利用超级编译技术优化串行程序*

张兆庆 乔如良

(中国科学院计算技术研究所,北京 100080)

摘要 现代的超标量(Superscalar)和超流水线(Superpipeline)高速处理器通常都带有二级高速缓冲存储器(cache),以及较多的快速寄存器(register). 利用程序变换来改善程序性能,是当今编译技术研究的热门课题之一. 本文重点讨论超级编译技术中的循环分布、交换、合并和 strip_mining 对串行程序的优化应用和效果.

关键词 数据依赖,循环变换,cache 命中率,数据局部性.

超级计算机是指具有向量和/或并行体系结构的计算机. 超级编译是针对这种体系结构的编译程序,它所用到的主要技术是数据依赖测试和各种循环变换. 前者是变换的基础,后者的作用在于尽可能产生高效的并行代码.

现代的超标量(Superscalar)和超流水线(Superpipeline)高速处理器通常都带有二级高速缓冲存储器(一级 cache 做在芯片中,另一级在芯片外),以及较多的快速寄存器(register),就是普通的工作站也都有适量的 cache 和 register. 如何利用程序变换来改善串行程序的性能,这是当今编译技术研究的热门课题之一. 众所周知,程序的大量运行时间是花在循环上,而循环通常描述的是数组运算. 研究结果表明,通过循环变换组织好数组运算,使之充分利用 cache 和 register,可以极好地提高程序的性能.

本文第 1 节介绍数据依赖的有关概念,第 2 节说明各种循环变换的功能和变换策略,第 3 节用例子说明超级编译技术应用到串行程序上的效果,最后是结束语.

1 数据依赖概念

在这一节,我们用例子简单介绍数据依赖的有关概念. 下面是一个只含简单变量的直线代码段,语句前的标号 S_i 只起标识作用.

例 1:考察下面的直线代码:

S_1: $A=1.0$

S_2: $B=A+1.4142$

S_3: $A=0.3333*(C-D)$

---

* 本文 1994－05－25 收到,1994－08－26 定稿

本课题是国家高技术 863 资助项目. 作者张兆庆,女,1938 年生,研究员,主要研究领域为并行语言编译和工具环境. 乔如良,1937 年生,研究员,主要研究领域为超级编译和并行程序设计工具.

本文通讯联系人:张兆庆,北京 100080,中国科学院计算技术研究所

……

S_4:$A=(B*0.38)/0.7125$

当语句S_2在求右端表达式$A+1.4142$时,使用的A值是由语S_1赋给A的,即1.0.我们称从语句S_1到语句S_2有真依赖(true dependence),用$S_1\delta^t S_2$指称,或者我们说S_2真依赖于S_1.这意味着这两个语句在上下文中的次序必须保持,不能互换,否则会导致语义错误.接下去看S_2和S_3,在S_3中对A重新赋值,如果S_2和S_3互换,那么S_2中使用的是S_3赋给A的值,而不是S_1赋的值,这样也会引起语义错误.这两个语句在上下文中的顺序也必须保持,我们称从S_2到S_3有反依赖(anti dependence),用$S_2\delta^a S_3$指称.最后S_3和S_4之间的依赖关系称为输出依赖(output dependence),用$S_3\delta^o S_4$指称,就象对待真依赖和反依赖一样,S_3和S_4的相对次序必须得到保持,以保证语义的正确性.真依赖是语言固有的,后两类依赖可以通过引进新变量将它们消除,因此有人称它们为伪依赖或人工依赖.

数据依赖的概念已隐含着方向,如果S'依赖于S,那么S可能依赖于S',也可能不依赖.假定语句S是在语句S'之前执行,并且两个语句访问某个相同的变量,其中至少有一个语句是对该变量的赋值,那么我们说存在从语句S到语句S'的依赖,依赖类型可以从它们对该变量的定义和使用情况加以区分.

我们真正关心的是实际程序中嵌套DO循环结构中语句之间的数据依赖关系.在超级编译系统中有几种方法对DO循环体中含数组元素引用的语句做系统的有效的数据依赖测试,并且用数据依赖方向向量刻划依赖类型和依赖发生的循环层次.目前常用的数据依赖测试方法有可分性测试(精确方法),gcd和Banerjee测试(近似法,给出依赖发生的必要条件).

有关数据依赖,方向向量,测试方法,以及下一节要介绍的各种循环变换,可以在文献[1-3,11,12]中找到,本文仅作上述简单介绍.

