基于 BLACS 的 2.5D 并行矩阵乘法

廖 霞[1] 李胜国[1] 卢宇彤[2] 杨灿群[1),3)]

[1)](国防科技大学计算机学院 长沙 410073)

[2)](中山大学国家超级计算中心 广州 510006)

[3)](国防科技大学并行与分布处理重点实验室 长沙 410073)

摘 要 并行矩阵乘法是线性代数中最重要的基本运算之一,同时也是许多科学应用的基石.随着高性能计算(HPC)向 E 级计算发展,并行矩阵乘法的通信开销所占比重越来越大.如何降低并行矩阵乘法的通信开销,提高并行矩阵乘的可扩展性是当前研究的热点之一.本文提出一种新型的分布式并行稠密矩阵乘算法,即 2.5D 版本的 PUMMA(Parallel Universal Matrix Multiplication Algorithm)算法,该算法是通过将初始的进程分成 c 组,利用计算节点的额外内存,在每个进程组上同时存储矩阵 $\boldsymbol{A}$、$\boldsymbol{B}$ 和执行 $1/c$ 的 PUMMA 算法,最后通过规约操作来得到矩阵乘的最终结果.本文基于 BLACS(Basic Linear Algebra Communication Subprograms)通信库实现了一种从 2D 到 2.5D 的新型数据重分配算法,与 PUMMA 算法相结合,最终得到 2.5D PUMMA 算法,可直接替换 PDGEMM(Parallel Double-precision General Matrix-matrix Multiplication),具有良好的可移植性.与国际标准算法库 ScaLAPACK(Scalable Linear Algebra PACKage)中的 PDGEMM 等经典 2D 算法相比,本文算法缩减了通信次数,提高了数据局部性,具有更好的可扩展性.在进程数较多时,例如 4096 进程时,系统测试表明相对 PDGEMM 的加速比可达到 2.20~2.93.进一步地,本文将 2.5D PUMMA 算法应用于加速计算对称三对角矩阵的特征值分解,其加速比可达到 1.2 以上.本文通过大量数值算例分析了 2.5D PUMMA 算法的性能,并给出了实用性建议和总结了未来的工作.

关键词 2.5D 并行矩阵乘算法;ScaLAPACK;PUMMA 矩阵乘算法;SUMMA 算法;分布式并行

中图法分类号 TP301 **DOI 号** 10.11897/SP.J.1016.2020.01037

New 2.5D Parallel Matrix Multiplication Algorithm Based on BLACS

LIAO Xia[1] LI Sheng-Guo[1] LU Yu-Tong[2] YANG Can-Qun[1),3)]

[1)](*College of Computer Science, National University of Defense Technology, Changsha* 410073)

[2)](*National Supercomputer Center in Guangzhou, Sun Yat-Sen University, Guangzhou* 510006)

[3)](*Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha* 410073)

Abstract Matrix-matrix multiplication is one of the most important basic operations in linear algebra and a building block of many scientific applications. As HPC (High Performance Computing) moves towards Exa-scale, the degree of parallelism of HPC systems is increasing. The communication cost of parallel matrix multiplication will be more and more important. How to reduce the communication cost, improve the scalability of parallel matrix multiplication and make full use of the advantages of supercomputer computing resources are well-studied subjects. In this paper, a novel distributed parallel dense matrix multiplication algorithm is proposed, which can be regarded as 2.5D PUMMA (Parallel Universal Matrix Multiplication Algorithm) algorithm. The underlying idea of this implementation is improving the scalability by decreasing the data

收稿日期:2020-02-01;在线发布日期:2020-08-26.本课题得到科技部重点研发计划项目(2018YFB0204301)、国家重点研发计划(2018YFB0204303)、国家自然科学基金项目(61872392,U1811461)、国家数值风洞项目(NNW2019ZT6-B20、NNW2019ZT6-B21、NNW2019ZT5-A10)、广东省自然科学基金(2018B030312002)、广东省"珠江人才计划"引进创新创业团队(2016ZT06D211)、湖南省面上项目(2019JJ40339)与校科研项目(ZK18-03-01)资助.**廖 霞**,博士研究生,主要研究方向为高性能计算、大数据分析与处理和并行计算.E-mail:liaoxia@nudt.edu.cn.**李胜国**,博士,助理研究员,主要研究方向为高性能计算、计算流体动力学、数值代数和特征值问题.**卢宇彤**,博士,教授,中国计算机学会(CCF)会员,主要研究领域为高性能计算、容错计算和系统软件.**杨灿群**,博士,研究员,主要研究领域为高性能计算、异构计算和大规模并行软件优化.

transfer volume between processors at the cost of the extra memory usage. The 2.5D matrix multiplication algorithm in this paper firstly divides the initial processes into c groups, stores matrix $\boldsymbol{A}$ and $\boldsymbol{B}$, and perform $1/c$ PUMMA algorithm simultaneously on each process group by utilizing the extra memory on the computing nodes, and finally gets the final result of matrix multiplication through MPI communications. Based on BLACS(Basic Linear Algebra Communication Subprograms), this paper implements a new data redistribution algorithm from 2D to 2.5D, combined with the PUMMA algorithm, and finally gets the 2.5D PUMMA algorithm which can directly replace PDGEMM (Parallel Double-precision General Matrix-matrix Multiply). Compared with classic 2D algorithms such as PDGEMM in ScaLAPACK (Scalable Linear Algebra PACKage), the algorithm in this paper reduces the number of communications, improves data locality, and has better scalability. The performance experiments are carried out on a supercomputer system with 300 computing nodes, each of which consists of two 10-core Xeon E5-2692 v3 CPUs. These nodes are interconnected by Tianhe 2 interconnection networks (TH-Express 2) with fat tree structure. The effectiveness and efficiency of the 2.5D PUMMA algorithm are evaluated with various of the matrix size, degree of blocking, the stack size (duplication factor) and the process number. In the case that the number of processes is high, such as 4096 processes, system tests show that the acceleration ratio relative to PDGEMM can reach 2.20—2.93. When the number of processes is small, such as 64 or 256 processes, the performance of the 2.5D PUMMA algorithm may be worse than the PDGEMM function. In this case, besides the additional overhead of data redistribution, the computation is the dominant factor rather than the communication. Furthermore, the 2.5D PUMMA algorithm proposed in this paper is applied to solve symmetric eigenvalue problems, which is used to accelerate the tridiagonal eigenvalue decomposition step. The speedup over the original implementation in ScaLAPACK can reach more than 1.2. The next stage of work is to study the 2.5D PUMMA algorithm for special matrices, such as Cauchy matrix, Toeplitz matrix and Vandermonde matrix, which are widely used in scientific, industrial and clinical fields. In this paper, the performance of 2.5D PUMMA algorithm is analyzed through a large number of numerical experiments, practical suggestions are given and the future work is summarized and forecasted.

