基于切片技术的并行化研究

桑春雷[1,2] 张兆庆[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)[1]
(中国科学院研究生院 北京 100039)[2]

摘 要 程序可以看作由很多计算组成(例如一个循环或一个平直代码片断),它们彼此相关或者无关,共同为计算最终的结果服务,其中彼此不相关的计算是并行性的重要来源。程序切片(Program Slicing)是一种程序分解技术,能够根据切片标准从程序中提取出特定的计算,切片技术的应用很广泛,例如程序调试、理解、维护等软件工程应用。切片技术作为一项程序分解技术,也可以用来帮助串行程序并行化。研究利用切片技术表示和发掘程序中的无关计算带来的并行性。首先提出一种基于OpenMP扩展的切片并行编程模型,用以表达程序中的切片并行性。另外,开发了一个基于切片的并行化分析系统,用来辅助程序员发掘程序中的切片并行性。

关键词 切片,并行化,OpenMP,分析工具

Slice Parallelization Research Based on Program Slicing Technique

SANG Chun-lei[1,2] ZHANG Zhao-qing[1]
(Key Laboratory of Computer System and Architecture, Institue of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China)[1]
(Graduate School of the Chinese Academy of Sciences, Beijing 100039, China)[2]

Abstract Computer program consists of a lot of computation units, which depend on each other or not, and serve to final computation result. The independent computation can be parallelized to accelerate the whole program. Program slicing can extract independent computation from large program according to slicing criterion, which is one set of variable and one program position. Program slicing can help exploiting parallelism from serial program, which we call slicing parallelism. This paper extended OpenMP to model slicing parallelism, and we also developed one slicing analysis tool, which could recognize slicing parallelism and help programmer to parallelize serial program.

Keywords Slicing, Parallelization, OpenMP, Analysis tool

1 引言

程序切片技术作为一项程序分解技术可以将串行程序分解为若干独立的程序[2],因而能够将串行程序自动地并行化。文献[2]重点讨论将整个程序分解之后如何构建原始的串行程序行为,但将整个程序按最后输出分解为若干程序的情形并不常见。我们发现在程序的局部存在更多的可分解的计算,即无关计算,我们称这样的并行性为切片并行性。程序员或者自动化分析工具可以识别、标记这样的无关计算区域,进而指定切片方式来表达这样的切片并行性。传统的自动并行化编译器大都依赖程序的数据并行性,研究如何识别和划分程序中的可并行循环。而切片并行在此基础上为程序并行化提供了更为广阔的模型。

OpenMP编程模型为程序员提供两种任务划分模型:循环划分和段(section)划分,分别用制导来描述。切片并行也可以看作是段并行的扩展,段并行(parallel sections)局限于连续的程序片断,而切片并行则不受这样的限制,可以跨越连续语句、结构化语句,甚至一般的控制语句。

程序切片[1,3-5]技术最早是由Mark Weiser提出的,能够根据切片标准从程序中提取出特定的计算。切片标准一般为程序中某处的变量或者变量集合,例如可以用行号和变量名字配对$C=\langle n,V\rangle$表示一个切片标准,而程序切片是所有程序中所有影响变量V在行号n处的值的代码片断。切片的方法主要有两种[5]:基于数据流的方法和基于图到达算法的方法。它们的基础都是程序的定义使用(def/use)分析。Mark Weiser提出切片技术时首先将它应用于程序调试。一般说来,程序员在发现程序出错时,只关心计算出错误值的程序代码片断,因此切片恰好是一个提取这些程序代码片断的有效工具。之后的研究主要集中在将切片应用于程序理解、调试、维护、重构等软件工程方面。

随着计算需求的增加和计算机体系结构的发展,开发大规模并行程序也变得越来越重要。过去几十年并行计算机体系结构和编程模型的开发取得了一定成就,但远不能满足需求;尤其是最近单处理器的运算能力达到了极点,多核处理器

到稿日期:2010-08-20 返修日期:2010-12-15 本文受国家高技术研究发展计划(863)项目(2008AA01Z115)资助。
桑春雷(1979—),男,博士生,主要研究方向为编译技术、程序切片,E-mail:scl@ict.ac.cn;张兆庆(1938—),女,研究员,博士生导师,主要研究方向为编译技术。

