PLASMA 自适应调优与性能优化的设计与实现

吕渐春[1,2] 张云泉[1] 王 婷[1] 肖玄基[1,2]
(中国科学院软件所并行计算实验室 北京 100190)[1] (中国科学院研究生院 北京 100190)[2]

摘 要 PLASMA是一个高效的线性代数软件包,其数据分布结合分堆、细粒度并行以及乱序执行机制等大大提高了程序的性能。但PLASMA仍然存在一些问题,比如分块大小对程序性能的影响非常大,以及产生了大量的数据拷贝等。通过对比传统的LAPACK和PLASMA的实现机制,分析了PLASMA中存在的优势和不足,介绍了两种弥补PLASMA自身不足的方法。针对PLASMA的架构,经过大量的测试与分析,提出了边缘矩阵的概念并分析了其对性能的影响,据此提出了一种自适应调优的方法。并通过数据拷贝与计算并行的运行方式,进一步提高了PLASMA性能,最后通过大量的测试验证了该优化方法的效果。

关键词 LAPACK,PLASMA,自适应调优,优化
中图法分类号 TP302 **文献标识码** A

Design and Implementation for PLASMA Auto-tuning and Performance Optimizing

LV Jian-chun[1,2] ZHANG Yun-quan[1] WANG Ting[1] XIAO Xuan-ji[1,2]
(Lab of Parallel Computing, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)[1]
(Graduate University of Chinese Academy of Sciences, Beijing 100190, China)[2]

Abstract PLASMA is a high performance linear algebra package. Its innovative approach such as block data layout with tiling, fine grain parallelism and out of order execution mechanism greatly improves the performance of the program. However, there are still some problems, for example, the size of block plays a severe role in performance and this mechanism brings some data copy. In this paper, by comparing the traditional LAPACK and PLASMA's mechanism, we aimed to analyze the advantages and disadvantages of PLASMA, and proposed two methods to make up the disadvantages. As to the PLASMA architecture, we proposed a concept of marginal matrix and analysed their impact on performance via extensive testing and analysis, and then proposed a method of auto-tuning. Besides, we also found a way to further improve the performance of PLASMA, which is adopting data transmission and computing in parallel. Finally, we verified the effect of optimized method by doing a large number of testing.

Keywords LAPACK, PLASMA, Auto-tuning, Optimization

1 引言

LAPACK(Linear Algebra Package)[1]是采用标准Fortran77编写的线性代数库,用来求解大多数常见的线性代数问题。LAPACK主要利用分块技术并且尽可能多地调用level-3BLAS子程序来适应多级存储层次,提高数据局部性从而提高性能,并且随着多核平台的不断发展,又提出了调用并行版BLAS程序(OpenBLAS、ATLAS等)和look-ahead技术来获得并行性。

PLASMA(The Parallel Linear Algebra for Scalable Multi-core Architectures)[2]是由美国田纳西大学ICL(The Innovative Computing Laboratory)部门的Jack Dongarra教授等人发起的项目。该项目的目标是设计一个用来编写可移植、可扩展的高性能并行程序的软件框架。其主要工作是设计新的求解线性代数问题的高效算法以适应多核平台的新特性。其中主要包含了一般矩阵LU分解、正定矩阵LU分解和一般矩阵QR分解算法的实现。PLASMA提出了一种新的算法,主要利用Block Data Layout和分堆(Tiling)、细粒度并行和乱序执行的思想。

BLAS(Basic Linear Algebra Subprograms)[3]是采用Fortran77编写的基本线性代数子函数,为向量和矩阵操作提供标准的构建模块,采用单线程来操作。

OpenBLAS[4]是由中国科学院软件研究所并行软件与计算科学实验室的张先轶等人发起的开源项目,基于GotoBLAS2[5] 1.13 BSD版本开发的一款高性能BLAS库。当前主要工作为针对龙芯3A平台提供BLAS的高效实现。

