软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2014,25(6):1154−1168 [doi: 10.13328/j.cnki.jos.004425]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

面向规则 DOACROSS 循环的流水并行代码自动生成*

刘晓娴[1,2], 赵荣彩[1,2], 赵 捷[1,2], 徐金龙[1,2]

[1](中国人民解放军信息工程大学,河南 郑州 450002)

[2](数学工程与先进计算国家重点实验室,河南 郑州 450002)

通讯作者: 刘晓娴, E-mail: xiaoxian0321@gmail.com

摘 要: 发掘DOACROSS循环中蕴含的并行性,选择合适的策略将其并行执行,对提升程序的并行性能非常重要.流水并行方式是规则 DOACROSS 循环并行的重要方式.自动生成性能良好的流水并行代码是一项困难的工作,并行编译器对程序自动并行时常常对DOACROSS循环作保守处理,损失了DOACROSS循环包含的并行性,限制了程序的并行性能.针对上述问题,设计了一种选择计算划分循环层和循环分块层的启发式算法,给出了一个基于流水并行代价模型的循环分块大小计算公式,并使用计数信号量进行并行线程之间的同步,实现了基于 OpenMP 的规则 DOACROSS 循环流水并行代码的自动生成.通过对有限差分松弛法(finite difference relaxation,简称 FDR)的波前(wavefront)循环和时域有限差分法(finite difference time domain,简称 FDTD)中典型循环以及程序 Poisson,LU 和 Jacobi 的测试,算法自动生成的流水并行代码能够在多核处理器上获得明显的性能提升,使用的流水分块大小计算公式能够较为精确地计算出循环流水并行时的最佳分块大小.自动生成的流水并行代码与基于手工选择的最优分块大小的流水并行代码相比,加速比达到手工选择加速比的 89%.

关键词: 流水并行;自动并行;DOACROSS 循环;代价模型

中图法分类号: TP314

中文引用格式: 刘晓娴,赵荣彩,赵捷,徐金龙.面向规则 DOACROSS 循环的流水并行代码自动生成.软件学报,2014,25(6):1154−1168. http://www.jos.org.cn/1000-9825/4425.htm

英文引用格式: Liu XX, Zhao RC, Zhao J, Xu JL. Automatic generation of pipeline parallel code for regular DOACROSS loops. Ruan Jian Xue Bao/Journal of Software, 2014,25(6):1154−1168 (in Chinese). http://www.jos.org.cn/1000-9825/4425.htm

Automatic Generation of Pipeline Parallel Code for Regular DOACROSS Loops

LIU Xiao-Xian[1,2], ZHAO Rong-Cai[1,2], ZHAO Jie[1,2], XU Jin-Long[1,2]

[1](PLA Information Engineering University, Zhengzhou 450002, China)

[2](State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450002, China)

Corresponding author: LIU Xiao-Xian, E-mail: xiaoxian0321@gmail.com

Abstract: To obtain as much performance improvement as possible for sequential applications, it is important to exploit parallelism lurking in DOACROSS loops and find good schemes for their parallel execution. Pipelining is such a parallelizing method which can work well for regular DOACROSS loops. However, it is so hard to maintain high performance pipeline parallel codes automatically that parallel compilers always treat DOACROSS loops conservatively. Compilers usually serialize DOACROSS loops, which loses the inherent parallelism of DOACROSS loops and affects the performance of generated parallel programs. To solve this problem, automatic generation of pipeline parallel code for regular DOACROSS loops is implemented for multicore platform based on OpenMP. Firstly, a heuristic is proposed to choose the partition loop and the tiling loop of regular DOACROSS loops. Secondly, a formula based on pipelining cost model is given to compute the optimal tiling size. Lastly, the synchronization between threads is implemented with counter semaphores. Measuring against the wavefront loops of finite difference relaxation, the representative loops of finite difference time

* 基金项目: “核高基”国家科技重大专项(2009ZX01036-001-001-2)
收稿时间: 2012-12-10; 修改时间: 2013-03-07; 定稿时间: 2013-05-03

domain, and Poisson, LU and Jacobi procedures, the pipeline parallel loops automatically generated by the proposed method increase execution efficiency on multicore platform with the average speedup up to 89% of the optimal speedup obtained manually.

Key words: pipeline parallel; automatic parallelization; DOACROSS loops; cost model

随着主频的提高,处理器功耗以指数速度急剧上升,使得以提高主频来提升处理器性能的方法不再有效[1].多核处理器已成为处理器体系结构发展的一个主要方向[2].尽管多核处理器能够提升多线程程序的性能,但早已存在的诸多单线程程序无法从中获益,程序员也习惯于编写单线程程序.自动并行化技术是将单线程程序移植到多核上的重要手段,通过对单线程程序中蕴含并行性的分析与发掘,采用程序变换技术自动生成适合多核处理器运行的多线程程序.根据 Amdahl 定律,应用程序通过并行获得的性能提升受限于程序中串行执行部分所占的比例,因此,充分发掘程序的并行性,是提升程序并行性能的有效手段.计算机科学中的二八法则表明,程序中 20%的代码占据了程序 80%的执行时间,这些花费大量时间开销的代码往往是程序中的循环.因此,循环是发掘程序并行性的主要对象.

根据蕴含并行性的不同,Cytron[3]将循环分为串行循环、能够完全并行的 DOALL 循环和 DOACROSS 循环.DOACROSS 循环因为携带跨迭代的依赖关系,并行性介于 DOALL 循环和串行循环之间.Chen 和 Yew[4]通过测试表明,将程序中的 DOACROSS 循环串行执行会带来程序并行性的巨大损失.因此,发掘 DOACROSS 循环中蕴含的并行性,选择合适的策略将其并行执行,对提升程序的并行性能非常重要.通过判断循环携带依赖的依赖距离是可精确确定的常量还是变量,DOACROSS 循环可分为规则 DAOCROSS 循环和不规则 DAOCROSS 循环[5].与不规则 DOACROSS 循环相比,规则的 DOACROSS 循环因为携带依赖关系的规则性,更适合使用编译器实现自动并行.

流水并行是指将循环的各次迭代分配给不同的线程,线程间流水执行来获得并行性,迭代间的依赖通过某种方式的同步得到维持.流水并行方式是规则 DOACROSS 循环并行的重要方式,但流水并行循环要获得良好的性能,既需要保持并行线程之间的负载平衡,又需要控制同步开销在执行总开销中所占的比例[6],这要求循环在流水并行时选择恰当的计算划分方式和线程同步方式,以及合适的循环分块大小.上述流水并行的要求,使得自动生成性能良好的流水并行代码成为一件困难的事情,因此,并行编译器常常对 DOACROSS 循环作保守处理.如:开源编译器系统 Open64[7]的自动并行化模块能够识别出 DOALL 循环和 DOACROSS 循环,但是只对 DOALL 循环插入 OpenMP 编译指示实施自动并行,不处理 DOACROSS 循环;Intel 公司发布的商业编译器也提供了对串行程序中循环自动并行化的功能,但是只能够识别和并行 DOALL 循环,对包含循环携带依赖的 DOACROSS 循环当作串行循环处理.上述的编译器处理方式损失了程序中 DOACROSS 循环包含的并行性,从而限制了程序并行性能的进一步提升.