成为主流，并行程序设计显得更为重要。并行程序设计不再局限于科学计算领域，不再是少数专业程序员的工作，而将成为多数程序员的日常工作。MPI 与 OpenMP 是最为成功的两个并行编程模型和语言(库)，几乎所有的并行计算机系统厂商都提供对这两种模型的支持。然而，MPI 的程序书写相当困难，易出错，开发周期长；OpenMP 编程相对简单，但它只支持共享内存多处理机上的数据与任务并行性开发。随着多核(multi-core)和众核(many-core)成为处理器研发的重要方向，探寻新的并行性和编程模型显得更为重要[7]。

Mark Weiser 曾经提出利用切片技术帮助串行程序自动并行化[2]。切片技术是一项程序分解技术，Mark Weiser 将程序输出作为切片标准，据此将串行程序分解为若干进程，而每个进程计算一个输出从而达到多进程并行化的目的。Mark Weiser 的文章主要关注如何从切片后的程序中构建串行行为。据我们观察，一般程序的输出很少，并且存在内在的相互关联，因此 Mark Weiser 的方法不实用。

程序切片对应程序中的一个特定计算，如果这些计算是相互独立的，那么它们将是并行性的来源。我们的观察发现，将整个程序的输出划分独立计算是很困难的，然而程序中存在很多无关的计算，它们存在于程序的局部。例如图 1(a)中的程序存在两个无关的规约计算。如果能将这两个计算分解开来，如图 1(b)所示，并且将两个计算放在不同的线程上执行，那么程序会获得并行性。

```
for (i = 0; i < N; i++) {
   s = s + a[i];
   t = t + b[i];
}
```

(a)

```
/* Thread #1 */
for (i = 0; i < N; i++)
   s = s + a[i];

/* Thread #2 */
for (i = 0; i <N; i++)
   t = t + b[i];
```

(b)

图 1　程序片段

再如，SPEC CPU2000 swim 中的 CACL2 函数(见图 2)，含有对 3 个数组 UNEW，VNEW，PNEW 的运算。我们利用切片将这 3 个计算分开，获得线程级并行性。swim 中有 4 个函数具有这样的切片并行性。

我们扩展 OpenMP 编程模型以支持切片并行模型，同时开发一个基于切片技术的自动并行化系统，分析程序中的切片并行性。基于 OpenMP 的扩展编程模型可以有效地表达切片并行性，并且能和 OpenMP 很好地结合在一起，书写方便。我们的自动化分析系统可以部分发掘程序中的切片并行性，也可以为程序员提供切片分析指导。

OpenMP 关心的是程序中的数据并行性，计算划分主要集中于循环之上。例如图 1(a)的程序，OpenMP 可以帮助程序员将两个规约循环并行化，但无法将两个规约循环分解开来分别运算。OpenMP 也支持段并行(section)，例如可以将图 1(a)的循环体分为两个 section 计算，但这样的划分粒度过小，很难获得好的性能。而切片并行则可以更好地并行化这段代码，因为两个规约是独立的。如果分别以变量 s 和 t 作为切片标准，我们可以将这段代码分解为图 1(b)的形式进行并行化(图 5 给出了我们的制导书写方式)。在多处理器集群(SMP cluster)中可以使用 MPI 与 OpenMP 的混合编程模型，发掘程序中的多层并行性。我们的 OpenMP 扩展同样可以平滑地与 MPI 结合在一起，发掘 MPI 程序进程内的切片并行性。

```
        TDTS8 = TDT/8.D0
        TDTSDX = TDT/DX
        TDTSDY = TDT/DY

C SPEC removed CCMIC$ DO GLOBAL
        DO 200 J=1,N
        DO 200 I=1,M
        UNEW(I+1,J) = UOLD(I+1,J)+
       1
TDTS8*(Z(I+1,J+1)+Z(I+1,J))*(CV(I+1,J+1)+CV(I,J+1)+CV(I,J)
      2          +CV(I+1,J))-TDTSDX*(H(I+1,J)-H(I,J))
        VNEW(I,J+1) = VOLD(I,J+1)-TDTS8*(Z(I+1,J+1)+Z(I,J+1))
      1          *(CU(I+1,J+1)+CU(I,J+1)+CU(I,J)+CU(I+1,J))
      2          -TDTSDY*(H(I,J+1)-H(I,J))
        PNEW(I,J) = POLD(I,J)-TDTSDX*(CU(I+1,J)-CU(I,J))
      1          -TDTSDY*(CV(I,J+1)-CV(I,J))
  200 CONTINUE

C
C       PERIODIC CONTINUATION
C
        DO 210 J=1,N
        UNEW(1,J) = UNEW(M+1,J)
        VNEW(M+1,J+1) = VNEW(1,J+1)
        PNEW(M+1,J) = PNEW(1,J)
  210 CONTINUE
        DO 215 I=1,M
        UNEW(I+1,N+1) = UNEW(I+1,1)
        VNEW(I,1) = VNEW(I,N+1)
        PNEW(I,N+1) = PNEW(I,1)
  215 CONTINUE
        UNEW(1,N+1) = UNEW(M+1,1)
        VNEW(M+1,1) = VNEW(1,N+1)
        PNEW(M+1,N+1) = PNEW(1,1)
```