Keywords 2.5D parallel matrix multiplication algorithm; ScaLAPACK; parallel universal matrix multiplication algorithm; scalable universal matrix multiplication algorithm; distributed parallel

1 引 言

并行矩阵乘法是科学计算、大数据分析和机器学习中的重要运算，也是 LAPACK[1]、ScaLAPACK[2] 库的重要运算函数. 面向未来 E 级计算机，如何提高并行矩阵乘法的可扩展性，充分发挥巨型机计算资源多的优势，成为大家研究的焦点. 本文的研究目标就是改进 ScaLAPACK 库中并行矩阵乘函数(PDGEMM)的可扩展性，介绍如何基于 BLACS 进行实现，进而可简单地替换具体应用程序中的 PDGEMM 函数，从而提高应用程序在巨型机上的性能.

根据进程的网络拓扑，并行矩阵乘算法可大致分为 1D 算法、2D 算法和 3D 算法[3-4]. 目前常用的算法都是 2D 算法，例如 Cannon[5]、Fox[6]、SUMMA[7] (Scalable Universal Matrix Multiplication Algorithm) 算法等. 2D 算法是将矩阵分布式地存储在二维网格的处理器上，并在 2D 网格上进行数据通信. 与 2D 算法相对应，Agarwal 等人提出了一种 3D 矩阵乘算法[8]，该算法是将进程组织成 3D 网格，并冗余地存储矩阵 $\boldsymbol{A}$，$\boldsymbol{B}$ 和矩阵 $\boldsymbol{C}$ 的临时块，各进程独立地对 $\boldsymbol{C}$ 的各块副本进行更新，最后通过规约来得到矩阵乘的结果. 虽然 3D 算法在通信次数和通信量方面都可证明是最优的[9-13]，但是需要更多的计算资源(例如内存和计算节点等)，并不是很实用.

ScaLAPACK 库中的 PDGEMM 函数是基于 2D 算法——DIMMA(Distribution-Independent Matrix Multiplication)算法. DIMMA[14] 是 SUMMA 的优化版本,进一步利用了 LCM(Least Common Multiple)块的概念,并采用了流水线通信方案,可有效地实现计算与通信的重叠.

在 2011 年,Solomonik、Demmel 等人[15-16] 提出了 2.5D 的 Cannon 矩阵乘法,其核心思想是利用计算节点额外的内存,在 2D 进程网格上模拟 3D 算法,从而降低通信开销. 在计算资源的需求方面,介于 2D 算法与 3D 算法之间. 相较于 2D 算法,2.5D 算法具有更好的数据局部性、更少的通信次数等优点. 后来很多学者[17-19] 对 2.5D 的矩阵乘算法进行研究,例如日本的 Mukunoki、Imamura 实现了基于 SUMMA 的 2.5D 并行矩阵乘算法[20],并考虑了 2D 进程拓扑向 2.5D 进程拓扑的数据转换开销,研究表明即使考虑了数据的转换开销,2.5D 的矩阵乘算法仍可比 ScaLAPACK 中的 PDGEMM 函数快. Solomonik 等人[15] 实现了 2.5D Cannon 算法,但其只适用于方形进程网格,也没有考虑从 2D 到 2.5D 的数据转换开销.

作者和其他学者[21-23] 研究发现,许多实际应用问题中的矩阵乘法占据了绝大部分的计算时间,而且矩阵往往是具有低秩结构的,通常称为秩结构矩阵(Rank-structured matrix),例如三对角特征值问题[24] 等. 但是,2.5D SUMMA 算法[20] 是无法利用该低秩结构的. 虽然 2.5D Cannon 算法可以利用该特殊结构,但是该算法没有有效的从 2D 到 2.5D 的数据重分配算法,具体可参看第 3.3 节的讨论,文献[20]中的数据重分配算法并不适用于 2.5D Cannon 算法. 为了利用矩阵块的低秩性,本文则采取研究 PUMMA 算法[9,25],结合文献[20]中的数据重分配方法,提出了 2.5D PUMMA 算法,并基于 ScaLAPACK 库中的 BLACS 通信库进行了实现. 目前,作者还没有发现有相关工作介绍 2.5D PUMMA 算法. 本文算法将采用与 PDGEMM 相同的函数接口,在函数内部实现进程拓扑的转换,从而获得与 ScaLAPACK 更好的兼容性. 最后,本文通过大量的数值算例测试了 2.5D PUMMA 算法的性能,并与 PDGEMM 算法进行了对比,结果表明其可扩展性要优于 PDGEMM 算法,充分说明了新提出的 2.5D 算法的有效性.

本文第 2 节简单介绍 BLACS 通信库,并行矩阵乘算法的研究概况,包括 Cannon、Fox 和 SUMMA 算法等,并详细介绍 PUMMA 算法;第 3 节详细论述基于 BLACS 的 2.5D PUMMA 算法的流程,并介绍 2.5D 矩阵乘算法的相关工作,分析本文算法与先前算法的异同;第 4 节测试本文新提出的 2.5D PUMMA 算法的性能,在采用不同矩阵维度、分块大小、矩阵备份数及进程数的情况下,分别与 PDGEMM 的性能进行对比,给出详细的数值结果与实用性的建议,并将广义的特征值问题作为应用算例,简单介绍其基本算法流程以及本文算法在该问题上的具体应用过程,最后给出相应的测试结果;第 5 节对本文工作进行总结与展望.

