分支嵌套循环的自动并行化研究

丁丽丽 李雁冰 张素平 王鹏翔 张庆花
(解放军信息工程大学数学工程与先进计算国家重点实验室 郑州 450002)

摘 要 GCC编译器是一种受广大研究者青睐的开源优化编译器,但它仅仅能够对完美嵌套循环进行依赖分析。为了更好地挖掘嵌套循环粗粒度的并行,深入研究了GCC5.1数据依赖分析过程,提出了一种能够处理分支嵌套循环的依赖测试方法。首先识别出分支嵌套循环,然后分析数组下标与分支嵌套循环外层索引变量的关系,最后计算出外层循环索引变量的距离向量,并通过检测距离向量判断循环是否存在依赖。实验结果表明,该方法能够正确、有效地分析出分支嵌套循环的依赖关系。

关键词 数据依赖分析,GCC,完美嵌套循环,分支嵌套循环,距离向量
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.05.003

Auto-parallelization Research Based on Branch Nested Loops

DING Li-li LI Yan-bing ZHANG Su-ping WANG Peng-xiang ZHANG Qing-hua
(State Key Laboratory of Mathematical Engineering and Advanced Computing,
PLA Information Engineering University,Zhengzhou 450002,China)

Abstract GCC compiler is an open source compiler system which has won favour among many researchers,however,it is only able to analyze the dependence of perfect nested loop. In order to efficiently explore the granularity parallelism of nested loop,we deeply analyzed the data dependence of GCC5.1 and put forward a dependence testing method of handling branch nested loop. At first,the branch nested loop is recognized. Then,the relationship between array index and index variable of outer loop is identfied. At last,the distance vector of outer loop is computed,and whether the loop has carried dependence or not is decided by examining the distance vector. The experimental results show that the proposed method can effectively recognize the dependence of branch nested loop.

Keywords Data dependence analysis,GCC,Perfect nested loop,Branch nested loops,Distance vector

1 引言

人们对计算机运算速度无止境的追求使得CPU的运算速度不断提升。尤其是超级计算机,其运算速度已达到每秒千万亿次级别,如:由中国自主研发并获得2016年TOP500冠军的"神威·太湖之光",其浮点运算速度高达每秒125.436千万亿次。但在实际应用中,超级计算机的计算速度仅能达到理论峰值速度的30%,甚至更低[1]。经计算机界众多研究者分析,之所以出现这一结果是因为并行软件的开发落后于硬件开发,不能充分利用计算资源。目前,解决这一问题的重要方法是采用并行化编译系统识别出串行程序中潜在的并行性,将其转换为并行程序。程序并行化的本质是对程序内的语句进行重排序变换,然而并不是所有的程序都能在重排序变换前后保持语义不变,所以必须在程序进行变换之前对其进行依赖关系分析。依赖关系分析技术是用来判断程序中语句之间是否存在依赖的方法,同时也是并行化编译开发程序并行性的一个基本步骤。对于并行化编译而言,依赖分析能力的强弱直接影响其挖掘并行性的能力。

由GNU开发研制的GCC(GNU Compiler Collection)编译器对串行程序中的完美嵌套循环进行依赖分析,对非完美嵌套循环则直接跳过;Stanford大学研制的自动并行化编译器SUIF[2]能够很好地识别串行程序中的完美嵌套循环,但是对于非完美嵌套循环的并行识别能力却有限;中国科学院和Delaware大学维护开发的Open64[3]编译器在识别并行循环之前通过大部分循环优化将循环转换为完美嵌套循环,由此提高并行循环的识别能力。这些系统的发展促使对依赖检测技术的研究达到了一个高潮,各种依赖检测方法也相继被提出来,如:单数组下标的依赖测试方法主要包括GCD测试法[4]和Banerjee不等式测试法[5],耦合数组下标的依赖测试方法主要包括Delta测试法[6]和Omega测试法[7]等。但是这些编译器有一个共同的缺陷,即对非完美嵌套循环的依赖分析能力较弱,有些编译器甚至只能对完美嵌套循环的依赖关系进行检测。

到稿日期:2016-03-07 返修日期:2016-08-10 本文受国家高技术研究发展计划(863计划)(2009AA01220),"核高基"重大专项(2009zx01036-001-001-2)资助。

丁丽丽(1992—),女,硕士,主要研究方向为先进编译技术,E-mail:1551523054@qq.com;李雁冰(1989—),男,博士生,主要研究方向为先进编译技术;张素平(1991—),女,硕士,主要研究方向为先进编译技术;王鹏翔(1988—),男,硕士,主要研究方向为先进编译技术;张庆花(1991—),女,硕士,主要研究方向为先进编译技术。