图 2　SPEC CPU2000 swim 中的程序片断

切片并行化方式同时提供一种局部性变换的描述，在将独立无关计算分开的同时，也可以将独立无关的数据分开，从而减少单个线程的数据足迹。例如，上面的例子中，数组 a，b 被同时访问到，但利用切片并行化可以将数组 a，b 的访问分割到不同的线程上，从而为局部性优化提供更大的空间。

第 2 节将简单介绍切片技术和我们的实现；第 3 节介绍如何利用切片技术将串行程序划分为多线程程序；第 4 节介绍我们的试验结果；最后是总结和未来工作。

2　切片技术

我们开发了一个程序切片工具，作为切片并行化分析的基础。采用基于程序依赖图(Program Dependence Graph，PDG)的切片方法[3-5]，利用程序控制流分析获得控制依赖关系，利用定值引用关系获得语句之间的数据依赖关系(这里不计算数组元素之间的依赖关系；利用别名分析的结果，如果分析不清则做保守假设)。这里不详细介绍 PDG 的构造过程[3]，只给出示例。图 3 是图 2 中程序的 PDG，其中节点{1，2，6，10}代表源程序中的 DO 语句，而其他节点{3，4，5，7，8，9，11，12，13}代表循环体内的数组赋值语句，其他节点(例如标量赋值、函数出口和入口)并未画出。PDG 中的边表示程序语句之间的依赖关系，控制依赖(例如〈1，2〉，〈2，3〉)用细线条画出，而数据依赖关系(例如〈3，6〉，〈6，10〉)用粗线条画出。

图 3 中，节点表示程序语句，边表示语句依赖关系(其中细边为控制依赖，粗边为数据依赖)。节点{1，2}表示代表循环 200 中的 J 循环和 I 循环，节点{3，4，5}分别表示其内的 3 个语句；节点 6 表示循环 210，节点{7，8，9}表示其内的 3 个语

句；节点 10 表示循环 215，节点{11,12,13}表示其内的 3 条语句。

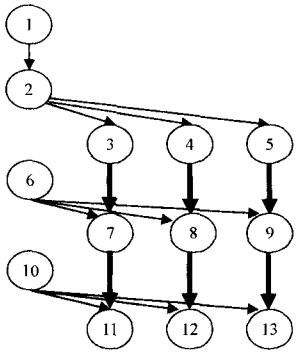

图 3　图 2 中 swim 程序片断的 PDG

在 PDG 上切片标准的表述稍有不同。每个 PDG 节点 n 都可以作为切片标准，也称为切片种子。节点 n 的切片记为 slice(n)，它是 PDG 中所有能够到达节点 n 的节点集合。为了自动分析的方便，我们一次计算出所有节点的切片。图 4 给出基于程序依赖图的切片算法。

```
基于程序依赖图的切片算法：
输入：程序依赖图(PDG)
输出：每个 PDG 节点的切片
算法：
      // 初始化
      foreach n in PDG
            slice(n) = {n}
      changed = true
      while (changed)
            changed = false
            foreach edge m->n in PDG // 按拓扑序遍历
                  if (slice(n) does not include slice(m))
                        slice(n) = slice(n) union slice(m)
                        changed = true
```