到稿日期:2011-08-11 返修日期:2011-11-11 本文受国家"863"曙光6000千万亿次高效能计算机系统研制项目(2009AA01A129),国家"863"高效能计算机及网格服务重大项目(2009AA01A134),国家重大专项核高基项目(2009ZX01036-001-002),中国科学院知识创新工程重大项目课题(KGCX1-YW-13),国家重大科研装备研制项目(ZDYZ2008-2),国家自然科学基金项目(61100073, 61133005, 61100066)资助。
吕渐春(1987—),男,硕士生,主要研究方向为并行软件与计算科学,E-mail:jianchunlv@gmail.com;张云泉 博士,研究员;王 婷 博士,副研究员;肖玄基 硕士。

本文对 LAPACK 和 PLASMA 的架构做了简单的介绍，通过调用 BLAS 和 OpenBLAS 作为底层 BLAS 库在多种平台上进行了测试，提出了一种适合 PLASMA 架构的自适应调优的方法，通过对 PLASMA 架构的分析，提出了性能优化策略，并对优化后的性能进行了测试。本文的所有测试都是在以下两种平台上进行的，如无特殊说明，本文测试平台为，Intel 平台：Xeon 5472 处理器，二路四核，16GB 内存；AMD 平台：Opteron 8378 处理器，八路四核，64GB 内存。

2 传统线性代数分解算法架构分析

本部分以一般矩阵 QR 分解的实现方式为例，对 LAPACK 做了简单的介绍[6]。

QR 分解算法[7]是用矩阵分解的方法来求解最小二乘问题等，对矩阵 A 的 QR 分解为：$A=QR$，其中 Q 为正交矩阵，R 为上三角矩阵，一般 QR 分解有 Schmidt 正交化方法、初等旋转变换法和 Householder 变化法 3 种，其中 4 * 4 矩阵 Householder 的实现方法如图 1 所示，其中 A_1、A_2、A_3 等为当前迭代的剩余子矩阵。

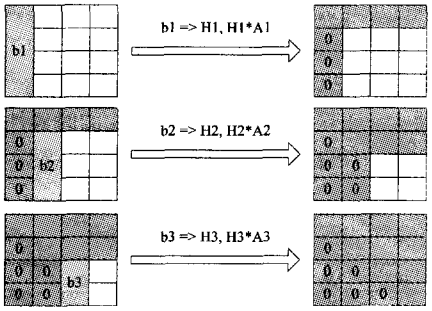


图 1 QR 分解基本实现方法[8]

算法由 $n-1$ 步循环组成，每一步首先根据当前列求解其 Householder 变换，然后根据求解出的矩阵 H_n 对剩余子矩阵进行更新，更新完之后则当前列下面的元素全部变为 0，求解完成后，右上角的三角矩阵即为 R，所求的 $Q=H_m * H_{m-1} * \cdots * H_1$，此时 $A=QR$。

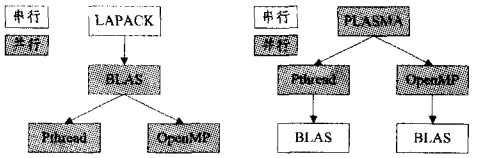
由于以上算法存在大量的 Level-2 运算和外积更新，而且按行(列)的存储方式也会增加 cache 失效率，为了改善程序的性能，LAPACK 采用了分块的方法来增加数据的重用，比如对一个 12 * 12 的矩阵，若块大小(NB)为 3，首先根据图 1 的算法产生 H_1、H_2、H_3，不同于以上算法将所有的 H_i 应用于子矩阵 A，而是先产生一个块 $H_1H_2H_3$ 的乘积，然后用这个乘积来更新子矩阵。分块算法不断循环执行以下两个基本步骤[7]：

1)块分解(panel factorization)：利用不分块 QR 算法 DGEQR2 直接对当前 $nb * nb$ 大小矩阵进行分解，这一步主要利用 Level-2 BLAS 操作完成；

