格子玻尔兹曼方法计算程序的循环优化技术研究

崔元桢 刘 松 王 倩 伍卫国

(西安交通大学计算机科学与技术学院 西安 710049)

摘 要 格子玻尔兹曼方法(Lattice Boltzmann Method,LBM)在计算流体力学领域中得到广泛应用,但传统的LBM计算程序耗时巨大,如何优化LBM的计算程序具有重要研究意义.现有的优化方法较少关注LBM计算程序中时间步迭代中潜在的大量数据重用收益,造成计算性能损失.本文通过对LBM计算程序核心循环代码进行循环优化,将其巨大的迭代空间划分成满足cache容量的分块,从而提高数据重用性,同时开发粗粒度循环并行性.分块大小在对迭代空间划分时起到了影响性能的关键作用.本文根据LBM的程序特征提出了一种混合的分块大小选择方法——LBM_TSS方法.该方法从LBM计算程序的访存行为、局部性收益、并行效率以及同步开销四个方面进行静态分析,在约束条件限定的搜索空间内进一步对分块大小寻优,从而计算出性能最优的分块大小.本文在一个共享内存多核系统上对LBM_TSS方法的有效性进行了全面的验证和分析.实验结果表明,在最优情况下,采用LBM_TSS方法计算的分块大小所实现的LBM循环优化方法,与其他3种LBM并行优化方法相比,将LBM程序性能提高了16.79%.

关键词 格子玻尔兹曼方法;局部性优化;并行优化;分块大小选择

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2020.01086

Research on Loop Optimization for Lattice Boltzmann Method Computation Program

CUI Yuan-Zhen LIU Song WANG Qian WU Wei-Guo

(*School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an* 710049)

Abstract The Lattice Boltzmann Method (LBM) is widely used in the field of computational fluid dynamics. However, the traditional LBM computation is time-consuming. Therefore, it has great significance to optimize LBM computation program. The existing algorithms pay little attention to the potential large amount of data reuse benefits in time step iterations of the LBM computation program, resulting in the loss of performance. This paper performs loop optimizations on the core loops of LBM computation program to divide the huge iteration space into blocks or tiles that match the cache capacity, thereby improving data reuse and developing coarse-grained loop parallelism. The tile size has an important impact on program performance when partitioning the iteration space. A hybrid tile size selection method, LBM_TSS, is proposed, based on the program characteristics. LBM_TSS performs static analysis from the four aspects of LBM computation program's memory access behavior, locality benefits, parallel efficiency and synchronization cost to construct constraints of tile sizes. Then, LBM_TSS performs empirical search to tune the best

收稿日期:2019-08-30;在线出版日期:2020-02-12.本课题得到国家重点研发计划(2017YFB0203003)、国家自然科学基金(91630206,61672423)资助.**崔元桢**,硕士,主要研究方向为并行计算和程序优化.E-mail:cuiyuanzhen@stu.xjtu.edu.cn.**刘 松**,博士,助理研究员,中国计算机学会(CCF)会员,主要研究方向为编译优化和高性能计算.**王 倩**,硕士,主要研究方向为程序优化和高性能计算.**伍卫国**(通信作者),博士,教授,博士生导师,中国计算机学会(CCF)会员,主要研究领域为高性能计算机体系结构、存储系统和嵌入式系统.E-mail:wgwu@xjtu.edu.cn.

tile size in the constrained search space. The effectiveness of LBM_TSS is fully verified and analyzed on a shared memory multi-core system. Experimental results show that our loop optimization method with the tile size calculated by LBM_TSS improves the performance of LBM program by 16.79% at best, compared with other three LBM parallel optimization methods.

Keywords　LBM; locality optimization; parallel optimization; tile size selection

1 引　言

格子玻尔兹曼方法(Lattice Boltzmann Method, LBM)[1]是一种介于流体的分子动力学模型(微观层面)和连续模型(宏观层面)之间的介观模型,是计算流体力学[2]的重要方法之一.由于 LBM 可以显式求解线性的问题,具有天然的高度可并行性,在相关研究领域的计算机仿真中具有广泛应用.例如,大型客机气动噪声就可以使用 LBM 进行大规模模拟,进而解决一些关键的科学问题.包含 LBM 在内的大多数计算流体力学问题,其核心逻辑的数学表现形式往往是存在复杂内在联系的大型线性代数方程组[2],而其编程实现的计算程序则是由大量带嵌套循环代码的基本线性代数子程序和模板计算构成.从计算机科学的角度分析,这些科学计算程序绝大多数的 CPU 时间耗费在多层嵌套循环代码上,嵌套循环代码也因此成为这些科学计算应用程序的热点,是优化这类科学计算应用程序运行性能的关键.

针对上述问题,循环优化技术(loop optimization)[3-4]提供了一种有效的解决方案,它包含了循环融合、颠簸、偏斜、展开、分块等多种循环变换(loop transformations)方法,通过对程序进行等价变换来开发循环代码的数据局部性和并行性.一方面,循环优化技术可以利用仿射变换技术,在不改变循环体源代码的情况下,改变循环迭代空间的访存次序,充分利用宝贵的 cache 资源,尽可能地让迭代访存数据在 cache 中得到充分的重用后才会被替换出去,从而有效减小高时延访存代价,提高程序访存的局部性,缓解高性能计算中普遍存在的"存储墙"问题.另一方面,循环优化技术也可以开发程序循环级的粗粒度并行性,如循环优化技术中的循环分块方法可以将嵌套循环的迭代空间划分成多个分块,在满足数据依赖关系的基础上,实现循环分块的并行,而且通过循环优化技术也可以开发循环代码的向量化能力,实现 SIMD 性能.值得一提的是,有研究表明高性能计算的能耗主要来自于大规模的数据移动[5],循环优化技术可以有效提高 cache 利用率,减少片外访存所造成的数据移动,从而缓解数据移动造成的能耗问题.

LBM 计算程序的核心计算过程包含多个嵌套循环,最外层为迭代时间步,在每个时间步内计算多个嵌套循环,所有的时间步具有完全相同的访存序列.如果能够保证每一次时间步的所有访问数据能够缓存在片上 cache 中,程序就只需进行一次片外访存,剩余时间步的迭代均可从高速 cache 中获取数据,这将会大大提高数据重用率和访存速度.但原始迭代访问的数据量往往大于 cache 容量,由于缓存容量限制导致 cache 失效,从而引发频繁的缓存数据替换行为,最终造成程序执行性能下降.因此,如何通过循环优化技术解决 LBM 计算程序中的性能瓶颈,并充分开发计算机的硬件能力,从而提高 LBM 计算程序的执行性能,具有重要的研究意义.然而,之前关于 LBM 的研究工作主要关注于更高效的计算模型[6-7]、更快收敛速度的网格模型[8-9]、改进的 LBM 物理算法[10-11],或者是研究异构节点上数据的布局与通信方法[12-13],使得加速设备能更好地实现并行性能.这些研究工作虽然从物理模型和程序实现这两方面对 LBM 的发展和应用起到了重要作用,但是目前仍缺乏深度开发 LBM 计算程序局部性的相关研究工作.

本文从程序代码实现和性能优化的角度,在共享内存多核系统上致力于开发 LBM 程序中核心计算循环代码的循环并行性,并以提高数据局部性为主要优化目标,具体开展了如下工作.首先,通过对 LBM 程序的核心循环代码实施循环融合、循环偏斜、循环分块等循环优化技术,开发时间步上的数据重用,并将空间维迭代空间进行分块,实现一种高效的波阵面并行化方法,挖掘循环分块间的粗粒度并行性.其次,为了更好地对迭代空间进行分块,需要考虑有效的分块大小选择方法(Tile Size Selection, TSS).本文提出了一种基于静态分析和经验搜索的混合分块大小选择方法,LBM_TSS 方法.该方法从 LBM 循环代码的访存行为、局部性收益、并行性效

率以及同步开销 4 个方面进行静态分析建模，构建计算分块大小的约束条件，然后在此基础上采用经验搜索对分块大小进行寻优，确定最优的分块大小.将 LBM_TSS 方法计算出的分块大小用于波阵面并行化方法中实施循环优化的过程，从而最终实现有效开发 LBM 计算程序的粗粒度并行性和数据局部性的目标.最后，本文进行了广泛的实验验证.实验结果表明，采用 LBM_TSS 方法计算的初始分块大小和经验搜索后分块大小的 LBM 程序，其性能分别可以达到采用全局搜索获得的最优分块大小的 LBM 程序性能的 83.20% 和 97.22%.相比经典的空间维循环并行化方法、局部性优化的空间维循环并行化方法和基于 POST/WAIT 函数的时间维循环并行化方法，本文最终实现的 LBM 循环并行优化方法分别实现了 14.61%、16.79% 和 13.10% 的平均性能提升.

2 LBM 循环代码的并行优化方法

2.1 D2Q9 模型

LBM 是一种能够很好地描述复杂流体流动情况的数值方法.它将流体视作很多没有体积的可以向若干方向流动的微粒，每个微粒每时每刻都与周围的微粒发生碰撞.LBM 则通过简单规则的微观粒子运动代替复杂多变的宏观现象.目前主流的 LBM 模型主要是基于文献[14]提出的基础 LBM 模型，其核心逻辑包括当前时间步内的微粒碰撞、流动的结果迭代到下一时间步的微粒碰撞、流动的更新.本文选取了广泛用于科学研究的标准模型之一的单松弛模型 LBGK(Lattice Bhatnagar-Gross-Krook)[15].该模型可以很好地代表大多数 LBM 模型的程序计算特点，所以针对该模型展开的程序优化研究可以轻松拓展到其他 LBM 模型.常用的 LBGK 模型包括 D2Q9、D3Q19、D3Q27 等.本文侧重于如何开发 LBM 计算程序中大量潜在的数据重用和循环并行性，并不关注 LBM 模型本身，因此为了更好地描述本文提出的 LBM 计算程序优化方法，选用了 LBGK 模型中最基础的 D2Q9 模型.该模型描述了 2 维空间中每个格子中心的微粒和它周围的 8 个微粒进行碰撞，其网络结构如图 1 所示，黑色实心圆点表示一个微粒.微粒的运动过程可以分为两个步骤：碰撞(collision)和流动(streaming)[14].碰撞指的是微粒根据模型定义好的方向运动时和相向的微粒发生二体碰撞导致速度改变；流动指的是微粒按照速度方向移动到最近的网格节点上，实现新一轮的宏观物理量更新.本文采用简单松弛的微粒分布函数来完成碰撞过程的计算.

