基于分布式系统的可并行循环动态识别技术*

阳雪林, 于 勐, 陈道蓄, 谢 立

(南京大学 计算机软件新技术国家重点实验室,江苏 南京 210093);
(南京大学 计算机科学与技术系,江苏 南京 210093)
E-mail: yxl@dislab.nju.edu.cn
http://www.nju.edu.cn

摘要: 针对分布式环境下可抽取观察循环的不规则串行程序循环的动态依赖关系分析问题,提出了一个基于观察/执行模型的动态分析算法.其贡献是:(1) 算法可并行执行于分布式系统;(2) 直接分析具有拷入和最后赋值操作的循环;(3) 给出了循环的并行化方法;(4) 并不要求循环是完全可并行的,对某些部分可并行循环,也支持其并行执行.理论分析和实验表明,在处理器数量适当的情况下,循环可以并行时,可以获得很好的加速比;不能并行时,对串行执行增加的开销也是小的.从而为分布式环境下开发更多的循环并行性提供了一种新的手段.

关 键 词: 分布式系统;循环并行性分析;动态并行;观察/执行模型
中图法分类号: TP311 文献标识码: A

随着芯片运算速度和网络传输速度的不断提高,超大规模的并行机系统、并行工作站和工作站网络等不同形式的并行计算环境的计算能力大大提高,性能价格比也不断提高,并行计算环境得到普及.但编写适应并行计算环境的软件却依旧是烦琐、易出错和困难重重的工作.自动并行编译技术为并行化现有串行程序及编写新的并行程序提供了一种重要的手段.由于循环是程序中蕴含并行性最丰富的一种结构,也往往是程序执行最耗时的部分,开发循环的并行性是自动并行编译研究的重点,而循环并行性分析又是开发循环并行性的必要前提.按分析的时间划分,循环并行性分析方法可分为静态分析方法(即在编译时进行分析的方法)和动态分析方法(即在运行时进行分析的方法).目前对静态分析方法的研究已取得相当大的成就,大部分规则程序(即内存存取模式编译时已知的程序)的循环可采用静态分析方法来分析.动态分析的目标是对不规则的串行程序(即内存存取模式使用特别的映射来描述的程序,在 Fortran 或 C 中此映射一般用下标数组(subscript array)(即数组的值为另一数组的下标)来表示.如图 1 中的程序所示[1])进行分析,也希望能对目前的静态分析算法所不能分析的规则串行程序,如存在复杂的非线性表达式、包含子程序调用的深度嵌套循环的串行程序进行分析.由于不规则的程序常见于流体力学、量子力学、分子生物学等系统模拟程序,往往这些程序执行又十分耗时,开发动态的分析方法来挖掘这些程序的并行性进而加快程序的运行,就显得很有必要[1].

已有一些研究从不同的角度支持了动态并行性的挖掘[1~4].动态并行化循环常采用的一种模型是观察/执行模型,即从循环中抽取一个观察(inspector)循环,通过观察循环的执行结果来判断循环能否并行化.若不能并行化,则随后串行执行循环,否则并行执行循环.Kennedy 等人对循环分布的研究[2]可用于观察循环的抽取.Voss[3]和 Nguyen[4]等人进行了基于共享内存处理器的动态调度的研究,目标是让程序自动确定循环并行执行还是串行执行效率更高,更进一步就是确定并行执行时使用几个处理器效率最高,并自动调节处理器的使

* 收稿日期: 2000-11-27; 修改日期: 2001-05-22
基金项目: 国家 863 高科技发展计划资助项目(863-306-ZT02-0301)

作者简介: 阳雪林(1967－),男,广西桂林人,博士,主要研究领域为并行与分布式计算;于勐(1972－),男,辽宁鞍山人,博士生,主要研究领域为并行与分布式计算;陈道蓄(1947－),男,江苏苏州人,教授,博士生导师,主要研究领域为并行与分布式计算;谢立(1942－),男,江苏常熟人,教授,博士生导师,主要研究领域为并行与分布式计算,网络安全.

用.Rauchwerger 等人系统地研究了基于共享处理器的可并行循环动态识别问题[1].