早期关于 DOACROSS 循环的研究主要集中于并行过程中使用的同步策略[3,8−12],还有一些研究工作专注于流水并行中的同步优化和流水计算粒度的优化[13−19].以上研究均是针对 DOACROSS 循环并行的某一方面问题展开,往往缺少对所需的同步开销、存储空间的说明和对所能获得性能提升的定量测试.文献[20]中,面向能够静态确定依赖关系的 DOACROSS 循环(即规则 DOACROSS 循环)中的单层并行性,提出了一种编译时和运行时相结合的自动并行化算法.该算法在实施并行变换前没有对进行计算划分的并行层进行选择,因此在面向多层循环时无法获得最大的性能加速.另外,该算法需要运行时的支持,限制了其应用的广泛性.

本文以多核处理器为目标平台,面向规则 DOACROSS 循环实现了基于 OpenMP[21]的循环流水并行代码的自动生成.首先设计了一种启发式算法,从规则 DOACROSS 循环中选择合适的计算划分层和循环分块层,尽量保证最大限度地挖掘循环中存在的并行性;然后,基于流水并行的代价模型给出一个循环分块大小的计算公式,用于选择并行时的最佳分块大小.代价模型能够将并行过程中对性能带来影响的多方面因素纳入考虑的范围,也更加利于流水计算粒度选择的进一步优化和扩展;最后,使用计数信号量实现并行线程之间的同步,避免了 OpenMP 中 barrier 同步方式简单、同步开销大的问题,并且将每个线程使用的同步变量个数限制为 1,优于文献[20]中的算法对同步变量数目的要求.本文在 Open64 编译器后端的循环嵌套优化遍(loop nest optimization,简称 LNO)[22]中实现了 OpenMP 流水并行源代码的自动生成,因而能够面向更多的平台,有更好的应用广泛性.通过

对有限差分松弛法(finite difference relaxation,简称 FDR)的波前(wavefront)循环[23]和时域有限差分法(finite difference time domain,简称 FDTD)[24]中典型循环以及程序 Poisson,LU 和 Jacobi 的测试表明:本文算法自动生成的流水并行代码能够在多核处理器上获得明显的性能提升;本文使用的流水分块大小计算公式能够较精确地计算出循环流水并行时的最佳分块大小.自动生成的流水并行代码与基于手工选择的最优分块大小的流水并行代码相比,加速比达到手工选择加速比的 89%.

1 流水并行模型

可并行循环分为 DOALL 循环与 DOACROSS 循环两种类型.DOALL 循环的各次迭代之间无依赖,可以按照任何顺序调度执行,即完全并行;DOACROSS 循环中包含循环携带依赖,因而无法完全并行.

对于 DOACROSS 循环,常常将循环的各次迭代分配给不同的线程,线程间进行流水并行.

流水并行通过线程之间某种方式的同步维持原有的跨迭代依赖,为存在跨迭代依赖的循环提供了并行的机会.同一线程两次同步之间的计算工作量大小称为流水计算粒度.因为流水并行中线程之间的同步较为频繁,所带来的开销较大,所以流水粒度的大小影响着循环的并行性能.根据流水粒度的大小,可将流水并行分为细粒度流水和粗粒度流水.

细粒度流水是指将被划分的循环层放置在循环嵌套的较内层,因此每次的计算量都很小,下一结点能够较快获得所需数据进行计算,有效减少结点的空闲等待时间.但由于同步延迟的存在,如果粒度太小,流水会引起较多的同步次数,从而导致较高的同步代价和较低的流水并行性能.粗粒度流水是指将计算划分的循环层放到循环嵌套的较外层,增大每个分块的计算量和同步数据量,因此后续的计算结点在开始计算之前需要更多的等待时间,但减少了同步的次数,同步代价相对较小.选择循环流水粒度的目标是平衡并行粒度和同步代价两者的关系,通常通过结合循环交换[23]和循环分块[23]这两种循环变换方式来完成.

2 流水并行代码的自动生成

本节讨论基于 OpenMP 实现 DOACROSS 循环的自动流水并行过程中要解决的问题,然后给出流水并行代码的自动生成算法.下面以第 2.1 节的图 1(a)所示的循环为例对各个问题进行说明,该循环是 FDR 的波前循环,包含 *i*,*j* 两层循环、4 个数组 *a* 的读引用和一个数组 *a* 的写引用.根据依赖关系分析的结果得出该循环的 *i* 层和 *j* 层都携带依赖,无法完全并行,是一个 DOACROSS 循环.

2.1 选择计算划分层和循环分块层

DOACROSS 循环的各层都携带依赖,导致无法完全并行.在流水并行时,被划分的循环层携带的数据依赖是形成流水的依据,这样的循环层被称为计算划分层.本文的代码自动生成算法中实现单层循环的计算划分.为提高并行程序的性能,在进行并行分解时常常增大并行粒度来减少通信和同步的代价[23].本节从循环的各层(除计算划分层)中选择一层实施循环分块来调整流水并行的粒度,以期获得最佳并行效果,该层称为循环分块层.

要生成高效的流水并行代码,就需要从循环各层中选择合适的计算划分层和循环分块层.但是,循环选择问题是非常困难的[23],因此,本文设计了一种启发式算法来选择流水并行时的计算划分层和循环分块层,并使用循环交换技术将它们交换到循环的最外层和次外层,以尽可能减少并行时的同步开销.该启发式算法思想如下:首先判断 DOACROSS 循环的层数是否大于等于 2,因为单层的 DOACROSS 循环很难获得并行性能,因而不实施自动并行;收集各循环层的相关信息,包括循环的上下界、循环携带依赖的依赖距离和依赖相关的数组;维持一个由候选循环层组成得的集合,初始时该集合中包含了循环的所有循环层,由循环层的索引表示,再依次按照下面列表的顺序,对各个条件进行判断,并将不符合以下条件的循环层从候选循环层集合中去除:

1) 循环层上下界的值在执行过程中不发生变化;
2) 循环层携带的依赖能够精确地确定依赖方向和依赖距离,即循环层是规则的 DOACROSS 循环;
3) 循环层对应的依赖方向向量中不包含“>”的依赖方向,能够往循环的外层进行交换;

4) 循环迭代次数/max(循环携带依赖的依赖距离)不小于 4.

经过以上条件的判断后,如果候选集中循环层的数目大于 1,则从中选择出涉及依赖的数组最少、依赖距离最小的循环层作为计算划分层,并将其交换到循环的最外层.

选出计算划分层后,对循环分块层进行选择.重新对候选循环层集合初始化,初始化后的集合中包含除计算划分层之外的所有层,然后依次按照下面列表的顺序对各个条件进行判断,并将不符合以下条件的循环层从集合中去除:

1) 循环上下界的值在执行过程中不发生变化;

2) 不包含“>”的依赖方向,能往循环的外层交换;

3) 循环迭代次数不小于 32.

经过以上条件的判断后,若满足条件的循环层(计算划分层已经选好,不包含在内)数目大于 1,则选择在原循环层顺序中靠近外层的循环层.在选择计算划分层和循环分块层时,都需要循环迭代次数的信息,如果静态编译时无法确定具体的值,可通过与程序员交互或是将循环预运行来获得.启发式算法按照以上要求选择流水并行时的计算划分层和循环分块层,如果无法选择出符合条件的循环层,则该循环无法进行流水并行,并行失败.