2　循环变换与变换策略

在超级编译中实施的循环变换有许多种,每一种变换都是有条件的,笼统地讲就是变换后的循环在迭代过程中发生的数据依赖关系应与原循环一致.精确刻划合法变换的条件涉及到复杂的依赖方向向量和其它概念,在此我们不作赘述,仅就几种对优化串行程序效果显著的循环变换,介绍它们的功能和变换策略.这些变换是:循环分布,循环交换,循环条块划分和循环合并.

2.1　循环分布(loop distribution)

在超级编译中,循环分布是把循环体分解成几个组,然后将它们重构成几个相邻的循环,以利于并行化.先来看一个例子.

例2:考察下面的DO循环L:

```
L:    DO I=2,N
  S1:     A(I-1)=B(I+2)+1
  S2:     C(I-1)=3
  S3:     D(I)=C(I-1)+B(I+1)
```

```
S4:     B(I)=A(I)
      END DO
```

经数据依赖测试，得知体中语句之间的依赖关系如下：

$$S_1\delta^a S_4,\quad S_2\delta^t S_3,\quad S_3\delta^a S_4,\quad S_4\delta^a S_1$$

它们构成的依赖关系图中有一个环. 经过进一步分析和拓扑排序，可将这四个语句划分成三组(簇)：$\{S_2\},\{S_3\},\{S_1,S_4\}$. 对它们做循环分布，得到与 L 等价的三个循环 L_1,L_2,L_3：

```
L1:     DO I=2,N
S2:       C(I-1)=3
        END DO
L2:     DO I=2,N
S3:       D(I)=C(I-1)+B(I+1)
        END DO
L3:     DO I=2,N
S1:       A(I-1)=B(I+2)+1
S4:       B(I)=A(I)
        END DO
```

在 2.4 节中我们将讨论 L_1 和 L_2 的合并问题. 此例表明，循环分布不能对体中语句作随意划分，分布是有条件的，分布算法也比较复杂.

就本文讨论的主题而言，循环分布的作用在于将非紧嵌循环(imperfectly nested loops)，变换成几个相邻的紧嵌循环，以便进一步做其它的变换. 例 3 是我们将要分析的主要对象.

例 3：内积法矩阵乘程序：

```
    DO I=1,N
      DO J=1,N
        C(I,J)=0.0
        DO K=1,N
        C(I,J)=C(I,J)+A(I,K)*B(K,J)
        ENDDO K
      ENDDO J
    ENDDO I
```

对它实施循环分布后，得到两个相邻的循环 L_1,L_2，且都是紧嵌的：

```
L1:   DO I=1,N
        DO J=1,N
          C(I,J)=0.0
        ENDDO J
      ENDDO I
L2:   DO I=1,N
        DO J=1,N
```

DO $K=1,N$

$C(I,J)=C(I,J)+A(I,K)*B(K,J)$

……

其中 L_1 将矩阵 C 先置成 0;L_2 做矩阵乘 $A*B$.

2.2 循环交换(loop interchange)

循环交换是一种强有力的变换,在保持与原循环体内语句之间数据依赖关系一致的前提下,适当交换循环层,可以使程序能更好地遵守数据局部性(data locality)原则,提高 cache 命中率和减少虚存缺页次数. 这是当前程序优化的重要目标.

按嵌套循环变量出现的顺序,我们用(ijk)指称例 3 中做矩阵乘的算法形式. 通过循环交换我们可以获得 3! =6 种合法的形式,即(ijk),(jik),(kij),(ikj),(jki)和(kji). 我们用"*"表示 i 或 j 或 k,那么这 6 种形式可分为三组:(* * k),(* * j),(* * i). 实际上它们分别代表了矩阵乘的内积,外积和中积算法. 在 MIPSR2000 上用最高级优化选项做编译,三种算法的运行效果如图 1 所示.

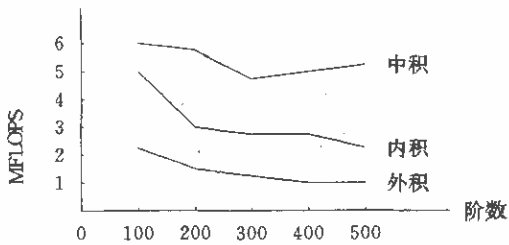

图1 三种矩阵乘算法的运行效果对比

此图显示出中积效果最好. 我们还做了更高阶测试,内积和外积的性能(MFLOPS)都随矩阵阶数增加而下降;中积基本上稳定在 4.5MFLOPS 上. 图 1 基本上反映了它们的性能状态. 现在我们来分析一下产生这种现象的原因. 以下讨论中,我们假定 cache 的调度无数据干扰,cache 的组织是理想的结构.