Rauchwerger 的 RPD(reduction privatizing doall)测试采用了观察/执行模型,基本解决了在共享处理器下可抽取观察循环的不规则串行程序的循环的动态依赖关系分析问题,但在算法中没有考虑具有拷入(copy-in)和最后赋值(last assignment)操作的循环并行化问题.

```
W[1:8]=[1,3,2,3,7,5,6,12]
R[1:8]=[1,9,2,2,7,8,8,12]

S1:Do i=1,8
S2:   A[W[i]]=…
S3:   …=A[R[i]]
S4: Enddo
```

Fig.1 An example of irregular programs

图 1 一个不规则程序的例子

考虑到分布式系统在并行计算中的广泛应用以及拷入和最后赋值操作是分布式系统中并行计算的常见操作,需要开发基于分布式系统的动态分析算法.针对可抽取观察循环的不规则串行程序循环的动态依赖关系分析问题,考虑到分布式系统的特点,我们提出了适用于分布式系统的 DPP(distributing partial parallell)循环测试算法.我们的贡献是:(1) 算法可并行执行于分布式系统;(2) 直接分析具有拷入和最后赋值操作的循环;(3) 给出了循环的并行化方法;(4) 并不要求循环是完全可并行的,对某些部分可并行循环,也支持其并行执行,从而支持更多的循环并行.

1 算法描述

1.1 背景介绍

一个循环可完全并行执行当且仅当不同迭代的执行顺序对循环的结果不产生任何影响.为确定迭代执行顺序是否影响循环的结果,需要分析循环中语句的数据依赖关系.存取同一内存位置的语句存在 3 种依赖关系:流依赖(写读)、反依赖(读写)、输出依赖(写写).如果迭代间不存在依赖关系,则循环可以完全并行执行.如图 2(a)所示的循环中不存在跨迭代的依赖关系,循环可以并行执行.

```
S1: DO i=1,n
S2:   A[i]=2*A[i]
S3: ENDDO
```

(a) A loop without data dependences

(a) 无数据依赖的循环

```
S1: DO i=1,n/2
S2:     tmp=A[2*i]
S3:     A[2*i]=A[2*i−1]
S4:     A[2*i−1]=tmp
S5: ENDDO
```

(b) A loop which data dependences can be removed by privatizing

(b) 可用私有化技术消除数据依赖的循环

```
S1: DO i=2,n
S2:   A[i]=A[i]+A[i−1]
S3: ENDDO
```

(c) A loop with flow dependences

(c) 存在流依赖的循环

Fig.2

图 2

流依赖反映了数据生产者和消费者的依赖关系,如果在循环的不同迭代间存在流依赖,则循环一般不能并行执行.如图 2(c)所示.

反依赖和输出依赖则是由于内存(程序变量)重用而引起的,常可以通过对循环施以“私有化”变换而消除.私有化是指在合作并行执行循环的各个处理器上复制引起反依赖和输出依赖的程序变量.如图 2(b)所示的循环,即可通过私有化消除反依赖以并行执行.最常采用的数组私有化的标准为:

在循环 L 中被引用的共享数组 *A* 能被私有化的标准是满足以下条件:在循环 L 中的任一迭代中对数组 *A* 的元素的读取都在对同一元素写之后.

然而,这个私有化标准只适用于处理只作为迭代内工作空间的变量所引起的依赖关系.不能处理拷入问题,即一个共享变量通过读一个在循环外计算的值来初始化的问题.为了同时私有化具有拷入性质的变量的循环,我们扩充了数组私有化的标准:

在循环 L 中被引用的共享数组 A 能被私有化的标准是在任一迭代中对数组 A 的元素的读取至少满足以下条件之一:(1) 在同一迭代中对此元素写之后;(2) 在此迭代前的所有迭代中此元素均未被写.

除拷入问题外,私有化的算法还需要考虑最后赋值问题,即一个被私有化的变量在循环结束后仍是活跃的,私有化的算法还需要保证此变量在循环结束后被正确地赋值.