虽然循环分布[4]技术常被用来将分支嵌套循环分解为完美嵌套循环，以弥补大多数编译器不能直接对分支嵌套循环进行依赖分析的不足，但是该方法存在以下问题：

1)并不是所有的分支嵌套循环均能使用循环分布处理；

2)分支嵌套循环进行循环分布变换后，循环的总迭代次数和执行并行化语句的同步次数都会成倍增加；

3)可能会破坏数据的局部性，降低 cache 命中率。

作为 GCC 开发者之一的 Daniel Berlin 在文献[8]中提出了基于 Tree-SSA 的适用于完美嵌套循环的循环变换和数据依赖分析框架，该框架极大地提高了程序自动并行化和向量化的性能。文献[9]实现了分裂递链的数据依赖分析方法和针对 Fortran 语言的非线性数组下标依赖分析算法，从而加强了 GCC 数据依赖分析的能力。文献[10]提出了一种针对含有跨迭代数据依赖的循环进行 OpenMP 自动并行化的方法。文献[11]设计了一个用于在编译时检测并行程序数据依赖特征的插件——Cit，该插件能够对被检测为 May Confilct 的依赖做进一步的分析，并且能够将分析出的依赖信息以可视化的形式反馈给程序员。以上研究均在一定程度上提升了 GCC 的并行化性能，但没有解决 GCC 编译器不能对分支嵌套循环进行依赖分析的问题。针对这一问题，本文基于 GCC5.1 提出了一种处理分支嵌套循环的依赖测试方法。

本文首先根据嵌套循环的结构特点，将嵌套循环分为完美嵌套循环、分支嵌套循环和复杂嵌套循环；然后深入分析了 GCC5.1 中的依赖分析模块；最后对 GCC 编译器中的依赖关系分析进行补充，实现了针对分支嵌套循环的依赖测试方法。实验结果表明，该方法使得程序的并行效率提升了 14%以上。

2　相关概念

循环是并行化编译系统进行并行性分析的主要对象，也是面向共享存储结构 OpenMP 并行化的基本单位。在并行化编译系统中，当循环被判定为不存在依赖或存在循环无关依赖时，循环可并行执行。本节首先给出了分支嵌套循环的定义，然后介绍了数据依赖的基本定义。

2.1　嵌套循环的分类方法

应用程序中嵌套循环结构多种多样，其形式影响着循环是否可以并行执行。传统的嵌套循环分类方法把循环分为完美嵌套循环和非完美嵌套循环。大多数并行编译系统只能有效地发掘出完美嵌套循环中潜在的并行性。但是在 NPB 和 SPEC 等科学计算程序中存在着较多的分支嵌套循环，若能高效地分析出这些循环的并行性，可能会进一步提高程序的性能。并行化是一种粗粒度的优化策略，并行代码的初始化和同步操作的开销也阻碍了程序性能的提升，因此必须尽可能地挖掘循环外层的并行性，才能获得更好的性能提升。根据嵌套循环的结构将嵌套循环分为完美嵌套循环、分支嵌套循环和复杂嵌套循环。为了更好地理解分支嵌套循环的结构，下面给出了嵌套循环分类的定义。

定义 1　如果一个 N 层嵌套 for 循环只有最内层的循环体是由一条或多条非循环赋值语句构成的，并且这些语句满足无过程调用和过程返回语句，其他所有循环的循环体只由一条 for 循环语句构成，那么该嵌套循环的每层循环都是完美嵌套循环(Perfect Nested Loop，PNL)。

定义 2　如果一个 N 层嵌套 for 循环的第 $k(0<k\leqslant N)$ 层循环存在至少两个完美嵌套循环，那么该循环为分支嵌套循环(Branch Nested Loop，BNL)。

定义 3　如果一个 N 层嵌套 for 循环即不是完美嵌套循环也不是分支嵌套循环，那么该循环为复杂嵌套循环(Complex Nested Loop，CNL)。

2.2　数据依赖的定义

依赖的相关理论和技术是保证程序在变换前后保持语义等价性的基础。依赖关系是确定指令执行顺序的一种偏序关系，可分为数据依赖关系和控制依赖关系。数据依赖是由存储访问引起的，控制依赖是由程序中的控制流引起的。大多数情况下循环体中的控制语句可以通过 if-转换(IF-conversion)转化为数据依赖进行处理，而且在现实应用中，尤其是在科学计算程序中，循环体内极少包含控制流语句，因此数据依赖是本文研究的重点。本文所叙述的依赖若无作特殊说明均为数据依赖。下面介绍依赖的相关定义及性质。

定义 4　从语 S1 到语句 S2 存在数据依赖(语句 S2 依赖于语句 S1)，当且仅当：

1)两个语句访问相同的存储单元，并且其中至少有一个语句存入此单元；

2)存在一条从 S1 到 S2 的可能的运行时执行路径。

定义 5　假设从 n 层循环嵌套的迭代 i 中的语句 S1 到迭代 j 中的语句 S2 有依赖，则依赖距离向量 $d(i,j)$ 定义为长度为 n 的向量，使得 $d(i,j)_k=j_k-i_k(1\leqslant k\leqslant n)$。

定义 6　假设从 n 层循环嵌套的迭代 i 中的语句 S1 到迭代 j 中的语句 S2 有依赖，那么依赖方向向量 $D(i,j)$ 定义为长度为 n 的向量，使得

$$D(i,j)_k=\begin{cases}\text{“<”}, & \text{如果 } d(i,j)_k>0\\ \text{“=”}, & \text{如果 } d(i,j)_k=0,1\leqslant k\leqslant n\\ \text{“>”}, & \text{如果 } d(i,j)_k<0\end{cases}$$

定义 7　循环携带依赖的层是此依赖的 $D(i,j)$ 的最左非“=”的索引。

语句 S2 的数据依赖于 S1，要求存在一条可能的执行路径使 S1 和 S2 两者引用相同的存储单元，同时 S1 引用的执行发生在 S2 引用的执行之前。可能满足上述条件的方式有两种：