Code segment 1-1: Sequential form of example loops

```
1  do j=2, jend
2     do i=2, iend
3        a(i, j)=0.25*(a(i−1,j)+a(i,j−1)+
4                      a(i+1,j)+a(i,j+1))
5     end do
6  end do
```

(a)

Code segment 1-2: Pipelined form of example loops

```
1       include “omp_lib.h”
2       real a(iend,jend)
3       integer i, j
4       integer isync(0:256), mthreadnum, iam
5       common/threadinfo1/isync
6       common/threadinfo2/mthreadnum, iam
7    !$omp threadprivate(/threadinfo2/)
8    !$omp parallel default(shared) private(i,j) shared(a)
9        mthreadnum=1
10   !$   mthreadnum=omp_get_num_threads()
11      iam=1
12   !$   iam=omp_get_thread_num()+1
13      isync(iam)=0
14   !$omp barrier
15      do i=2, iend, b
16         call sync_left(iend,jend,a)
17   !$omp do schedule(static)
18         do j=2, jend
19            do ii=i, min((i+b−1),iend), 1
20               a(i,j)=0.25*(a(i−1,j)+a(i,j−1)+
21                            a(i+1,j)+a(i,j+1))
22            end do
23         end do
24   !$omp end do nowait
25         call sync_right(iend,jend,a)
26      end do
27   !$omp end parallel
```

(b)

Code segment 1-3: Subroutine *sync_left*(*di*,*dj*,*a*)

```
1       integer di, dj
2       real a(di,dj)
3       integer isync(0:256), mthreadnum, iam
4       common/threadinfo1/isync
5       common/threadinfo2/mthreadnum, iam
6    !$omp threadprivate(/threadinfo2/)
7       integer neighbour
8       if (iam.gt.0.and.iam.le.mthreadnum) then
9          neighbour=iam−1
10         do while (isync(neighbour).eq.0)
11   !$omp flush(isync)
12         end do
13         isync(neighbour)=0
14   !$omp flush(isync,a)
15      end if
16      return
17      end
```

(c)

Code segment 1-4: Subroutine *sync_right*(*di*,*dj*,*a*)

```
1       integer di, dj
2       real a(di,dj)
3       integer isync(0:256), mthreadnum, iam
4       common/threadinfo1/isync
5       common/threadinfo2/mthreadnum, iam
6    !$omp threadprivate(/threadinfo2/)
7       if (iam.lt.mthreadnum) then
8          do while (isync(iam).eq.1)
9    !$omp flush(isync)
10         end do
11   !$omp flush(isync,a)
12         isync(iam)=1
13   !$omp flush(isync)
14      end if
15      return
16      end
```

(d)

Fig.1 Serial and pipelining version of the wavefront loops of FDR

图 1 FDR 波前计算循环的串行版本和流水并行版本

现对启发式算法中的选择循环层的准则说明如下:计算划分层的条件 4)和循环分块层的条件 3)对循环迭代次数与依赖距离的比值和循环迭代次数本身给出了一个经验性的数值要求,当不符合条件时,循环一般无法获得并行收益.另外,选择计算划分层时选择涉及依赖数组少的循环层,能够减少线程之间数据同步的开销;选择依赖距离小的循环层,能够实现更多线程之间的流水执行,增强了并行的可扩展性.选择循环分块层时选择在原循环中最靠近外层的循环层,尽量保留内层,保持了循环执行时良好的数据局部性,有助于获得更大的性能提升.图 1(a)中,循环包含 *i*,*j* 两个循环层,并且都满足计算划分层和循环分块层的条件,因此选择 *j* 层为计算划分层,*i* 层为循环分块层.

2.2 计算划分

计算划分层选定之后,要选择一个计算划分和循环迭代调度方式,以保证并行时各线程之间不存在依赖环,能够流水执行.首先,使用 OpenMP 的 schedule 子句将调度方式指定为 static.在对循环迭代调度时,按照 OpenMP 中定义的 static 方式,循环的各次迭代分得一个从 0 到 $N-1$ 的逻辑编号,N 是循环包含的迭代次数,逻辑编号的顺序与循环串行执行时各次迭代的执行顺序一致.迭代空间按照线程组中包含的线程数目被等分成几个部分,每个线程分得一部分.因为各线程分得的是循环中连续的迭代空间,所以循环携带的依赖在各线程之间单向流动,不存在依赖环,满足流水并行的要求.

为说明自动流水并行时能够处理的流水并行的情况,将某一携带依赖的循环层划分给多个线程进行流水并行时,满足:

计算划分层迭代数/并行线程数≥max(循环携带依赖的依赖距离)

的流水并行称为规则流水并行.选择计算划分层时的条件 4)正是为了保证满足该要求.图 2 给出了规则流水并行与非规则流水并行的一个例子.

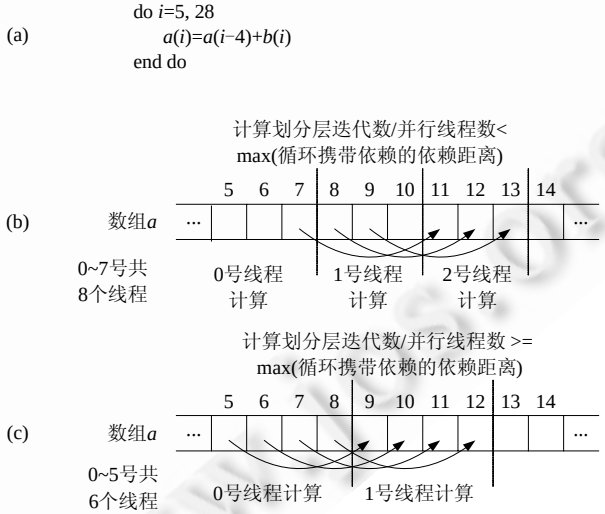

Fig.2 Regular pipelining and irregular pipelining

图 2 规则流水并行与非规则流水并行

为简单起见,图 2(a)给出了一个单层的 DOACROSS 循环.图 2(b)中将迭代划分给 8 个线程流水执行,每个线程分得 3 次迭代,不满足规则流水并行的条件.这时:2 号线程计算数组元素 *a*[11]时,需要与 0 号线程同步来获取 *a*[7]的值;计算 *a*[12]和 *a*[13]时,需要与 1 号线程同步来获取 *a*[8]和 *a*[9]的值.即,2 号线程在流水并行过程中需要与 0 号和 1 号两个线程进行左同步.图 2(c)中,1 号线程只需与 0 号线程同步来获取计算时所需的数组元素值,其他线程也是类似的,这样的流水并行是规则流水并行.本文在进行自动流水并行时只处理规则流水并行,因此要避免图 2(b)情况的出现.图 1(a)中循环的计算划分层 *j* 的依赖距离为 1,因此不会出现非规则流水并行的情况.

2.3 选择分块大小

流水并行较完全并行有更大的同步开销,单个循环迭代中的工作量往往不足以作为调度的单位.在这种情况下,把多个迭代组成集合,每个集合作为一个调度单位,能够更有效地利用并行性.循环分块技术能够实现这样的功能.本节讨论基于代价模型选择循环分块大小的问题,首先基于以下几点假设建立流水计算的模型:

1) 线程之间的同步是阻塞的,即同步完成前,不能进行其他的工作,类似于分布存储结构下的同步通信模式;
2) 线程数目为 p;
3) 流水循环是二维矩阵计算区域 $N_1\times N_2$,N_1 是计算划分层的迭代数,N_2 是循环分块层的迭代数;
4) 线程间调度采用 static 方式,每个线程分得的计算区域是$(N_1/p)\times N_2$;
5) 使用循环分块优化流水并行循环,调整计算代价与同步代价之间的比率,尽可能获得好的并行性和较低的同步代价,分块大小为 n_2;
6) 流水计算的调度:第 1 步,Thread0 计算分块 $B(0,0)$,进行右同步,Thread1 进行左同步;第 2 步,Thread0 和 Thread1 开始计算 $B(0,1)$和 $B(1,0)$;之后的步骤依此类推.整个流水计算过程如图 3 所示.