对类似于图 2 中那些内存存取模式确定的例子,静态分析即可确定其循环是否可并行.但对图 1 中这样的例子,对于变量是一个其下标索引在编译时未知的数组元素这样的情况,静态分析不起作用,只能采用动态分析的方法.此时,可通过运行一个测试循环,首先判断循环是否存在不可消除的依赖关系,如果不存在,则并行执行,否则,串行执行循环.即采用所谓的观察/执行模型,这是动态分析中常用的一种方法.我们的 DPP 测试即采用了这一方法.由于在分布式系统上频繁调度代价较高,我们的目标确定为判断循环能否以一定的分布在各处理器上并行,对循环是否可完全并行不作要求.由于测试执行需要消耗时间,因此,要求测试本身可并行执行,以降低开销增加可扩性.由于循环的并行方式和测试方式密切相关,我们同时给出了循环并行的方法.

1.2 算法描述

综合考虑以上问题,算法描述如下:

1. 测试:

1.1. 处理器数量确定.根据可用处理器数量和估计循环粒度,确定使用处理器数量p,处理器分别记为0,1,…,p−1.

1.2. 迭代空间划分.将迭代空间划分为p份,每一份记为 $\vec{I}_i(0 \le i < p)$,且满足:

$\forall i, \forall \vec{j} \in \vec{I}_i, \forall \vec{k} \in \vec{I}_{i+1}$,有$\vec{j} < \vec{k}$

将迭代子空间 $\vec{I}_i(0 \le i < p)$ 赋给处理器i.即在任一处理器上执行的循环迭代空间在串行执行时是连续的,且在处理器i上执行的迭代在串行执行时总在处理器i+1上执行的迭代之前.

1.3. 标记.对编译时不能确定依赖关系的共享数组$A[1:s]$,处理器i定义标志数组$A_{ri}[1:s]$,$A_{wi}[1:s]$,(0≤i<p),将这两个标志数组的元素初始化为假,并按如下方法标记:

对在处理器i上的迭代子空间 $\vec{I}_i(0 \le i < p)$,按如下方式处理对数组元素$A[k]$(1≤k≤s)的存取:

(a) 写:设置$A_{wi}[k]$为真;

(b) 读:若$A_{wi}[k]$为假,则置$A_{ri}[k]$为真;

1.4. 分析.对每一共享数组A:

(a) 如果∃i,k满足0≤i≤p,1≤k≤s,使得$f_{Aw}(i-1)[k] \wedge A_{ri}[k]$为真,则循环不能并行. $f_{A_w}(i)$ 为与A同大小的布尔数组,定义如下:

$$f_{A_w}(i)[k] = \begin{cases} A_{w0}[k], & \text{若} i = 0, \\ f_{A_w}(i-1)[k] \vee A_{wi}[k], & \text{若} 0 < i < p. \end{cases}$$

(b) 否则,循环可并行执行.

2. 可并行时循环地执行:

若循环是可并行的,根据1.2的迭代子空间划分,将迭代子空间 $\vec{I}_i(0 \le i < p)$ 赋给处理器i.将数组A被处理器i读取部分复制于处理器i,记为$A_i(0 \le i < p)$.方法如下:

若$A_{ri}[k]$为真,则置$A_i[k]$为$A[k]$.

并行计算中,所有对$A[k]$的访问均指向$A_i[k]$.

并行计算结束后,根据A_{wi}的结果,计算最后结果A.方法如下:

定义$g_A(i)$为与A同大小、同类型的数组,

令$g_A(0)[k]=A_0[k]$,若$A_{w0}[k]$为真,

$$g_A(i)[k] = \begin{cases} g_A(i-1)[k], & \text{若} A_{wi}[k] \text{为假} \\ A_i[k], & \text{若} A_{wi}[k] \text{为真} \end{cases}, \text{对} 0 < i < p.$$

则$\forall k$,若$f_{A_w}(p-1)[k]$为真,$A[k] = g_A(p-1)\ [k]$,否则,$A[k]$不变.

1.3 实现优化

(1) p 值的确定.可由用户根据程序以往执行情况指定,也可由程序在执行中动态确定.

(2) 迭代空间的划分.划分并不要求均匀,可根据不同处理器的计算能力划分,这特别适合于结点处理能力不同的异构 NOW 环境.

(3) 标志数组的实现.处理器 i 上的标志数组 $A_{ri}[1:s]$,$A_{wi}[1:s]$,$0\le i<p$,初始化时无须数据传送.为减少存储消耗,如果硬件支持,可采用位数组存储.若对 A 的读写次数不多,也可采用 hash 表实现.