我们用 C_d 指称计算机的 data cache 容量. 如果矩阵乘用到的三个数组 A,B,C 的全部元素都能装入 cache 中,即 $3N^2 \leqslant C_d$,那么这三种算法的性能应当没有多少差异. 事实上现代计算机的 C_d 还不够大,大致在 4k—512k bytes 之间,最常见的是 16k bytes,以单精度计算合 4K 字,这对解大的线性代数问题是远远不够的. 现在我们来考察做最内二层循环计算时会用到哪些数据. 众所周知,FORTRAN 语言的数组元素在存储器中是按列存放的,调度到 cache 中的批数据必须是地址连续的一片数据. 如果计算时用到整个数组或一列元素,那么它们可以整体被调度到 cache 中,达到很高的命中率;如果用到数组的一行元素,则必须多次调度 cache,呈现较低的命中率. 略加分析可知,三种算法做矩阵乘的最内二层循环计算时,需要用到一个数组和 2 列或 2 行数组元素,即要用到 N^2+2N 个元素. 如果 $N^2+2N \leqslant C_d$,以 $C_d=4K$ 字计算,当 $N<64$ 时,计算应有很好的性能(事实也是如此). 实际上我们测试的矩阵阶数 $N \gg 64$. 现在我们进一步深入到最内层计算. 首先来看中积(jki)形式,它的最内层循环是 I,计算是将矩阵 A 的第 K 列元素分别乘上元素 $B(K,J)$,然后分别累加到 C 矩阵的第 J 列元素上. 这就是说,计算用到了 A 和 C 的各一列元素和 B 的一个元素,共计有 $2N+1$ 个元素. 如果 $2N+1 \leqslant C_d$,那么该循环就可以保持在 cache 中进行. 另外,值得注意的是矩阵 C 的元素,在计算中有一次使用和一次定义. 这是对 cache 中数据重用(data reuse)的体现. 数据重用(包括 register 中的数据)也是编译优化的重要目标.

对内积(ijk)形式来说，做最内层循环 K 要用到 A 的一行元素和 B 的一列元素，以及元素 $C(I,J)$. 实际上一次能调度到 cache 中的元素个数为 $N+2$. 外积要用到 C 的一行元素，$A(I,K)$和 B 的一行元素，这是最坏的情况，cache 的命中率极低.

令某循环计算一次可调到 cache 中的数据量为 M，$D=C_d-M$ 称为 cache 未命中数(the number of misses). 优化的目的在于使 D 尽可能的小. 从以上分析中不难看出中积最好，内积次之，外积最差的原因就在于 cache 命中率的不同.

结论：循环交换的策略应使循环计算中有最佳的 cache 命中率，并且其中重用数据愈多愈好. 其实这个结论适用于所有的循环变换.

循环交换可以应用到初值，步长均不为 1 的非规则嵌套循环上，也可以直接应用到非紧嵌的循环上，还可以对三角循环(即内层循环的初值或终值为其外层循环变量的线性函数)做循环交换，在此不一一讨论了.

2.3 循环条块划分(strip mining)

循环条块划分是将一个 DO 循环分割成条块状. 以 DO $I=1,N$ 为例，该变换将它变换成：

```
DO I'=1,N,ib
  DO I=I',min(I'+ib-1,N)
```

我们称前一个循环为控制循环(有人称之为 tiling 循环)，后者称为计算循环(有人称之为 element循环)，ib 称为条宽.

现在我们对 2.2 节中分析过的中积形式(kji)做二次条块划分，然后再应用循环交换将控制循环移到最外层，即得：

```
DO J'=1,N, ia
  DO I'=1,N, ib
    DO K=1,N
      DO J=J',min(J'+ia-1,N)
        DO I=I',min(I'+ib-1,N)
          C(I,J)=C(J,J)+A(I,K)*B(K,J)
        ......
```

我们用 $j'i'(kji)$指称这种形式，它的计算过程可以用一种称为 DCD 图表示(Data and Computation Diagram)，如图 2.