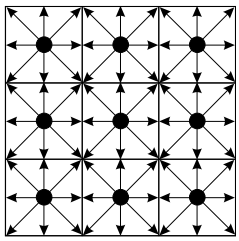


图 1 D2Q9 模型网络结构示例

2.2 LBM 计算程序的局部性优化

图 2(a)展示了 D2Q9 模型计算程序的核心循环的伪代码.该循环的最外层为时间维循环(time dimension)，不同时间步迭代内的程序执行路径相同.时间维循环内包含 3 个子循环，它们分别描述微粒的碰撞过程、流动过程和宏观量更新过程，本文将实现这 3 个过程的循环统称为空间维循环(space dimensions).虽然不同时间步迭代的空间维循环的执行路径相同(也就是访存地址序列相同)，但是宏观量更新过程在每次时间迭代都会更新这些数据.因此，在 LBM 计算程序的时间维循环上具有大量潜在数据重用的机会.但是，图 2(a)所示的原始计算过程中，每个时间步迭代内的空间维循环将访问庞大的地址空间，远超过 cache 的容量限制，使访问的数据在被重用之前就由 cache 替换策略从 cache 中替换出去，从而无法实现数据重用.

本文采用循环优化技术实现 LBM 计算程序时间维循环上的数据重用.首先，利用循环融合(loop fusion)将空间维的 3 个循环融合成 1 个嵌套循环.原始的 3 个循环需要依次执行，将它们融合后可以减少两次的循环迭代开销[3]，还可以增强一定的局部性.循环融合也是后续进行其他循环优化工作的基础.然而，描述碰撞过程和流动过程的循环之间存在数据依赖，如果直接将 3 个循环进行融合，将破坏程序的依赖关系，无法得到正确结果.循环变换的合法条件是变换后的循环不能含有依赖距离为负的依赖向量[3].在描述碰撞过程和流动过程的 2 个循环语句中，存在阻止合法循环变换的依赖向量(−1,0)、(0,−1)和(−1,−1).通过一种称为循环颠簸(loop bump)的循环变换方法，利用仿射变换将这 2 个循环语句的依赖向量在迭代空间上按(1,1)进行偏移，使非法的依赖向量转换为满足合法变换条件的依赖向量(0,1)、(1,0)和(0,0).图 2(b)展示

```
for(int t=0; t<ST; t++){
 //The procedure of collision
 for (int i=0; i<SI; i++) {
  for (int j=0; j<SJ; j++) {
   F[i][j] = function1(f[i][j],rho[i][j],u[i][j]);
  }
 }
 //The procedure of streaming
 for (int i=0; i<SI; i++) {
  for (int j=0; j<SJ; j++) {
   f[i][j] = function2(F[i+1][j+1],F[i+1][j],
                      F[i][j+1],F[i][j],F[i-1][j],
                      F[i][j-1],F[i+1][j-1],
                      F[i-1][j+1],F[i-1][j-1]);
  }
 }
 //The procedure of updating the macro quantity
 for (int i=0; i<SI; i++) {
  for (int j=0; j<SJ; j++) {
   rho[i][j] = function3(f[i][j]);
   u[i][j] = function4(f[i][j]);
  }
 }
}
```

(a) 原始循环

```
for(int t=0; t<ST; t++){
 for (int i=0; i<SI+1; i++) {
  for (int j=0; j<SJ+1; j++) {
   if(i<SI&&j<SJ)
   F[i][j] = function1(f[i][j],rho[i][j],u[i][j]);
   if(i>0&&j>0){
   f[i-1][j-1] = function2(F[i][j],F[i][j-1],F[i-1][j],
                          F[i-1][j-1],F[i-2][j-1],F[i-1][j-2],
                          F[i][j-2],F[i-2][j],F[i-2][j-2]);
    rho[i-1][j-1] = function3(f[i-1][j-1]);
    u[i-1][j-1] = function4(f[i-1][j-1]);
   }
  }
 }
}
```

(b) 循环颠簸、融合后的循环

```
for(int ii=0; ii<iT; ii++){
 for(int jj=0; jj<jT; jj++){
  for(int t=0; t<ST; t++){
   for (int i=ii*Ti+t*2; i<(ii+1)*Ti+t*2; i++) {
    for (int j=jj*Tj+t*2; j<(jj+1)*Tj+t*2; j++) {
     if(i<SI&&j<SJ)
      F[i][j] = function1(f[i][j],rho[i][j],u[i][j]);
     if(i>0&&j>0){
      f[i-1][j-1] = function2(F[i][j],F[i][j-1],F[i-1][j],
                             F[i-1][j-1],F[i-2][j-1],F[i-1][j-2],
                             F[i][j-2],F[i-2][j],F[i-2][j-2]);
       rho[i-1][j-1] = function3(f[i-1][j-1]);
                             u[i-1][j-1] = function4(f[i-1][j-1]);
     }
    }
   }
  }
 }
}
```

(c) 循环偏斜、分块后的循环

图 2　LBM 计算程序的循环变换过程

了经过循环颠簸后和循环融合后的循环伪代码.

接着采用循环分块(loop tiling)[3,4,16]将庞大的循环迭代空间划分为若干个子空间,每个子空间称为一个分块,使其可以完全存放在 cache 中,从而实现分块数据的重用.同样地,图 2(b)中的循环语句中存在跨迭代的依赖关系(loop-carried dependence),阻碍了合法的循环分块.例如,在时间步 $t=n$ 时对 $F[i-2][j-2]$ 进行更新时,需要等待时间步 $t=n-1$ 内($1\leq n\leq ST$,ST 为图 2 所示的时间步迭代次数)所有对 $F[i-2][j-2]$ 的读取操作完成后才能进行.如果对该循环直接分块,则无法维持这个依赖的正确性,导致程序错误.文献[17]指出,对一个循环进行合法分块的前提条件是,分块后的循环不能含有字典序为负的依赖向量.因为本文采用矩形分块方法[3],所以要求所有依赖向量不能含有负元素.对图 2(b)中的循环语句可以计算出阻碍合法循环分块的依赖向量有(0,−1)、(−1,0)、(−1,−1)、(−2,−1)、(−1,−2)、(0,−2)、(−2,0)和(−2,−2).通过一种称为循环偏斜(loop skewing)的循环变换方法,采用时间步变量 t 和偏斜因子为 2 对空间维循环变量 i 和 j 构造仿射变换($2\times t+i$,$2\times t+j$),可以消除所有字典序为负的依赖向量.上述依赖向量经过仿射变换后变为(2,1)、(1,2)、(1,1)、(0,1)、(1,0)、(2,0)、(0,2)和(0,0).此时偏斜后的循环可以进行合法的循环分块.图 2(c)展示了图 2(b)的循环经过循环偏斜和循环分块后的循环伪代码.其中,变量 T_i 和 T_j 分别表示 i 循环和 j 循环采用的分块大小.分块大小的计算方法将在第 3 节详细论述,本节只讨论 LBM 循环代码的并行优化方法.

2.3　波阵面并行化方法

文献[18]将不存在数据依赖关系的循环称为 DOALL 循环,可以完全并行,如图 2(a)中单独描述微粒碰撞过程的嵌套循环;而将存在跨迭代依赖关系的循环称为 DOACROSS 循环,如图 2(c)所示的循环.大部分模板(stencil)计算的循环代码就属于 DOACROSS 循环.由于存在数据依赖,这类循环只能开发严格遵循依赖关系的受限并行性,如波阵面(wavefront)并行性或流水(pipeline)并行性.

本文通过对循环优化后的 LBM 计算程序实施波阵面并行来开发粗粒度循环并行性.根据前文所述的循环融合、偏斜、分块等循环优化方法处理后,LBM 计算程序的迭代空间经过仿射变换和空间划分,形成了如图 3 所示的仿射迭代空间.其中,虚线方框表示划分后的一个迭代子空间,即一个循环分块;分块内的空心圆点表示具体的迭代实例;实线箭头表示各个分块之间的数据依赖关系.分块的迭代执行过程必须严格遵循依赖关系以保证程序执行的正确性.如图 3 所示,处于相同列上的分块之间没有数据依赖,这些分块组成一个波面,同一波面上的分

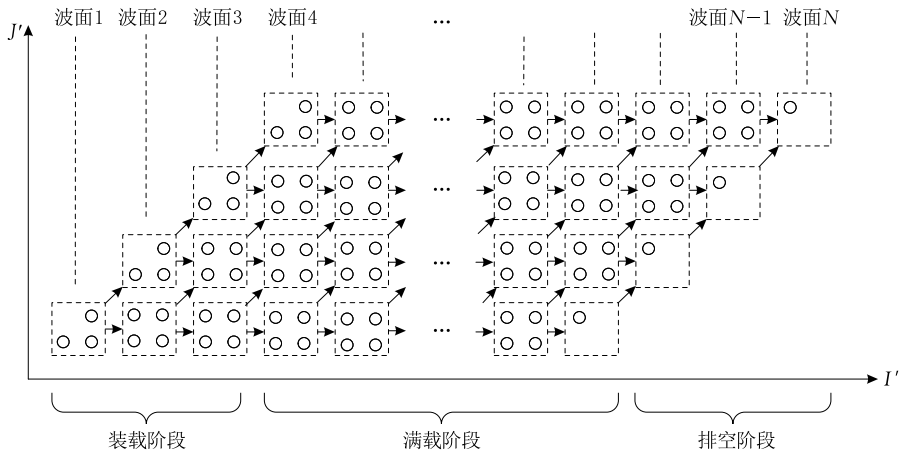


图 3 波阵面并行示例图

块可以并行执行.而波面之间由于存在数据依赖,需要从左到右顺序执行.这种并行执行过程称为波阵面并行,其执行过程可以分为装载、满载和排空 3 个阶段.若不考虑线程数的影响,满载阶段的并发度将达到波阵面并行过程的最大化.

一个波面上的所有分块都依赖于前一个波面上的相应分块,为了保持依赖关系,该波面上的每个分块必须要等它依赖的分块都执行完才能开始执行.因此,所有相邻波面上的分块需要进行同步通信.在基于共享内存的计算系统中,通常采用 OpenMP 并行制导语句实现波阵面并行,OpenMP 将自动为相邻波面之间设置隐式栅栏(implicit barriers)进行线程同步,并静态地为线程分配波面上的分块并行执行.由于迭代空间发生了仿射变换,一些处于波面边缘的分块包含的有效迭代实例较少(如图 3 中装载阶段各波面最上端分块和排空阶段最下端分块),导致同一波面上分块的执行时间存在差异.这样执行同一波面的并行线程在同步结束之前需要等待其他未执行完的线程,造成计算资源浪费,不利于并行效率.