图 4　基于程序依赖图的切片算法

按拓扑序遍历每条边，因此 while 循环的收敛速度很快，与 PDG 中环的数目成正比，一般可认为是个小常数。例如在图 3 中，slice(2)={1,2}，slice(3)={1,2,3}，slice(11)={1,2,3,6,7,10,11}。

3　基于切片的程序划分技术

首先定义切片并行性的描述语言，扩展 OpenMP 制导描述基于切片的任务划分；之后介绍我们的切片并行性分析工具；最后简单介绍如何基于切片制导生成多线程程序的细节。

3.1　基于切片制导的程序划分

本节扩展 OpenMP 编程模型，支持切片并行性。OpenMP 支持共享内存多处理器上的循环级并行(for construct directive)和段并行(section construct directive)。循环级并行只能处理规则的循环，而段并行只能处理连续的程序片断。这里我们为 OpenMP 增加一个新的构造制导(construct directive)以支持用切片作为任务划分的方式。我们的目标是程序员不需要重写程序，直接在串行程序上添加 OpenMP 制导就可以表达程序中的切片并行，同时为编译器并行化提供直接的并行语义，而不需要编译器做额外的并行化分析。

我们设计了 slice 制导，如图 1(a)中的两个规约计算的切片并行化可以用图 5 中的形式表达。

```
#pragma omp parallel slice (s, t)
    for (i = 0; i < N; i++) {
      s = s + a[i];
      t = t + b[i];
    }
```

图 5　切片制导

含义是该程序片断被并行执行(parallel construct)，每个线程要执行的代码由 slice 制导来分配。它根据切片标准变量 s 和 t 将程序划分为两个切片，每个线程执行一个切片。

slice 制导的形式化语法描述如下。

语法：

#pragma omp slice(list)

　　structured-block

list 是一个变量列表，与 OpenMP 标准中其他变量 list 的语法结构相同，即用逗号隔开的变量名字，变量可以是标量也可以是数组变量和结构体变量。在 slice 制导中，该变量列表指定一组切片标准，每个变量指定的程序切片由一个线程执行。slice 制导与 for，do 等制导类似，作用于其后的结构化代码块上。同 for，do 类似，slice 制导需要放在 parallel 制导内，可以合写为 parallel slice 形式。slice 制导表达如下语义：

1)slice 作用在其后的结构化代码块 structured-block 上，指定了切片并行性的来源，为编译器界定了切片的范围，也界定了并行化的范围，或者说指定了并行化区域。

2)list 指定了若干切片标准，为编译器提供了切片的依据(切片标准)；同时指定了线程划分的策略和线程的个数。

3)OpenMP 数据制导，例如 private(list)可以作用于 slice 制导，含义不变，用于表述切片线程之间的数据访问方式。例如 private(list)表示 list 中的变量在每个切片线程内有一个私有拷贝。

4)程序员不需要保证指定的切片覆盖整个结构化代码块 structured-block，而由编译器保证未在指定切片内的代码构成额外的一个线程。程序员指定的切片标准如果恰好覆盖原程序，就可以更清晰地表达并行性，避免发生错误。

基于上面的语义，slice 确定地表达了程序中的切片并行性，即基于切片的线程划分策略；也为编译器实现这样的切片并行性提供了足够的支持，这样编译器只需要进行程序切片和线程代码生成即可。

3.2　基于切片分析的自动程序划分

上节介绍了如何描述切片并行性，但完全由程序员发掘程序中的切片并行性比较繁琐，为此，我们开发了一个切片并行性分析工具，它能够分析两个切片是否可以并行执行，也能够从函数内找到可以并行的切片。我们以图 2 中的代码和图 3 中的 PDG 为例，说明与切片相关的几个概念和我们的切片并行化算法。

备选切片：备选切片是我们分析的起点。我们的工具首先分析各切片的切片标准节点，含有数据计算的作为备选切片，而控制节点的切片不作考虑，例如图 3 中节点{3,4,5,7,8,9,11,12,13}的切片可以作为备选切片，而节点{1,2,6,9}是控制节点，因此{1,2,6,9}的切片不做考虑。进一步含有副作用的切片，例如写非局部变量的切片是我们更关心的，而写局部变量的切片不作为备选切片。