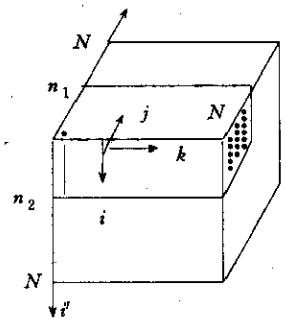

图2 $j'i'(kji)$形式的DCD图

令 $ia=n_1, ib=n_2$，从图上不难看出最内层的 3 个计算循环是做子矩阵乘(分块乘)，在外层循环控制下，每次计算一个长方块，其迭代空间为 n_1*n_2*N.

现在考察做了循环条块划分的二个最内层循环，计算时要用到 C 的一个子数组，大小为 n_1*n_2，A 的第 k 列元素，列长为 n_2，以及 B 的第 k 行元素. 在实际应用中常取 $ia=ib=n$，故只要 $n^2+n+1\leqslant c_d$，即 $n\approx\sqrt{c_d}$，那么就可以保证最内二层循环计算基本上

是在 cache 中进行,并且 C 的子数组为重用数据. 这种划分完全可行. 对于更大的线性代数计算问题,我们还可以做三层条块划分. 就矩阵乘而言,二层条块划分已经可以了. 测试结果在第 3 节中给出.

条块划分是在向量化编译中发展起来的,原则上可以无条件地应用到循环上,但是划分的方向要保持正交. 对于分块的大小要注意优化,因为它不仅与 C_d 有关,而且还与 cache 的结构有关(是直接映射的,还是成组相联的,这二种结构对数据的干扰作用是不一样的). 总之,应用条块划分时要对循环体中的计算做细致的分析,采用适宜的变换策略,才能收到良好的效果. 可以说对这种变换的研究现在还很不完善,有不少问题有待进一步深入探讨.

2.4 循环合并(loop fusion)

循环合并是将两个相邻的合并无阻碍循环(合并后保持语义等价)合并成一个循环. 例如,在 2.2 中对例 2 做循环分布得到的循环 L_1 和 L_2 就是可以合并的相邻循环. 合并后为:

```
DO I=2,N
  C(I-1)=3
  D(I)=C(I-1)+B(I+1)
END DO
```

合并前,二个循环要做 $2(N-1)$ 次增量_测试运算,合并后只需做 $N-1$ 次增量_测试运算. 当然这对程序性能改善是微不足道的. 但是,分开做时 C 的全部元素先要被保存起来,然后再全部取出来使用. 合并后,可以减少这种 load/store 操作,而这类操作很费时间. 循环合并在减少寄存器与存储器之间的传输方面是有益的. 对串行程序优化,首先要做的就是循环合并.

3 测试结果

我们在 1993 年 6 月研制成功了一个"并行优化重构工具 PORT",它的基本功能是将 FORTRAN 77 程序自动转换成并行化程序. 此系统是在共享存储紧耦合并行机 SGI POWER 4D/240 和曙光 I 号上实现的. 在测试和应用过程中发现,它对串行程序的优化也有很好的效果. 下面 3 个程序的测试结果是在 MIPSR2000 上,用最高级优化选项(-O3)编译、运行获得的. 前 2 个程序由中国科学院计算中心提供(源文件名为 b1.f 和 b3.f),后 1 个是由美国 livermore 实验室提供的国际标准性能测试程序(源文件名为 livermore.f),经 PORT 变换后的源文件名是在原文件名前缀 p_.

3.1 矩阵乘

b1.f 是矩阵乘程序,它用了内积,外积和中积三种算法,经 PORT 做循环分布和交换后,实际上都变成了中积. 我们对中积做二次循环条块划分,得即 $j'i'(kji)$ 形式的矩阵乘算法. 下表是 (kji) 和 $j'i'(kji)$ 算法运行性能(MFLOPS)的对比,平均可得到 1.5 加速比.

阶数	100	200	300	400	500
(kji)	5.88	4.71	4.59	4.59	4.62
$j'i'(kji)$	7.69	7.58	7.51	7.52	7.54

3.2 高斯消去法解线性方程组

PORT 应用到 b3.f 上，只对其中的二个嵌套循环做了交换，其效果令人瞩目. 下面是测试的结果：

阶数	100	200	300	400	500	600
b3.f	2.60	1.60	1.25	1.01	0.87	0.73
P_b3.f	3.91	4.08	3.80	3.70	3.76	3.68

从上表可以看到 p_b3.f 的性能对方程组的阶数不敏感，直至 2000 阶，它的性能仍稳定在 3.6MFLOPS；而原程序的性能下降得很快.