(4) 标志数组的传送.若对 A 的读写次数不多,可只传送值为真的 $f_{Aw}(i)$ 元素下标.

(5) 分析.依处理器递增顺序,处理器 $i(0<i<p-1)$接收 $f_{Aw}(i-1)$,检测是否与 A_{ri} 冲突,若冲突,广播不能并行消息,否则计算 $f_{Aw}(i)$ 并传送到处理器$(i+1)$.处理器 0 只需传送 $f_{Aw}(0)$,不需接收和检测;处理器$(p-1)$不需要传送 $f_{Aw}(p-1)$.处理器$(p-1)$发现无冲突,则广播可并行消息.

(6) 结果收集.并行计算结束后,依处理器递增顺序,处理器 $i(0<i<p-1)$接收 $g_A(i-1)$,计算 $g_A(i)$ 并传送到处理器$(i+1)$.处理器 0 不需接收和计算,只需传送 $g_A(0)$.

(7) 测试结果重用.如果循环以同样的存取模式又一次执行,则可以重用第 1 次测试的结果,这样如果可并行,可以在多次并行间分摊测试的开销;如果不能并行,则可以降低测试的开销.

2 复杂性分析

假设处理器运算速度是均匀的 k,处理器两两之间通信速率为 v,通信启动时间为 u,p 表示处理器数量,n 表示循环的迭代次数,s 表示共享数组的元素数量,a 表示在一个单独迭代中对共享数组的最大存取数量,则测试需要的时间是

$$T(p,k,v,u,n,s,a)=O\big(na/(pk)+p\times(u+s/v+s/k)\big).$$

若 $na<s$,则测试需要的时间是

$$T(p,k,v,u,n,s,a)=O\big(na/(pk)+p\times(u+na/v+na/k)\big),$$

其中标记相耗时 $O(na/(pk))$.处理器i在私有标记数组A_{ri},A_{wi}中记录读写操作是常量时间的操作,因为每一处理器只负责 $O(na/p)$ 个存取,所有的存取标记能在 $O(na/(pk))$ 时间内完成.

分析相耗时 $O(p\times(u+s/v+s/k))$.将 $f_{Aw}(i-1)$ 从处理器$(i-1)$传送至处理器 i 耗时 $O(u+s/v)$.比较 $f_{Aw}(i-1)$ 与 A_{ri} 耗时 $O(s/k)$.将 $f_{Aw}(i-1)$ 与 A_{wi} 合并计算 $f_{Aw}(i)$ 耗时 $O(s/k)$.即分析相耗时与 $\mathrm{Max}(p\times(u+s/v),p\times s/k)$ 成比例.若 $na<s$,则将 $f_{Aw}(i-1)$ 从处理器$(i-1)$传送至处理器 i 耗时 $O(u+na/v)$.比较 $f_{Aw}(i-1)$ 与 A_{ri} 耗时 $O(na/k)$.将 $f_{Aw}(i-1)$ 与 A_{wi} 合并计算 $f_{Aw}(i)$ 耗时 $O(na/k)$.即分析相耗时与 $\mathrm{Max}(p\times(u+na/v),p\times na/k)$ 成比例.此时分析相耗时 $O(p\times(u+na/v+na/k))$.

从以上的分析我们可以得出结论:由于存在通信开销,DPP 测试的性能随处理器数量的增大而增大到一个最高点后,处理器的继续增加会带来性能的下降.在循环不变的情况下,最佳处理器数受处理器运算速度、处理器间通信速率以及通信启动时间的影响,处理器运算速度愈快,处理器间通信速率愈快,通信启动时间愈小,则最佳处理器数量可以愈大.

