第32卷 第24期 计 算 机 工 程 2006年12月
Vol.32 No.24 Computer Engineering December 2006
·软件技术与数据库· 文章编号：1000—3428(2006)24—0037—04 文献标识码：A 中图分类号：TP311

OpenMP 并行程序的编译器优化

张 平，李清宝，赵荣彩
(解放军信息工程大学信息工程学院，郑州 450002)

摘 要：OpemMP 标准以其良好的可移植性和易用性被广泛应用于并行程序设计。该文讨论了 OpenMP 并行程序的编译器优化算法，在编译过程中通过并行区合并和扩展，实现并行区重构，并在并行区中实现了基于跨处理器相关图的 barrier 同步优化。分析验证表明，这些优化策略减少了并行区和 barrier 同步的数目，有效地提高了 OpenMP 程序的并行性能。

关键词：跨处理器相关；barrier 同步；并行区重构；数据相关图

Compiler Optimization Algorithm for OpenMP Parallel Program

ZHANG Ping, LI Qingbao, ZHAO Rongcai
(School of Information and Engineering, PLA Information and Engineering University, Zhengzhou 450002)

【Abstract】 OpenMP is widely used in parallel programming for its portability and simplicity. This paper introduces the compiler optimization algorithms for OpenMP parallel program. In compiling, parallel regions are reconstructed through extension and combination. And a barrier synchronization optimization algorithm based on cross-processor dependence graph is developed to eliminate redundant barriers in each parallel region. Analysis show that these strategies reduce the number of parallel region and barrier synchronization, and can improve the parallel performance of OpenMP program.

【Key words】 Cross-processor dependence; Barrier synchronization; Parallel region reconstruction; Data dependence graph

OpenMP 是共享内存并行程序设计的工业标准，其目标是为具有统一地址空间的并行系统提供可移植、可扩展的开发接口，它通过编译指示和运行时库函数扩展 C、C++和 Fortran 语言支持并行。

OpenMP 为程序员提供了一种简单的并行程序设计方法，可以在串行程序的基础上方便地开发出并行程序。但应用程序员往往缺乏对程序并行性的分析(如数据相关性分析和通信分析等)，许多 OpenMP 程序的性能也并不理想，并行效率较低；另一方面，如果要求程序员在编写并行程序时进行深入的程序分析就会增加程序设计的难度，违背 OpenMP 的易用性原则，也是不现实的。因此，我们考虑在编译过程中实现 OpenMP 程序的优化。

本文讨论了循环级并行的 OpenMP 并行程序的编译器优化策略：通过并行区的扩展和合并重构并行区，减少并行区的数目；在重构后的并行区中，依据计算的分配调度进行跨处理器相关性分析，建立跨处理器相关图，消除冗余 barrier 同步。

1 OpenMP 并行程序

OpenMP 利用编译指示、运行时库函数和环境变量描述程序的并行特性。

1.1 OpenMP 编译指示

OpenMP 的编译指示分为并行结构、工作共享结构、同步编译、数据环境指示等几类。常用的编译指示包括：(1)#pragma omp parallel 说明并行结构，其中的代码被多个工作线程执行；(2)#pragma omp for 说明工作共享结构，指示循环被分配给多个工作线程并行执行；(3)#pragma omp barrier 为同步指示，标记线程在此等待，直到所有线程都执行到这个点，再继续向下执行。

在 omp parallel, omp for 的结束都隐含着一个 barrier 同步点，为减少不必要的同步，OpenMP 提供了 no wait 子句，指示可以不进行 barrier 同步。

1.2 OpenMP 的执行模式

OpenMP 程序遵循 *fork-and-join* 执行模式(如图 1)。程序从主线程开始执行，当遇到并行结构，主线程启动(创建或唤醒)一组工作线程并行执行其中的语句。当遇到工作共享结构，工作负荷由线程组中的各个线程分担；并行区结束，线程组中的线程同步，工作线程终止(消亡或睡眠)，主线程继续执行。