2)尾子矩阵更新(trailing submatrix update)：利用上一步的结果更新右端剩余的矩阵部分，此步骤利用 Level-3 BLAS 操作完成。

经过分块操作后可知，程序大部分工作由 Level-3 BLAS 完成，减少了子矩阵的更新次数，同时提高了 cache 的重用率，从而增加了程序的局部性[12]。

BLAS 中的程序比较容易并行化，传统的 LAPACK 是通过调用并行的 BLAS 来获得并行性[6]，如图 2(a)所示，这种架构既保持了原有接口的一致性，又能使程序并行化进而获得比较高的性能。



(a) 基于并行 BLAS 架构　　(b) PLASMA 中的架构

图 2 并行分解算法架构[9]

3 PLASMA 架构分析

通过上一节对 LAPACK 架构的分析可知，传统的 LAPACK 的运行方法如图 2(a)所示，LAPACK 本身不并行，只能通过调用并行的 BLAS 库获得并行性，这样在程序本身便存在大量的串行操作，由 Amdahl 定律可知，串行部分会限制程序的加速比。为了能够进一步获得加速比，PLASMA 提出了全新的设计方法，如图 2(b)所示，通过自身的并行来调用底层的 BLAS 库，其中 BLAS 库必须设置为单线程，不再依赖 BLAS 而获得并行，这样可以进一步挖掘程序中可能并行的部分，实践证明已经获得了不错的加速比[8,9]。

PLASMA 打破了以往的设计思路，在计算前对数据进行了分堆，所有的计算操作都是在分堆之后的数据上进行，计算完成后再将堆内的数据回写到原数据中。因此 PLASMA 的操作主要由以下 3 步来完成：

1)分堆。先开辟一段空间，将原数据拷贝到堆中，拷贝的过程对堆中的数据进行了重新组合，使得在逻辑上相邻的元素在物理内存上也是相邻的。分堆示意图如图 3(Lapack_to_tle)所示。分堆后可以大大地减少缓存缺失，提高缓存的利用率。

2)计算。所有的计算都是在分堆后的数据上进行。因为 PLASMA 改变了传统方法的架构，所以其计算次数也发生了一些变化。以 QR 分解举例，传统的 LAPACK 计算次数为 $2N2(M-N/3)$，而 PLASMA 需要的计算次数为 $2N2(M-N/3)(1+s/(4b))$，其中 s 为外块和内块的比值[11]。虽然增加了计算次数，但其多核情况下的运行效率还是比传统的 LAPACK 有了很大的提高。

3)回写。回写就是分堆的逆过程，将计算完成后的数据按照原方法回写到原矩阵中。其过程如图 3(Tile_to_lapack)所示。

仍然以 QR 分解为例，图 3 展示了 PLASMA 的工作过程，包括分堆、计算和回写。假设把原矩阵分成 3 * 3 个分块，通过分堆，对数组中的元素进行了重新安排，每个分块被分配到一块连续的内存空间，这样逻辑上相邻的元素在物理位置上也相邻，大大地增加了 cache 重用率和 TLB 的命中率[13]，从而提高了程序的效率，计算过程被抽象为一个有向无环图，可以乱序执行，每个分块只要满足依赖条件就可以执行，因此获得了很高的并行性。

PLASMA 的分块由两层组成，外块(tile)和内块(inner block)，本文中若无特殊说明，提到的块是这两个块的统称。其中外块决定了分堆的大小，一旦确定之后不可改变，程序会按照该外块大小对矩阵进行分堆，内块是对外块的进一步分块，其中外块大小必须为内块的整数倍。

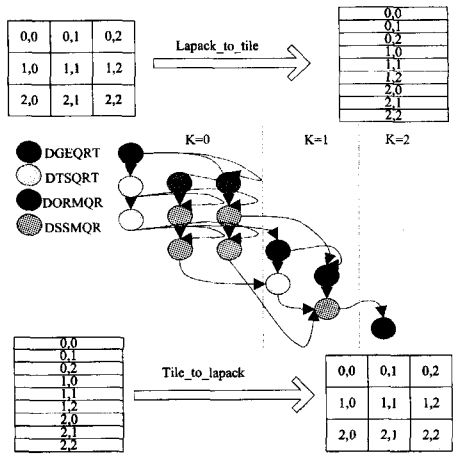


图 3 PLASMA 中的并行 QR 分解算法的全过程

4 PLASMA 自适应调优

目前稠密矩阵采用的自适应策略主要有 3 种模型，生成和测试模型(generate-and-test model)，性能分析模型(analytical performance model)，忽略缓存模型(cache-oblivious model)[10]。