切片之间的关系可以概括为3种：完全无关、完全包含、部分重叠。

完全无关：这样的切片显然可以并行执行，但一般说来这样的关系较少。

完全包含：完全包含关系也可以称为父子切片关系。父子切片之间不存在并行性，但可以作为两个线程执行，这样的线程之间存在顺序依赖关系；考虑尽可能获得大粒度的计算。我们需要一个切片最大化的过程，显然所有切片都是整个程序的子切片，如果完全最大化，就只剩下一个切片，这对我们的切片分析没有任何意义，因此我们采用一个启发式的策略将备选切片最大化。我们的策略是：如果切片 S1 是 S 的子切片，且是唯一的子切片，那么忽略 S1，只考虑切片 S；否则 S1 与 S 作为两个切片单独考虑。如图6所示，节点6的切片完全包含节点3的切片，因此节点3的切片将不作为并行化的备选切片。切片最大，可以有效降低分析复杂度。

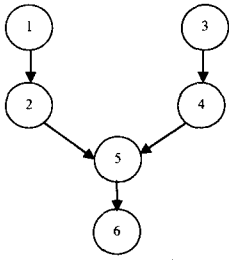

图6　切片最大化示例

图6中，slice(2)与 slice(4)完全包含在 slice(5)内，但 slice(2)与 slice(4)显然都不是 slice(5)的唯一子切片。我们将 slice(2)，slice(4)和 slice(5)作为3个切片单独考虑；进一步分析，slice(5)是 slice(6)的唯一子切片，利用最大化原则，我们将忽略 slice(5)，只考虑 slice(6)。

部分重叠：两个切片含有部分相同的 PDG 节点。部分重叠的切片是最常见的，我们或者利用冗余计算或者截断切片，使得切片能够并行执行；例如图3中的节点3和节点4的切片有部分重叠，即节点1和节点2。考虑到节点1和节点2是控制语句节点(DO 语句)，它们可以被复制，冗余执行。如果两个切片的重叠部分是不可冗余执行的(例如重叠部分含有副作用，写同一全局变量)，我们将截断切片，以重叠部分作为新的线程的备选切片。

那么，哪些代码可以冗余执行呢？这主要有两个考虑：(1)必须是结构化代码片段，即单入单出的代码区域；(2)只能写局部变量的代码片段，且不会因为数据并行化引入数据竞争(可私有化的)。最常见的这种代码片段就是循环控制语句、计算初始化语句等。

图7是切片并行性分析算法。以图3为例演示我们的算法。第1步，分析节点的副作用。节点{1,2,6,10}由4个 DO 语句构成，不含有任何副作用；而其他节点{3,4,5,7,8,9,11,12,13}都是数组赋值语句，写非局部数组变量，因此含有副作用，是我们关心的节点。

第2步通过切片最大化，进一步削减备选切片集合。从 PDG 上不难看出，slice(3)被包含在 slice(7)中，而 slice(7)又被包含在 slice(11)中，因此在我们的算法中 slice(3)和 slice(7)从备选切片集合中被删除，而 slice(11)则保留其中。类似考虑其他切片，最后备选切片集合为{11,12,13}。

第3步将备选切片集合{11,12,13}划分为线程组。例如 slice(11)与 slice(12)的重叠部分为节点集合{1,2,6,10}，全部是可冗余执行的控制语句集合。它们可以任意复制多份，不阻碍 slice(11)与 slice(12)的并行执行，因此 slice(11)与 slice(12)属于同一线程组。类似地，我们可以得到一个线程组{11,12,13}，程序因此可以划分为3个可以并行执行的线程。

```
切片并行性分析算法：
输入：PDG 及所有 PDG 节点的切片
输出：若干切片组，同组的切片可以并行执行
算法：
    第一步：计算备选切片 // 利用副作用分析初始化备选集合
        分析副作用，如果节点 n 的语句含有副作用，那么 slice(n)作为备选切片
    第二步：切片最大化 // 进一步削减备选切片集合
        如果存在边 e：m → n，且 slice (m) 和 slice (n)都是备选切片，且节点 m 只有一个后继节点有副作用(即节点 n)，节点 m 只有一个前驱节点(即节点 m)，那么 slice(m)将不再考虑，只考虑 slice (n)。
    第三步：给定 k 个备选切片 S = {s1,s2,...sk}，我们根据切片之间的关系将 S 划分为线程组，
        如果 si,sj 两个切片可并行执行(完全无关)，那么 si 与 sj 属于同一个线程组；
        如果 si,sj 部分重叠
            如果重叠部分无副作用，那么 si 与 sj 属于同一线程组，重叠部分冗余执行。如果重叠部分有副作用，那么将重叠部分作为新的备选切片，单独并且先于 si 与 sj 执行，而 si 与 sj 仍可以放在同一线程组并行执行。
        否则，si 与 sj 属于不同的线程组
```