3.3 livermore 测试

livermore.f 由 24 个 kernel 组成，它们都是应用程序中经常出现的普通循环，为了测得高速计算机的 CPU 时间，故在每一个核心 DO 循环外面加了一层形如 DO $L=1$, LOOP(在我们的例子中 LOOP 取值为 2000)的简单重复循环. livermore 的 kernel 1,7,9,12 具有如下的形式：

```
      DO L=1, LOOP
        DO K=1, N
S:      A(…)=f(…)
        END DO K
      END DO L
```

其中计算语句 S 的左部为下标变量 $A(\cdots)$，右部为不含数组 A，但含多个其它数组元素的复杂表达式，显然这二个循环可以交换，PORT 对它们做了循环交换. 交换后的内层循环具有极好的数据局部性，加之 MIPS 编译程序在代码级做了窥孔优化(peephole optimization)，使得这四个交换后的循环，执行速度得到了极大的提高，f 计算越复杂，效果越好. 下面是从 livermore.f 运行报告中摘取的部分结果.

kernel	1	7	9	12
livermore.f	7.35	7.77	7.05	6.6389
p_livermore.f	175.18	382.08	370.88	32.69

livermore.f 24 个 kernel 的速度在 0.9129 到 7.8339 MFLOPS/sec. 平均速率为 5.5197 MFLOPS/sec. p_livermore.f 24 个 kernel 的速度在 1.5464 到 382.0822 MFLOPS/sec. 平行速率为 44.3963 MFLOPS/sec.

4 结束语

高性能计算机的潜在能力是否能充分发挥出来，关键在软件，特别是优化编译系统. 当前国际上非常重视编译优化技术，像全局数据流分析这样的“古老”技术，现在又有人对它做更深入研究，使之更加实用，因为精确的数据流分析是当代超级编译的重要基础. 我们在 PORT 的实现中，做了过程间数据流分析，对数据依赖测试做了不少改进，使得 PORT 不仅成为一个很好的并行重构工具，而且也是一个极好的串行程序优化工具，本文的结果就是在 PORT 支持下完成的.

参考文献

1 Allen J R, Kennedy K. Automatic translation of Fortran programs to vector form. ACM TOPLAS, 1987,9(4): 491—542.

2 Banerjee U. Dependence analysis for supercomputing. Boston MA: Kluwer, 1988.

3 Wolfe M. Optimizing supercompilers for supercomputers. Combridge MA: The MIT Press, 1989.

4 Zima H P. Supercompilers for parallel and vector computers. New York: ACM Press, 1990.

5 Polychronopoulos C. D. Parallel programming and compilers. Boston MA: Kluwer, 1988.

6 Allen J R. and Kenndey K. Automatic lool interchange. Proc. ACM SIGPLAN'84 Sym. on Compiler Construction, SIGPLAN Notices, 1984,19(6):233—46.

7 Wolfe M J. Advanced loop interchanging. Proc. 1986 Int. Conf. Parallel Proc., 1986. 536—542.

8 Lam M S, Rothberg E E, Wolf M E. The cache performance and optimizations of blocked algorithms. ASPLOS, 1991. 67—74.

9 Carr S, Kennedy K. Compiler blockability of numerical algorithms. Proc. of the Supercomputing'92 Conf., 1992. 114—124.

10 Chen D. Hierarchical blocking and data flow analysis for numerical linear algebra. ACM Int. Conf. Supercomputing, 1991. 12—19.

11 Kennedy K, Mckinley K. Optimizing for parallelism and data locality. Proceedings of ICS'92, 1992.

12 张兆庆,乔如良. 并行优化重构工具集:PORT. 计算机学报,1994,17(12).

13 高念书等. 实用数据依赖测试方法. 计算机学报,待发表.

USING SUPERCOMPILING TECHNIQUES TO OPTIMIZE SEQUENTIAL PROGRAMS

Zhang Zhaoqing　　Qiao Ruliang

(*Institute of Computing Technology, The Chinese Academy of Sciences, Beijing* 100080)

Abstract　Recent superscalar and superpipelined processors commonly have at least two levels caches and a large register set. Using the transformations to improve performance of programs that is one of the research projects of compiling technique. This paper discusses optimization of sequential programs by using do_loop distribution, do_loop interchange, do_loop fusion, and strip_mining used in supercompiler. These tramsformations have been shown to result in dramatic improvements.

Key words　Data dependence, loop transformation, cache hit ratio, data locality.