针对上述问题,本文作者在之前的工作中提出了一种动态的波阵面并行化方法[19],本文称之为 MDPS 方法.该方法将迭代空间中的分块依赖关系映射成一个状态矩阵,并归约出依赖保持条件,动态地为线程分配分块执行,同时采用互斥操作替代隐式栅栏实现线程同步.MDPS 方法可以避免线程空等现象,使线程并行更加灵活,有效地提高并行效率.本文采用该方法为循环优化后的 LBM 计算程序实现波阵面并行化过程.需要注意的是,分块大小的选择将直接影响循环优化后 LBM 计算程序的性能和波阵面并行效率,而 MDPS 方法并不提供确定分块大小的方法.将在下节中详细描述本文提出的分块大小选择方法.

3 循环分块大小选择方法

分块大小选择方法(TSS)[20-23]对循环分块代码的性能有着重要的影响,不同的分块大小将会产生截然不同的优化效果.但是,TSS 受到硬件架构、运行环境、循环代码特征等诸多因素的影响,难以计算出最优解,而分块大小又与程序的访存行为、局部性收益、并行效率、线程同步开销等多种性能因素密切相关,所以 TSS 一直是研究者关注的重点.本文根据 LBM 循环的特征,提出一种结合了静态分析和经验搜索的混合循环分块大小选择方法 LBM_TSS.该方法分别从访存行为、局部性收益、并行效率以及同步开销 4 个方面分别通过静态分析构造分块大小选择的约束条件,然后在此基础上进一步结合经验搜索方法,对计算的分块大小进行调优,使其性能接近最优.

3.1 访存行为分析

首先,对于时间维循环的分块大小进行选择.由于本文工作的基本出发点是开发时间步迭代之间大量潜在的数据重用,所以当时间维循环的分块越大时,数据被重用的次数也就越多.因此,本文直接选择最大的时间维分块大小 ST,也就是对时间维循环并不分块.当循环偏斜采用的仿射变换只包含空间维循环变量而不包含时间维循环变量时,相邻的两个时间步内访问的分块数据是完全相同的,在迭代空间上两者重叠.对于这种情况,只需要保证每个分块访问的数据总量不大于 cache 容量即可.设 $Tile\_Data\_Size$ 为分块访问的数据总量,TS_i 和

TS_j 分别为 i 循环和 j 循环的分块大小，$Iteration\_Data\_Size$ 为一次迭代包含的数据总量，$Cache\_Capacity$ 为 cache 容量，则这些变量需要满足式(1).

$$Tile\_Data\_Size = TS_i \times TS_j \times Iteration\_Data\_Size \leq Cache\_Capacity \quad (1)$$

然而，如 2.2 小节所述，对 LBM 计算程序进行循环偏斜的仿射变换中为空间维循环变量引入了时间维循环变量，循环迭代空间将沿着时间步发生位移(如图 4 所示)，此时相邻的两个时间步内访问的分块不再完全重叠，因此需要对式(1)进行修正.

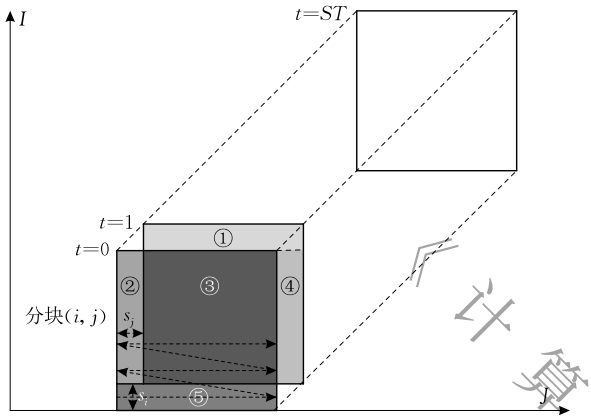


图 4　空间维分块相对时间维位移的过程

在图 4 中，两个相邻时间步 $t=0$ 和 $t=1$ 的分块只存在部分数据重叠(③区域)，④和①区域表示在 $t=1$ 时需要新访问的数据. 其中 s_i 和 s_j 分别为空间维循环变量 i 和 j 的偏斜因子. 目前一般采用理论最优的 LRU 策略进行 cache 数据替换[24]. 如图 4 所示，迭代访存将从 $t=0$ 的分块的最左下侧开始，向右执行到分块边界，然后 i 迭代递增，依此类推，访存过程由虚线箭头表示. 最先被访问的是图中⑤区域，然后交错访问②区域和③区域. 当程序执行到③区域的最右上角时，对于分块 (i,j) 在 $t=0$ 时刻的访问就全部完成了，此时进入 $t=1$ 的访问. $t=1$ 时刻的访存过程和 $t=0$ 时刻一样，只是从③区域的最左下角开始进行. 因为③区域的数据在 $t=0$ 时刻已经访问过，理论上这部分数据可以从 cache 中读取. ④区域的数据从未被访问过，需要进行访存并将数据缓存到 cache 中. 如果此时 cache 已满，根据 LRU 策略就需要替换出最长时间未被使用的数据，也就是图中⑤区域最左下角的数据. 这些数据在 $t=1$ 时刻不会被使用，可以直接替换. 依此类推，每当程序需要使用一块④区域的新数据，就相应地替换掉一块同样大小的⑤区域旧数据，直到程序执行到了①区域. 由于①区域也是未访问过的新数据，需要从 cache 中替换出旧数据. 此时，④区域是最新访问数据，③区域的数据在 $t=1$ 时刻又被重新全部访问了一遍，而②区域的数据将被 LRU 策略最优先替换出去. 于是，程序在执行①区域时，cache 将依次替换②区域数据，直到程序执行到①区域的最右上角. 此时，$t=1$ 时刻分块执行结束，cache 中将完整地保存①、③以及④区域的所有数据. 在下一个时间步 $t=2$ 时刻，程序的访存过程依此类推，直到 $t=ST$ 时刻，该分块的执行结束.

根据上面的分析，需要保证⑤区域的面积不小于④区域，因此，选择的分块大小在满足式(1)的情况下还需要满足式(2). 本文根据 LBM 计算程序的依赖向量，采用 $s_i=s_j=2$，式(2)只需满足 $TS_j \geq TS_i-2$ 即可. 有研究表明，对于 stencil 计算而言，方形分块会引起更少的 cache 失效[25]. 而 LBM 计算程序就是一种典型的模板计算，因此本文选择了 $TS_i=TS_j$ 的方形分块，即空间维循环分块大小相等. 这样，式(2)条件得到满足.

$$s_i \times TS_j \geq s_j \times (TS_i - s_i) \quad (2)$$

3.2　局部性收益分析

根据上节分析，选择方形分块时 $TS_i=TS_j$，可以由式(1)得到式(3). 目前主流的计算机都拥有多级的 cache. 对于一个拥有 3 级 cache 的通用计算机而言，L1 级 cache 的容量往往较小，无法缓存较多的数据，但其数据访问速度最快；而 L3 级 cache 容量最大，但 L3 级 cache 往往是共享 cache，其访存行为可能会受到其他计算核的影响，并且其速度要慢于 L1/L2 级 cache. 为了获得较好的 cache 局部性收益，本文折衷地初步选取 L2 级 cache 来缓存分块数据，即 $L2\_Cache\_Capacity$ 作为计算分块大小的 $Cache\_Capacity$ 指标. L1 级 cache 和 L3 级 cache 会在后文采用经验搜索进一步对分块性能调优的过程中使用.

$$TS_i = TS_j \leq \left\lfloor \sqrt{\frac{Cache\_Capacity}{Iteration\_Data\_Size}} \right\rfloor \quad (3)$$

3.3　并行效率分析

分块大小决定了分块数量，而并行线程的数量往往是确定的，因此分块大小也将影响线程的并发度和程序执行的并行性能. 本小节将对并行效率进行分析. 设 $Tile\_Amount$ 为分块数量，它可由式(4)计算出来，其中 SI 和 SJ 分别表示空间维 i 循环和 j 循环的迭代总数. 为了获得更高的线程并发度，需要获得更多的分块数量，从而让每个线程执行更多

的分块.文献[26]定义了一种类似度量标准来衡量每个线程执行的分块数,称为并行粒度(granularity),其研究结论是 granularity 的值需要大于等于 2,也就是每个线程至少要分配 2 个分块才能保持较好的并发度.本文根据文献[26]的研究结果得到式(5),其中 $num\_of\_threads$ 表示并行线程数量.

如 2.3 小节所述,LBM 计算程序将实现波阵面并行性.根据前文分析,在不考虑线程数的情况下,波阵面并行的最大并发度是满载阶段一个波面上的分块数,用 $Max\_Parallel\_Tiles$ 表示.如果 $num\_of\_threads$ 大于 $Max\_Parallel\_Tiles$,那么在满载阶段有($num\_of\_threads-Max\_Parallel\_Tiles$)个线程处于空闲状态,造成计算资源浪费.因此,$Max\_Parallel\_Tiles$ 应不小于 $num\_of\_threads$,由此可以推导出式(6).

$$Tile\_Amount=\left\lceil\frac{SI+s_i*(ST-1)}{TS_i}\right\rceil\times\left\lceil\frac{SJ+s_j*(ST-1)}{TS_j}\right\rceil \tag{4}$$

$$\frac{Tile\_Amount}{num\_of\_threads}\geq 2 \tag{5}$$

$$Max\_Parallel\_Tiles=\left\lceil\frac{SJ+s_j*(ST-1)}{TS_j}\right\rceil\geq num\_of\_threads \tag{6}$$

由图 3 可知,在波阵面的装载和排空阶段依然存在可并行分块数小于 $num\_of\_threads$ 的情况,这是波阵面并行无法避免的.为了提高波阵面并行效率,应该尽可能地让可并行分块数小于 $num\_of\_threads$ 的阶段在整个波阵面并行过程中所占的比例小,而所有线程满负载的阶段所占的比例大.由式(6)可知,线程只可能在图 3 的装载和排空阶段处于非满负载的状态.需要注意,如果线程数较少,在波阵面装载阶段的后部分和排空阶段的前部分线程即可达到满负载的状态.例如,假设只有 3 个线程执行图 3 所示波阵面,则只有波面 1、波面 2、波面 $N-1$ 和波面 N 存在线程处于非满负载的状态.因此,在式(6)的约束条件下,线程的满负载率 $Fullload\_Ratio$ 由 $num\_of\_threads$ 决定.根据上述分析可以推导出了式(7).LBM_TSS 方法要求式(7)的值尽量大,从而保证整个波阵面执行过程具有较好的并行效率.

$$Fullload\_Ratio=1-\frac{num\_of\_threads\times(num\_of\_threads-1)}{Tile\_Amount} \tag{7}$$