在程序执行过程中，可多次执行*fork-and-join*过程，需要在串行执行和并行执行间进行多次切换。实验表明[1]，OpenMP执行过程中，串行执行和并行执行切换带来的额外执行开销是影响OpenMP程序性能的一个重要因素。

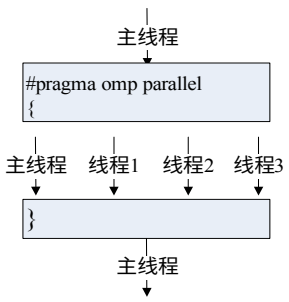

图 1 OpenMP 程序的执行模式

基金项目：国防科研基金资助重点项目
作者简介：张 平(1969－)，女，博士生，主研方向：并行识别，并行编译；李清宝，副教授；赵荣彩，教授、博导
收稿日期：2006-02-24 **E-mail：**lqb215@vip.371.net

OpenMP 实现共享内存编程模式，当两个或多个处理器读写相同的内存单元时，要依靠同步来保证程序的正确性，barrier 是一种常用的同步方式。实验表明，barrier 也是影响并行性能的重要因素之一。

因此，考虑在编译过程中通过程序分析对 OpenMP 程序进行优化，减少并行区的数目和消除冗余 barrier，提高 OpenMP 并行程序的性能。

2 并行区的扩展和合并

2.1 并行区扩展和合并的方法

并行区扩展和合并的目标就是通过并行区重构尽可能地扩大并行区，减少程序中并行区的数目，以减少并行执行和串行执行切换的开销，同时为消除 barrier 同步建立基础。

(1)相邻并行区的合并：当两个并行区相邻或两个并行区间的串行代码符合合并的条件，可以将它们合并到一个并行区中。

(2)并行区向外层扩展：如果一个高级控制结构，如 if、for、do-while 等语句体中的所有语句都被包含在一个并行区中，则该并行区可以被扩展到此控制结构之外。

因为实际程序中代码情况比较复杂，所以将串行代码合并或扩展到并行区时要考虑：

(1)重复运行：可以在多个处理器间重复执行的语句；

(2)条件运行：可以被限定在一个特定的处理器上执行的语句。

在并行区中加入重复计算，增加了必须执行的运算量。条件运行则增加了评估条件表达式的运算，降低了处理器利用率。只有当这些语句位于两个并行区之间或位于循环体内，才运用上面两个条件进行并行区合并、扩展。

2.2 并行区重构算法

采用自底向上的递归调用方式，从最内层代码进行并行区合并操作，如果同层的所有语句都包含在一个并行区中，则向外层进行扩展，直到遇到无法合并到并行区中的语句。算法如下：

```
PRConstruct(CodeSeg)
{针对每个并行区，建立一个 block 结构；
  parallel=0；  //并行区合并标志
  For  CodeSeg 中的每一结构 CS
  {Case 结构类型：
    block 结构： //CS 为并行区
            parallel =1;
            break;
    loop 结构： //CS 为循环语句，递归处理 loop 循环体
            parallel=PRConstruct(loop_body);
            break;
    if 结构:：   //CS 为 if 语句
            递归处理 if 语句的 then 语句体和 else 语句体
            parallel=PRConstruct(then_body)                    &&
PRConstruct(else_body)；
            break;
    instruction 结构： //CS 为指令节点
              if 该语句符合合并条件 parallel=1;
              else parallel=0;
            break;
    if  parallel==1  将 CS 合并到前面并行区中；
    else    置并行区结束标志
    }
  if  CodeSeg 所有结构都包含在同一并行区中 return 1
  else return 0;
  }
```

在重构过程中采用了保守的方法，对于合并到并行区的串行代码限定由单线程执行，默认在其结束处存在隐含的 barrier，且在每个并行循环的开始和结束都由隐含的 barrier，这些同步保证程序语义的正确性，但其中很多是冗余的，影响了程序的性能。