2　相关背景简介

2.1　BLACS 通信库

BLACS[26] 是 ScaLAPACK 库重要的通信库,负责所有进程间的通信. 基于该库,可实现不同进程(组)间的广播、点对点通信、进程(组)间规约操作等. 如图 1 所示,ScaLAPACK 库的架构由核心函数、PBLAS 和底层的 BLACS、LAPACK、BLAS 构成. 本文中主要提到的 PDGEMM 函数便隶属于 PBLAS.

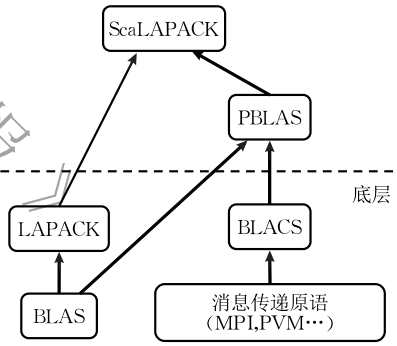


图 1　ScaLAPACK 的组织架构图

本文算法在实现时主要采用了以下的 BLACS 函数:点对点通信函数 DGESD2D 和 DGERV2D,进程间的广播函数 DGEBS2D 和进程间的规约函数 DGSUM2D. 需要说明的是 BLACS 库中的函数种类相对缺乏,只对常用的 MPI 函数进行了封装,缺少许多 MPI 函数,例如 MPI_Allgather 等操作函数. 本文之所以采用 BLACS 库来实现 2.5D 算法是为了最大限度地保持与 ScaLAPACK 的兼容,方便直接替换 PDGEMM 函数,而并非处于性能考虑. 为了追求更高性能,建议采用直接调用底层的 MPI 函数的方式.

2.2　并行稠密矩阵乘法

并行稠密矩阵乘法一直是科学计算的重点研究

课题. Cannon 算法[5]最早可追溯到 1969 年，又称为滚动-滚动-计算算法，可证明理论上是通信最优的，但是该算法只适用于方形进程网格. Fox 算法[6]可追溯到 1988 年，又称为广播-滚动-计算，是通过行处理器间施行一到多广播，列处理器间施行循环单步上移. 该两类算法都采用二维分块式数据划分方式，相较于先前的一维数据划分算法，减少了对计算机的存储需求，提高了运行效率. 1981 年，Dekel、Nassimi 和 Sahni 提出了 DNS[27]算法，超立方连接的矩阵乘法，该算法属于 3D 算法. 1994 年，Choi 等人[25]提出了 PUMMA 算法，是对原来 Fox 算法[6]的推广，并采用了二维块循环存储格式，与 ScaLAPACK 的数据存储格式一致，可避免数据格式的转换，同时进一步提出了矩阵转置的乘法，即 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}$、$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}^{\mathrm{T}}$ 等，这也是为什么算法名称中有 Universal 的原因. PUMMA 算法可适用于任意的进程拓扑，详细细节可参看下一小节.

1995 年，Van de Geijn 和 Watts[7]提出了一个简单高效的矩阵乘算法——SUMMA 算法. SUMMA 是基于向量的外积运算，可通过采用流水线(pipeline)技术，实现计算与通信的重叠，从而提高计算效率[28]. ScaLAPACK 中的 PDGEMM 函数也主要是基于 SUMMA 算法，额外利用了 *LCM* 块的概念. 与 Cannon、Fox 算法相比，虽然 SUMMA 算法可适用于任意的进程拓扑，而且需要更少的内存开销；但是作者发现 Cannon、Fox 以及 PUMMA 算法可额外地利用矩阵块的低秩性，从而来降低算法的计算和通信开销，具体可参看文献[24]. SUMMA 算法很难利用矩阵块的低秩性，这是由于 SUMMA 算法是基于向量外积运算的，而 Cannon 和 Fox 算法是基于向量内积运算的. 基于内积运算，可简单地对矩阵块进行低秩逼近，但是基于外积运算，则无法直观地利用低秩逼近.

2.3 PUMMA 算法

PUMMA 是块循环存储格式的 Fox 算法[6]，适用于任意的二维进程拓扑，而 Fox 算法是基于二维块数据结构. 假设矩阵 $\boldsymbol{A}$ 被划分为 M_b 个行块和 K_b 个列块，矩阵 $\boldsymbol{B}$ 被划分为 K_b 个行块和 N_b 个列块，则乘积矩阵 $\boldsymbol{C}$ 的计算公式如下：

$$\boldsymbol{C}(i,j)=\sum_{k=0}^{K_b-1}\boldsymbol{A}(i,k)\boldsymbol{B}(k,j) \tag{1}$$

其中 $i=0,1,\cdots,M_b-1$，$j=0,1,\cdots,N_b-1$.

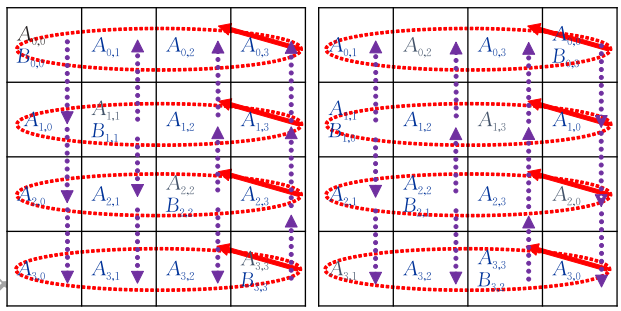
在介绍 PUMMA 算法之前，首先回顾下 Fox 算法. Fox 算法是假设 $M_b=K_b=N_b$，矩阵块 $\boldsymbol{A}(i,j)$，$\boldsymbol{B}(i,j)$ 和计算得到的 $\boldsymbol{C}(i,j)$ 都存储在 (i,j) 进程上. 每个进程在计算式(1)时，都是从对角块开始的. 正如前面所述，Fox 算法又称为广播-滚动-计算，其计算过程是循环地执行下面的步骤，直到矩阵 $\boldsymbol{A}$ 完全向左循环滚动一轮，最终求得矩阵 $\boldsymbol{C}$ 的值.