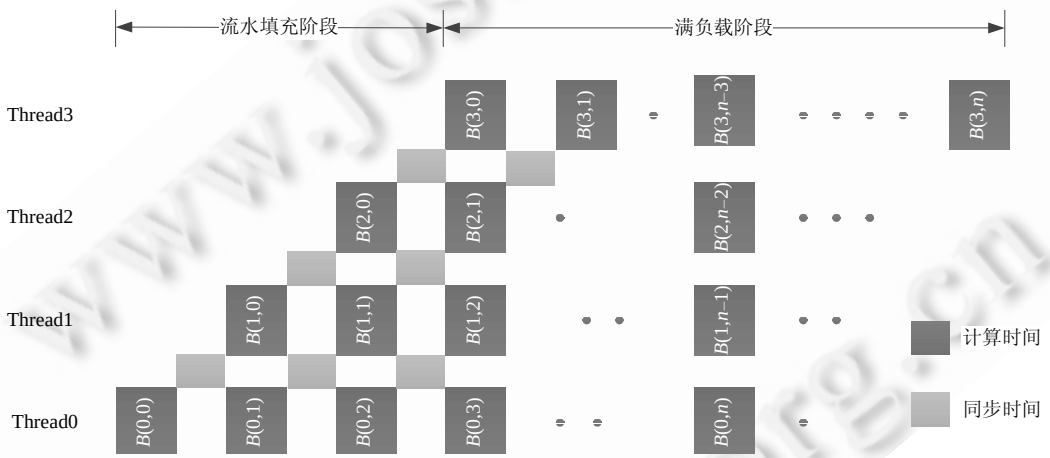

Fig.3 Process of pipeline computing

图 3 流水计算过程

流水并行过程中计算与同步是不可重叠的,因此,每一分块的执行时间是分块计算与同步时间的总和,即

$$T_{tile}=T_{comp}+T_{comm}=((N_1/p)\times n_2\times t_1)+t_2,$$

其中,t_1 是循环内层计算语句单次执行的时间,t_2 是单次线程同步的时间.整个流水执行时间描述为

$$T=(M_s+M_p)\times T_{tile},$$

其中,M_s 是流水填充阶段最后一个计算结点开始计算前的分块个数,其值为 $p-1$;M_p 是最后一个结点需计算的分块总数,其值为 $\frac{N_2}{n_2}$,流水计算的运行时间 T 具体表示为 $T=\left(p-1+\frac{N_2}{n_2}\right)\times((N_1/p)\times n_2\times t_1+t_2)$.对该式求解 $\frac{\mathrm{d}T}{\mathrm{d}n_2}$,得到最佳分块:$n_{2opt}=\sqrt{\frac{N_2\times t_2\times p}{N_1\times t_1\times(p-1)}}$,此时的流水并行时间最短.

对于图 1(a)中的循环,*jend*−1 对应上面公式中的 N_1,*iend*−1 对应于 N_2,p 可在运行时调用 OpenMP 的库函数 *omp_get_thread_num*()得到并存放在整型变量 *iam* 中,t_1 和 t_2 的值通过调用编译器中包含的代价模型和对基准测试集的测试得到.将这些值代入上面的公式中,就能够计算出流水并行时最优分块大小.

2.4 同步的实现

图 4(a)给出图 1(a)循环实施循环分块后的并行代码,代码中使用 *post* 和 *wait* 操作示意并行时的同步. *Post*(*EV*(*i*))操作发布事件 *EV*(*i*)已经发生的信号而 *wait*(*EV*(*i*))阻塞直至事件被发送.该流水并行循环的执行过程如图 4(b)所示.下面对图 4(b)中灰色背景的迭代块的左同步和右同步分别进行分析,并基于 OpenMP 来实现.

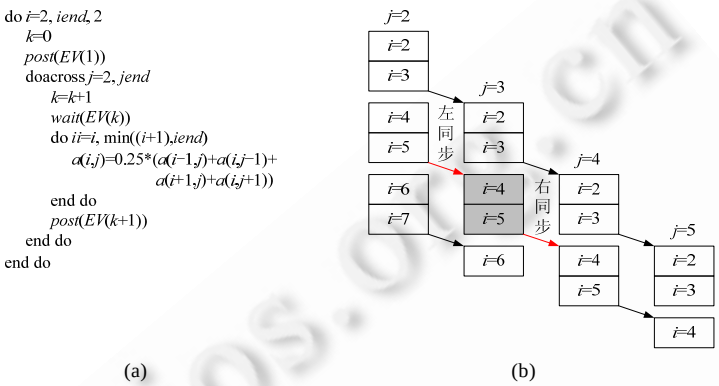

Fig.4 Pipelining with synchronization

图 4 流水并行同步

流水并行的正确性依赖于线程之间同步的正确性.OpenMP 中提供的 barrier 构造能够实现一组线程的栅障同步,组中的所有线程在 barrier 构造指示的同步点等待,直至最后一个线程到达.这样的线程同步方式代价较大,且无法满足流水同步方式的要求.流水并行时,相邻线程之间是典型的生产者-消费者关系,灰色迭代块要读取其左侧迭代块写入的数据,而它写入的数据要被其右侧迭代块读取,因此可使用计数信号量来实现同步.定义两个整型的计数信号量 *isync*1 和 *isync*2,分别用于灰色迭代块的左同步和右同步.实现左同步和右同步的代码段如下所示:

```
c 左同步代码段
c 等待左侧的线程完成计算并将isync1置为1
          do while( isync1 .eq. 0)
!$omp      flush(isync1)
          end do
c 将左侧线程置1的isync1置为0
          isync1 = 0
c 更新共享变量isync1，告诉左侧的线程a中本
c 次迭代的计算结果已被读取过
!$omp  flush(isync1, a)
```

```
c 右同步代码段
c 等待右侧线程将上一迭代更新的数据取出
          do while( isync2 .eq. 1)
!$omp     flush(isync2)
          end do
c 将本次迭代更新的数据写入共享内存
!$omp flush(isync2, a)
c 修改共享变量isync2
          isync2 = 1
c 更新共享变量isync2，告诉右侧的线程本
c 次迭代的数据已经更新过
!$omp flush(isync2)
```

由于 OpenMP 存储模型的弱一致性,当对计数信号量修改后,需要使用 *flush* 操作来保证修改能够被其他线程看到.为了将以上的左同步和右同步应用于流水并行执行时所有线程之间的同步,定义一个计数信号量数组,另外定义两个辅助变量,如下所示:

interger isync(0:256), *mthreadnum*, *iam*.

数组 *isync* 的元素初始化为 0,其值为 1 时,表明前一个线程已经完成一个迭代块的计算;其值为 0 时,表明后一个线程已经读取了上一个迭代块的计算结果.需要指出的是,流水时的第 1 个线程无需进行左同步,最后一个线程无需进行右同步.整型变量 *mthreadnum* 用于存放 *omp_get_num_threads* 库函数所取得的当前线程组中的线程总数,*iam* 用于存放当前线程在线程组中编号,可调用库函数 *omp_get_thread_num* 得到.

虽然本节定义的同步数组中包含 256 个元素,但根据以上同步代码可知,本节设计实现的流水并行同步方式仅为每个线程分配一个同步元素,同步过程中使用的同步元素数目与并行时的线程数目相等,因此在实际生

成代码时,根据可用的并行线程数定义包含相应数目元素的同步数组,对存储空间的要求低.图 1(a)中循环流水并行后的左同步子例程和右同步子例程分别如图 1(c)和图 1(d)所示.