生成测试模型是本次自适应调优借鉴的一个过程，其主要思想为用测试的方法来找出最优参数。本设计通过简单的穷举测试等，确定出最优内块的范围，为进一步的分析提供数据支持。

性能分析模型也是本次自适应调优借鉴的一个过程，其主要思想是通过分析当前环境或者程序的特性，给出比较优化的参数。本设计通过对程序架构的分析，总结出了一套适合该架构的分析方法，在常数时间内得到最优或者接近最优的块大小。

忽略缓存模型的主要思想为在运行时递归地减少分块的大小，总会有一种分块的方式适合程序的运行，从而使性能达到一定的优化，但是这种方法优化的效率并不是很高，可能会错过接近最优分块的机会。而且在 PLASMA 的架构中，分块大小一旦确定后，将不可更改，因此此种方法不适合 PLASMA 的自适应调优。

本方法借鉴以往的性能分析方法和生成测试方法，对这两种方法进行了整合，针对矩阵为方阵的情况，提出了一种适合 PLASMA 架构的混合方法。

4.1 生成和测试

生成测试法是最简单的一种自适应调优策略，因其与硬件条件或者其它依赖条件无关的特性，在某些情况下也是首选的方法。比如在本设计中，因为 PLASMA 要调用 BLAS 层，但是事先不知道用户会调用哪个 BLAS，也不知道像 MKL 等非开源 BLAS 的内部工作机制，因此不易通过分析的方法得出最优内块，只能通过生成测试的方法来完成。通过测试 PLASMA 计算函数所有块大小的性能来寻找最优块的方法虽然可行，但是需要大量的测试时间。

本方法能够找出最优外块的范围和最优内块的范围，由于矩阵的规模跟分块有着很大的关系，并且把所有规模的矩阵都测一遍来选取最优值耗时太长，因此不能找出一个合适的分块来适合所有矩阵规模的情况，本过程只是为以后的性能分析过程提供基础数据，在程序运行时动态地分析出合适的分块组合。

4.2 性能分析

由第 3 节的分析可知，PLASMA 并不负责底层的具体操作，而是通过调用 BLAS 来完成，因而对 PLASMA 的自适应调优增加了难度，表 1 是在 Intle 平台上分别用不同的 BLAS 和不同的矩阵规模的测试结果，PLASMA 版本为2.1.0，测试函数为 DGEQRF，线程数均为 4，选取每种测试结果的前十名最优分块组合，通过比较表 1(a)和表 1(b)可以发现，在矩阵规模相同的情况下，不同的 BLAS 最优的分块并不相同；通过比较表 1(b)和表 1(c)可以发现，在相同的 BLAS 下，不同的矩阵规模对应的最优分块也不相同。因此，不易直接通过性能分析法来找出最优外块和内块的组合，本设计将寻找最优外块和内块范围的任务用生成测试法来完成。

PLASMA 中分块大小对程序的影响比较大，程序的性能会随着分块的大小的增加而出现很多波动，并不是平滑地上升和下降，这是因为外块大小必须为内块大小的整数倍，对于不同的外块大小不一定有合适的内块，比如外块大小为 194 的时候，因为 194 的因数只有 2 和 97，内块大小只能为 2 或 97，如果 2 和 97 与最优内块差别比较大，则外块为 194 时性能会非常低，而 192 的因数为 2、4、6、8、12、16、24、32、48、64、96，其覆盖面比较广，有更多的机会接近最优内块。因此虽然这两种外块大小很接近，但是外块为 194 的时候性能可能会很差，而 192 的时候性能要高很多。

