础编译器为 gcc4.0.1,编译优化选项为 -O3.

表 2 测试用例信息

| 测试用例 | 描述 |
|------------|---------------------------------------|
| libquantum | 一个量子计算机的模拟器,在模拟器上运行 Shor 多项式时间分解算法 |
| namd | 模拟大型分子生物系统,测试集中含有 92224 个载脂蛋白 A-I 的原子 |
| astar | 2 维地图的路径搜索库,包括著名的 A* 算法 |
| hmmmer | 使用 profile 隐式马尔可夫模型搜索基因排序 |
| bzip2 | 一个压缩程序,版本号为 1.0.3 |

表 3 测试平台信息

| 机器 | 主要信息 |
|-------------|-----------------------------|
| Intel Xeon | 4 核,2 个 CPU,2 核/CPU,2 个线程/核 |
| AMD Opteron | 4 核,2 个 CPU,2 核/CPU,1 个线程/核 |

4.2 实验结果及分析

在区域平均执行时间分析中,静态被插桩的循环体和函数调用点的比例见表 4 的第 1 列. 这些区域是程序并行执行时性能收益的来源. 插桩库函数的执行时间占程序总执行时间的比例见表 4 的第 2 列. 从中可以看出,插桩库函数的执行时间所占比例非常低,这样插桩库函数的执行对区域平均执行时间分析精度的影响就可以忽略. 在区域间数据依赖分析中,被多个可能并行区域访问且访问操作包含

① PathScale Compiler. <http://www.pathscale.com/>

写操作的堆所占的比例见表 4 的第 3 列. 从中可以看出, 可能导致区域间数据依赖关系的堆所占的比例比较小. 这样, 本文提出的堆数据的依赖分析方法可以大幅度提高区域间数据依赖分析的效率.

| 表 4 两种分析的效率 | | | |
|-------------|------------------|-------------------|------------------------|
| | 静态被插桩 区域的比例/% | 插桩函数执行 时间的比例/% | 被多个可能并行区域 访问的堆的比例/% |
| libquantum | 7.8 | 0.3 | 23.6 |
| namd | 3.4 | 0.5 | 1.1 |
| astar | 4.0 | 0.7 | 0.2 |
| hmmer | 1.5 | 0.2 | 0 |
| bzip2 | 9.9 | 0.6 | 6.9 |

利用区域平均执行时间分析和区域间数据依赖分析, 本文从 namd、astar、hmmer 和 bzip2 等 4 个测试用例中发掘出了粗粒度并行机会, 从 libquantum 中只发掘出了循环迭代间的细粒度并行机会. 在 astar 和 namd 中, 本文以区域间数据依赖分析得到的导致区域间数据依赖关系的变量 y1rnd、y2rnd 和 y3rnd 为切片标准计算程序切片, 然后利用冗余计算消除这 3 个变量导致的数据依赖关系. 在 hmmer 中, 以导致区域间数据依赖关系的变量为切片标准计算程序切片, 然后通过把切片剥离可能并行区域消除部分数据依赖关系. 另外, 本文通过把可能并行区域的输出写入各自创建的临时文件, 在可能并行区域结束后按照顺序写入目标文件的方法消除 I/O 操作带来的依赖关系. 下面, 我们应用面向行为的投机并行系统对测试用例实施并行. 由于面向行为的投机并行系统主要针对程序中的粗粒度并行性, 所以下面只对 namd、astar、hmmer 和 bzip2 进行分析.

根据面向行为的投机并行系统的需求, 我们只需要在可能并行区域的入口和出口插入接口函数 BeginPPR(P) 和 EndPPR(P) 来标识可能并行区域即可. 在基于区域平均执行时间的并行收益分析中, 我们已经得到了区域所在的文件名和行号信息. 利用这些信息, 本文使用离线的脚本自动地修改源文件插入面向行为的投机并行系统的接口函数.

图 5 给出了相关的程序性能加速比. 对每个测试用例, 本文给出了 4 组加速比数据. 前面两组是在 Intel Xeon 上得到的, 后面两组是在 AMD Opteron 上得到的. 两组 BOP 的数据是基于本文的分析结果应用面向行为的投机并行系统得到的, 两组 Ideal 的数据是删除 BOP 系统中的投机并行开销得到的. 对于所有测试用例, 在应用面向行为的投机并行系统的时候, 本文对所有测试用例 (astar, namd,

bzip2, hmmer) 在 Intel Xeon 和 AMD Opteron 上分别得到了 270%、400%、220%、360% 和 220%、350%、210%、280% 的性能加速. 对比删除投机并行开销前后的性能数据, 对本文的测试用例来说, 投机并行的开销均小于 3.5%. 投机并行开销相对较小的原因是: (1) 本文发现的并行执行的粒度比较大, 分摊了投机并行的开销; (2) 利用区域间数据依赖分析的结果, 本文通过程序切片技术消除了区域间部分依赖关系, 降低了投机失败的概率. 整体上来说, 本文对所有测试用例在 Intel Xeon 和 AMD Opteron 上分别得到了 300% 和 260% 的平均性能加速.