2.5 算法实现

图 5 给出了完整的流水并行代码自动生成算法,该算法以规则 DOACROSS 循环 L 的中间表示作为输入,对其进行分析和变换,在并行成功的情况下,输出并行后循环 L 的中间表示;否则,循环 L 并行失败,照原样输出.图 1(a)中循环经过自动流水并行算法处理后,得到的并行代码如图 1(b)~图 1(d)所示.

```
1  procedure Automatic_Pipeline(L)
2  //功能:实现循环的OpenMP自动流水并行
3  //输入:一个由中间语言表示的DOACROSS循环L;
4  //输出:parallelized;当parallelized为TRUE时,表示循环已并行,L中存放并行后
5          循环的中间表示;为FALSE时,表示没有并行,L中存放原本的循环.
6
7     parallelized=FALSE;
8     使用循环选择算法从循环L中选出计算划分层l1和循环分块层l2;
9     if l1==NULL ||l2==NULL then
10       return parallelized;
11    else begin
12       将循环层l1打上标记,并插入OpenMP的Loop构造指示对该层进行计算划分;
13       使用代价模型计算循环层的最佳分块大小;
14       往循环L中插入实现线程同步所需的变量的定义和初始化;
15       根据计算划分层携带依赖的信息,生成左同步子例程和右同步子例程;
16       parallelized=TRUE;
17       return parallelized;
18    end
19 end Automatic_Pipeline
```

Fig.5 Algorithm of automatic generation of pipelining code

图 5 流水并行代码自动生成算法

3 实验结果与分析

本节对使用本文算法生成的流水并行循环的性能和算法中使用的循环分块计算公式的有效性进行测试.测试平台为 IBM x3650 系列的服务器,其中包含 4 个 Intel Xeon X5670 CPU,每个 Intel Xeon X5670 CPU 中包含 6 个主频为 2.93GHz 的处理器核,内存为 40GB,使用的操作系统为 Red hat Enterprise 5.5.

本节选择 FDR 的波前循环和求解电磁问题的 FDTD 的典型循环作为测试用例.物理学和其他学科领域的许多问题在被分析研究之后,往往归结为偏微分方程(partial differential equation,简称 PDE)的求解问题.直接求解 PDE 比较困难,所以常常使用有限差分法(finite differential method,简称 FDM)来实现 PDE 的数值求解.在 PDE 中,用差商代替偏导数,得到相应的差分方程,通过解差分方程得到 PDE 的近似解.FDM 被广泛应用于诸如气象预报、流体力学的模拟、弹力力学等研究领域.FDM 中采用各种迭代法,如点逐次超松弛方法、迭代的交替方向隐式方法等,求解差分方程组,其中蕴含大量的 DOACROSS 并行性,适用于 DOACROSS 循环自动并行的测试.本节使用的两个测试用例均属于 FDM 的迭代求解法.

3.1 并行性能测试

3.1.1 对 FDR 的测试

经过并行识别后,FDR 的波前循环被标记为 DOACROSS 循环,经本文算法自动并行后,得到如图 1(b)~图 1(d)中所示的并行代码.为充分测试本文算法的有效性,选择了 256×256,512×512 和 1024×1024 这 3 种规模,分别测试在这 3 种规模下自动流水并行生成的循环代码的加速比,如图 6 所示.

图 6 的测试结果显示:

1) 自动生成的 FDR 波前循环的流水并行版本在不同迭代规模和不同线程数目时都能够获得明显的性能加速;
2) 线程数目相等时,循环迭代规模越大,获得的性能加速越明显.这是因为循环迭代规模越大,线程两次

同步之间的计算量越大,计算时间也就越长,同步开销占总时间开销的比例越小,因而加速效果越明显;但随着规模的增大,加速比增长趋势变缓.因为当规模大到一定程度,并行同步的开销在总开销中所占的比例非常小,对循环的加速比带来的影响也很小;

3) 对于循环迭代的相同规模,加速比随着线程数目的增加而增大;但随着线程数目的增大,加速比增长趋势变缓.因为流水并行过程由流水填充和满负载两个阶段组成,随着并行线程数的增加,流水填充阶段也越长,满负载阶段越短,因此,由增加并行线程数带来的性能提升不会无止境地上升,而是趋近于一个上界;

4) 不同线程数目下的平均加速比均能达到最大线性加速比的 67%.

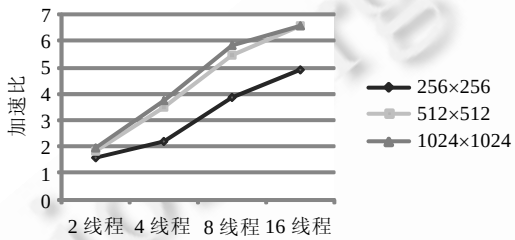

Fig.6 Speedup of the wavefront loops of FDR

图 6 FDR 波前循环的加速比

3.1.2 对 FDTD 的测试

FDTD 典型循环的测试结果如图 7 所示,其串行代码和本文算法自动生成的流水并行代码分别如图 8(a)和图 8(b)所示.因为该循环与 FDR 波前循环相比,单次迭代完成的工作量要大得多,所以选择较小的循环规模 128×128 来替代规模 1024×1024,另外两个循环规模不变.

图 7 的测试结果显示:

1) 自动生成的流水并行代码能够在测试平台上获得明显的性能提升;

2) 与 FDR 波前循环的测试结果不同,FDTD 典型循环的性能加速在并行线程数目相等的情况下,随着循环迭代规模的增大而呈下降趋势;

3) 对于循环迭代的相同规模,加速比随着并行线程数目的增加呈增长趋势且增势渐缓;

4) 在各线程数目下的平均加速比均能达到最大线性加速比的 71%.

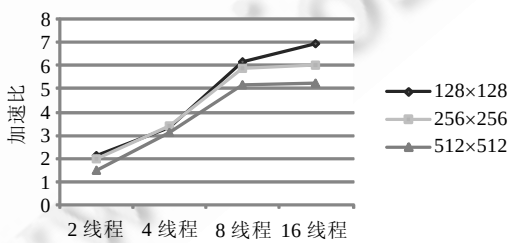

Fig.7 Speedup of representative loops of FDTD

图 7 FDTD 典型循环的加速比

Code segment 8-1: Sequential form of example loops in FDTD

```
1  do i=2, iend−1
2    do j = 2, jend−1
3      do k=2, kend−1
4      invmu=1.0/mu(i,j,k)
5      tmpx=rx*invmu
6      tmpy=ry*invmu
7      tmpz=rz*invmu
8      hx(i,j,k)=hx(i,j,k)+tmpz*(ey(i,j,k+1)−ey(i,j,k))−
9                          tmpy*(ez(i,j+1,k)−ez(i,j,k))
10     hy(i,j,k)=hy(i,j,k)+tmpx*(ez(i+1,j,k)−ez(i,j,k))−
11                         tmpz*(ex(i,j,k+1)−ex(i,j,k))
12     hz(i,j,k)=hz(i,j,k)+tmpy*(ex(i,j+1,k)−ex(i,j,k))−
13                         tmpx*(ey(i+1,j,k)−ey(i,j,k))
14     end do
15     do k=2, kend−1
16     invep=1.0/ep(i,j,k)
17     tmpx=rx*invep
18     tmpy=ry*invep
19     tmpz=rz*invep
20     ex(i,j,k)=ex(i,j,k)−tmpz*(hy(i,j,k)−hy(i,j,k−1))+
21                        tmpy*(hz(i,j,k)−hz(i,j−1,k))
22     ey(i,j,k)=ey(i,j,k)−tmpx*(hz(i,j,k)−hz(i−1,j,k))+
23                        tmpz*(hx(i,j,k)−hx(i,j,k−1))
24     ez(i,j,k)=ez(i,j,k)−tmpy*(hx(i,j,k)−hx(i,j−1,k))+
25                        tmpx*(hy(i,j,k)−hy(i−1,j,k))
26     end do
27   end do
28 end do
```