(1)将 $\boldsymbol{B}$ 的矩阵块 $\boldsymbol{B}(j,j)$ 沿列广播给其它进程，使得同列进程都获得一份拷贝，并执行本地运算 $\boldsymbol{A}(i,j)\cdot\boldsymbol{B}(j,j)$，如图 2(a)所示；

(2)将 $\boldsymbol{A}$ 的矩阵块向左循环移动一步(即滚动)，如图 2(b)所示；

(3)将 $\boldsymbol{B}$ 的矩阵块 $\boldsymbol{B}(j+1,j)$ 沿列广播给其它进程，并执行本地矩阵块的乘法运算，如图 2(b)所示；

(4)将 $\boldsymbol{A}$ 的矩阵块向左循环移动一步.



(a)步骤Ⅰ (b)步骤Ⅱ、Ⅲ

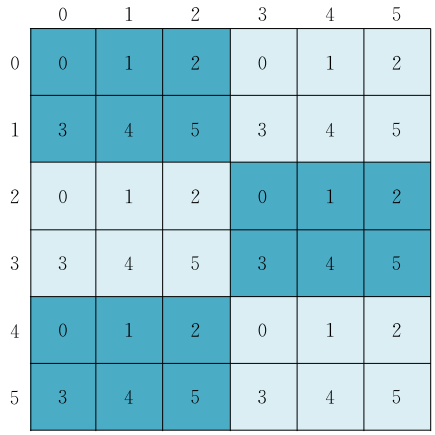
图 2 Fox 算法前三步计算过程的示意图

图 2 中展示的是矩阵 $\boldsymbol{B}$ 按列进行广播，而将矩阵 $\boldsymbol{A}$ 向左循环滚动. 另一种策略是矩阵 $\boldsymbol{A}$ 按行进行广播，而将矩阵 $\boldsymbol{B}$ 向上循环滚动. 第一种策略有利于按列存储的 Fortran 程序. 本文主要基于第一种策略来描述算法. 相较于 Cannon 算法，Fox 算法需要广播通信，如果巨型机的高速互连通信网络的广播操作比较耗时，则 Fox 算法的性能可能会较低.

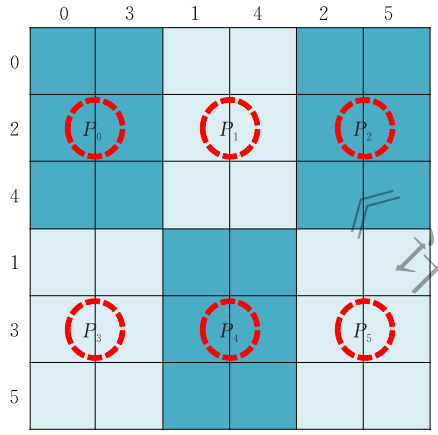
类似于 ScaLAPACK，PUMMA 算法也假设矩阵是按照块循环格式存储的，这主要是为了兼顾其他数值代数算法，因为对于矩阵乘法(计算密集型)，矩阵的分块越大则性能通常也会越高. 图 3 给出了 6×6 矩阵在 2×3 进程网格上的存储方式.

图 3(a)是从矩阵分布的角度，每个阴影和非阴影区域代表不同的划分区域. 每个小方块表示矩阵块，方块上的数字则表示矩阵块在进程网格中的存储位置，标记为相同数字的所有块都存储在同一个处理器中. 矩阵的左侧和顶部，分别表示块行和块列的全局索引. 图 3(b)是从处理器分布的角度，每个处理器拥有 3×2 个块.

最简单形式的 PUMMA 算法是类似于 Fox 算



(a) 矩阵分布示意图



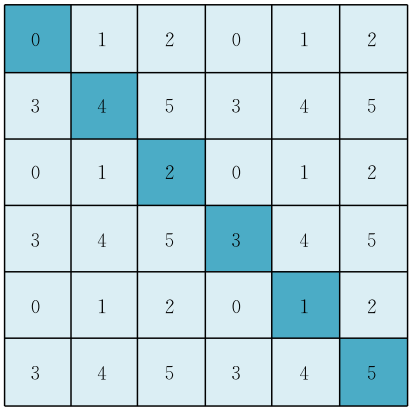
(b) 处理器分布示意图

图 3　6×6 的分块矩阵在 2×3 进程网格上的存储形式

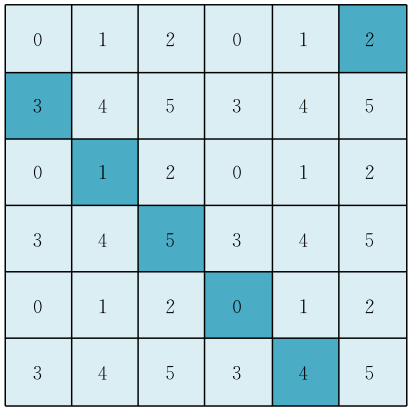
法，每次广播一个矩阵 $\boldsymbol{B}$ 的小块，共需执行 K_b 次广播，该算法在文献[25]被称为 SDB(Single Diagonal Broadcast)算法. 第 1 步是拥有主对角线上矩阵块的进程，将其广播给同列的其它进程，参看图 4(a)；第 2 步则是拥有次下对角线上矩阵块的进程，将其广播给同列的其它进程，参看图 4(b)；依次类推执行 K_b 次.