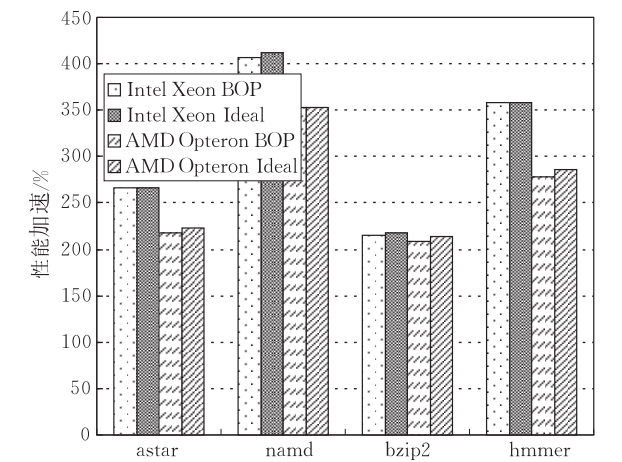


图 5 投机并行的性能

5 相关工作

与本文所关注的投机并行程序的可能并行区域识别问题相关的研究可以分为如下几个方面:

当前, 大多数自动并行的研究主要关注循环迭代间的细粒度并行性. 与此不同的是, Ortega 等关注的是粗粒度的循环体^[10], Shen 等关注的是跨越多个循环体和函数调用点的行为阶段^[11]. 本文不仅考虑到了循环迭代间的细粒度并行性, 也考虑到了循环体和函数调用点间的粗粒度并行性.

在可能并行区域构造方面, 目前有两种方式: (1) 对已有多线程程序进行改造; (2) 利用某种分析方法标记可能并行区域. Chung 等人基于已有的 35 个多线程程序, 使用把程序中的并行和同步原语映射为 transaction 边界的方法标识程序中的可能并行区域^[4]. 但是, 投机并行提供的并行机制比基于锁的并行机制适用范围更广, 例如: 对于一个只有少量迭代之间存在数据依赖的循环, 基于锁的并行机制

不能对其实现高效的并行,但是投机并行可以. 这样,仅仅分析已有多线程程序中的并行区域对投机并行的研究有一定的限制. 因此,人们开始尝试多种分析方法标记可能并行区域. von P C 等人基于程序基本块标记可能并行区域,把连续的多个基本块组合成一个可能并行区域^[12]. 但是,基于程序基本块标记可能并行区域的代价非常高,而且他们的算法只能识别循环迭代间的并行性. 本文基于循环体和函数调用点使用区域平均执行时间分析和区域间数据依赖分析来标记程序中的可能并行区域.

数据依赖分析方面的工作分为两类:静态数据依赖分析和动态数据依赖分析. 静态数据依赖分析是在编译时刻分析程序运行过程中的不同语句之间或者同一语句的不同执行实例之间的相互作用. 静态数据依赖分析都采用保守的分析策略,即当不能肯定没有依赖的时候就保守地认为有依赖. 这种保守的策略导致很多并行变换无法实施. 动态数据依赖分析是指在程序运行时或者利用 profile 的方法进行数据依赖分析. 相对于传统的静态分析方法,动态数据依赖分析能够得到性能更高的激进的优化建议. 但是,动态数据依赖分析得到的数据依赖信息不能保证对程序的所有输入都是正确的. Chen 等利用 profile 的方法得到程序中的数据依赖关系^[13],使用特殊硬件来保证由该数据依赖关系指导的程序变换的正确性. 本文也是使用 profile 的方法得到区域间的数据依赖关系,同时使用面向行为的投机并行系统来保证程序变换的正确性.

6 总结和工作展望

可能并行区域的识别对提高投机并程序的性能非常重要,本文提出的基于区域平均执行时间和数据依赖信息的可能并行区域识别方法,利用区域平均执行时间进行并行收益分析发现程序热点,不仅可以发掘程序中蕴含的循环迭代间的细粒度并行性,而且可以发掘循环体和函数调用点间的粗粒度并行性,同时通过区域间数据依赖分析提高可能并行区域投机并行的成功率,为构造高效的投机并行程序提供了重要依据. 本文使用面向行为的投机并行系统来保证程序变换的正确性,对 SPEC2006 中的 4 个测试用例有 300% 和 260% 的平均性能加速.