(a)

Code segment 8-2: Pipeline form of example loops in FDTD

```
1  !$omp  parallel default(shared) private(i,j,ii,k,invmu,invep
2  !$omp&  tmpx,tmpy,tmpy) shared(hx,hy,hz,ex,ey,ez,mu,ep)
3        mthreadnum=1
4  !$    mthreadnum=omp_get_num_threads()
5        iam=1
6  !$    iam=omp_get_thread_num()+1
7      isync(iam)=0
8  !$omp barrier
9  do i=2, iend−1, b
10   do j=2, jend−1
11     call sync_left(iend,jend,kend,hx,hz)
12 !$omp do schedule(static)
13     do ii=i, min((i+b−1),(iend−1))
14       do k=2, kend−1
15       invmu=1.0/mu(i,j,k)
16       tmpx=rx*invmu
17       tmpy=ry*invmu
18       tmpz=rz*invmu
19       hx(i,j,k)=hx(i,j,k)+tmpz*(ey(i,j,k+1)−ey(i,j,k))−
20                          tmpy*(ez(i,j+1,k)−ez(i,j,k))
21       hy(i,j,k)=hy(i,j,k)+tmpx*(ez(i+1,j,k)−ez(i,j,k))−
22                          tmpz*(ex(i,j,k+1)−ex(i,j,k))
23       hz(i,j,k)=hz(i,j,k)+tmpy*(ex(i,j+1,k)−ex(i,j,k))−
24                          tmpx*(ey(i+1,j,k)−ey(i,j,k))
25       end do
26       do k=2, kend−1
27       invep=1.0/ep(i,j,k)
28       tmpx=rx*invep
29       tmpy=ry*invep
30       tmpz=rz*invep
31       ex(i,j,k)=ex(i,j,k)−tmpz*(hy(i,j,k)−hy(i,j,k−1))+
32                          tmpy*(hz(i,j,k)−hz(i,j−1,k))
33       ey(i,j,k)=ey(i,j,k)−tmpx*(hz(i,j,k)−hz(i−1,j,k))+
34                          tmpz*(hx(i,j,k)−hx(i,j,k−1))
35       ez(i,j,k)=ez(i,j,k)−tmpy*(hx(i,j,k)−hx(i,j−1,k))+
36                         tmpx*(hy(i,j,k)−hy(i−1,j,k))
37       end do
38     end do
39 !$omp end do nowait
40       call sync_right(iend,jend,kend,hx,hz)
41   end do
42 end do
43 !$omp end parallel
```

(b)

Fig.8 Serial and pipelining version of representative loops of FDTD

图 8 FDTD 典型循环的串行版本和流水并行版本

通过对 FDTD 典型循环自动并行过程的分析发现,在 256×256 和 512×512 两个规模时,由第 2.3 节的公式计算出的最佳分块大小均小于 1,因此取分块大小为 1 时获得的并行性能并不是理论上的最佳并行性能,这是出现上述第 2 条结果的原因.

对 FDR 波前循环和 FDTD 典型循环的测试表明,经本文算法生成的流水并行循环能够在多核平台上获得良好的并行性能.

3.1.3 与 XL Fortran13.1 的对比测试

与本节工作最为接近的是 Unnikrishnan 等人在文献[20]中提出的 DOACROSS 循环自动并行算法.他们在 XL Fortran 13.1 编译器中实现了文献[20]中的算法,并且给出了 Poisson,LU,SOR 和 Jacobi 这 4 个程序的测试结果.因为 SOR 和 Jacobi 两个程序核心循环的结构非常相似,本节选择 Poisson,LU 和 Jacobi 这 3 个程序对包含

DOACROSS 自动并行算法的 Open64 编译器的自动并行性能(包括 DOALL+DOACROSS 自动并行)进行测试.图 9~图 11 分别给出了 3 个程序经添加本文算法的 Open64 编译器并行后的性能加速,同时给出了 XL Fortran 13.1 编译器的 DOALL 并行和 DOALL+DOACROSS 并行性能作为对比.

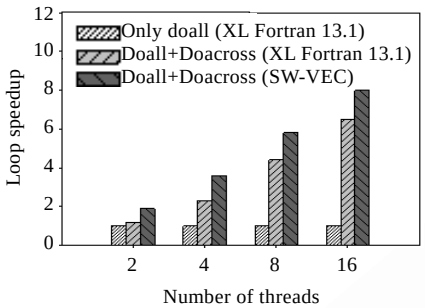

Fig.9 Speedup of Poisson

图 9 Poisson 的加速比

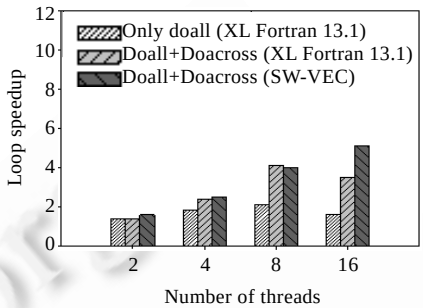

Fig.10 Speedup of LU

图 10 LU 的加速比

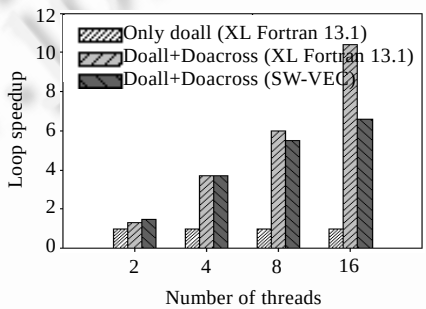

Fig.11 Speedup of Jacobi

图 11 Jacobi 的加速比

程序 Poisson 的自动并行性能对比如图 9 所示,经 Open64 编译器自动并行化后,Poisson 能够获得显著的性能提升,在 16 线程时的加速比为 8.0,优于 XL Fortran 13.1 编译器.自动并行方法中设计了一种启发式算法,用于计算划分层和循环分块层的选择,尽量保证最大限度地挖掘循环中存在的并行性;但文献[20]的方法中没有进行该项工作,因此在面向类似 Poisson 这样的多层嵌套循环时无法获得最佳的性能加速.

程序 LU 中在 blts 和 buts 子例程中包含两个 DOACROSS 循环,经 Open64 编译器自动并行后,能够获得显著的性能提升,在 16 线程时的加速比为 5.1,如图 10 所示.另外,图 10 的性能对比表明,程序 LU 经 Open64 编译器自动并行后的总体性能加速优于 XL Fortran 13.1 编译器.

程序 Jacobi 经 Open64 编译器自动并行后,在线程数目较少时的性能提升与 XL Fortran 13.1 编译器基本一致,但随着线程数目的增大,XL Fortran 13.1 编译器所获得的自动并行性能更优,如图 11 所示.出现这种现象主要是因为 Jacobi 中的 DOACROSS 核心循环的循环体工作量较小,而基于 OpenMP 中 flush 操作的同步方式与文献[20]中的同步方式相比开销较大,因此需要通过循环分块来增加流水计算粒度,给循环的并行性带来部分损失;随着线程数目的增加,这种损失愈加明显.因此,随着线程数目的增加,所获得的性能提升不及 XL Fortran 13.1 编译器.

以上测试结果表明,包含 DOACROSS 自动并行算法的 Open64 编译器能够有效地对程序实施并行,并行后代码能够获得明显的性能提升,在大多数情况下,性能优于 XL Fortran 13.1 编译器.