假设进程网格拓扑为 $P\times Q$，则用 LCM 表示 P 和 Q 的最小公倍数. 由于每行或每列间隔 LCM 的块在同一个进程上，文献[25]将这些块进行合并，这样外层循环只需要执行 LCM 个迭代步. 在每个迭代步，$\lceil K_b/LCM\rceil$ 个 $\boldsymbol{B}$ 矩阵块沿进程列同时广播给其它进程，从而提高每次的通信量，降低延迟开销，该算法被称为 MDB1(Multiple Diagonal Broadcast 1)，具体细节可参看文献[25]的第 3.2 节.

在每步计算的过程中，矩阵 $\boldsymbol{A}$ 的块会向左循环滚动. 如果再考虑到矩阵 $\boldsymbol{A}$ 的状态，则通信的延迟可进一步降低，算法的外循环只需要执行 Q 个迭代步. 每列间隔为 Q 的矩阵块可合并，同时进行广播.



(a) 广播主对角块

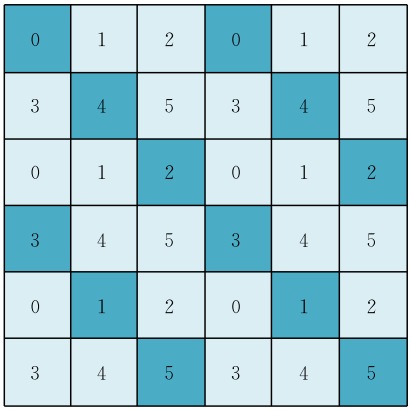


(b) 广播次下对角块

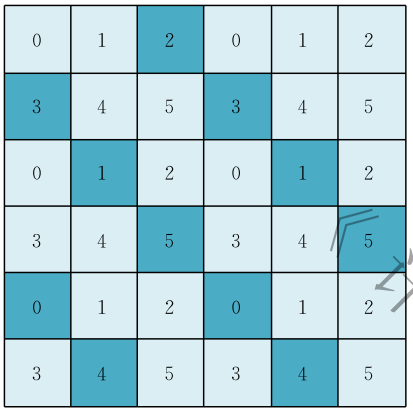
图 4　SDB 的前两次需广播的矩阵块

此时，$\lceil K_b/Q\rceil$ 个矩阵 $\boldsymbol{B}$ 块按列广播给其它进程. 该算法在文献[25]中称为 MDB2(Multiple Diagonal Broadcast 2). 如果矩阵的行块或列块数多于 LCM，即 $\lceil K_b/LCM\rceil>1$，则 MDB2 会适当地合并矩阵 $\boldsymbol{A}$ 的列块和矩阵 $\boldsymbol{B}$ 的行块，从而将多个小矩阵乘法合并为一个大的矩阵乘法，提高计算密集度和计算性能. 图 5 给出了 MDB2 算法前两次迭代需要广播的矩阵块.

PUMMA 算法的重要特点是通过合并矩阵块，使得通信和矩阵规模最大化，从而减少通信延迟，提高计算效率. 但是，PUMMA 算法的不足是由于每步都使得通信和计算最大化，导致计算和通信无法重叠. 另一个不足是为了最大化计算，需要对矩阵 $\boldsymbol{A}$ 的列块和 $\boldsymbol{B}$ 的行块进行重排，导致需要额外的内存. 本文 2.5D 算法是基于 MDB2 实现的，每个进程层执行 Q/c 个外迭代步，因此通信次数可直观地降低为原来的 $1/c$，从而显著地降低了延迟开销，其中 Q 是进程网格的列数，c 是 2.5D 算法中进程网格层数，具体参看下面第 3 节.



(a) 第一步迭代



(b) 第二步迭代

图 5 MDB2 中前两次迭代需要广播的矩阵块

3 基于 BLACS 的 2.5D 矩阵乘法

3.1 2.5D 矩阵乘法

2.5D 矩阵乘法的核心想法是利用计算节点的额外内存,用少量进程来存储整个矩阵,并将矩阵在不同进程组上进行备份,通过减少矩阵乘算法的迭代步数,来降低通信开销. 其核心计算流程介于 2D 与 3D 矩阵乘算法之间,因此被称为 2.5D 矩阵乘法. 2.5D 算法的核心计算流程如图 6 所示.

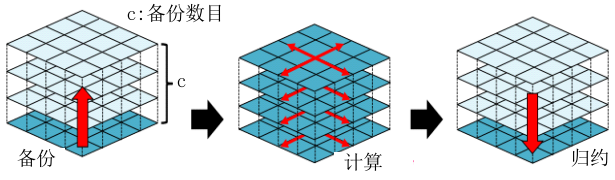


图 6 2.5D 算法核心计算流程

首先,将 2D 的进程拓扑组织为 3D 形式的进程拓扑,参看文献[15,20]中的算法,假设矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 被备份了 c 份;然后,每层的进程组执行 $1/c$ 的 2D 算法,即每层执行 $1/c$ 的 SUMMA 或者 PUMMA 算法等;最后,沿垂直方向,通过局部的规约来得到最后矩阵乘的解.

3.2 2.5D PUMMA 并行矩阵乘法

本小节主要介绍本文新提出的 2.5D PUMMA 并行矩阵乘法. 为了便于描述,只介绍没有转置的 2.5D PUMMA 算法,计算矩阵乘 $\boldsymbol{C}=\alpha\boldsymbol{A}\cdot\boldsymbol{B}+\beta\boldsymbol{C}$ (其中 α 和 β 是标量值,$\boldsymbol{A}$、$\boldsymbol{B}$ 和 $\boldsymbol{C}$ 是密集矩阵). 其基本的流程如图 7 所示,具体的执行步骤如下:

(1) 从 2D 进程网格创建 3D 逻辑进程网格;

(2) 数据分层,将矩阵 $\boldsymbol{A}$、$\boldsymbol{B}$ 从 2D 重新分配到 2.5D;

(3) 计算 2.5D 矩阵乘法 $T=\alpha\boldsymbol{AB}$,每层的进程组执行 $1/c$ 的 PUMMA 算法;

(4) 对数据进行适当排序,通过规约操作得到最终计算结果 $\boldsymbol{C}=T+\beta\boldsymbol{C}$.