图7　切片并行性分析算法

3.3　基于切片的并行线程生成

我们在 PathScale EKOPath 2.4 编译器中实现了一个切片工具 OpenMP Slicer，它能够根据 slice 制导或者切片分析工具给出切片组，将程序划分为若干线程，利用 PathSacle EKOPath 2.4 中原有的 OpenMP 的实现策略和运行时支持实现切片并行化。

本工具从 slice 制导的变量列表中获得切片标准，在 slice 制导控制的代码范围内进行切片，每个切片绑定一个线程。

程序员指定的切片标准有时不能完全覆盖串行程序，我们将没有被覆盖的程序作为另外一个线程，单独执行。首先确定哪些程序语句没有被程序员指定的切片覆盖，之后同样利用切片技术，将这些语句作为切片标准进行切片，从而得到一个语义完整的新线程。这个新线程的大小取决于用户制导的准确性。我们的试验中用户都可以准确地描述覆盖整个代码段的切片。

例如图5中的切片制导可以改写为 # pragma omp slice (s)，那么我们的实现会将变量 s 的切片作为一个线程，其他代码作为一个线程，即 t 的切片。这样可以减轻程序员的负担，但我们还是认为程序员完整地表达并行切片语义更为重要。

```
#pragma omp parallel sections
{
#pragma omp section slice (s)
    for (i = 0; i < N; i++) {
      s = s + a[i];
      t = t + b[i];
    }
#pragma omp section slice (t)
    for (i = 0; i < N; i++) {
      s = s + a[i];
      t = t + b[i];
    }
}
```

```
#pragma omp parallel sections
{
#pragma omp section //slice(s)
    for (i = 0; i < N; i++) {
      s = s + a[i];

    }
#pragma omp section //slice (t)
    for (i = 0; i < N; i++) {

      t = t + b[i];
    }
}
```

图8　切片线程化过程

图 8 中，先将 slice 制导变换为 sections 制导，之后在每个 section 内切片。切片之后的程序满足 slice 制导语义，也满足 OpenMP 标准指定，因此可以用标准的 OpenMP 编译器实现代码生成线程。

在实现中，为了与标准的 OpenMP 实现更好地结合在一起，首先将 slice 制导转换为 sections 制导。转换的方法是：如果有 n 个 slice 标准，则将 slice 制导控制的 structured-block 复制 n 份，作为 sections 控制的 structured-block，而每份作为 section 制导的控制代码区域。图 8 演示了这个变换过程，它是图 5 中含有 slice 制导的代码进行线程化的过程。

本文方法的主要优点在于和通用的 OpenMP 实现结合得很好，同时我们的方法有利于将含有 slice 制导的 OpenMP 程序源变换为对应的标准 OpenMP 程序。

4 实验结果与分析

我们选取了 SPEC CPU2000 中的 3 个程序进行实验，它们分别是 swim，applu，equake。实验是 HP ProLiant DL385 平台，CPU 是 AMD Opteron 285 双核(core)，运行 RHEL4AS 操作系统。原始串行测试用-O3 选项，并行测试用-O3 选项同时打开切片并行选项。表 1 列出了测试数据。

表 1 切片并行性测试数据

程序	串行(s)	并行(s)	性能提升(%)
swim	229.90	179.94	27.77%
applu	149.41	144.34	3.51%
equake	145.99	125.61	16.22%

首先利用切片分析工具分析 swim 和 applu，搜索能够获得切片并行的代码。分析结果中有些是不希望并行的，例如有些切片的并行粒度很小，因此从中选取合适粒度的切片用 slice 制导来并行化。循环与数组读写是判断计算粒度的主要依据，如果程序中没有循环和数组读写，或者循环次数是小常数，将不再分析这样的代码片段。在手工分析中，可以使用轮廓(profiling)信息寻找热代码，进行切片并行性分析。