1)在相同循环迭代中 S1 和 S2 引用共同单元，并且 S1 的执行在 S2 之前；

2)在循环的一次迭代中 S1 引用存储单元而在后面的迭代中 S2 引用相同单元。

第一种情况属于循环无关依赖，第二种情况属于循环携带依赖，前者依赖取决于循环内代码的位置，而后者由循环迭代产生。

定义 8　语句 S2 对语句 S1 有一个循环无关依赖，当且仅当存在两个迭代向量 i 和 j，使得：1)S1 在迭代 i 中引用存储单元 M，S2 在迭代 j 中引用 M，且 $i=j$；2)迭代中有一条从 S1 到 S2 的控制流路径。

定义 9　语句 S2 对语句 S1 有一个循环携带依赖，当且仅当 S1 在迭代 i 中引用单元 M，而 S2 在迭代 j 中引用 M，且

$d(i,j)>0$(即 $D(i,j)$包含一个“<”作为它的最左非“=”分量)。

3 GCC 数据依赖分析

GCC 编译器是当前应用较为广泛的开源编译器,它能够自动挖掘出应用程序中完美嵌套循环潜在的并行性,生成面向共享存储结构的 OpenMP 并行代码。依赖分析是判断程序能否并行执行的基础,它不仅需要考虑循环体内数组引用的下标形式,还需考虑循环的结构。本节首先介绍了 GCC 的编译框架,描述了循环变换和数据依赖分析遍的组织流程,然后介绍了 GCC 中数据依赖分析的详细过程,并重点分析了逐下标计算冲突迭代的方法。

3.1 编译框架

GCC 编译器可接受 C/C++,Fortran 和 JAVA 等多种源语言,经过 GCC 中的各源语言对应的语言前端将它们解析为一种通用的抽象语法树——GENERIC 树。然后 GENERIC 通过 Gimplify 可以将代码转换为 GIMPLE 三地址中间代码。GIMPLE 再经由 GIMPLE-SSA 分析器处理转换为树节点表示的静态单赋值(Static Single Assignment,SSA)形式 Tree-SSA。数据依赖分析就是在 Tree-SSA 上进行的,而且 GCC 在该中间语言层面还会执行一系列优化遍,进行各项功能的优化,此过程即为 GCC 的中端(Middle-end)。源代码经过 Tree-SSA 的各种优化后会重新转换为 GIMPLE 中间代码;然后转为基于底层的 RTL 语言,此时,GCC 会在 RTL 语言上执行一系列的优化遍,进行寄存器分配、代码生成等工作,这一系列过程称为后端(Back-end)。图 1 所示为 GCC 编译框架。

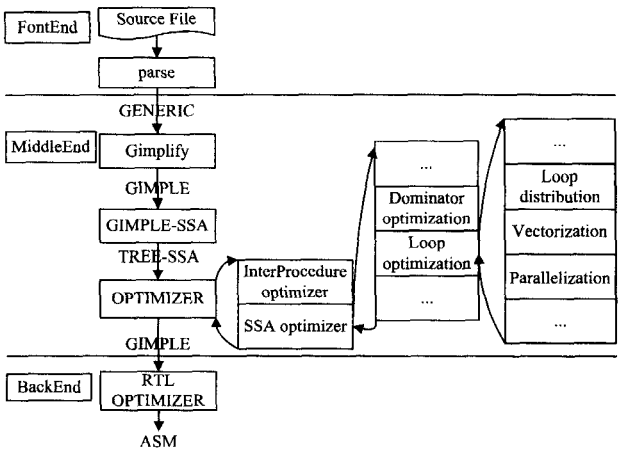

图 1 GCC 编译框架

3.1.1 循环变换和依赖分析的遍

GCC 中基于 Tree-SSA 和 RTL 中间表示的优化遍的执行均由源码文件 Passes.c 中的 execute_one_pass 函数调度。如图 1 所示,循环变换、向量化和并行化等循环优化操作均基于 Tree-SSA 中间表示。GCC 中的循环变换包括(依照遍的调用顺序)循环展开、if 外提、循环分布、循环分段以及循环交换等。循环变换的主要目的是提高指令局部性和数据局部性,并改变循环依赖关系以发掘更多的并行化,所以一般情况下循环变换遍先于并行化遍执行。但是为了使程序获得更多的优化机会,某些循环变换遍将可能被多次调用,比如执行循环并行化遍后,可能会再次调用循环展开遍。循环变换是对程序进行并行化和向量化,也是必不可少的关键程序变换技术,而数据依赖分析的结果是判断循环能否执行循环变换、向量化及并行化等循环优化的基础,所以调用循环变换、并行化或向量化遍前均需调用数据依赖遍对循环进行依赖分析。下面详细介绍 GCC 中数据依赖分析的过程。

3.2 依赖分析过程

GCC 自动并行化模块中的数据依赖分析器可以分析应用程序中完美嵌套循环的依赖状况并检测其能否并行化处理。该分析器对循环采取从外层到内层的分析策略,且只能对下标是循环索引变量的线性表达式的数组进行依赖分析。