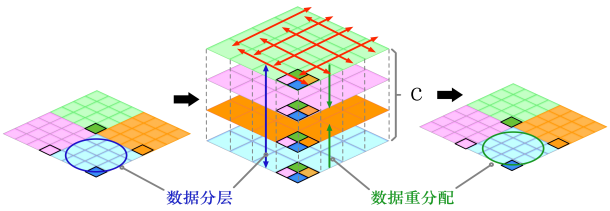


图 7 包含数据重分配的 2.5D 矩阵乘法

本文假设初始的进程拓扑为正方形网格,并且进程总数可被 4 整除,即矩阵被备份 4 份,其他情形可简单地类推得到,例如 $c=1,2,\cdots$. 本文选取 $c=4$,一方面是为了便于描述,另一方面是因为该策略通常会获得较好的性能.

从进程的角度考虑,本文算法将原来的进程网格划分为 4 个区域,每个区域含有相同的进程数,并且将处于 3D 网格的同一层. 最初是用 N_p^2 个进程共同存储矩阵 $\boldsymbol{A}$,现在用 1/4 的进程来共同存储矩阵 $\boldsymbol{A}$,每个进程要近似存储 4 倍多的数据量. 与文献[20]算法类似,本文采用将 4 个区域中相同位置的进程的数据进行组装,构成本地的数据. 文献[20]的数值结果表明,当划分的区域数目太多(即引入的层数太多)时,引入的额外开销是不容忽视的. 位于进程组的同一行的进程拥有相同的矩阵行数,位于相同列的进程拥有相同的矩阵列数,并且各个 3D 网格层上的相同位置的进程将存储相同的矩阵数据.

从矩阵的角度考虑,最初矩阵被划分为 $MB\times NB$ 大小的矩阵块,然后通过循环映射的方式分布存储在各个进程中. 在进行数据重分配时,即从 2D 网格转化到 2.5D 网格时,本文仍采用相同的矩阵块大小,即 $MB\times NB$,只是将原来划分好的矩阵块

重新分配到新的进程. 本文算法允许矩阵行的块数不能被进程的行数整除，也允许块大小 MB 或 NB 不能整除矩阵的行数或列数. 众所周知，经典的 Cannon 和 Fox 算法，只适用于方形的进程网格拓扑. 但是采用 LCM 概念后，该两类算法同样适用于任意的 2D 进程拓扑，具体可参看文献[25,29]. 本文算法是基于 PUMMA 算法设计的，也适用于任意的 2D 进程拓扑.

下面从进程拓扑重划分、数据的重分配和计算结果的规约与重分配 3 个方面，分别详细地介绍具体的基于 BLACS 的实现细节.

(1) 进程拓扑重划分：进程的重划分是通过 BLACS 函数 *BLACS_GRIDMAP* 实现的. 通过定义三维数据 $UMAP(:,:,:)$，将 $UMAP(i,:,:)$ 中的进程放在 3D 进程网格的同一层. 对于如图 8 所示的 6×6 初始进程网格，当被等分为 4 个区域时，$UMAP(1,:,:)$ 包含的数据为 0,1,2,6,7,8,12,13 和 14.

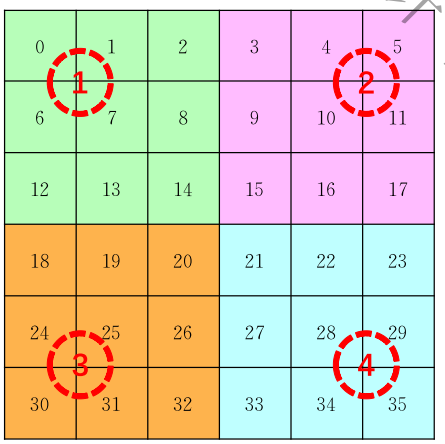


图 8　当选取 $c=4$ 时，进程组的划分方式

通过数组 $UMAP$，便可将原来的进程分为四组，具体的进程重分配的 Fortran 伪代码如下所示，其中假设 $sep\_nprows=sep\_npcols=3$ 为每层进程网格的行和列进程数，$c=4$ 是某平方数，并且 $NPCOL=6$ 为原来 2D 进程网格的列数.

```
irow=0; jcol=0;
DO IP=1, c
 DO I=1, sep_nprows
  DO J=1, sep_npcols
  UMAP (IP, I, J)=(irow * sep_nprows+I-1) *
          NPCOL+(jcol * sep_npcols+J-1)
  jtop=(I-1) * sep_npcols+J
  UTOP (IP, jtop)=UMAP (IP, I, J)
  END DO
END DO
CALL BLACS_GET (ICTXT, 0, CONTXT(IP))
CALL BLACS_GRIDMAP (CONTXT(IP), UMAP
        (IP, :, :), sep_nprows, sep_npcols)
irow=mod (IP,√c)
jcol=mod (jcol+1,√c)
END DO
```

需要说明的是进程组的排序方式要与最初进程的排序方式相同，即共同地都采用行优先或者列优先排序. 当实现进程拓扑的重划分后，需要将原来用 6×6 个进程存储的矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$，通过数据通信来存储到每个进程组的 3×3 个进程中.

(2) 数据的重分配：矩阵最初是分布存储在原来 2D 所有的进程中，当进程拓扑重划分后，需要将其它 3 个进程组中数据收集到当前进程组中. 本文根据各个进程组的划分方式，将对应进程上的数据组装到一起，来得到各个进程所需的数据. 2.5D 矩阵乘法中数据重分配的实现过程如图 9 所示.