3 barrier 同步的优化

在并行区中，通过对共享数据的读写实现处理器间通信，barrier 保证并行执行的处理器对内存访问的一致性。对同一共享数组元素的一对读写操作的执行会有两种情况：

(1)对该元素的读写操作被划分到同一处理器执行。实际上对该元素的读写是按程序的语法顺序串行执行的，不需同步即可保证正确的语义。

(2)对该数组元素的读写被分配调度到不同的处理器执行，引起跨处理器数据相关。这种情况下，需要插入 barrier 同步以保证正确的读写顺序。

barrier 同步优化首先要分析并行区中的跨处理器相关关系，如果可以证明两个区域中所有对同一共享数组元素的读写都被分配到同一个处理器中，则不存在跨处理器相关，无须 barrier 同步。然后建立相关关系图，应用 barrier 优化算法，用尽可能少的 barrier 消除所有的跨处理器相关。

在 OpenMP 标准中说明了并行结构和工作共享结构中程序代码的执行方式，对并行循环(omp for)标准中给出了 4 种调度的执行方式，包括：static，dynamic, guided, runtime。除了 runtime 方式外，其他 3 种方式都可以确定循环中计算的分配方式。可以利用这些信息，确定计算的分配调度方式，进而确定对同一数组引用的计算是否被分配到同一处理器，即代码间是否存在跨处理器相关关系。

3.1 跨处理器数据相关关系

定义 1 如果相关关系的源和目的在不同的处理器被访问，则称该相关关系为跨处理器数据相关关系。

显然，跨处理器数据相关关系在满足相关关系的前提下，要根据计算在处理器间的分配方式来确定。我们要确定的是对于同一数组元素读写的计算是否被分配到同一处理器，而不关心具体被分配到哪个处理器。如果两个代码段间存在跨处理器相关，则必定满足：

(1)在两个代码段存在对同一数组的引用：写—读，写—写，读—写；

(2)对同一数组的两次引用的下标相等；

(3)访问这两个数组引用的循环迭代被分配到不同的处理器。

令 f1 ($\vec{i}_1$)和 f2 ($\vec{i}_2$)分别表示同一数组的两个数组引用的下标访问函数，其中 $\vec{i}_1$、$\vec{i}_2$ 为循环迭代向量；$c_1(\vec{i}_1)$ 和 $c_2(\vec{i}_2)$ 分别为两个数组引用所在区段的计算划分函数；$p_1 = c_1(\vec{i}_1)$ 和 $p_2 = c_2(\vec{i}_2)$ 分别表示迭代 $\vec{i}_1$ 和 $\vec{i}_2$ 所被分配到的处理器。$b_1(\vec{i}_1)$ 和 $b_2(\vec{i}_2)$ 是循环迭代向量 $\vec{i}_1$ 和 $\vec{i}_2$ 必须满足的边界条件。则两个数组引用间存在跨处理器相关关系，必满足下列限定条件：

$$f_1(\vec{i}_1) = f_2(\vec{i}_2)$$
$$p_1 = c_1(\vec{i}_1)$$
$$p_2 = c_2(\vec{i}_2)$$
$$b_1(\vec{i}_1) \geq 0$$

$b_2(\vec{i}_2) \geq 0$

$p_1 \neq p_2$

这些限定条件构成了一个线性不等式系统，利用 Fourier-Motzkin 消除算法扫描该不等式系统来判断该不等式系统是否有解，如果系统有解表明两次数组引用间存在跨处理器相关。而对于并行区中代码间的跨处理器相关关系测试转化为对于代码段中所有共享数组数据读写引用对跨处理器相关关系的测试。