3.2.1 数组引用信息

数组的引用属性(是否为读)、别名信息及下标信息等均是从 Tree-SSA 中间代码中获取的,并使用递归链的形式表示数组下标的访问函数。递归链是一种表示定义在匀格函数的方法。阶乘、有理函数和多项表达式都可以使用递归链表示。使用递归链的方式描述数组下标不仅可以在循环迭代空间内快速求出任意一次迭代的数值,而且可以通过递归链的表示得出该循环归纳变量所在的循环表示形式。例如,图 2 中数组 $a[i][j]$中归纳变量 i 和 j 的递归链表示分别为{0,+,1}_1 和{0,+,1}_2。对于{0,+,1}_1,0 表示循环初始值;+表示循环操作数;1 表示循环跨副;_1 表示循环归纳变量所在的循环层。

```
for (i=0;i<U1;i++)
  for(j=0;j<U2;j++)
  {
    a[i][j]=b[i][j]+D;
    c[i][j]=a[i][j]+8;
  }
```

图 2 完美嵌套循环

3.2.2 下标的分类

术语下标是指一对数组引用中下标的位置之一。因为依赖测试总是考虑一对数组引用,所以可以使用术语下标来指代一对下标的位置。例如,图 2 中数组 a 包含两个下标,第一个下标为$\langle i,i\rangle$,第二个下标为$\langle i,j\rangle$;在图 3 中由数组 a 形成的第一个下标为$\langle i,i\rangle$,第二个下标为$\langle j,k\rangle$。在一个下标中出现的循环索引越多,其依赖测试就会变得越复杂,所以为了更为简单高效地测试这些下标,需将其进行分类。

```
for (i=L1;i<U1;i++)
{
  for(j=L2;j<U2;j++)
    a[i][j]=b[i][j]+D;
  for(k=L3;k<U3;k++)
    c[i][j]=a[i][k]+8;
}
```

图 3 分支嵌套循环

不包含循环索引变量的下标为 ZIV,只包含一个循环索引变量的下标为 SIV,任何包含多于一个循环索引变量的下标为 MIV。例如,$\langle 5,6\rangle$为 ZIV,$\langle i,i\rangle$为 SIV,$\langle j,k\rangle$为 MIV。

测试多维数组时,需要用可分性来描述下标之间是否会相互产生影响,所以还需根据下标之间的关系对其进行分类。

当下标中的循环索引变量不在其他下标中出现时，称该下标是可分的，即单数组下标。如果在不同的下标中出现了相同的循环索引变量，则它们是耦合的，称为耦合数组下标。

3.2.3　采用逐下标方式计算冲突迭代

GCC 中的数据依赖分析器以逐下标的方式使用传统解丢番图方程的方法求出相同数组每一个下标的冲突迭代，并用递归链进行表示。其基本思想是利用冲突迭代计算循环的方向向量矩阵和距离向量矩阵，然后对其进行矩阵变换，通过判断该变换是否合法来决定循环是否能够并行化。

首先使用 GCD 测试法简单快速地判断该方程是否存在解，若无解则证明不存在依赖，并立即终止该算法；否则对其使用 SIV 测试法以及 MIV 测试法以及 Banerjee 不等式测试法等计算出该下标的冲突迭代，并用递归链对其进行表示。计算出每一维的下标的冲突迭代函数后，若存在耦合数组下标，则对其使用耦合技术在数组访问的每一维之间传递约束信息，同样用递归链对其进行表示。采用逐下标的方式计算冲突迭代时只能在数组访问的每一维之间传递循环依赖距离。根据冲突迭代计算距离向量矩阵和方向向量矩阵。对距离向量和方向向量进行矩阵变换，若变换合法则可并行，若不合法则不可并行。通过矩阵变换来检测循环是否可以并行化的判别方式只对完美嵌套循环才有效，所以 GCC 的数据依赖分析只能对完美嵌套循环进行依赖分析。

从以上分析可知，GCC 的数据依赖分析器能够快速有效地对完美嵌套循环进行依赖分析，但是不能挖掘出分支嵌套循环潜在的并行性。针对这种情况，本文在 GCC 的依赖分析的基础上进行了改进，提出了一种相关性依赖测试方法，该方法能够更加简且洁高效地挖掘出分支嵌套循环中的并行性，极大地提高了程序性能。下面将具体介绍这种相关性方法的算法实现并举例说明如何使用该方法判断分支嵌套循环的依赖关系。

4　面向分支嵌套循环的依赖检测技术

基于分支嵌套循环的依赖检测方法不再采用 GCC 中以逐下标的方式检测数组下标之间的关系，而是通过判断数组下标与当前循环索引的相关性进行处理。该方法比以逐下标形式计算数组下标之间的冲突迭代的方法更为快捷有效，且该方法只需检测循环最外层的距离向量是否为零向量即可判断该循环能否并行，而不需检测整个循环的距离矩阵和方向矩阵的变换的合法性，因此更加简单易懂。

4.1　算法的基本思想

并行化的基本单位是循环，只需要考虑循环携带依赖，即当循环无依赖或存在循环无关依赖时均可并行执行。也可以根据循环当前层的距离向量的值来判断循环依赖的类型，若距离向量为零向量则为循环无关依赖，否则为循环携带依赖。正是根据这个原理，该方法只需计算出被检测的分支嵌套循环当前层的距离向量即可。而又因为只需计算出当前循环的距离向量，所以不需要以逐下标的方式去计算数组下标含有的所有循环索引的冲突迭代函数，只需观察数组下标所含的循环索引变量与当前层的关系，再决定是否去计算冲突迭代。