3.4 同步开销分析

在分析分块的并行效率时,还需要考虑同步开销这一重要的因素.同步开销是指在并行过程中,一个工作单元(通常为一个线程或者进程)执行完分配的任务后与其他并行工作单元进行数据同步所产生的开销.由 2.3 小节可知,本文采用 MDPS 方法实现波阵面并行,该方法引入了互斥机制.因此,本文同步开销由内存读写、互斥锁构成.式(8)表示线程的工作效率 $Work\_Efficiency$,其中,$Real\_Compute\_Time$ 表示线程真正进行有效计算的时间,$Sync\_Overhead$ 表示同步开销.该式忽略程序启动等一次性开销.

$$Work\_Efficiency\approx\frac{Real\_Compute\_Time}{Real\_Compute\_Time+Sync\_Overhead} \tag{8}$$

$$Real\_Compute\_Time=TS_i\times TS_j\times Iteration\_CPU\_Time \tag{9}$$

$$Work\_Efficiency\approx\frac{TS_i^2\times Iteration\_CPU\_Time}{TS_i^2\times Iteration\_CPU\_Time+Sync\_Overhead} \tag{10}$$

对于本文研究的 LBM 循环,$Real\_Compute\_Time$ 的值可以由式(9)计算,$Iteration\_CPU\_Time$ 表示每次迭代花费的 CPU 时间.由于本文采取方形分块,即 $TS_i=TS_j$,将式(9)代入式(8)可以得到式(10).$Sync\_Overhead$ 和 $Iteration\_CPU\_Time$ 需要根据实际情况在相应的物理机上进行测量,而且对于不同的程序或并行模式,它们的值都不一样.对于 $Iteration\_CPU\_Time$ 的计算,可以通过对相应的循环代码进行时间插桩,由单线程运行的总时间除以迭代次数,并多次重复此过程计算平均值,得到 $Iteration\_CPU\_Time$ 的近似值.因为访存行为的差异会导致 cache 命中率不同,从而影响 $Iteration\_CPU\_Time$,无法得到准确值.对于 $Sync\_Overhead$ 的计算,则稍微复杂.考虑使用互斥锁机制的并行方法,互斥锁的运行时间较长,而且当有线程访问互斥区时,其他线程只能等待,所以线程等待的时间也应该记入 $Sync\_Overhead$,这就更加增大了 $Sync\_Overhead$ 的不确定性.由于互斥区访问的随机性,使得几乎不存在静态方法可以估算 $Sync\_Overhead$ 的值.针对这种情况,本文采用基于概率的方法对互斥区访问进行建模,并以 $Sync\_Overhead$ 的数学期望作为近似值.本文将不需要等待而直接访问互斥区的确定性开销定义为 $Mutex\_Overhead$,将在互斥区前等待的不确定性开销定义为 $Wait\_Overhead$,并由式(11)描述这 3 种

开销之间的关系. *Mutex_Overhead* 可以使用类似于计算 *Iteration_CPU_Time* 的方法测算出来，而 *Wait_Overhead* 则需要使用基于概率的方法计算. 假设每个线程访问互斥区的概率为 *Mutex_Probability*，且各线程访问互斥区的时间点无相关性，这样可以推导出式(12). 在同一时刻有 *num_of_threads* 个线程同时执行，它们访问互斥区的概率都是 *Mutex_Probability*，所以在每个时刻访问互斥区的线程数的数学期望为 $num\_of\_threads \times Mutex\_Probability$. 由此推导出式(13). 将式(12)和式(13)代入式(11)可以得到式(14)，该式变量名过于复杂，为了简洁性使用缩写 *nt* 表示 *num_of_threads*，*MO* 表示 *Mutex_Overhead*，*ICT* 表示 *Iteration_CPU_Time*，而这 3 个变量都是可以测得的常量. 最后将式(14)代入式(10)，可以得到 *Work_Efficiency* 关于 TS_i 的一元函数，如式(15)所示.

$$Sync\_Overhead = Mutex\_Overhead + Wait\_Overhead \tag{11}$$

$$Mutex\_Probability = \frac{Mutex\_Overhead}{Mutex\_Overhead + Iteration\_CPU\_Time \times TS_i^2} \tag{12}$$

$$Wait\_Overhead = num\_of\_threads \times Mutex\_Probability \times Mutex\_Overhead \tag{13}$$

$$Sync\_Overhead = \frac{nt \times MO^2}{MO + ICT \times TS_i^2} + MO \tag{14}$$

$$Work\_Efficiency \approx \frac{TS_i^2 \times ICT}{TS_i^2 \times ICT + \left(\frac{nt \times MO^2}{MO + ICT \times TS_i^2} + MO\right)} \tag{15}$$

3.5　静态分析与经验搜索的结合

以上 4 小节，分别从 LBM 循环的访存行为、局部性收益、并行效率以及同步开销 4 个方面进行了静态分析，在此进行以下总结.

(1) 对于 LBM 循环的访存分析表明，只要在满足式(1)的情况下，使得 1 次时间迭代的分块可以完全放入 cache，那么 cache 命中率将达到最大值；另外，使用方形分块可以满足式(1).

(2) 对局部性收益的分析表明，考虑到 cache 的容量、速度以及独占性，本文初步选择 L2 级 cache 缓存分块数据，L1 级和 L3 级 cache 在后续经验搜索调优的过程中使用.

(3) 对并行效率的分析表明，分块粒度越小则并行度越高，分块粒度的选择需要满足式(4)～式(6)，才能获得较好的并行效率；同时，线程满负载率也受到分块大小的影响，选择分块大小时也需要考虑线程满负载率.

(4) 对同步开销的分析表明，工作效率受到分块大小和同步开销的影响，在无法缩小同步开销的情况下，只有增大分块大小才能提高工作效率.

结合 LBM 循环的访存行为、局部性收益、并行效率以及同步开销 4 点进行分析，将经验搜索和静态分析结合到一起，利用静态分析构造的约束条件来缩小经验搜索的范围，进一步提高分块大小的性能. 本文所说的经验搜索是指通过直接或间接的观察、经验等先验知识，修剪搜索空间大小，避免无意义的搜索以减少时间成本的一种搜索方式. 例如，对于分块大小的经验搜索，可以按 2 的幂作为步长进行搜索(获得更好的空间局部性)，而不是以 1 为步长搜索；对于分块容量大于 cache 容量的候选分块大小搜索空间，可以直接不搜索.

算法 1 展示了完整的混合分块大小选择方法 LBM_TSS 的具体实现算法的伪代码. 其中，第 1 至 7 行是初始化阶段；第 8 至 14 行是约束区间计算阶段，也就是前文所述的静态分析过程的形式化表示；第 15 至 17 行处理了如果约束区间交集为空的特殊情况，直接输出一个分块大小；第 18 至 26 行则进行了经验搜索过程，最终输出分块大小.

算法 1.　LBM_TSS 算法.

输入：循环优化后满足合法分块条件的循环 *LBM_Loop_M* 和所需的计算机硬件参数

输出：分块大小

/* 初始化过程 */

1. ST、SI、SJ = *LBM_Loop_M* 的 t、i、j 维循环迭代总数；
2. si、sj = *LBM_Loop_M* 的 i、j 维循环偏斜因子；
3. *IDS* = *LBM_Loop_M* 每次迭代访问的数据量；
4. *ICT* = *LBM_Loop_M* 每次迭代消耗的时间；
5. *MO* = *LBM_Loop_M* 波阵面并行过程中线程访问互斥区所用的时间；
6. *L1CC*、*L2CC*、*L3CC* = L1、L2、L3 级 cache 容量；
7. *num_of_threads* = 线程数；

/* 静态分析构建约束条件 */

8. $TS_i == TS_j$；
9. 根据式(3)，使 $Cache\_Capacity = L1CC$、$L2CC$、$L3CC$ 时分别计算出满足 L1、L2、L3 级 cache 容量的 TS_i，分别记为 TS_i_L1、TS_i_L2、TS_i_L3；
10. 根据式(4)、(5)计算出 TS_i 的取值范围 $[1, Upper1]$；
11. 根据式(6)计算出 TS_i 的取值范围 $[1, Upper2]$；
12. 根据式(7)计算出 $Fullload\_Ratio \geq 90\%$ 时 TS_i 的取值范围 $[1, Upper3]$；

13. 根据式(10)计算出 $Work\_Efficiency \geq 95\%$ 时 TS_i 的取值范围[$Lower0$,min(SI,SJ)];
/* 将上述各约束条件进行整合形成完整的约束条件 */
14. [$Lower$, $Upper$] = [TS_i_L1, TS_i_L3] ∩ [1, $Upper1$]∩[1,$Upper2$]∩[1,$Upper3$]∩[$Lower0$, min(SI,SJ)];
/* 特殊情况处理 */
15. if ([$Lower$,$Upper$]==∅)
16. return (ST,TS_i_L2,TS_i_L2);
17. end if
/* 经验搜索调优过程 */
/* 将最小执行时间变量和最优分块大小变量初始化为无穷大和 0 */
18. $Min\_Time=\infty$, $Best\_TS_i=0$;
/* 在完整约束条件内以 4 为步长进行搜索 */
19. for TS_i,TS_j=$Lower$: $Upper$ by 4 do
20. 使用分块大小(ST,TS_i,TS_j)运行 LBM_Loop;
21. $current\_time$=运行时间;
22. if ($current\_time<Min\_Time$)
23. $Best\_TS_i=TS_i$; /* 最优分块大小 */
24. end if
25. end for
26. return (ST,TS_i,TS_j);

该算法如果不考虑经验搜索过程,其时间开销是常数,复杂度仅为 $O(1)$. 若考虑经验搜索开销,算法使用多个约束条件已经极大地压缩了搜索空间,且方形分块将搜索的复杂度限定在一次幂. 因此,考虑经验搜索的时间开销也很小. 为了尽量利用空间局部性,搜索步长一般设置为 2 的幂,该算法选择以 4 为步长进行搜索. LBM_TSS 方法本身并不局限于 LBM 循环,只要类似 LBM 的计算特点,都可以根据 LBM_TSS 方法进行适配.

3.6 LBM_OPTIMIZATION 方法

目前 TSS 主要有基于静态分析的方法[20,27]、基于经验搜索的方法[21,28]、基于机器学习的方法[22,29]以及混合的方法[23,30]. 基于静态分析的方法通常计算速度快,但是分析建模过程较复杂,难以考虑到所有对 TSS 有影响的因素,导致计算的分块大小性能和最优性能存在一定差距. 基于经验搜索的方法往往可以获得接近最优性能的分块大小,而且方法简单,但是对搜索空间近似穷举式地遍历往往需要消耗较长时间. 基于机器学习的方法获得的分块大小性能依赖于程序和硬件特征的提取是否准确,而且需要自主构建训练数据集. 虽然人工神经网络或者回归网络训练成功后预测分块大小的速度较快,但是前期构造和训练网络的开销巨大,当程序运行环境发生变化时往往需要重新训练网络,该方法的普适性仍待提高. 而混合方法则是将上述 2 种甚至 3 种方法进行结合,互补缺点,实现高效的 TSS 方法. 本文就是采用静态分析和经验搜索相结合的混合方法,通过深入分析分块数据在访存行为、局部性收益、并行效率和同步开销的基础上构建有效的约束条件,然后在约束条件限制的搜索空间内寻优,从而获得最优性能的分块大小.