在 swim 中选取 SHALOW(主函数)、CALC2 和 CALC3 3 个函数并行化。每个并行化代码片段都是指定 3 个切片变量，因此串行代码被分配到 3 个线程上执行。并行执行有 27.77%的性能提升。

在 applu 中选取 jacld 和 jacu 两个函数并行化。jacld 中并行化代码片段指定 4 个切片变量，jacu 中的代码片段指定 3 个切片变量，因此最多需要 4 个线程执行 applu。并行版本有 3.51%的性能提升。

在 equake 中，并行化核心热函数 smvp 指定 3 个切片变量，因此最多需要 3 个线程执行 equake。并行版本获得 16.22%的性能提升。

结束语 我们发现程序中含有一定数量的无关计算，这些计算可以并行执行并且能够获得一定的加速比。尤其是在程序的局部经常会有一些无关的计算被放在一起。本文提出的切片并行能够发掘程序中的无关计算带来的并行性，可以作为对数据并行的有益补充。基于 OpenMP 扩展的编程模型可以有效地表达这样的并行性，为程序员编程模型提供了方便。本文的切片分析可以帮助程序员更快捷、准确地寻找程序中的切片并行性，避免手工分析速度慢、易出错的问题。

目前的实验例子是手工写的，所以实验的充分性受到局限。将来的工作将考虑编译器全自动化地识别程序中的切片并行性。全自动的切片并行需要编译器自动识别并行区域、切片标准，并且进行切片，控制切片的大小和相互关系，评价切片并行的收益。这些问题对编译器都是极大的挑战，但问题的解决也将为切片并行带来更为广阔的应用前景。

参 考 文 献

[1] Weiser M. Program Slicing [J]. IEEE Transaction on Software Engineering, 1984, 10(4): 352-357

[2] Weiser M. Reconstructing Sequential Behavior from Parallel Behavior Projections[J]. Information Processing Letters, 1983, 17(3): 129-135

[3] Ferrante J, Ottenstein K J, Warren J D. The Program Dependence Graph and Its Use in Optimization[J]. TOPLAS, 1987, 9(3): 319-349

[4] Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs[J]. ACM Transactions on Programming Languages and Systems, 1990, 12(1): 35-46

[5] Tip F. A survey of program slicing techniques[J]. J. Programming Languages, 1995, 3(3): 121-189

[6] Asanovic K, et al. The Landscape of Parallel Computing Research: A View from Berkeley[R]. UCB/EECS-2006-183. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[7] OpenMP Specifications [EB/OL]. http://www.openmp.org/drupal/

[8] Binkley D, Gold N, Harman M. An Empirical Study of Static Program Slice Size[J]. ACM Transactions on Software Engineering and Methodology(TOSEM), 2007, 16(2)

[9] Saleem M, Hussain R, Ismail V, et al. Cost effective software engineering using program slicing techniques[C]// Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human. Seoul, Korea, 2009: 768-772

(上接第 87 页)

[5] 朱江，徐斌阳，李少谦. 一种基于马尔可夫决策过程的认知无线电网络传输调度方案[J]. 电子与信息学报，2009，31(8)：2019-2023

[6] Watkins C J C H. Learning from Delayed Rewards [D]. Cambridge: Kings College, University of Cambridge, 1989

[7] Watkins C J C H. Q-Learning [J]. Machine Learning, 1992, 8(3): 279-292

[8] Lu Shou-feng, Liu Xi-min, Dai Shi-qiang. Incremental multistep Q-learning for adaptive traffic signal control based on delay minimization strategy[C]// Proceedings of the 7th World Congress on Intelligent Control and Automation. USA: IEEE, 2008: 2854-2858

[9] 陈圣磊，吴慧中，肖亮，等. 基于 Metropolis 准则的多步 Q 学习算法与性能仿真[J]. 系统仿真学报，2007，19(6)：1284-1287

[10] 周浦城，洪炳镕，韩学东，等. 基于多 Agent 的并行 Q-学习算法[J]. 小型微型计算机系统，2006，27(9)：1704-1707