表 1 Intel 平台上不同 BLAS，不同矩阵规模的测试结果

(a) BLAS 4000 * 4000

排序	外块	内块	时间(s)
1	200	10	9.011
2	200	8	9.019
3	192	8	9.047
4	252	12	9.048
5	252	6	9.053
6	192	12	9.057
7	192	6	9.072
8	224	8	9.076
9	192	16	9.084
10	236	4	9.099

(b) OpenBLAS 4000 * 4000

排序	外块	内块	时间(s)
1	448	64	2.585
2	448	56	2.595
3	336	56	2.597
4	312	52	2.598
5	336	48	2.599
6	364	52	2.604
7	456	76	2.612
8	340	68	2.614
9	372	62	2.619
10	312	78	2.621

(c) OpenBLAS 4234 * 4234

排序	外块	内块	时间(s)
1	476	68	3.024
2	480	60	3.047
3	480	80	3.058
4	480	48	3.062
5	392	56	3.063
6	360	60	3.066
7	360	72	3.075
8	480	96	3.078
9	396	66	3.084
10	532	76	3.091

PLASMA 的最优外块也不是一成不变的，而会根据矩阵的规模发生一定的变化，比如表 1 所列，由表 1(b)和(c)可以看出，在矩阵为 4000 * 4000 的时候可能为 448，而在 4234 * 4234 的时候可能为 476。这是由于“边缘矩阵”的性能引起的，分块后位于原矩阵最右面一列和最下面一行的矩阵称之为“边缘矩阵”，如果分块不够合适，比如 4000 的时候选择 476 或者 4234 的时候选择 448，这时会导致“边缘矩阵”细长或者矮宽，极端情况下，如果边缘矩阵只有一行或者一列，这

时同样是在 ONB^2)的数据上,但计算将会由 $O(NB^3)$退化为 $O(NB^2)$,其中 NB 为外块的大小,比如矩阵乘(GEMM)将退化为矩阵向量乘(GEMV),这样会大大影响计算的效率。边缘矩阵的计算次数所占的比例在整个计算中的比例比较大,假设矩阵有 $NT*NT$ 个分块,则总计算次数为 $1^2+2^2+3^2+\cdots+NT^2=(NT*(NT+1)*(2*NT+1))/6$ 次。而边缘矩阵的计算次数为 NT^2 次,边缘计算次数占总计算的比例为 $6*NT/(2NT^2+3NT+1)$。虽然所占比例会随 NT 的增大而减小,当 NT 为 20 时,仍然占据 14%左右的计算次数。选择合适的分块,使边缘矩阵尽量方阵化,也就是将其与矩阵规模做除法,小数部分比较大或者为 0 的分块,可以有效地发挥出 PLASMA 的性能。

为了保证程序充分的并行性,还应该保证总分块数远远大于线程数,若分块过少而线程数较多将限制多线程的优势。有了最优外块和最优内块的范围之后,在程序运行时,根据输入矩阵的规模和线程数,通过分析法,找出满足条件的分块组合。