本文采用 MDPS 方法实现循环优化后 LBM 计算程序的并行性,但是 MDPS 方法并未提供有效的 TSS 方法,而合理的分块大小是实现高效 MDPS 方法的必要条件. 因此,本文提出了 LBM_TSS 方法,将计算出的最优分块大小用于 MDPS 方法的实现过程,从而使最终实现的 LBM 优化程序能够具有良好的数据局部性和粗粒度并行性. 在实现本文研究的 LBM 计算程序循环优化时,两者是一个有机的整体. 本文将这一完整的面向 LBM 计算程序的循环优化方法称为 LBM_OPTIMIZATION 方法.

4 实验与分析

本文所有实验均在一台 4 处理器 Intel Xeon E7-4820 v2 服务器上进行. 每个处理器具有 8 个计算核,每个计算核具有 32 KB 的 L1 级私有 cache 和 256 KB 的 L2 级私有 cache,8 个计算核共享一个 16 MB 的 L3 级 cache. 4 个处理器共享 128 GB 的 DDR3 主存. 该服务器的操作系统为 CentOS Linux release7.1.1503,使用内核版本为 Linux kernel 3.10.0,使用编译器版本为 gcc4.8.5. 测试程序均基于 LBGK 程序 D2Q9 模型代码使用 C 语言实现.

本节从整体和局部两个层面验证 LBM_TSS 方法的高效性. 4.1 小节对 LBM_TSS 方法的整体性能进行了测试,4.2 小节验证空间维分块大小选择的有效性,4.3 小节验证时间维分块大小选择的有效性,4.4 小节将 LBM_OPTIMIZATION 方法与其他 LBM 并行优化方法进行了实验比较. 本文只测量 LBM 计算程序的核心循环的执行时间,使用的指标是钟表时间(Real_Time).

4.1 LBM_TSS 方法的整体性能验证

为了充分展示 LBM_TSS 方法的高效性,本文在 t、i、j 三维循环构成的迭代空间进行全局搜索来获取最优的分块大小 $optimum\_ts$,其时间复杂度为 $O(ST\times SI\times SJ)$. 由于无法承受的时间开销,全局

搜索不能够简单地进行穷举计算.例如,对于 $ST=200$、$SI=SJ=1000$ 的较小问题规模而言,仅全局搜索 i 和 j 两维空间,假设执行一次搜索需要 30 s,则一共需要 $1000\times1000\times30$ s,约 347 天才能执行完,如果对 t 维也进行穷举,则需要约 190 年.实践证明,当分块大小的取值较大时,较小的变化量对执行时间的影响很小.因此,本文缩小了全局搜索空间来降低时间复杂度.但本文的全局搜索空间依然远大于经验搜索空间,从而尽可能地覆盖更多性能优异的分块大小.

根据算法 1,分别计算出初始的分块大小 *original_lbm_ts* 和经验搜索的区间,以及寻优后的分块大小 *searched_lbm_ts*.图 5 展示了使用 *original_lbm_ts*、*searched_lbm_ts* 的性能(执行时间)相对于使用 *optimum_ts* 的性能的百分比.其中,纵坐标为性能百分比,图中两种条柱分别表示采用 *original_lbm_ts* 和 *searched_lbm_ts* 分块后的 LBM 程序执行时间除以采用 *optimum_ts* 分块后的 LBM 程序执行时间的百分比.该值越接近 100% 表示该分块大小的性能越接近最优分块大小的性能,也说明该分块大小越好.所有实验均采用 32 线程.

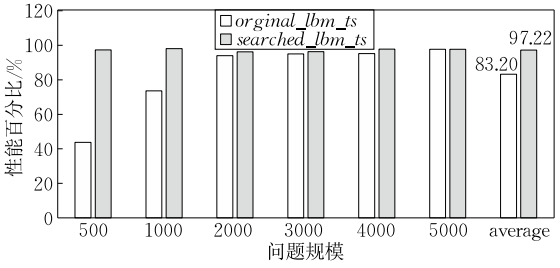


图 5　LBM_TSS 算法计算的分块大小的性能

实验结果表明,*original_lbm_ts* 的性能接近最优分块大小的性能,所有问题规模下的平均 *original_lbm_ts* 性能可以达到 *optimum_ts* 性能的 83.20%.在问题规模较小时(即 500 和 1000),*original_lbm_ts* 性能表现不理想.这是因为当问题规模较小时,由初步选择 L2 级 cache 容量计算出的 *original_lbm_ts* 分块大小较大,导致并行分块数变少,从而影响了并发度,降低了并行性能.在这两种问题规模下实际的 *optimum_ts* 比较小,因而维持了较好的并行性能.*searched_lbm_ts* 相比 *original_lbm_ts* 表现出了更好的性能,它在所有问题规模下的平均性能达到了 *optimum_ts* 性能的 97.22%.对于 *original_lbm_ts* 表现不佳的较小问题规模,*searched_lbm_ts* 也表现良好,分别在 500/1000 问题规模下达到了 *optimum_ts* 性能的 93.74% 和 98.07%.实验结果说明,*original_lbm_ts* 由于通用性选择 L2 级 cache 作为折衷的缓存容量,在某些特殊情况下表现不够理想.但通过经验搜索,可以增加对 L1 级和 L3 级 cache 的补充考虑,从而弥补只考虑 L2 级 cache 的不足,实现对 *original_lbm_ts* 的性能调优.

4.2　空间维分块大小验证

分块大小的语义除了包含容量,还包含了形状,所以一个分块大小应该由分块大小的容量(Size of Tile Size,STS)和分块大小的形状(Shape of Tile Size,ShTS)唯一确定.同一个 STS 可能对应多个不同形状的分块.例如,对于 3 种不同 ShTS 的非方形分块$(TS_i,TS_j)=(4,16)$、$(8,8)$和$(16,4)$,它们对应的 STS 均为 64.如前文所述,让一次时间步迭代内所有的数据可以完全放入 cache 中,并采用方形分块.该思想前半部分限制了分块大小的 STS,后半部分则限制了分块大小的 ShTS.本小节将用实验依次验证两者的有效性.所有实验均采用 32 线程.

图 6 给出了问题规模为 $ST=200$、$SI=SJ=2000$ 时,STS 和分块的执行时间之间的关系.因为同一 STS 对应多个不同 ShTS 的分块大小,它们的执行时间各不相同,所以该实验记录了同一 STS 分块执行时间的最大值(图中点线)、最小值(图中实线)和平均值(图中虚线).暂时先不考虑这 3 条曲线都具有的周期性波动,可以看出,在 STS 较小的初始阶段 3 条曲线基本保持相似的差值上下波动.而随着 STS 进一步增大,3 条曲线都呈现逐渐增大的趋势.这是因为 STS 增大到某个临界点时,将无法完全放入 cache,此时出现 cache 失效.随着 STS 继续增大,cache 失效率也将持续增大,导致访问主存的频率越来越高,因此延长了程序的执行时间.实验结果验证了 LBM_TSS 方法的核心思想之一,即将一次时间步迭代内访问的数据完全放入 cache 中有利于 LBM 计算程序的执行性能.

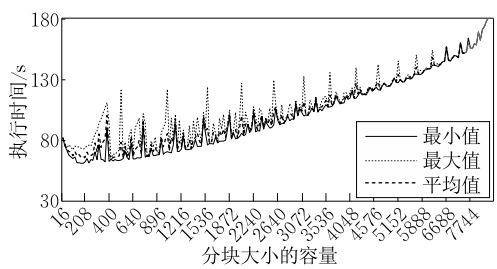


图 6　STS 对执行时间的影响

图 6 中各曲线周期性波动的原因是很小的 STS 变化将会引起执行时间的明显波动.本文对最大值

曲线的每个波动峰值进行了检视，发现这些运行较慢的分块大小的 TS_i 取值都是 128，同时 TS_j 的值小于 TS_i. 根据前文 LBM 计算程序的访存行为分析和推导的式(2)约束条件，要求 $s_i \times TS_j \geq s_j \times (TS_i - s_i)$. 在 $s_i = s_j$ 的情况下，则要求 $TS_j \geq TS_i - s_i$，如果 $TS_j < TS_i$ 则 cache 失效率将会上升. 点线也恰恰对这一结论进行了验证. 对于实线和虚线的分析几乎相同，此处不再赘述. 而且，同一 STS 最大执行时间(点线值)大概是最小执行时间(实线值)的 2 倍. 实验结果验证了分块大小的容量对其执行性能的影响是不可忽略的.

此外，图 6 也从侧面间接地反映了相同容量的分块大小由于分块形状不同也会影响访存行为. LBM_TSS 方法采用了方形分块方法，简单有效地满足了式(2)的约束. 图 7 对方形分块形状的性能进行了进一步验证.

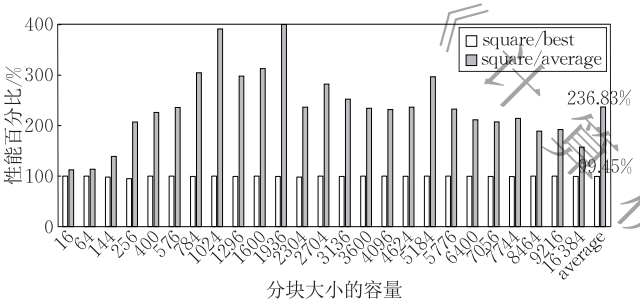


图 7 方形分块的有效性验证

在图 7 中，横坐标为分块大小的容量，纵坐标为性能百分比，表示采用方形分块的 LBM 计算程序执行性能与采用最优性能或平均性能分块的 LBM 计算程序执行性能的比值. 其中，每组对比实验的两个分块的 STS 相同，浅色条柱表示方形分块的性能与最优性能分块的性能的比值，深色条柱表示方形分块的性能与平均性能分块的性能的比值. 浅色条柱越接近 100%，深色条柱越大于 100%，则说明方形分块的性能更优. 由图 7 可以看出，方形分块的平均性能达到了相同 STS 最优性能分块的 99.45%. 而且该数据很平滑，即使在最差情况下方形分块仍可以达到最优分块性能的 94.93%. 另外，方形分块在每组实验中都比相同 STS 的平均性能分块更优，其性能平均达到了这些处于平均性能分块的 236.83%. 实验结果说明方形分块具有近似最优的性能，适合计算模式与 LBM 相似的 stencil 计算程序.