3.2 分块大小选择测试

本节分别测试循环自动流水并行和使用手工选择最优分块大小时的并行加速比,将两个加速比进行对比.对于 FDR 波前循环,选择 512×512 的迭代规模,对于 FDTD 典型循环,选择 128×128 的迭代规模.测试结果如图 12 和图 13 所示.

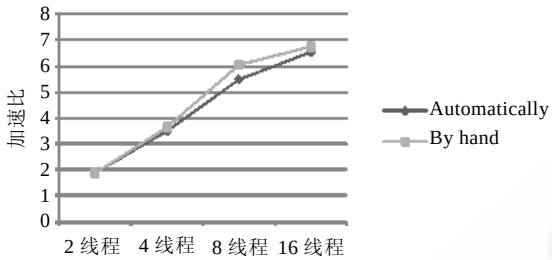

Fig.12 Speedup obtained automatically and by hand of the wavefront loops of FDR

图 12 FDR 波前循环的分块大小选择测试结果

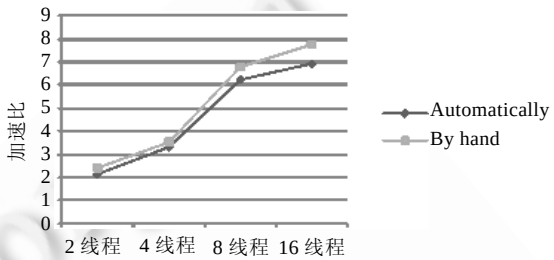

Fig.13 Speedup obtained automatically and by hand of representative loops of FDTD

图 13 FDTD 典型循环的分块大小选择测试结果

根据图 12 和图 13 的测试结果,本文算法生成并行程序与手工选择分块大小的程序相比,FDR 波前循环的加速比在不同线程数目下均能达到手工加速比的 90%,在 2 线程时为 98.2%;FDTD 典型循环的加速比在不同线程数目下能够达到手工加速比的 89%,在 4 线程时最为接近,为 95.0%.上述测试结果表明,本文使用的分块大小计算公式基本能够选择合适的分块大小,自动生成的并行代码能够在测试平台上获得明显的性能提升.

4 相关研究

早期关于 DOACROSS 循环的研究主要集中于并行过程中使用的同步策略.Cytron[3]研究了如何根据流水并行时各迭代之间的迟延对 DOACROSS 循环进行调度,以维持循环中存在的依赖.Padua 和 Midkiff[8]关注于保证单嵌套 DOACROSS 循环中循环携带依赖的同步策略,他们为循环中的每个数据依赖使用一个同步变量,但没有考虑多维循环的情况.Wolfe[91]研究了 4 种不同的同步机制,包括在循环中每一个存在数据依赖的位置进行同步,将循环划分为多个流水执行的语句段,将栅障同步插入循环中的各同步点和使用有序的关键区域,但仍然只考虑了单嵌套循环的情况.Su 和 Yew[10]面向流水并行提出了几种同步策略,其中,面向数据(data-oriented)的策略为循环中依赖所涉及的每一个数据使用一个同步变量,面向语句(statement-oriented)和面向过程(process-oriented)的策略则分别为每个语句和每次迭代使用一个同步变量.他们考虑了嵌套循环中的单层并行性和嵌套并行性,但是没有给出实验结果.Li[11]给出了一个基于数组下标和循环边界生成同步代码的算法,该算法中不要求数据依赖的距离为常量,因而能够处理任意嵌套循环.Tang 等人[14]提出了一个同步策略,能够对包含复杂的迭代间数据依赖的一般嵌套循环实施并行.同步策略的选择、实现和优化对 DOACROSS 循环并行方法的实用性有重要影响.上述文献[3,8−12]提出了很多有用的同步策略,但是没有对所需的同步开销、存储空间进行说明或是缺少对所能获得性能提升的定量测试.

还有一些研究工作专注于同步的优化.Krothapalli[13]通过消除包含常量依赖的简单循环中的冗余依赖实现了冗余同步的消除.Rajamony 和 Cox[14]使用线性规划得到 DOACROSS 循环中的同步最小化,同时保持循环的并行性.Chen 和 Yew[15−17]针对同步优化问题进行了一系列的研究.文献[15]中提出了一种编译优化策略,通过对 DOACROSS 循环实施语句重排序,以达到并行性最大化和同步最小化的目标.文献[16]中提出了一种静态调度策略和一个对应的同步策略,使用依赖一致化(dependence uniformization)技术使用一组一致(uniform)的依赖向量对 DOACROSS 循环中可能存在的所有依赖进行覆盖.文献[17]对包含多条语句和控制流分支的循环嵌套中存在的冗余同步进行标识,从而达到冗余同步消除的目的.

面向静态无法确定依赖关系的 DOACROSS 循环存在一些运行时并行的研究.Chen 等人[25]面向编译时无法确定依赖关系的情况,提出了实现 DOACROSS 并行的运行时算法.Jeyaraman 和 Krothapalli[26]提出的运行时并行方法能够处理各种类型的数据依赖,且不要求有任何特殊的硬件支持,在 inspector 阶段不要求进行同步,在 executor 阶段有时需要同步操作的支持.Xu 和 Chaudhary[27]开发了一种时间戳算法,实现包含间接数组访问的 DOACROSS 循环的运行时并行.

为了调整同步开销占循环执行时间的比例,需要对流水计算粒度进行优化.Pan 等人[18]使用循环分块技术增加并行的粒度,提出了计算最优分块大小的公式,结论是:与动态调度相比,静态调度策略往往具有更优的分块间数据局部性,因此能够获得更优的性能.Lowenthal[19]给出了一个运行时确定流水并行粒度的方法,该方法使用运行时的信息构建循环的执行模型,从而选择一个合适的分块大小.

以上研究或是针对 DOACROSS 循环并行的某一方面问题展开研究,或是面向静态无法确定依赖关系的 DOACROSS 循环进行运行时的并行.本文提出了一种完整的编译时 DOACROSS 自动并行化算法,在解决自动并行的各个子问题时,对以上文献中提出的方法进行了借鉴.

与本文工作最为接近的是 Unnikrishnan 等人在文献[20]提出的 DOACROSS 循环自动并行算法,文献[20]中只考虑单层的并行性和能够静态确定依赖关系的 DOACROSS 循环(即规则 DOACROSS 循环),提出了编译时和运行时相结合的优化方法,称为依赖打包(dependence folding).该方法使用一个保守依赖(conservative dependence)表示循环携带的所有依赖,从而将每个线程使用的同步变量个数限制在与嵌套循环的层数相等;另外,文献[20]还提出了一个收益分析方法来指定循环展开和循环分块,选择最优的流水计算粒度.

与文献[20]类似,本文面向的也是规则DOACROSS循环中的单层并行性,但与之相比,本文算法有以下几个特点:

首先,本文中设计了一个启发式算法从循环嵌套的各层中选择合适的计算划分层和循环分块层,尽量保证最大限度地挖掘循环中存在的并行性.文献[20]中没有进行该项工作,因此在面向多层并行循环时无法获得最大的性能加速,正如文献[20]中对 Poisson 测试程序的性能测试结果所表明的;

其次,本文的同步策略将每个线程使用的同步变量个数限制为 1,优于文献[20]中的算法对同步变量数目的要求;

再次,本文对流水计算粒度的选择基于流水并行的代价模型,因而能够将并行过程中对性能带来影响的多方面因素纳入考虑的范围,也更加利于流水计算粒度选择的进一步优化和扩展;

最后,文献[20]中的算法需要运行时的支持.本文实现了 OpenMP 并行源程序的自动生成,能够面向更多的平台,因此有更好的应用广泛性.