通过性能分析,能够在已知最优外块和内块大致范围的情况下,常数时间内给出准最优外快的大小,但是需要生成测试法的配合,不能单独处理。

4.3 混合调优法

本设计的方法容纳了以上两种过程,其思想为通过生成和测试,找出其中的最优内块和最优外块大小的范围,然后运行时通过性能分析选择合适的分块。针对方阵的情况,其流程图如图 4 所示,其中测试部分可以用两个样本矩阵规模来找出最优外块的范围,首先随机选择一个矩阵规模,比如 4000,测试完成后确定出最优外块的初等范围,比如 336～448,然后再取该区间平均数的一半与原矩阵规模相加作为第二次的矩阵样本。因此第二次的矩阵样本为 4000+(336+448)/4=4196,再用 4196 规模的矩阵进行测试,这样便可以找出比较全面的外块的范围,其中寻找外块范围时也用了分析法的思想,比如只查找特定外块的数据,这些数据与矩阵规模做除法,小数部分大于 0.8 或者等于 0,只有这样的数据才可能为候选块,这样可以筛掉大部分的测试数据,从而缩短寻找最优外块范围的时间。测试完成后,对最优外块范围内的数进行进一步的测试,用来找出其对应的最优内块,NB 为外块,IB 为内块,则最后的产出为 NBMIN、NBMAX、IBMIN、IBMAX。性能分析部分则首先确定一个初始分块,当矩阵规模比较小并且线程数比较多时,为了保证充分并行,应该满足分块数大于线程数,也就是保证 $ARG*THREADS*NB^2<M^2$(其中 M 为矩阵的行或列的大小,THREADS 为线程数,在这里 ARG 取 4,保证分块数大于等于线程数的 4 倍),若 NBINIT 为初始化分块,此时应该初始化 NBINIT 为 sqrt((M*M)/(4*THREADS)),在矩阵比较大的情况下,此时设置初始分块大小为最优分块范围的中间值即可。因此,NBINIT=MIN((NBMAX+NBMIN)/2, sqrt((M*M)/(4*THREADS)))。找出 NBINIT 之后,在该数的两侧进行搜索,其中搜索的必要条件为内块必须在最优内块范围内,若同时外块满足 $M\%NB==0||(M\%NB)/NB>0.8$ 则直接返回,如果 $(M\%NB)/NB<0.8$ 但是比以前的 $(M\%NB)/NB$ 值要大,则替换掉之前的记录,即如果没有满足 $M\%NB==0||(M\%NB)/NB>0.8$ 条件的情况,则选 $(M\%NB)/NB$ 值最大的一组组合。

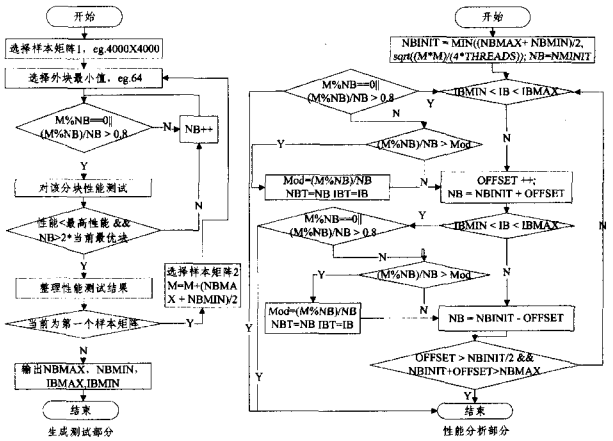
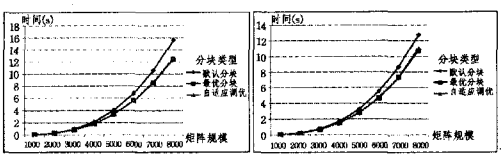


图 4 自适应调优流程图

实践证明,本方法能够有效地在很短的时间内找出准最优值,并且给出准最优外块大小与其对应内块的大小,测试结果在下节中给出。

4.4 自适应调优的测试

测试结果如图 5 所示,分别在 AMD 平台和 Intel 平台上进行了测试,两个平台都用的 8 线程,测试函数为 DGEQRF,图中显示了运行时间随矩阵规模的变化情况。



(a) AMD 平台 (b) Intel 平台

图 5 两个平台 8 线程分块策略随矩阵规模的变化情况

由图 5 可见,在两个平台上,自适应调优后的性能与最优性能基本相当,最多时会有接近 2%的性能差距,默认分块的性能则与最优块的性能差距比较大,矩阵规模为 8000 时会有 20%左右的性能差距。