4.2　下标相关性分析

下标的相关性分析即检测下标中是否包含外层循环的索引变量。当检测的下标不包含外层循环索引变量，则说明该数组下标索引不会因外层循环迭代的改变而发生变化，此时无论下标中是否还含有别的循环索引变量，都不需要再对其计算求解冲突迭代，可直接将该下标对外层的依赖距离置为零。当检测的下标包含且仅包含外层的循环索引变量时，只需计算其外层的依赖距离。若不相同的下标对外层携带的依赖距离不同，则不存在依赖，终止算法。当检测的下标不仅包含外层循环索引变量还包含其他的索引变量时，因为情况较为复杂，本文所提出的算法还不能对其进行准确的处理，所以采取保守分析方式，最后检测外层循环的依赖距离向量是否为零向量即可。

如图 2 所示的分支嵌套循环，数组 a 的引用形成了两对下标$\langle i,i\rangle$和$\langle j,k\rangle$，首先分析$\langle i,i\rangle$，发现其只包含外层循环索引变量，计算其外层的依赖距离为 0。然后分析下标$\langle j,k\rangle$，得知其并不包含外层循环索引变量，下标中的值并不会因外层循环索引值的变化而变化，所以该下标对外层循环的依赖距离也可设为零。故该分支嵌套循环外层的距离向量为(0，0)，不存在循环携带依赖，该外层循环可以并行执行。下面介绍具体的算法。

4.3　算法的具体步骤

本文提出的算法主要针对分支嵌套循环，所以本节首先给出判断循环为分支嵌套循环的算法，然后介绍检测分支嵌套循环是否可并行化的依赖测试方法。将该算法添加到了 GCC 源码文件 tree-data-ref.c 中。

(1)判断输入循环是否为分支嵌套循环

因为 GCC 中已存在对复杂嵌套循环的过滤算法，所以判断嵌套循环是否为分支嵌套循环的算法不需要再考虑过程跳转和调用等情况，只需遍历循环，检测在循环的同一层是否存在分支即可。将该算法添加到了函数 compute_data_dependences_for_loop中，具体步骤如图 4 所示。

```
输入：循环 loop
输出：若是分支嵌套循环则返回 true，否则返回 false，并把该循环入
      栈放入 loop_nest 中
Procedure find_loop_BNL(loop, loop_nest)
{
1. loop_nest->safe_push(loop); //循环入栈
2.   if(loop->inner) //循环是否是嵌套循环
3.     loop=loop->inner
4.     while(loop)
5.       if(loop->next) //循环是否有兄弟节点
6.         return true;
7.       loop=loop->inner
8.     end while
9.   end if
10. return false;
}
```

图 4　判断循环为分支嵌套循环的算法

(2)处理数组引用信息

算法的主要流程：首先分析数组别名问题，不互为别名的数组不存在依赖。然后分析相同数组下标的表达式，如果存在包含循环索引的非线性表达式的下标，那么就判断为有依赖，并停止依赖检测；接下来分析数组下标与分支嵌套循环外层索引变量的关系，计算其在外层的依赖距离；最后分析分支

嵌套循环外层循环的距离向量，若为零向量，则该循环可并行执行，否则判断其存在循环携带依赖。具体步骤如下。

输入：记录数组引用信息的向量 Datarefs

输出：无依赖返回 true，否则 false

```
Procedure computer_BNL_dependence(Datarefs,loop)
{
1. while Datarefs do
2.   if 数组 a&&b 为读操作 then
3.     continue;
4.   end if
5.   if 数组 a,b 不互为别名 then
6.     continue;
7.   end if
8.   if 数组 a,b 的数组下标为非线性表达式 then
9.     return false ; //给出保守结果存在依赖
10.  end if
11.  while 数组 a,b 形成的下标对 do
12.    //对应维数组 a,b 的下标分别为 chrec_a,chrec_b
13.    //下标为 ZIV
14.    if chrec_a 和 chrec_b 为 ZIV then
15.        td <— chrec_a—chrec_b ;
16.        if td !=0 then
17.            d <— 0 ; //该数组对外层的依赖距离为 0,存入向
               量 d 中
18.            回到步骤 1 ;
19.        end if
20.        tempd <—0 ;//依赖距离为 0,并存入向量 tempd 中
21.        end if
22.        //下标为 SIV
23.        if chrec_a 和 chrec_b 为 SIV then
24.          if chrec_a 或 chrec_b 为外层索引变量表达式 then
25.                tempd <— chrec_a—chrec_b ;//计算依赖距
                   离,并存入向量 tempd 中
26.          else if chrec_a 和 chrec_b 均不是外层循环索引表达式
27.                tempd <— 0 ;//依赖距离为 0,并存入向量
                   tempd 中
28.        end if
29.    end if
30.    //下标为 MIV
31.        if chrec_a 和 chrec_b 为 MIV
32.          if chrec_a 或 chrec_b 有外层循环索引变量存在
33.                return false ; //给出保守结果存在依赖
34.        else
35.            tempd <— 0 ;
36.        end while
37.        if tempd 中的数值均为 0,或存在两个不相同且不为 0 的值
38.            d<— 0 ;//该数组对外层循环的依赖距离为 0,并
               存入 d 中
39.        else return false;//存在外层循环携带依赖
40.        end if
41. end while
42. if d 为零向量 //d 所有数组对外层循环索引变量依赖距离构成的
    向量
43.    return true;
}
```