4.3 时间维分块大小验证

本小节对时间维分块大小选择的有效性进行验证，实验结果如图 8 所示. 所有实验均采用 32 线程. 图中横坐标为空间维的分块大小(STS 均满足 cache 容量)，纵坐标为程序的执行时间，加粗短线、长点线、点划线和实线分别代表时间维分块大小为 32、64、128 和 200 时的执行时间. 可以看出，这 4 条线在不同空间维分块大小上的执行时间大致相同，没有出现较大的差异. 而实线表示的 $TS_t = 200$(即时间维循环不分块)的执行时间一直处于比较低的位置.

虽然实验结果并未证明选择 ST 作为 t 维循环的分块大小比其他分块大小具有显著的性能优势，但是这种策略的有效性已经得到验证. LBM_TSS 方法的思想是在短时间内选择性能较高的分块大小，根据对时间维循环的分析，选择了时间维循环不分块的策略. 与时间维循环分块的实验对比结果已经表明该策略达到了 LBM_TSS 方法的目的.

另外，前文分析认为，时间维分块越大，理论上重用次数越多，程序应该执行得越快，但图 8 的数据却并未有效地验证这一观点，4 种不同的 TS_t 执行性能差异不大. 我们分析了 $TS_t = 32$ 时的数据，发现当空间维分块大小较小时，即图中 $(TS_i, TS_j) = (8,8)$、$(16,16)$、$(24,24)$ 时，加粗短线的执行时间最长，而相反，在其他空间维分块大小较大的情况，加粗短线却表现较好. 这是因为当空间维分块较大时，空间维分块数较少，此时如果时间维不分块或者分块较大，则总的分块数较少，并发度较差；而当空间维分块较小时，空间维分块数较多，虽然此时时间维分块较小会有利于增加并发度，但是较小分块的数据重用率会降低. 因此，时间维分块的选择需要在数据重用率和并发度之间进行权衡. 考虑到实际应用中的 SI、SJ 通常较大，而算法 1 计算的空间维循环分块大小不会太大，所以本文认为选择时间维循环不分块是一种较优的选择.

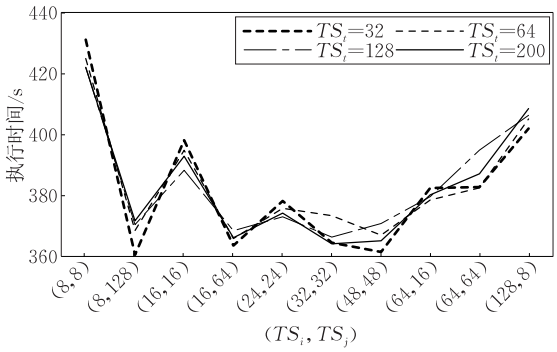


图 8 时间维分块大小选择的有效性验证

综上所述，LBM_TSS 方法是一种性能优良且算法复杂度较低的分块大小选择方法，它充分结合了静态分析的速度优势和经验搜索的性能优势.

4.4　LBM_OPTIMIZATION 性能验证

实际应用中 LBM 计算程序的时间步迭代总数往往能够达到百万级甚至亿级，为了让实验时间在可控范围内，本节实验选用问题规模为 $ST=200$，$SI=SJ=500$、1000、2000、3000、4000、5000. 实验结果为 3 次独立执行结果的算术平均值.

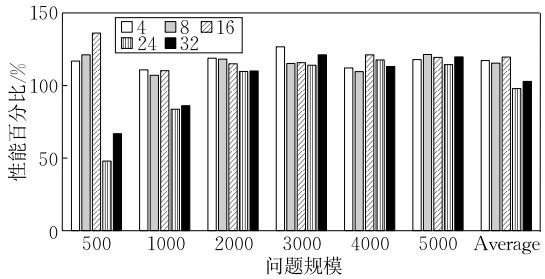
为了验证 LBM_OPTIMIZATION 方法的整体性能，本文选择了以下 3 种面向 LBM 计算程序的并行优化方法进行比较. 为了简洁性，用 LBM_OPTIMIZATION 表示采用本文提出的 LBM_OPTIMIZATION 方法实现的 LBM 程序.

(1) LBGK 模型的空间维循环并行化方法. 该并行化方法是在实际问题中应用最广泛，也是最经典和最简单有效的 LBM 并行化方法之一. 该方法将描述空间维微粒碰撞、流动和宏观量更新 3 个过程的 DOALL 循环分别采用 OpenMP 并行制导语句直接实现，而让时间步迭代串行执行. 本文用 DOALL 表示基于该并行化方法实现的 LBM 程序.

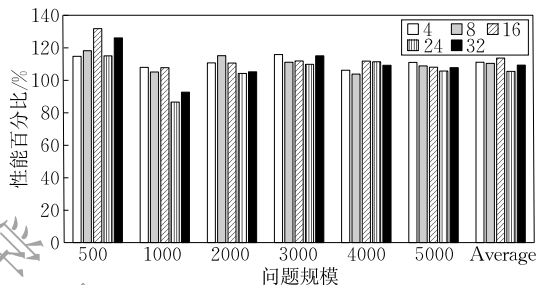
(2) 局部性优化的 LBGK 模型的空间维循环并行化方法. 很多前人的研究工作在上述(1)方法的基础上提出在空间维对 LBM 计算程序进行局部性优化. 文献[31-32]发现描述微粒流动的循环是 LBM 计算程序开发局部性的关键之处，并采用循环分块方法对该循环进行了 cache 优化，被认为是该领域最先进的方法之一. 本文基于文献[31-32]的核心思想，对描述微粒流动过程的 DOALL 循环代码实施了循环分块，并用 DOALL+TILE 表示基于该方法实现的 LBM 程序. 因为该方法没有提供具体的 TSS 方法，为了保证公平性，本文为 DOALL+TILE 采用算法 1 计算出的分块大小.

(3) 基于 POST/WAIT 函数的时间维循环并行化方法. 该方法也尝试开发时间维循环的数据重用性. 文献[33]在实现波阵面并行化过程中采用依赖折叠和通信打包的方式减少冗余同步，并提出了采用 POST/WAIT 函数代替 OpenMP 并行制导语句的通信策略，从而提高了波阵面并行效率. 本文基于文献[33]的思想和方法，实现了基于 POST/WAIT 函数通信优化、时间维循环并行的 LBM 程序，本文用 POST/WAIT 表示该程序. 同样地，由于文献[33]没有提供具体的 TSS 方法，本文依然为 POST/WAIT 采用算法 1 计算出的分块大小.

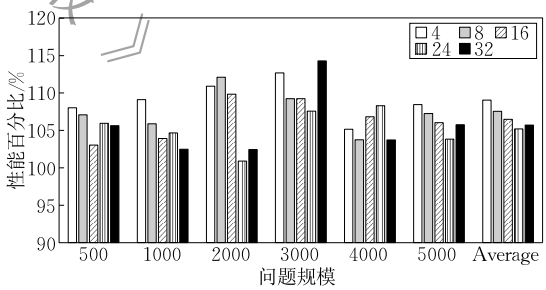
在不同线程数和问题规模下，图 9 展示了 LBM_OPTIMIZATION 采用 LBM_TSS 算法计算的初始分块大小的性能与其他 3 种采用相同分块大小 LBM 实现程序的性能比较，图 10 则展示了 LBM_OPTIMIZATION 采用经验搜索后分块大小的性能与其他 3 种采用相同分块大小 LBM 实现程序的性能比较. 所有图中，纵坐标“性能百分比”表示 LBM_OPTIMIZATION 的执行性能分别与其他 3 种 LBM 实现程序执行性能的比值.



(a) LBM_OPTIMIZATION和DOALL的性能比较



(b) LBM_OPTIMIZATION和DOALL+TILE的性能比较



(c) LBM_OPTIMIZATION和POST/WAIT的性能比较

图 9　采用初始计算分块大小的 LBM_OPTIMIZATION 性能比较

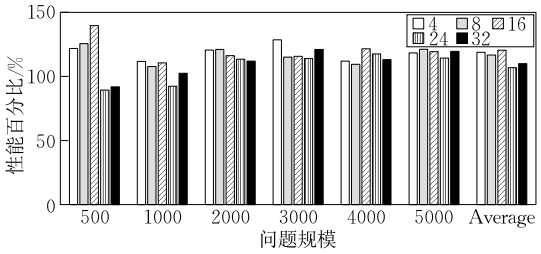
从图 9 的性能对比结果可以看出，根据 LBM_TSS 方法计算的初始分块大小可以使 LBM_OPTIMIZATION 获得较好的性能提升. 在图 9(a)中，除了问题规模在 500 和 1000、线程数为 24 和 32 的算例情况，LBM_OPTIMIZATION 在其他所有算例情况下都比 DOALL 具有更好的性能. 这 4 组结果不理想的算例，是因为问题规模较小时初始计算的分块大小(200,24,24)偏大，导致并发度较低，而 LBM_OPTIMIZATION 代码比 DOALL 相对复杂，其

控制流开销也较大，所以在线程数较大时（如 24 和 32）空闲线程增多，反而使 LBM_OPTIMIZATION 性能差于 DOALL. DOALL＋TILE 和 POST/WAIT 的代码复杂度和控制流开销明显大于 DOALL，因此在图 9(b)和图 9(c)的这类算例下并发度和控制流开销对性能的影响不太明显. 对于图 9 的所有算例实验，采用初始分块大小的 LBM_OPTIMIZATION 的性能平均分别达到采用相同分块大小的 DOALL、DOALL＋TILE 和 POST/WAIT 的 110.54%、110.03%和 106.80%.

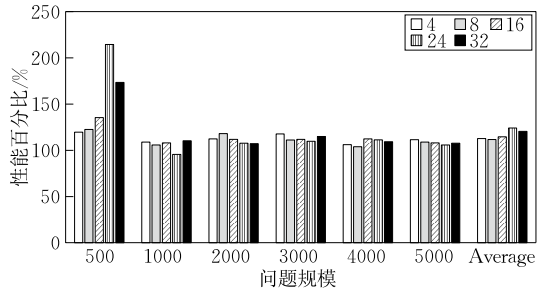
采用经验搜索后分块大小的 LBM_OPTIMIZATION 在所有算例上都获得了性能提升，如图 10 所示，LBM_OPTIMIZATION 的性能平均分别达到采用相同分块大小的 DOALL、DOALL＋TILE 和 POST/WAIT 的 114.61%、116.79%和 113.10%. 对于图 9 中表现不佳的 4 组算例，采用经验搜索后分块大小的 LBM_OPTIMIZATION 性能接近或优于 DOALL. 这是因为通过经验搜索，可以把 L1 级 cache 考虑进来，从而覆盖到更小的分块大小. 图 10 的实验结果表明，经验搜索后分块大小可以使 LBM_OPTIMIZATION 的性能有效提升.