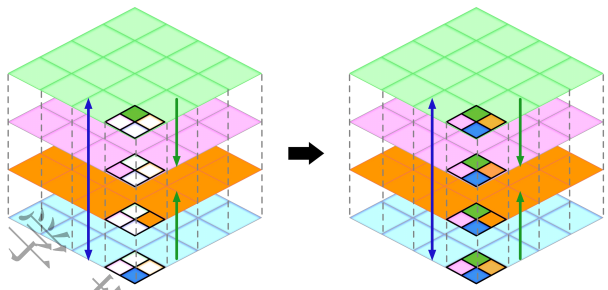


图 9　数据重分配的实现过程

本文采用文献[20]中的数据重分配算法. 只要进程组的个数是平方数，则按照进程组排列方式收集到的数据可直接拼装在一起，并不需要对收集到的数据进行重排序. 具体的数据重分配过程可分为如下 3 个步骤：

① 每个进程根据自己在当前进程网格层中位置，通过调用 NUMROC 来计算本地的局部矩阵 $\boldsymbol{LA}$ 和局部矩阵 $\boldsymbol{LB}$ 的维数，并开辟临时内存；

② 各个进程根据所属进程组的排序，将原有的本地矩阵 $\boldsymbol{A}$ 拷贝到 $\boldsymbol{LA}$ 的相应位置；

③ 各网格层中相同位置上的进程，通过 Allreduce 或 Allgather 来得到全部的数据.

在具体实现上述的数据重分配过程时，作者另外定义了一个全局的网格拓扑 TOP，其包含了所有的进程，所需的数据通信只需在相同进程列中进行即可. TOP 的同一行保存的是位于同一进程组的进程号，TOP 的同一列的进程应包含相同的数据. TOP 网格的进程排序方式如图 10 所示. 具体的数

据规约可通过调用 BLACS 中的函数 DGSUM2D 来实现.

0	1	2	6	7	8	12	13	14
3	4	5	9	10	11	15	16	17
18	19	20	24	25	26	30	31	32
21	22	23	27	28	29	33	34	35

图 10 顶层网格 TOP 的排序方式

如上所述,本文的数据重分配算法与文献[20]中的策略是相同的,只是在具体的实现细节稍有不同. 为了与 ScaLAPACK 兼容,本文采用 BLACS 库来实现进程间的通信,优点是可直接使用 BLACS 原有的通信器,编程比较简单. 本文实现的缺点是由于 BLACS 中通信函数接口不是很多,缺少 MPI_Allgather 功能的函数接口,只能通过 DGSUM2D(规约求和)来模拟实现 Allgather 功能. 因此,导致通信量有所增加,但是通信次数(延迟开销)没有增加. 利用 BLACS 函数 DGEBS2D(广播操作)也可模拟 Allgather 功能. 文献[20]是先利用 MPI_Allgather 函数收集到所有的数据块,再在本地拷贝到适当的位置,编程更加底层和复杂,但性能可能会更好些.

(3)计算结果的规约与重分配:各个进程组在得到所需的数据后,即可同时执行 PUMMA 矩阵乘算法的不同计算步骤. 需要强调的是每层只需执行原来算法 $1/c$ 的迭代步. 在计算得到结果后,通过将位于 TOP 相同列中的进程的数据相加求和,来得到最终的矩阵乘的计算结果. 本文仍是通过调用 DGSUM2D 函数,来使得每个进程都得到一份最终的计算结果,即完成数据的规约. 最后,各个进程根据自己所属进程组的位置,将自己所需的数据从局部矩阵 $\boldsymbol{LA}$ 中拷贝到全局矩阵 $\boldsymbol{A}$ 中(此处是局部操作,并不需要进程间的数据通信).

3.3 进一步讨论

本文提出的基于 BLACS 的进程与数据的重分配方式,与 ScaLAPACK 具有紧密联系,具有相同的函数接口,设计得到的 2.5D 矩阵乘算法可直接用于替换 PBLAS 的函数 PDGEMM. 与 SUMMA 算法相比,PUMMA 算法需要大量的额外内存,这是该类算法不足;但是对于某类矩阵,PUMMA 算法可充分地利用矩阵的非对角块低秩性,从而降低算法的通信和计算开销,可参看文献[24]. 本文的下一步工作是针对特殊矩阵乘法设计 2.5D PUMMA 算法,例如 Cauchy、Toeplitz、离散 Fourier 变换 DFT(Discrete Fourier Transform)矩阵等,进一步地降低通信开销和计算开销. 这些特殊矩阵乘法在科学计算中具有重要应用,例如,求解三对角矩阵特征值分解的分而治之 DC(Divide-and-Conquer)算法需计算大量的矩阵乘法,其中一个矩阵是 Cauchy 型矩阵[24];快速傅里叶变换 FFT(Fast Fourier Transform)的分布式并行可扩展性不是很好,可尝试利用 DFT 矩阵的结构进行加速等.

数据重新分配的效率会严重影响算法的性能. 在提出本文算法之前,作者曾尝试过调用 BLACS 函数 PDGEMR2D 来实现 2D 进程到 2.5D 进程的数据重分配,发现由于需要同时转换 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 两个矩阵,该函数的开销非常大,导致 2.5D 算法的最终性能会劣于 PDGEMM 函数. 本文算法的数据重分配算法要明显地优于调用 PDGEMR2D 函数的策略. 由于 Cannon、Fox 算法采用的二维块存储格式,不能采用本文或文献[20]中的数据重分配方法,只能采用类似函数 PDGEMR2D 形式的重分配方法,因此性能会受到影响.