该方法的优点如下：

1)更快捷有效。不再以逐下标的方式计算所有下标中的冲突迭代，而先判断下标与外层循环索引变量的相关性，只计算与外层循环索引变量有关的下标的冲突迭代。

2)不仅时间复杂度低而且更为通俗易懂。不再计算整个循环的方向向量和距离向量，然后再对其进行矩阵变换，通过判断其合法性来判断循环是否可并行。只需计算出外层循环的距离向量，判断其是否为零向量即可。

但该方法在处理分支嵌套循环时仍然有一些限制条件：

1)数组引用的下标必须是循环索引变量的线性表达式；

2)当该数组下标包含外层循环索引变量时，不能再含有其他循环索引变量。

本文提出的方法对于不满足以上条件的循环采取保守测试态度，判断循环不可并行。

对于处理循环数组引用依赖信息的算法，其时间开销主要集中在所有数组下标对的比较上，因此不难得出算法的时间复杂度为 $O(n(n-2)/2)$，其中 $0\leqslant i,j<n$，且 $i<j$，n 为数组的个数，$n(n-2)/2$ 为数组比较次数，a_{ij} 为存储在 datarefs 中下标为 i 和下标为 j 的数组组成的下标对数目。

5 实例分析与测试

5.1 实验平台

本文基于 GCC5.1 开源编译器实现了 GCC 对分支嵌套循环的依赖分析，编译系统为 Linux 操作系统，版本为 Red Hat Enterprise Linux Server release 5.5 (Tikanga)。实验平台采用 intel 至强处理器 5500，内存为 4GB，主频为1600.00Hz，L1 数据 cache 为 32kB，L2 cache 为 256kB。

5.2 程序分析及测试

为验证本文算法的正确性和有效性，使用 NPB3.3.1 标准测试集进行实验。NPB 是 1991 年美国 NAS 项目所开发的并行基准测试程序，其目的就是用来比较各种并行机的性能，NPB3.3.1 版本共有 10 个程序。将本文提出的依赖测试方法分别用于分析这 10 个程序中的嵌套循环，结果如表 1 所列。然后采用 MG 程序来具体分析算法的实现及其加速比，结果如图 5 所示。MG(MultiGrid)用 4 个 V 循环多重网格算法来求解三维波松方程的离散周期近似解，其核心程序为分支嵌套循环，很明显地展示了本文算法的功能。

表 1 本文依赖测试方法分析 NPB 程序的结果

程序	嵌套循环总数(多层)	BNL	本文能处理的 BNL 总数	不能处理的 BNL 总数
MG	18	6	6	0
CG	10	3	2	1
SP	38	5	5	0
BT	35	8	5	3
LU	44	9	7	2
UA	68	1	1	0
EP	1	0	0	0
FT	8	0	0	0
DC	11	0	0	0
IS	0	0	0	0

由表 1 可知，在 NPB3.3.1 标准测试集中，总共有 32 个分支嵌套循环，本算法能够分析出其中的依赖关系，使其并行

的分支嵌套循环有 26 个，不能处理的分支嵌套循环有 6 个。不能处理的原因是对下标进行分析时发现所检测的下标不仅包含外层循环索引变量还包含某些内层的循环索引变量。

因为在程序 EP，FT，IS 和 DC 中并不存在分支嵌套循环，在程序 UA 中存在的一个分支嵌套循环不在其热点函数中，所以该算法对这些程序并没有加速效果。而在程序 CG，SP，BT 和 LU 中，其加速效果可达到 8%左右。因为程序 MG 的热点函数的核心循环为分支嵌套循环，所以本文所提出的算法对其加速效果最为明显，在四线程时可达到手工并行代码的 60%。下面对 MG 程序进行分析。

MG 中的热点函数为 resid 和 psinv，分别占串行执行时间的 53.35%和 27.24%。两个函数结构相似，内部均仅包含了一个三层嵌套循环，并且该三层嵌套循环是一个分支嵌套循环。循环代码分别如图 5(a)、图 5(b)所示。

```
do i3=2,n3-1
  do i2=2,n2-1
    do i1=1,n1
      u1(i1)=u(i1,i2-1,i3)+u(i1,i2+1,i3)+u(i1,i2,i3-1)+
             u(i1,i2,i3+1)
      u2(i1)=u(i1,i2-1,i3-1)+u(i1,i2+1,i3-1)+u(i1,i2-
             1,i3+1)+u(i1,i2+1,i3+1)
    enddo
    do i1=2,n1-1
      r(i1,i2,i3)=v(i1,i2,i3)-a(0)*u(i1,i2,i3)-a(2)*
                  (u2(i1)+u1(i1-1)+u1(i1+1))-a(3)*
                  (u2(i1-1)+u2(i1+1))
    enddo
  enddo
enddo
```