本方法在两个平台上均取得了不错的效果,可以在短时间内获取到内块和外块的范围,通过生成测试法得到初试数据之后,均可以在常数的时间内找出其所需要的准最优分块。

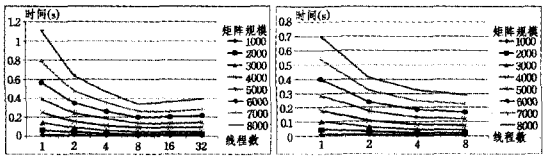
5 PLASMA 性能优化

本章针对前面对传统 LAPACK 算法和 PLASMA 架构的分析,提出了一种优化策略:数据拷贝与计算并行策略。本节以双精度一般矩阵 LU 分解(DGETRF)为例,对该方法进行了说明和测试。

5.1 数据拷贝与计算并行的优化方法

通过第 3 节分析可知,PLASMA 为了完成一个计算,需要进行分堆和回写,这给程序带来了大量的数据拷贝,在 PLASMA 中,虽然这些数据拷贝也是通过并行来完成的,但是因为传输带宽的限制,其效率仍然十分低下,大部分的 CPU 处在空闲状态,图 6 为数据拷贝时间随线程数的变化情况。

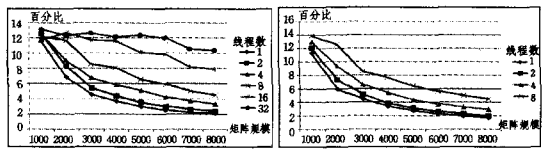
由图 6 可知,在 AMD 平台,4 线程之后加速比变得不是很明显,在 16 线程和 32 线程,甚至还出现了减速;在 Intel 平台,2 线程之后,加速比开始变得不是很明显,4 线程和 8 线程基本没有加速比。本节展示了一种进一步提高其性能的方法,通过拷贝与计算并行的方式来减少数据拷贝对性能的影响。



(a) AMD平台　　(b) Intel平台

图6　两个平台拷贝时间随线程数的变化情况

数据拷贝占总运行时间的比例如图7所示，由前面的图6可知，AMD平台的数据的拷贝时间在4线程之后变化不大，而Intel平台在4线程之后基本没有变化，但是计算时间却随着线程数的增大而继续减少，因此，其占的比例会随着线程数的增加而增加。数据拷贝的时间复杂度为$O(N^2)$，而计算部分的时间复杂度为$O(N^3)$，所以，数据拷贝所占的比例会随着矩阵规模的增大而减小。由图7(a)可知，在32线程的情况下，所有矩阵规模的拷贝时间均在10%以上，小矩阵的情况下所占的比例也很大，矩阵规模为1000时Intel平台可以占到14%左右，AMD平台也在12%以上，因此如果可以缩短这部分的时间，将会使程序继续获得一定的加速比。



(a) AMD平台　　(b) Intel平台

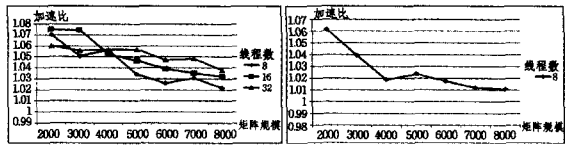
图7　数据拷贝所占的比例随矩阵规模的变化情况

将数据拷贝与计算并行是本节采用的提高PLASMA性能的方法。PLASMA中LU分解的过程与第3节描述的QR分解的计算过程类似，如图3所示，假设有3*3个分块，需要经过3次迭代。在每次分块计算之前都需判断是否是第一次迭代，如果是第一次迭代则将数据从原矩阵拷贝到堆中，如果不是，则说明数据已经在堆中，无需再次拷贝，每次迭代结束都会产生很多块内计算已经完全完成但不再使用的分块，此时，可以利用已有的线程将那些不再使用的分块回写到原矩阵中，与其他计算线程并行进行。

拷贝过程分为分堆和回写两部分，其中分堆部分获得的加速比并不是很大，因为在程序刚开始运行时，会需要大量的数据，因此，拷贝仍然比较集中。在回写阶段则可以获得比较不错的加速比，因为回写阶段产生的数据比较松散，每一步迭代都会产生一些计算完成的块，并且随着迭代次数的增加产生的块数越来越少，最后一次迭代只产生一个块，回写过程可以分散在整个运行过程中，这样便可以很好地让拷贝和计算并行执行。