因此，判断两个代码段间是否存在跨处理器相关的核心问题转化为构造线性不等式限制系统的过程。如下给出了基准测试程序 tomcat 的一个程序片断：

```
#paragma omp parallel
  #paragma omp for schedule（static）
     for (i=2;i<N;i++) {
        x[i][M]= x[i][M]*d[i][M];
        y[i][M]= y[i][M]*d[i][M];
     }
#paragma omp parallel
  #paragma omp for schedule（static）
     for (i=2;i<N;i++)
        for (j=M-1;j>0;j--) {
           x[i][j]= x[i][j]+x[i][j+1];
           y[i][j]= y[i][j]+y[i][j-1];
        }
```

其中，两个并行循环可以通过扩展和合并构成一个并行区，如下所示：

```
#paragma omp parallel{
 #paragma omp for schedule（static）
    for (i=2;i<N;i++) {
        x[i][M]= x[i][M]*d[i][M];
       y[i][M]= y[i][M]*d[i][M];
    } //隐含 barrier
#paragma omp for schedule（static）
    for (i=2;i<N;i++)
      for(j=M-1;j>0;j--) {
        x[i][j]= x[i][j]+x[i][j+1];
        y[i][j]= y[i][j]+y[i][j-1];
}   //隐含 barrier
}
```

下面给出了判断两个工作共享结构中 X[i][M]和 X[i][j+1]引用对间是否存在跨处理器相关关系的不等式限制系统，利用 Fourier-Motzkin 消除算法可以证明该不等式系统无解。因为采用相同的方法来测试可以证明两个循环中其它的读写引用对间也不存在跨处理器相关，所以两个循环间不存在跨处理器相关关系。

符号不等式系统：

循环索引边界条件：

$2 \leq i_1 \leq N\text{-}1$

$1 \leq j_2 \leq M\text{-}1$

$2 \leq i_2 \leq N\text{-}1$

数组下标函数：

$i_1 = i_2$

$M = j_2 + 1$

计算划分：

$b*nprocs = N$

$b*p_1 \leq i_1 < b*(p_1 + 1)$

$b*p_2 \leq i_2 < b*(p_2 + 1)$

不同处理器：

$p_1 < p_2$ 或

$p_1 > p_2$

3.2 barrier 优化算法

barrier 优化算法以并行区为单位，采用自底向上的原则，从并行区代码开始，递归处理每个复杂结构的语句体：调用 barrier 消除其中的跨处理器相关。然后将整个结构看作为一个语句，作为没有被内部 barrier 消除的相关关系的源和目的，递归算法继续应用到上层代码，直到并行区中所有代码被处理完成。

3.3 barrier 消除算法

并行区的语句间存在跨处理器相关关系，为保证程序的正确执行需要在源和目的两条语句间的所有执行路径上插入 barrier 同步以消除相关，而设置同步点的灵活性很大，对于同一个相关在程序的执行路径上往往可以有多个插入位置，如果插入点合适，可以同时消除其他相关。

性质 1 一个 barrier 可以消除在这条执行路径上所有源点在该 barrier 之前，汇点在 barrier 之后的跨处理器相关关系。可用尽量少的 barrier 消除并行区中的所有的跨处理器相关。

定义 2 跨处理器相关图 DG=(S，E)为有向图，其中 S 为顶点集，每个顶点表示一个语句，E 为边集，表示顶点间的跨处理器相关。顶点 s1 到顶点 s2 的弧表示 s1 到 s2 存在跨处理器相关，s1 称为源点，s2 称为汇点。

为了表示方便，将复杂结构看成一个语句，所有处理的代码段都作为基本块对待，对其中的语句按顺序进行编号，为每个顶点建立一个相关链，其中的顶点按编号顺序排列。如果存在从语句i到语句$a_{i,j}$的跨处理器相关，则顶点$a_{i,j}$在顶点i的邻接表链中。基本块代码如下：

```
(1) a[i]=i;
(2) b[i]=e[i]+1;
(3) c[i+1]=a[i];
(4) d[i+2]=a[i]+c[i+1];
```