(a)

```
do i3=2,n3-1
  do i2=2,n2-1
    do i1=1,n1
      r1(i1)=r(i1,i2-1,i3)+r(i1,i2+1,i3)+r(i1,i2,i3-1)+
             r(i1,i2,i3+1)
      r2(i1)=r(i1,i2-1,i3-1)+r(i1,i2+1,i3-1)+r(i1,i2-
             1,i3+1)+r(i1,i2+1,i3+1)
    enddo
    do i1=2,n1-1
      u(i1,i2,i3)=u(i1,i2,i3)+c(0)*r(i1,i2,i3)+c(1)*(r(i1-
                  1,i2,i3)+r(i1+1,i2,i3)+r1(i1))+c(2)*
                  (r2(i1)+r1(i1-1)+r1(i1+1))
    enddo
  enddo
enddo
```

(b)

图 5　resid 和 psinv 中的核心循环代码

因为图 5(a)和图 5(b)中的循环结构相似，所以只分析函数 resid 中的核心循环(见图 5(a))。图 5(a)中的循环为分支嵌套循环，GCC 不能对该循环进行依赖分析，采用保守判断，只对第三层的两个并列循环进行深入的依赖检测，得出其不存在依赖，对其做并行化处理。可以看到，这种做法不仅不能充分挖掘该循环的并行性，还带来了更多的同步操作，极大地影响了程序的性能。而本文提出的算法不但可以对该循环做依赖分析，而且能够正确分析出外层循环不存在循环携带依赖。具体分析过程：数组 $u1(i1)$ 和数组 $u(i1,i2-1,i3)$ 不互为别名，它们对外层不存在循环携带依赖，即其依赖距离为零；数组 $u1(i1)$ 和数组 $u1(i1-1)$ 组成的下标为 $(i1,i1-1)$，不含外层循环索引变量的表达式，所以其外层的依赖距离也为零，该下标对外层的距离向量为零向量，即外层循环的方向向量全为“=”，故外层能够进行并行化。图 6 所示为程序 MG 使用数目不同的线程时，手工并行化代码和使用不同的并行化分析方法时的加速比。

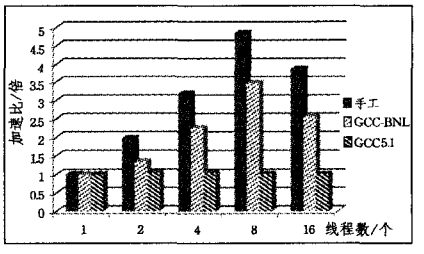

图 6　MG 程序加速比

因为 GCC5.1 只能分析分支嵌套循环中所嵌套的完美嵌套循环的依赖关系，并行化内层循环，这就造成程序在并行执行时进行了较多的同步操作。对于图 5(a)中的循环，GCC 同样只能并行化最内层，而其同步的次数至少为 $2(n2-2)(n3-3)$。由于同步的花销太大，导致其并行化几乎没有加速。

从表 1 和图 6 可以得出结论：本文基于 GCC5.1 提出的针对分支嵌套循环的依赖分析方法能够快速地分析出该类循环潜在的并行性，增加了代码并行的可能性，同时有效提升了程序执行速度。

结束语　完美嵌套循环是一种结构整齐且紧凑的嵌套循环，但是在应用程序中分支嵌套循环也经常出现，若不能对此类循环进行依赖分析，必将错失提升程序性能的机会。本文针对 GCC 不能对分支嵌套循环进行依赖分析的情况，提出了一种基于分支嵌套循环的依赖检测技术。首先介绍了 GCC 中的依赖测试流程，然后详细介绍了所提的方法。实验结果表明，本文提出的算法能够快速准确地分析出分支嵌套循环的依赖关系，并增强 GCC 的依赖分析能力。但是该算法也存在较多保守测试结果，算法性能还有待改进，这些将是我们下一步深入研究的内容。

致　谢　向对本文研究工作提供基金支持的单位和评阅本文的审稿专家表示衷心的感谢，向为本文研究工作提供基础和研究平台的前辈致敬。

参 考 文 献

[1] HAN L. Research on Consistent Optimization Techiniques of Parallel Decomposition for Distributed memory Architecture [D]. Zhengzhou: The PLA Information Engineering University, 2008. (in Chinese)
韩林. 面向分布存储结构的并行分解一致性优化技术研究[D]. 郑州：解放军信息工程大学，2008.

[2] HALL M W, AMARASINGHE S P, MURPHY B R, et al. Interprocedural parallelization analysis in SUIF[J]. ACM Transactions on Programming Languages and Systems, 2005, 27(4): 662-731.

(下转第 52 页)

结束语 本文研究了 CR-OFDM 的认知无线电资源分配问题，采用先分配子载波再分配子载波功率的分步式方式。为降低算法的复杂度，通过对已有的线性注水算法进行研究，提出了改进的线性注水算法来求解功率和干扰双重约束问题。仿真结果表明，本文提出的算法可以逼近最优功率分配算法，大于传统功率分配算法所能获得的系统容量，并且算法复杂度小，可作为 CR-OFDM 资源分配中的一种次优算法。

参 考 文 献

[1] HAYKIN S. Cognitive radio: brain-empowered wireless communications[J]. IEEE J. Sel. Areas Commun, 2005, 23(2): 201-220.

[2] WEISS T, JONDRAL F K. Spectrum pooling: an innovative strategy for the enhancement of spectrum efficiency[J]. IEEE Commun. Mag, 2004, 43(3): S8-S14.