5.2　性能测试

本测试用的PALSMA版本为2.1.0。将分堆过程和回写过程都整合到计算过程中，最后在AMD和Intel平台进行了测试，测试函数为DGETRF，加速比如图8所示，线程数较少的情况下加速不是很明显，因此AMD平台只列出了8、16和32线程的情况，Intel平台只列出了8线程的情况。可以发现，AMD平台在矩阵规模小于4000时，因为分块比较少，此时线程过多会导致产生结果数据的时间比较集中，所以32线程的加速比并不是很好；当矩阵规模大于4000时，分块比较多，此时程序可以充分地并行，线程数越多时加速比越大。Intel平台采用了2路4核共8核，因此只给出了8线程的情况，在小矩阵的情况下也获得了不错的加速比，随着矩阵规模的增大，因为数据拷贝所占的比例越来越小，所以争取回来的拷贝时所占的比例会随着矩阵规模的增大而减小。



(a) AMD平台　　(b) Intel平台

图8　两个平台加速后的性能测试结果

结束语　本文主要做了两部分的工作，通过分析PLASMA的架构和PLASMA的乱序执行和细粒度并行的工作方式，以及对比传统的线性代数分解算法和PLASMA中线性代数分解使用的分堆算法，针对PLASMA的架构进行了自适应调优和性能优化。其中自适应调优部分提出了两个过程，并对这两个过程进行了整合，生成测试法通过对可能的分块的测试来确定最优外块和内块的范围，分析法可以根据矩阵的规模快速地定位出合适的外块和内块组合。这两个过程的整合就是通过生成测试过程确定出最优块所在的范围作为基础数据，然后在运行时根据矩阵的规模和线程数，动态分析出最优外块和内块的组合，但是目前测试的平台还不是很多，下一步可用更多平台来对此策略进行测试。本次自适应调优只是针对方阵的情况进行了分析，下一步可以用同样的思路对非方阵的情况进行分析，结合多平台的测试，进一步完善该自适应调优策略。在性能优化部分中，提出了拷贝与计算并行的方法，该方法在多核和小矩阵的情况下获得了一定的加速比。

参考文献

[1] LAPACK-3.3.0[OL]. http://www.netlib.org/lapack/

[2] PLASMA-2.1.0 [OL]. http://icl.cs.utk.edu/plasma/software/

[3] BLAS[OL]. http://www.netlib.org/blas/

[4] OpenBLAS[OL]. http://xianyi.github.com/OpenBLAS/

[5] GotoBLAS [OL]. http://www.tacc.utexas.edu/resources/software/

[6] Anderson E, Bai Z, Bischof C, et al. LAPACK Users' Guide[Z]. SIAM, 1992

[7] Golub G H, Van Loan C F. Matrix Computations[M]. Beijing: Post & Telecom Press, 1996: 223-226

[8] Buttari A, Langou J, Kurzak J, et al. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures[R]. ICL Technical Report, UT-CS-07-600(also LAPACK Working Note 191). September 2007

[9] Buttari A, Langou J, Kurzak J, et al. Parallel tiled QR factorization for multicore architectures, Concurrency Computat[J]. Pract. Exper., 2008, 20(13): 1573-1590

[10] Kulkarni M, Pingali K. An Experimental Study of Self-Optimizing Dense Linear Algebra Software[J]. Proceedings of the IEEE, 2008, 96(5): 832-848

[11] Song F, Ltaief H, Hadri B, et al. Scalable Tile Communication-Avoiding QR Factorization on Multicore Cluster Systems[C]// Proceedings of the ACM/IEEE. Washington, DC, USA, November, 2010

[12] Castaldo A M, Whaley R C. Scaling LAPACK Panel Operations Using Parallel Cache Assignment[C]// Proceedings of the 2010 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Bangalore, India, January 2010: 223-231

[13] 王磊. PLASMA并行LU分解高效实现机制分析与测试[C]// . 全国高性能计算年会. 2009