图 2 为对应的跨处理器相关关系。

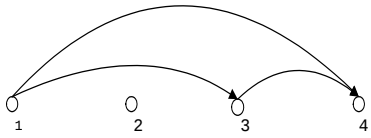

图 2 数据相关关系

顶点的相关链如下：

1 → 3, 4

3 → 4

barrier 优化的过程实际上是消除跨处理器相关的过程，优化算法分两步执行：(1)预处理，应用最近汇点算法，针对跨处理器相关图中的每个源点，消除冗余相关，降低问题的时间复杂性和空间复杂性。(2)利用第 1 汇点优化算法进一步消除相关，确定关键相关和最终必须保留的 barrier 同步点。

3.3.1 最近汇点相关消除算法

根据性质 1，在跨处理器相关图上 barrier 和跨处理器相关具有如下性质：

性质 2 在跨处理器相关图G中，对于所有源点为i的弧，令$a_{i,1}$是所有汇点中离i最近的顶点，则在语句i和语句$a_{i,1}$间插入一个barrier同步点，可消除i->$a_{i,j}$，j≥1 的所有相关。

根据性质 2，使用一个barrier即可消除相关图G中以i为源点的所有相关弧，可以只保留源点i到最近汇点$a_{i,1}$的弧，消除此弧的barrier即可消除所有以i为源点的弧，因此有最近汇点

相关消除算法：

(1)从顶点 1 开始执行，令 i=1；

(2)以i为源点，寻找最近的汇点，在相关图上删除除t_i外其余的以i为源点的弧；

(3)令 i=i+1，如果所有的源点都处理完，算法结束，否则转(2)。

最近汇点相关消除算法执行后，得到相关图G’，其中的每个顶点最多只有一条出边，即对于顶点i来说，$a_{i,1}$是唯一要存储的相关，对于有n个顶点的相关图，仅需长度为n的向量来保存所有的相关关系，这个过程同样减少了后续处理的时间复杂性。

3.3.2 第 1 汇点同步优化算法

性质 3 如果 s 是图 G’的一个顶点，t 是 s 后的第 1 个汇点，则在 t 前插入个 barrier 将会消除所有源点大于等于 s 而小于等于 t 的相关。

根据性质 3，有第 1 汇点同步算法：在图G’中，从顶点 1 开始，寻找顶点 1 后的第 1 个汇点t_1。迭代地计算，选择t_i作为所有源点大于等于t_{i-1}的相关关系的第 1 个汇点。如果对于某一t_l，没有相关弧的源点大于或等于t_l，算法终止。如果在$t_1,t_2,\ldots,t_l$前插入显示barrier，则可消除所有的相关。

利用第 1 汇点同步优化算法可确定基本块中最终所需的 barrier 同步点，而且可以证明文献[4]所确定的 barrier 数为最少，算法的时间复杂度为 O(n),n 为基本块中的语句数。

4 分析验证

我们利用斯坦福大学suif[5]工作组提供的ppopp测试包对上面所讨论的优化算法的性能作了分析和验证。ppopp中共有 6 个程序模块，分别为adi1k、erle64、lu1k、swm25 和tomvatv。

编译器优化的目标是通过减少并行区和 barrier 数目，降低并行执行的开销，提高程序的性能。表 1 给出了优化前、后并行区数和 barrier 同步次数的对比。从表 1 中可以看出，文中所讨论的优化算法有效地减少了程序中并行区和同步的数目。

表 1 静态分析结果

程序名	并行区数		同步次数		
	原始	优化后	原始	优化后	消除百分比(%)
adi1k	27	2	42	16	47.6
erle64	38	12	76	39	48.6
lu1k	1 025	1	2 050	1 027	49.9
sor	101	1	202	153	24.7
swm256	16	9	32	25	31.3
tomvatv	9	5	18	14	22
平均					37.35