前文 4.1、4.2 和 4.3 小节对 LBM_TSS 方法的实验并未和 4.4 小节一样选择多种问题规模或线程数的组合算例进行验证分析，这是因为这 3 小节已经对 LBM_TSS 方法进行了全面、详细的实验分析，实验结果也从多个方面有效地验证了第 3 节对 LBM_TSS 方法分析的正确性. LBM_TSS 方法在其他组合算例上的实验也基本符合前 3 小节分析的结论，由于篇幅限制本文不再赘述. 而 4.4 小节对 LBM_OPTIMIZATION 方法与其他 3 种并行优化方法进行性能比较，采用多种组合算例进行实验可以更好地反映本文优化方法在整体性能上的优势. 本文提出的 LBM_OPTIMIZATION 方法虽然只在 D2Q9 模型上进行了 LBM 程序实现，该方法也可以应用于 D3Q19 或 D3Q27 等模型上. 此外，理论上 LBM_OPTIMIZATION 方法也可以在类似 LBM 应用的其他流体应用程序上进行扩展和应用，只是具体实现细节略有差异，整体优化方法和思路相同.

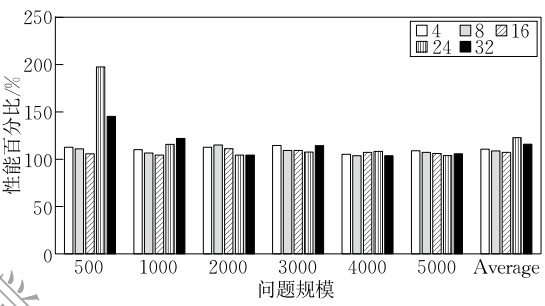
总的来说，LBM_OPTIMIZATION 方法集成了优良的并行化策略和 TSS 方法，既实现了灵活高效的波阵面并行效率，也可以有效地开发时间维循环的数据重用性. 在共享存储的实验平台上，LBM_OPTIMIZATION 方法相比另外 3 种典型的 LBM 并行优化方法，具有更好的性能优势.



(a) LBM_OPTIMIZATION和DOALL的性能比较



(b) LBM_OPTIMIZATION和DOALL+TILE的性能比较



(c) LBM_OPTIMIZATION和POST/WAIT的性能比较

图 10 采用经验搜索后分块大小的LBM_OPTIMIZATION 的性能比较

5 相关研究

对格子玻尔兹曼方法的研究主要分为对模型的研究和对应用优化方法的研究. 前者关注于 LBM 模型的改进，而后者则关注于 LBM 计算程序在不同应用场景下的优化实现. 本文的研究侧重于后者. 本节将对 LBM 相关研究现状进行综述分析.

5.1 格子玻尔兹曼方法的研究现状

格子玻尔兹曼方法是介于流体的宏观连续模型和微观分子动力学模型之间的介观模型，玻尔兹曼方程比 Navier-Stokes 方程蕴含了更多的物理学内涵，因此在流体模拟中得到广泛应用. 格子玻尔兹曼方法由 McNamara 等人[14]首次提出. Qian 等人[15]在此基础上提出了 LBGK 模型，该模型采用单一松弛时间系数来替代碰撞项的矩阵模式，极大地提高了计算效率. 但是当数值不稳定时，该模型会出现发散现象. 因此，许多研究采用了多松弛（MRT）格子

玻尔兹曼方法[34-35]，通过增加网格数来维持计算稳定性，但需要非常巨大的内存以及求解时间.

LBM 的网格划分技术也是相关领域的研究热点. 格子玻尔兹曼方法在某种程度上，可以被看作是连续玻尔兹曼方程的一种特殊离散形式，如果采用非均匀网格进行划分，则将其称为非标准格子玻尔兹曼方法. 国内外已有不少研究者针对非标准格子玻尔兹曼方法相继开展了研究，并分别提出了多种不同的计算模型，如与自适应网格方法类似的 FH 方法[8]、单向耦合方法[9]和基于有限体积法的 LBM 方法[36]等.

5.2　循环优化技术的研究现状

LBM 程序的计算核心是由循环代码实现. 嵌套循环也是众多计算密集型流体计算应用程序的性能热点. 研究者针对循环代码开展了一系列的循环优化技术. 这些优化技术通过循环融合、循环颠簸、循环偏斜、循环分块等多种循环变换方法，在不改变原始程序逻辑的基础上，以增强数据局部性和开发循环并行性为主要优化目标.

局部性原理作为计算机科学的理论基础之一，多年来一直受到广泛的研究. 一部分工作专注于局部性分析和量化的理论研究[37-38]，另一部分工作则侧重于通过局部性理论指导程序优化、作业调度、系统设计和性能分析等[39-40]. 本文侧重于利用局部性分析指导循环优化中的分块大小选择方法，充分开发分块数据在 cache 中的重用性.

分块大小选择方法(TSS)对循环优化，特别是循环分块后程序代码的性能起到了决定性的作用，因此一直是研究者关注的重点. TSS 方法主要分为静态分析方法、经验搜索方法、机器学习方法以及混合方法. 静态分析方法通过对循环结构和硬件的特征建立描述分块数据在 cache 中访存行为的模型来计算分块大小. 文献[20]在建立 cache 失效方程和数据重用率方程时引入 cache 组关联度对性能的影响，从而消除数据的自干扰和组干扰，提高 cache 利用率. 这类方法的性能依赖于静态模型的精准程度，而通过单一的静态模型完全考虑到影响 TSS 性能的众多因素是难以实现的. 基于经验搜索的 TSS 方法则是通过广泛的调试寻优获得性能较好的分块大小. 这类方法主要应用于性能调优系统，其中最具代表性的就是 ATLAS 系统[28]. 经验搜索方法虽然可以获得近似最优的分块大小，但是时间成本较大. 近年来随着机器学习方法的复兴，研究者采用人工神经网络或回归网络进行最优分块大小的预测[22,29]. 基于机器学习的方法将影响分块大小性能的众多因素留给网络模型，只需要关注软硬件特征值的提取. 但是这类方法缺乏有效的数据集，往往需要人工构造，建模成本较高，而且当程序模式或运行环境发生变化时，需要重新训练模型. 混合的 TSS 方法则是将上述方法中的两种或三种进行结合，汲取各种方法的优点，形成准确、高效的 TSS 方法. 其中，最常见的混合 TSS 方法是用静态分析方法构建满足优化目标的约束条件，从而有效地缩减搜索空间，提高性能寻优的速度[23,30]. 这样既可以保证分块大小的性能，也可以有效降低经验搜索的时间开销. 本文提出的 LBM_TSS 方法就是采用混合的方法，但是该方法在静态分析过程中相比已有方法考虑了更多更全面的优化目标.

此外，循环优化技术常用于开发粗粒度循环并行性，通过循环分块将迭代空间划分成若干分块空间，并在分块之间实现并行. 对于存在数据依赖的 DOACROSS 循环，则需要开发严格遵循依赖保持条件的波阵面或流水并行性[41-42]. 文献[43]采用启发式算法对循环层划分，实现单层循环并行性，并提出了流水并行粒度的代价模型. 文献[19]基于线程互斥锁操作首次实现了一种动态的波阵面并行化方法. 文献[44]提出一种基于主从并行模式的波阵面并行策略，进一步优化波阵面间的同步通信. 由于波阵面并行过程的装载和排空阶段限制了并行效率，文献[45]提出一种菱形分块并行方法，提高起始阶段的并发度，但该方法的代码复杂度较高. 此外，一些研究者将循环优化技术集成到多面体框架中，开发出自动循环优化工具[46-47]. 然而，这些工具只能处理代码结构相对简单的程序，在面对真实复杂的应用程序时，其性能难以满足要求.

5.3　LBM 计算程序优化方法的研究现状

LBM 计算程序具有显式的并行性，很多研究致力于开发 LBM 计算程序的并行性. 大量研究工作在通用多核系统上展开. 文献[31]对 LBM 计算程序的性能进行了细致分析，发现描述微粒流动过程的循环是优化 LBM 程序性能的关键，因此对该部分循环代码实施循环分块来提高空间维数据局部性. 文献[48]提出一种同时开发时间维和空间维局部性的并行化方法. 此外，许多工作利用异构加速器或超级计算机系统的大规模并行能力，开发高度并行化的 LBM 程序. 文献[49]提出了一种数据布局友好方法，将描述微粒碰撞和流动过程的循环代码融合，并实施循环分块，从而在 GPU 上实现 LBM

并行程序.文献[28]在天河二号超级计算机上采用循环分块、SIMD 友好的数据结构和通信重叠等优化方法实现 LBM 并行程序.目前的研究工作没有充分考虑时间维数据重用的开发,而且往往忽略了分块大小对 LBM 计算程序性能的影响.

6 总 结

本文针对 LBM 计算程序进行了一系列局部性和粗粒度并行性的优化研究.对 LBM 计算程序的核心循环进行了循环优化,将其巨大的迭代空间划分成满足 cache 容量的子空间,从而使连续时间步更迭时,空间维分块数据不会被替换出 cache,充分开发了数据重用,同时采用 MDPS 方法实现了分块数据的波阵面并行性.在对 LBM 核心循环进行空间划分时,提出了 LBM_TSS 方法.该方法从访存行为、局部性收益、并行效率以及同步开销四个方面进行静态分析,并在所构建的约束条件限定的搜索空间内对分块大小进行寻优,进一步优化分块大小的性能.LBM_TSS 方法可在常数时间内计算出近似最优的分块大小.LBM_TSS 方法与循环优化方法、MDPS 方法有机结合,形成一种完整的面向 LBM 计算程序循环优化的 LBM_OPTIMIZATION 方法.实验结果表明本文提出的方法可以有效地提升 LBM 计算程序的性能.针对 LBM_TSS 方法的验证实验表明,该方法计算的初始分块大小和经验搜索后分块大小的性能可以分别达到全局最优分块大小性能的 83.20%和 97.22%.同时,实验表明,LBM_OPTIMIZATION 方法相比经典的空间维循环并行化方法、局部性优化的空间维循环并行化方法和基于 POST/WAIT 函数的时间维循环并行化方法,其实现的 LBM 程序将性能分别提升了 14.61%、16.79%和 13.10%.下一步的工作将会把 LBM_OPTIMIZATION 方法扩展和应用到其他类似 LBM 的流体应用程序上,并根据新的应用程序特征提供新的优化方法.