[3] WEISS T, HILLENBRAND J, KROHN A, et al. Mutual interference in OFDM-based spectrum pooling systems[J]. Proc. IEEE Vehicular Technol. Conf. (VTC'04 Spring), 2004, 4(4): 1873-1877.

[4] WANG S, HUANG F, et al. Fast power allocation algorithm for cognitive radio networks[J]. IEEE Communications Letters, 2011, 15(8): 845-847.

[5] BANSAL G, HOSSAIN M, BHARGAVA V. Optimal and suboptimal power allocation schemes for OFDM-based cognitive radio systems[J]. IEEE Transactions on Wireless Communications, 2008, 7(11): 4710-4718.

[6] TANG L, WANG H, CHEN Q, et al. Subcarrier and power allocation for OFDM-based cognitive radio networks[C]// IEEE International Conference on Communication and Technology and Applications. IEEE, 2009: 457-461.

[7] XU L, LV T M, LI Q M, et al. Proportional Fair Resource Allocation Based on Chance-Constrained Programming for Cognitive OFDM Network[J]. Wireless Personal Communications, 2014, 79(2): 1591-1607.

[8] ZHANG D M, XU Y Y, CAI Y M. Linear water filling power allocation algorithm in OFDMA system[J]. Journal of Electronics & Information Technology, 2007, 29(6): 1286-1289. (in Chinese)
张冬梅，徐友云，蔡跃明. OFDMA 系统中线性注水功率分配算法[J]. 电子与信息学报，2007，29(16)：1286-1289.

[9] WU J, YANG L X, LIU X. Subcarrier and Power allocation in OFDM Based Cognitive Radio Systems[C]// International Conference on Intelligent Computation Technology & Automation. 2011: 728-731.

[10] YAN S C, REN P Y, LV F S. Power allocation algorithms for OFDM-based cognitive radio system[C]// Proc. WiCOM 2010. 2010: 1-4.

[11] ZHAO Q, SADLER B M. A Survey of Dynamic Spectrum Access[J]. IEEE Signal Processing Magazine, 2007, 24(3): 79-89.

[12] ALMALFOUH S M, STUBER G L. Interference-aware radio resource allocation in OFDMA-based cognitive radio networks [J]. IEEE Trans. Veh. Technol., 2011, 60(4): 1699-1713.

[13] LI W, ZHANG Y, SO A, et al. Slow adaptive OFDMA systems through chance constrained programming[J]. IEEE Trans. Signal Process., 2010, 58(7): 3858-3869.

[14] DANIELS R. Approximation methods for electronic filter design [M]. New York: McGraw-Hill, 1974.

[15] GOLDSMITH A, CHUA S. Variable-rate variable-power MQAM for fading channels[J]. IEEE Transactions on Communications, 1997, 45(10): 1218-1230.

[16] BANSAL G, Hossain M J, BHARGAVA V K. Adaptive Power Loading for OFDM-Based Cognitive Radio Systems with Statistical Interference Constraint[J]. IEEE Transactions on Wireless Communications, 2011, 10(9): 2786-2791.

(上接第 19 页)

[3] LIN M, YU Z Y, ZHANG D, ZHU Y M, et al. Retargeting the Open64 Compiler to PowerPC Processor[C]// Proceedings of Embedded Software and Systems Symposia, 2008. San Francisco: IEEE Computer Society Press, 2008: 152-157.

[4] ALLEN R, KENNEDY K. Optimizing Compilers for Modern Architectures[M]. California: Morgan Kaufmann Publisher, 2005.

[5] RADHIKA D. Venkatasubramanyam. Array Access Analysis in Open64[D]. Houston: University of Houston, 2004.

[6] ZHAO Q, BRUENING D, AMARASINGHE S. Umbra: efficient and scalable memory shadowing[C]// Proceedings of the 8th International Symposium on Code Generation and Optimization (CGO). 2010: 22-31.

[7] NETHERCOTE N, SEWARD J. How to shadow every byte of memory used by a program[C]// Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE). 2007: 65-74.

[8] BERLIN D, EDELSOHN D. High-level loop optimizations for GCC[C]// Proceedings of the Gcc Developers Summit. 2004: 37-54.

[9] ZENG Y L, YANG C Q, HUANG C. Analysis and Improvement of the GCC 4.1 Data Dependence Analyzer[J]. Computer Engineering & Science, 2006, 28(10): 104-106. (in Chinese)
曾利永，杨灿群，黄春. GCC 4.1 数据依赖分析器的分析与改进[J]. 计算机工程与科学，2006，28(10)：104-106.

[10] ZHANG Q S, LI Y, FAN Z D, et al. Automatic Parallelization for Loops Carried Data Dependence Between Itreations[J]. Journal of Chines Computer Systems, 2014(6): 1293-1297. (in Chinese)
张琼声，李莹，范志东，等. 含有跨迭代数据依赖关系循环的自动并行化[J]. 小型微型计算机系统，2014(6)：1293-1297.

[11] KUMAR S S, CHAHAR A, VAN LEUKEN R. Cit: A GCC Plugin for the Analysis and Characterization of Data Dependencies in Parallel Programs[J/OL]. http://cas.et.tudelft.nl/pubs/kumar_DCIS_2013.pdf.