3 实验结果

实验在用 10M 的 Ethernet 连接的 12 台 PC 上进行.PC 的处理器为 Celeron466,内存 32M,操作系统为 LINUX(RedHat6.2),通信平台为 PVM3.4.对测试软件包 PERFECT 中的程序 BDNA 的具有不规则数据存取的循环 240 进行了测试.循环 240 在程序中共执行 5 次,5 次执行接近程序串行执行总时间的 50%,是一个有代表性的循环.为了减少系统内部进程活动和网络负载变化对测试的影响,测试结果采用 6 次执行结果的最小值.测试结果为:当串行执行时,循环 240 执行一次耗时 1 539ms;将循环 240 用 DPP 测试改写后,循环可并行执行,并行执行的执行时间、开销和加速比见表 1.实验表明,由于循环 240 中的计算方法简单,测试开销占并行执行时间的比例是较大的,但由于测试的可扩性好,随处理器的增加,测试开销逐渐减小,当不能并行时,对串行所增加的时间也逐渐减少.实验也显示,即使是在传输速率如此低的网络,最佳处理器数量也不小.因而说明算法具有很好的可扩性.

Table 1 Results of parallel execution

表1 并行执行结果

Number of processors①	2	4	6	8	10	12
Parallel execution time②(ms)	1 648	858	596	461	376	340
Slowdown of test③(ms)	795	412	280	210	171	148
Slowdownof test/Sequential execution time④	0.52	0.27	0.18	0.14	0.11	0.096
Speedup⑤	0.93	1.79	2.58	3.34	4.09	4.53

①处理器数量,②并行时间,③测试开销,④测试开销/串行时间,⑤加速比.

4 结 论

与静态分析不同,动态分析要增加程序执行开销,因此,在考虑分析精度的同时,往往还要考虑分析的开销和并行可扩性等因素,具体的算法往往是这些因素的折衷.因此对不同的应用环境和需求,往往需要提出不同的算法.

理论分析和实验都表明,我们针对分布式环境下可抽取观察循环的不规则串行程序循环的动态依赖关系分析问题,考虑了拷入和最后赋值操作,提出的适用于分布式系统的 DPP 循环测试算法,在处理器数量适当的情况下,可以并行时,可以获得很好的加速比;不能并行时,对串行执行增加的开销也是小的,而其可扩性也是很好的,从而为分布式环境下开发更多的循环并行性提供了一种新的手段.

References:

[1] Rauchwerger, L. Run-Time prallelization: a framework for parallel computation [Ph.D. Thesis]. University of Illinois, Urbana-Champaign, 1995.

[2] Kennedy, K., McKinley, K.S. Loop distribution with arbitrary control flow. In: Proceedings of the Supercomputing'90. New York: ACM Press, 1990. 407~416.

[3] Voss, M., Eigenmann, R. Reducing parallel overheads through dynamic serialization. In: Proceedings of the 13th International Parallel Processing Symposium and the 10th Symposium on Parallel and Distributed Processing. Los Alamitos, CA: IEEE Computer Society, 1999. 88~92.

[4] Nguyen, T.D., Vaswani, R., Zahorjan, J. Maximizing speedup through self-tuning of processor allocation. In: Proceedings of the 10th International Parallel Processing Symposium. Los Alamitos, CA: IEEE Computer Society, 1996. 463~468.

A Run-Time Technique for Parallel Loop Identification Based on Distributed System*

YANG Xue-lin, YU Meng, CHEN Dao-xu, XIE Li

(*State Key Laboratory for Novel Software Technology*, *Nanjing University*, *Nanjing* 210093, *China*);

(*Department of Computer Science and Technology*, *Nanjing University*, *Nanjing* 210093, *China*)

E-mail: yxl@dislab.nju.edu.cn

http://www.nju.edu.cn

Abstract: An algorithm is presented to solve the problem of run-time parallel loop identification based on distributed system for the loops in irregular programs, from which inspective loops can be extracted. The contributions are: (1) The algorithm is fully parallel and can be run on a distributed system; (2) Loops with copy-in and last assignment attributes can be directly analyzed; (3) A method is given for a loop to parallel; (4) Some partial parallel loop can also be parallelized. The theoretical analysis and experimental results show that in adequate number of processors, if a loop is parallel, a good speedup can be obtained; if a loop is not parallel, the slowdown of serial executing is small. A new method is given for exploiting more loop paralizations on the distributed system.

Key words: distributed system; loop parallelization analysis; run-time parallel; inspective/executive model

* Received November 27, 2000; accepted May 22, 2001

Supported by the National High Technology Development 863 Program of China under Grant No.863-306-ZT02-0301