参 考 文 献

[1] Chen S, Doolen G D. Lattice Boltzmann method for fluid flows. Annual Review of Fluid Mechanics, 1998, 30(1): 329-364

[2] Li Jie, Xu He-Yong, Qu Kun. Computational Fluid Dynamics: Basics and Applications. Beijing: Aviation Industry Press, 2018(in Chinese)
(李杰,许和勇,屈崑. 计算流体力学:基础与应用. 北京:航空工业出版社,2018)

[3] Xue J. Loop Tiling for Parallelism. 2nd Edition. New York: Springer Science & Business Media, 2012

[4] Pouchet L N, Bondhugula U, Bastoul C, et al. Loop transformations: Convexity, pruning and optimization// Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Austin, USA, 2011: 549-562

[5] Llopis P, Isaila F, Blas J G, et al. Model-based energy-aware data movement optimization in storage I/O stack. The Journal of Supercomputing, 2017, 73(12): 5465-5495

[6] Luo L S. Theory of the Lattice Boltzmann method: Lattice Boltzmann models for nonideal gases. Physical Review E, Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics, 2000, 62(4 Pt A): 4982-4996

[7] Montellà E P, Yuan C, Chareyre B, et al. A hybrid multiphase model based on lattice Boltzmann method direct simulations//Proceedings of the VIII International Conference on Computational Methods for Coupled Problems in Science and Engineering. Sitaus, Spain, 2019

[8] Filippova O, Hnel D. Grid refinement for lattice-BGK models. Journal of Computational Physics, 1998, 147: 219-228

[9] Lin C L, Lai Y G. Lattice Boltzmann method on composite grids. Physical Review E, 2000, 62: 2219-2225

[10] d'Humières D, Bouzidi M, Lallemand P. Thirteen-velocity three-dimensional lattice Boltzmann model. Physical Review E, 2001, 63: 066702

[11] Lallemand P, Luo L S. Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invariance, and stability. Physical Review E, 2000, 61(6): 6546-6562

[12] Tomczak T, Szafran R G. Sparse geometries handling in lattice Boltzmann method implementation for graphic processors. IEEE Transactions on Parallel and Distributed Systems, 2018, 29(8): 1865-1878

[13] Herschlag G, Lee S, Vetter J S, et al. GPU data access on complex geometries for D3Q19 lattice Boltzmann method// Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium. Vancouver, Canada, 2018: 825-834

[14] McNamara G R, Zanetti G. Use of the Boltzmann equation to simulate lattice-gas automata. Physical Review Letters, 1988, 61(20): 2332

[15] Qian Y, d'Humières D, Lallemand P. Lattice BGK models for Navier-Stokes equation. Europhysics Letters, 1992, 17: 479-484

[16] Kelefouras V, Keramidas G, Voros N. Cache partitioning+loop tiling: A methodology for effective shared cache management//Proceedings of the 2017 IEEE Computer Society Annual Symposium on VLSI. Bochum, Gremany, 2017: 477-482

[17] Wonnacoott D G, Strout M M. On the scalability of loop tiling techniques//Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques. Berlin, Germany, 2013

[18] Cytron R. DOACROSS: Beyond vectorization for multiprocessors //Proceedings of the International Conference on Parallel Processing. 1986: 836-844

[19] Cui Y, Liu S, Zou N, et al. A dynamic parallel strategy for DOACROSS loops//Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region. Tokyo, Japan, 2018: 108-115

[20] Mehta S, Beeraka G, Yew P C. Tile size selection revisited. ACM Transactions on Architecture and Code Optimization, 2013, 10(4): Article No. 35

[21] Qasem A, Kennedy K. Profitable loop fusion and tiling using model-driven empirical search//Proceedings of the 20th Annual International Conference on Supercomputing. Cairns, Australia, 2006: 249-258

[22] Liu S, Cui Y, Jiang Q, et al. An efficient tile size selection model based on machine learning. Journal of Parallel and Distributed Computing, 2018, 121(2018): 27-41

[23] Shirako J, Sharma K, Fauzia N, et al. Analytical bounds for optimal tile size selection//Proceedings of the 21st International Conference on Compiler Construction. Tallinn, Estonia, 2012: 101-121

[24] Jiang B, Nain P, Towsley D. LRU cache under stationary requests. ACM SIGMETRICS Performance Evaluation Review, 2017, 45(2)

[25] Rivera G, Tseng C W. Tiling optimizations for 3D scientific computations//Proceedings of the ACM/IEEE Conference on Supercomputing. Dallas, USA, 2000: Article No. 32

[26] Mehta S, Garg R, Trivedi N, et al. TurboTiling: Leveraging prefetching to boost performance of tiled codes//Proceedings of the International Conference on Supercomputing. Istanbul, Turkey, 2016: Article No. 38

[27] Ghosh S, Martonosi M, Malik S. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. ACM Transactions on Programming Languages and Systems, 1999, 21(4): 703-746

[28] Whaley R C, Dongarra J J, Petitet A. Automated empirical optimizations of software and the ATLAS project. Parallel Computing, 2001, 27(1-2): 3-35

[29] Yuki T, Renganarayanan L, Rajopadhye S, et al. Automatic creation of tile size selection models//Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization. Toronto, Canada, 2010: 190-199

[30] Sato Y, Yuki T, Endo T. An autotuning framework for scalable execution of tiled code via iterative polyhedral compilation. ACM Transactions on Architecture and Code Optimization, 2019, 15(4): 1-23

[31] Wu X, Taylor V, Lively C, et al. Performance analysis and optimization of parallel scientific applications on CMP cluster systems//Proceedings of the International Conference on Parallel Processing-Workshops. 2008: 188-195

[32] Velivelli A C, Bryden K M. A cache-efficient implementation of the lattice Boltzmann method for the two-dimensional diffusion equation. Concurrency and Computation Practice & Experience, 2004, 16(14): 1415-1432

[33] Unnikrishnan P, Shirako J, Barton K, et al. A practical approach to DOACROSS parallelization//Proceedings of the 18th European Conference on Parallel Processing. Rhodes Island, Greece, 2012: 219-231

[34] Mccracken M E, Abraham J. Multiple-relaxation-time lattice-Boltzmann model for multiphase flow. Physical Review E, 2005, 71(3): 036701

[35] Yoshida H, Nagaoka M. Multiple-relaxation-time lattice Boltzmann model for the convection and anisotropic diffusion equation. Journal of Computational Physics, 2010, 229(20): 7774-7795

[36] Li W, Luo L S. Finite volume lattice Boltzmann method for nearly incompressible flows on arbitrary unstructured meshes. Communications in Computational Physics, 2016, 20: 301-324

[37] Xiang X, Ding C, Luo H, et al. HOTL: A higher order theory of locality//Proceedings of the 18th International Conference on Architecture Support for Programming Languages and Operating Systems. Houston, USA, 2013: 343-356

[38] Sabarimuthu J M, Venkatesh T G. Analytical miss rate calculation of L2 cache from the RD profile of L1 cache. IEEE Transactions on Computers, 2018, 67(1): 9-15

[39] Shi Q, Kurian G, Hijaz F, et al. LDAC: Locality-aware data access control for large-scale multicore cache hierarchies. ACM Transactions on Architecture and Code Optimization, 2016, 13(4): 1-28

[40] Cheng L, Murphy J, Liu Q, et al. Minimizing network traffic for distributed joins using lightweight locality-aware scheduling//Proceedings of the 24th European Conference on Parallel Processing. Torino, Italy, 2018: 293-305

[41] Baskaran M M, Hartono A, Tavarageri S, et al. Parameterized tiling revisited//Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization. Toronto, Canada, 2010: 200-209

[42] Venkat A, Mohammadi M S, Park J, et al. Automating wavefront parallelization for sparse matrix computations// Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Salt Lake City, UT, USA, 2016

[43] Liu Xiao-Xian, Zhao Rong-Cai, Zhao Jie, et al. Automatic parallel code generation for regular DOACROSS loops. Journal of Software, 2014, 25(6): 1154-1168(in Chinese)
(刘晓娴，赵荣彩，赵捷等. 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)

[44] Liu S, Cui Y, Zou N, et al. Revisiting the parallel strategy for DOACROSS loops. Journal of Computer Science and Technology, 2019, 34(2): 456-475

[45] Bondhugula U, Bandishti V, Pananilath I. Diamond tiling:

Tiling techniques to maximize parallelism for stencil computations. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(5): 1285-1298

[46] Bondhugula U, Acharya A, Cohen A. The PLuTo+algorithm: A practical approach for parallelization and locality optimization of affine loop nests. ACM Transactions on Programming Languages and Systems, 2016, 38(3): Article No. 12

[47] Liu Song, Zhao Bo, Jiang Qing, et al. A semi-automatic coarse-grained parallelization approach for loop optimization and irregular code sections . Chinese Journal of Computers, 2017, 40(9): 2127-2147(in Chinese)

(刘松，赵博，蒋庆等. 一种面向循环优化和非规则代码段的粗粒度半自动并行化方法. 计算机学报，2017，40(9)：2127-2147)

[48] Nguyen A, Satish N, Chhugani J, et al. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs //Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. New Orleans, USA, 2010: 1-13

[49] Tran N P, Lee M, Hong S. Performance optimization of 3D lattice Boltzmann flow solver on a GPU. Scientific Programming, 2017: 1-16

CUI Yuan-Zhen, M. S. His research interests include parallel computing and program optimization.

LIU Song, Ph. D., assistant researcher. His research interests include compiler optimization and high performance computing.

WANG Qian, M. S. Her research interests include code optimization and high performance computing.

WU Wei-Guo, Ph. D., professor, Ph. D. supervisor. His research interests include high performance computer architecture, storage system and embedded system.

Background

The Lattice Boltzmann Method (LBM) is a mesoscopic model between the molecular dynamics model of the fluid (microscopic level) and the continuous model (macro level). It is one of important methods of computational fluid dynamics (CFD). Because LBM can solve the linear problem explicitly, it is naturally suitable for high parallelism and is widely used in computer simulation of related research fields. Most of the computational fluid dynamics problems, including the lattice Boltzmann method, the mathematical representation of the core logic is large linear algebraic equations with complex intrinsic connections, and the computational program of its programming implementation is a large number of basic linear algebra subroutines with nested loops and template calculations. From the perspective of computer science, most of the scientific computing programs spend CPU time on multiple nested loops. Nested loops have become a hotspot for these scientific computing applications, which consumes the main time of running the program. This paper transforms the core loop of the LBM computation program to divide its huge iterative space, thereby improving the data reuse benefit. At the same time, when spatially dividing the LBM computation program, a tile size selection method, LBM_TSS, is proposed. LBM_TSS performs static analysis from the four aspects of the LBM computation program's memory access behavior, locality benefit, parallelism and synchronization overhead, and combines the empirical search method to select the tile size.

The work of this paper is mainly supported by the National Natural Science Foundation of China under Grant Nos. 91630206 and 61672423, the National Key Research and Development Plan under Grant No. 2017YFB0203003.