本文提出的可能并行区域识别方法能够识别出并行执行时间长且失败率低的高质量的可能并行区域. 现阶段,我们没有考虑可能并行区域并行粒度的

问题. 并行粒度不合理将导致系统负载不均衡,从而出现大量的空闲等待时间. 将来我们会在本文的基础上研究如何通过分裂和合并可能并行区域来动态调整并行粒度. 另外,可能并行区域中 I/O 操作的顺序性如何维护,以及如何通过采样的方法进一步加速区域平均执行时间分析和区域间数据依赖分析等都是将来要研究的课题.

参 考 文 献

- [1] Boulet P, Brandes T. Evaluation of automatic parallelization strategies for HPF compilers. Lecture Notes in Computer Science, 1996, 1067: 778-783
- [2] Steffan J G, Mowry T C. The potential for using thread-level data speculation to facilitate automatic parallelization//Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA). Las Vegas, 1998: 1-13
- [3] Hammond L, Wong V, Chen M et al. Transactional memory coherence and consistency//Proceedings of the International Symposium on Computer Architecture (ISCA). München, 2004: 102-113
- [4] Chung J W, Chafi H et al. The common case transactional behavior of multithreaded programs//Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA). Austin, 2006: 266-277
- [5] Sohi G S, Breach S E, Vijaykumar T N. Multiscalar processors//Proceedings of the International Symposium on Computer Architecture (ISCA). S. Margherita Ligure, Italy, 1995: 414-425
- [6] Kejariwal A, Tian X-M, Girkar M, Li W, Saito H, Banerjee U et al. Tight analysis of the performance potential of thread speculation using SPEC CPU2006//Proceedings of the Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP). California, 2007: 215-225
- [7] Ding C, Shen X-P, Kelsey K, Tice C, Huang R-K, Zhang C-L. Behavior-oriented parallelization//Proceedings of the Conference on Programming Language Design and Implementation (PLDI). California, 2007: 223-234
- [8] Luk C-K, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S et al. Pin: Proceedings of the Building Customized Program Analysis Tools with Dynamic Instrumentation//Proceedings of the Conference on Programming Language Design and Implementation (PLDI). Chicago, 2005: 190-200
- [9] Patterson D A et al. The landscape of parallel computing research: A view from Berkeley. University of California, Berkeley: Technical Report UCB-EECS-2006-183, 2006
- [10] Ortega D, Ayguade I M E, Valero M, Krishnan V. Quantifying

the benefits of specint distant parallelism in simultaneous multi-threading architectures//Proceedings of the 8th International Conference on Parallel Architectures and Compilation Techniques, California, 1999; 117-125

[11] Shen X-P, Ding C. Parallelization of utility programs based on behavior phase analysis//Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC), New York, 2005; 425-332

[12] von P C, Ceze L, Cascaval C. Implicit parallelism with ordered transactions//Proceedings of the Symposium on Principle and Practice of Parallel Programming (PPoPP), San Jose, California, 2007; 79-89

[13] Chen T, Lin J, Dai X-R et al. Data dependence profiling for speculative optimizations//Proceedings of the International Conference on Compiler Construction (CC), Barcelona, 2004; 57-72



ZHANG Chao, born in 1982, Ph. D. candidate. His research interests focus on high performance compile techniques.

WANG Lei, born in 1976, Ph. D. candidate. Her re-

search interests focus on high performance compile techniques.

XIANG Xiao-Ya, born in 1984, master. Her research interests focus on compile memory optimization.

FENG Xiao-Bing, born in 1969, Ph. D. , professor, Ph. D. supervisor. His research interests include compile optimization techniques and binary translation.

Background

Multi-core processors have become a common way of reducing chip complexity and power consumption while maintaining high performance. And how to construct high performance parallel programs becomes a hotspot of research again. Speculative parallelism is a promising way to this problem. And the quality of possibly parallel regions is very important for speculative parallelism. This paper focuses on

how to find high performance possibly parallel regions for speculative parallelism. The authors present a method based on execution time and data dependence profile to find possibly parallel regions. The new method can increase the degree of parallelism and the success rate of speculative parallelism. This work is supported by the National Basic Research Program (973 Program) of China under grant No. 2005CB321602.