5 总结

本文讨论了 OpemMP 并行程序的编译器优化算法，通过分析证明该算法通过并行区重构和 barrier 同步优化能够有效地减少并行区和 barrier 的数目，降低程序执行的额外开销。但目前，算法将串行代码合并到并行区时仅考虑了条件运行的情况，限制串行代码在单线程上运行；在 barrier 优化中针对程序代码的每一个层次，将其中的所有语法结构都看成是语句，进而将其看成一个基本块，在基本块中插入最少数目的 barrier，而没有考虑循环嵌套中循环携带相关的处理。这些将是我们下一步深入研究的主要内容。

参考文献

1 Tseng Chauwen. Compiler Optimizations for Eliminating Barrier Synchronization[C]. Proc. of the 5th ACM Symposium on Principles and Practice of Parallel Programming, Santa Barbara, CA, 1995.

2 Krawezik G, Cappello F. Performance Comparison of MPI and Three OpenMP Programming Styles on Shared Memory Multiprocessors[C]. Proc. of SPAA'03, San Diego, California, USA, 2003-06-07.

3 沈志宇, 胡子昂. 并行编译方法[M]. 北京: 国防工业出版社,2000.

4 张 平, 赵荣彩, 李清宝. 基于相关性的同步优化算法[J]. 计算机工程, 2005, 31(17): 68.

5 Wilson R P, French R S, Wilson C S. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers[J]. ACM SIGPLAN Notices, 1994, 29(12): 31-37.

(上接第 3 页)

防护系统起着与生物免疫系统相类似的作用。生物体的免疫系统保护机体免受外界病原体的侵害，而信息安全系统则防止网络中的数据遭到病毒的入侵以及异常行为的破坏。首先为基于优化的 Linux 系统的 NAS 设备建立了多层免疫模型。

用户认证利用账号和密码的限制，可以阻挡未经过身份验证的用户进入系统；文件权限层对不同的用户设置了不同的文件访问权限，可以拒绝用户未被授权的文件操作。穿过前面两道屏障的用户操作附带着用户的必要信息，如用户标识符(user id, uid)、组标识符(group id, gid)等。在用户阶梯层，“用户分阶”根据用户信息把它们分为只读用户、读写用户、管理员和系统用户等。不同的用户处于不同的阶梯上，在相应的阶梯中保存有该阶用户正常的、合法的行为模式库(“本体”库)。用户的操作命令按照分阶信息被发送到各自对应的阶梯中，分布于各个阶梯上的异常检测系统(Abnormity Detection System, ADS)分析用户命令请求的系统调用序列，识别“本体”和“异体”。用户行为若被识别为“本体”，则让其穿过第 3 道屏障被系统执行；若识别为“异体”，则进行免疫，禁止运行。实验证明 ADS 的引入对系统性能只产生很小的影响。

5 结论

数据中心采用基于 IP 的存储技术，实现了用户绕过服务器直接访问存储设备的存储结构，该结构相对于服务器存储能够提供更高的系统性能。应用于数据中心的网络化光盘库丰富了数据中心的存储层次，其中大容量磁盘缓存和光盘镜像技术的使用，提高了光盘库的性能。基于生物免疫思想的安全机制能有效阻止对数据中心存储节点的异常访问和操作，同时异常检测系统的引入对系统性能只产生很小的影响。IP 存储、NAS 与 SAN 的融合和存储虚拟化被认为是网络存储的发展趋势，它们在数据中心中都有所体现。

参考文献

1 Gibson G A, Meter R V. Network Attached Storage Architecture[J]. Communications of the ACM, 2000, 43(11): 37-45.

2 SNIA Technical Council. Shared Storage Model[EB/OL]. 2003. http://www.snia.org.

3 Mesnier M, Ganger G R, Riedel E. Object-based Storage[J]. IEEE Communications Magazine, 2003, 41(8): 84-90.