5 结束语

本文实现了基于 OpenMP 的规则 DOACROSS 循环的流水并行代码的自动生成.首先介绍流水并行模型,再详细讨论循环流水并行时要解决的几个问题,并给出流水并行代码的自动生成算法.实验结果表明,使用本文算法自动生成的流水并行代码能够在多核处理器上显著提升性能.下一步工作的重点是针对单次迭代包含较大工作量的循环,调整流水并行时的计算划分方式和循环分块选择,以获得更高的并行性能.

致谢 在此,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.

References:

[1] Benoit A, Melhem R, Renaud-Goud P, Robert Y. Power-Aware manhattan routing on chip multiprocessors. In: Proc. of the 26th Int'l Parallel and Distributed Processing Symp. (IPDPS 2012). Washington: IEEE Computer Society, 2012. 189–200. [doi: 10.1109/IPDPS.2012.27]

[2] Jin H, Jespersen D, Mehrotra P, Biswas R, Huang L, Chapman B. High performance computing using MPI and OpenMP on multi-core parallel systems. Parallel Computing, 2011,37(9):562–575. [doi: 10.1016/j.parco.2011.02.002]

[3] Cytron R. Doacross: Beyond vectorization for multiprocessors. In: Proc. of the Int'l Conf. on Parallel Processing. 1986. 836−844.

[4] Chen DK, Yew PC. An empirical study on DOACROSS loops. In: Proc. of the Supercomputing. New York: ACM Press, 1991. 620−632.

[5] Hurson AR, Lim JT, Kavi KM, Lee B. Parallelization of DOALL and DOACROSS loops—A survey. Advances in Computers, 1997, 45:53−103. [doi: 10.1016/S0065-2458(08)60706-8]

[6] Ma L. Tuning pipeline granularity based on feedback directed framework [MS. Thesis]. Beijing: Institute of Computing Technology of Chinese Academy of Sciences, 2005 (in Chinese with English abstract).

[7] Lin YT, Wang SC, Shih WL, Hsieh BKY. Enable OpenCL compiler with Open64 infrastructures. In: Proc. of the 13th IEEE Int'l Conf. on High Performance Computing and Communications (HPCC 2011). Washington: IEEE Computer Society, 2011. 863−868. [doi: 10.1109/HPCC.2011.123]

[8] Midkiff SP, Padua DA. Compiler algorithms for synchronization. IEEE Trans. on Computers, 1987,C-36(12):1485−1495. [doi: 10.1109/TC.1987.5009499]

[9] Wolfe M. Multiprocessor synchronization for concurrent loops. IEEE Software, 1988,5(1):34−42. [doi: 10.1109/52.1992]

[10] Su HM, Yew PC. On data synchronization for multiprocessors. In: Proc. of the 16th Annual Int'l Symp. on Computer Architecture. New York: ACM Press, 1989. 416−423. [doi: 10.1145/74925.74972]

[11] Li ZY. Compiler algorithms for event variable synchronization. In: Proc. of the 5th Int'l Conf. on Supercomputing. New York: ACM Press, 1991. 85−95. [doi: 10.1145/109025.109051]

[12] Tang P, Yew PC, Zhu CQ. Compiler techniques for data synchronization in nested parallel loop. In: Proc. of the 4th Int'l Conf. on Supercomputing. New York: ACM Press, 1990. 177−186. [doi: 10.1145/255129.255155]

[13] Krothapalli VP, Sadayappan P. Removal of redundant dependences in doacross loops with constant dependences. IEEE Trans. on Parallel and Distributed Systems, 1991,2(3):281−289. [doi: 10.1109/71.86104]

[14] Rajamony R, Cox AL. Optimally synchronizing doacross loops on shared memory multiprocessors. In: Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques. Washington: IEEE Computer Society, 1997. 214−224. [doi: 10.1109/PACT.1997.644017]

[15] Chen DK, Yew PC. Statement re-ordering for DOACROSS loops. In: Proc. of the Int'l Conf. on Parallel Processing. Washington: IEEE Computer Society, 1994. 24−28. [doi: 10.1109/ICPP.1994.186]

[16] Chen DK, Yew PC. On effective execution of non-uniform DOACROSS loops. IEEE Trans. on Parallel and Distributed Systems, 1996,7(5):463−476. [doi: 10.1109/71.503771]

[17] Chen DK, Yew PC. Redundant synchronization elimination for DOACROSS loops. In: Proc. of the 8th Int'l Parallel Processing Symp. Washington: IEEE Computer Society, 1994. 477−481. [doi: 10.1109/IPPS.1994.288260]

[18] Pan ZL, Armstrong B, Bae H, Eigenmann R. On the interaction of tiling and automatic parallelization. In: Proc. of the OpenMP Shared Memory Parallel Programming. Berlin, Heidelberg: Springer-Verlag, 2008. 24−35. [doi: 10.1007/978-3-540-68555-5_3]

[19] Lowenthal DK. Accurately selecting block size at runtime in pipelined parallel programs. Int'l Journal of Parallel Programming, 2000,28(3):245−274. [doi: 10.1023/A:1007577115980]

[20] Unnikrishnan P, Shirako J, Barton K, Chatterjee S, Silvera R, Sarkar V. A practical approach to DOACROSS parallelization. In: Proc. of the Euro-Par 2012. LNCS, Berlin, Heidelberg: Springer-Verlag, 2012. 219−231. [doi: 10.1007/978-3-642-32820-6_23]

[21] Thoman P, Jordan H, Pellegrini S, Fahringer T. Automatic OpenMP loop scheduling: A combined compiler and runtime approach. In: Proc. of the 8th Int'l Workshop on OpenMP (IWOMP 2012). Berlin, Heidelberg: Springer-Verlag, 2012. 88−101. [doi: 10.1007/978-3- 642-30961-8_7]

[22] Ramshankar R. X86 Open64 compiler suite. 2009. http://developer.amd.com/tools/cpu/open64/Documents/open64_ compiler_ developer_guide.html

[23] Allen R, Kennedy K. Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann Publishers, 2001. 63−68.

[24] Taflove A, Hagness SC. Computational Electrodynamics. 3rd ed., Artech House Publishers, 1995. 1−9.

[25] Chen DK, Oesterreich DA, Torrellas J, Yew PC. An efficient algorithm for the run-time parallelization of doacross loops. In: Proc. of the Supercomputing. Washington: IEEE Computer Society, 1994. 518−527. [doi: 10.1109/SUPERC.1994.344315]

[26] Jeyaraman T, Krothapalli VP, Giesbrecht M. Run-Time parallelization of irregular doacross loops. Lecture Notes in Computer Science, 1995,980:75−80. [doi: 10.1007/3-540-60321-2_5]

[27] Xu CZ, Chaudhary V. Time stamp algorithms for runtime parallelization of DOACROSS loops with dynamic dependences. IEEE Trans. on Parallel and Distributed Systems, 2001,12(5):433−450. [doi: 10.1109/71.926166]

附中文参考文献:

[6] 马琳.反馈指导的流水计算性能调优[硕士学位论文].北京:中国科学院计算技术研究所,2005.

刘晓娴(1985－),女,江西宜丰人,博士生,主要研究领域为并行编译.
E-mail: xiaoxian0321@gmail.com

赵捷(1987－),男,博士生,主要研究领域为并行编译.
E-mail: zjbc2005@163.com

赵荣彩(1957－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为并行编译,高性能计算,反编译技术.
E-mail: rczhao126@126.com

徐金龙(1985－),男,博士生,主要研究领域为并行编译.
E-mail: longkaizh@126.com