本文的数据重分配策略,同样适用于 c 取其他值的情形,只需合理地对原始的进程进行分组即可. 例如,当 $c=2$ 时,对于图 8 中的进程,可进行如图 11 所示的进程分组. 需要说明的是当 c 不是平方数时,通过规约操作得到重分配后的数据后,需要对矩阵的行块和列块进行适当的重排,以得到正确的块循环存储结构. 例如,图 11 中的进程 0 和进程 3 的数据合并在一起后,需要将进程 0 的列块(每 NB 个列)和进程 3 的列块交叉排列. 在下面第 4 节,本文对 $c=2$ 的情形,通过数值算例进行了验证. 当 c 是平方数时,只需要按照图 8 所示的方式,将不同进程

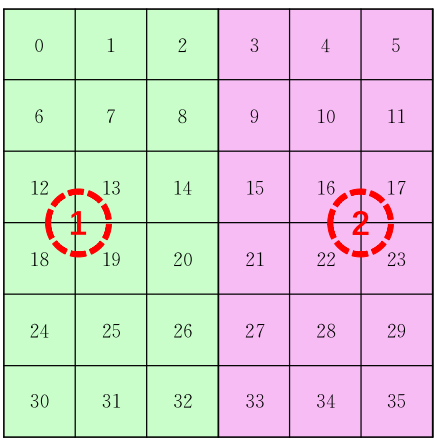


图 11 当选取 $c=2$ 时,进程组的划分方式

的数据组合在一起即可，不需要重新排序. 因此，通常建议选取 c 为某平方数，例如 $c=4,9,16$ 等，并且 c 不宜太大，否则会引入更多的额外内存开销和创建新的进程拓扑的开销.

3.4　相关 2.5D 矩阵乘法

2.5D 矩阵乘法是 Solomonik 和 Demmel 首先提出的[15]，结合了 2D 算法和 3D 算法[8]的概念，提出通过使用额外内存来模拟 3D 算法，并实现了 2.5D Cannon 算法. 该算法是针对方形矩阵和进程网格的，不适用于任意的进程网格. 后来，Solomonik 等人进一步实现了 2.5D SUMMA 算法，包含在框架 CTF(Cyclops Tensor Frameworks)中[30]，可适用于任意的进程拓扑. 上述两种算法的数值结果都没有包含矩阵从 2D 到 2.5D 的转换时间. Mukunoki 和 Imamura 提出一种高效的从 2D 到 2.5D 的数据重分配算法[20,25]，在日本京巨型机上实现了 2.5D SUMMA 算法. 数值实验结果表明，即使包含数据重分配时间，2.5D SUMMA 算法仍优于 PBLAS 中的 PDGEMM 算法. 需要说明的是该数据重分配算法[20]并不适用于 2.5D Cannon 算法. Cannon 等算法的通信最优都是基于正方形矩阵得到的，Demmel 等人要在文献[31]得到首个长方形矩阵乘法的通信最优算法，CARMA (Communication-Avoiding Recursive Matrix multiplication Algorithm)，在渐进意义下是通信最优的. CARMA 是递归算法，理论上只适用于进程数为 2 的幂次的情形.

本文算法是将数据重分配算法[20]与 PUMMA 算法相结合，提出一种 2.5D PUMMA 算法，可适用于任意的进程拓扑，并且在考虑数据重分配时间情况下，仍可优于 PDGEMM 算法. 相较于前期算法，本文算法的突出优势是可进一步利用矩阵块的低秩性，从而针对某类特殊矩阵可降低计算复杂度和通信量，具体可参看文献[24]. 本文算法的另一个优势是基于 BLACS 实现，最大限度地保持了与 ScaLAPACK 的兼容性，可在应用程序中直接替换 PDGEMM 函数.

4　算法测试与分析

本节首先对本文提出的 2.5D PUMMA 矩阵乘算法与 ScaLAPACK 中的 PDGEMM 函数进行对比，并在一台巨型机系统上进行了大规模的测试. 作者选取了不同的矩阵维数、分块大小、矩阵备份数和进程数进行评估. 由于稠密矩阵乘法的性能与矩阵元素的关系不大，本文采用随机生成的矩阵进行数值实验，矩阵的元素都服从高斯分布. 在第 4.3 节中，将本文提出的 2.5D PUMMA 矩阵乘算法应用于广义特征值问题，并与 ScaLAPACK 中的相应函数进行了数值对比.

本节数值实验是在一台具有 300 个计算节点的巨型机系统上进行的，该测试平台采用胖树结构的高速互联网络，每个计算节点有两块 CPU 处理器，处理器型号为 Intel E5-2660 v3 CPU，共 20 个计算核，128GB 内存. 具体的测试平台的信息与编译环境，详见表 1.

表 1　测试平台计算节点配置表

平台信息	配置列表
处理器	E5-2660 v3 64 位处理器(2.6GHz,10 核)
内存	128GB(DDR3)
操作系统	Linux 内核版本 3.10.0
编译器	Intel 的 fortran 编译器 ifort，版本号为 Compilers_and_libraries_2019.4.243
编译选项	-O3-mavx-qopenmp

在进行测试时，在原则上每个节点运行 20 个 MPI 进程，每个计算核运行 1 个 MPI 进程，选取线程数为 1.

4.1　分块大小对性能的影响

在本算例中，作者采用矩阵维数为 16 384 的随机矩阵进行测试，分别选取不同的进程数 NP 和分块大小 NB，其中进程数分别为 $NP=64,256,1024$ 和 4096，分块大小分别为 $NB=32,64,128$ 和 256. 具体的计算结果列在图 12 和图 13 中.

从图 12、图 13 显示的具体的计算结果中，可以得到如下结论：

(1) PDGEMM 的计算性能与 NB 的选取有较大关系，当选取 $NB=128$ 或 256 时，性能要明显地优于选取 $NB=32$ 和 64 时. 2.5D PUMMA 算法则与 NB 的选取关系不是很紧密.

(2) 2.5D PUMMA 算法具有更好的可扩展性，这也与算法的设计初衷是相符的. 当进程数为 1024 和 4096 时，2.5D PUMMA 算法的性能均要优于 PDGEMM 算法，尤其是当 $NP=4096$ 时，加速比可达到 2.20～2.93. 因此，当进程数较多时，建议选取 2.5D PUMMA 矩阵乘算法.

需要说明的是在实际应用中，NB 的选取需要根据具体的应用来综合考虑确定. 选取较大的 NB，往往可提高矩阵乘法的性能，但是可能降低算法其他部分的性能，尤其是当并行矩阵乘在整个应用中所占的时间比并不是很大时，例如对于稠密矩阵的

特征值分解算法，ScaLAPACK 中的算法通常选取 $NB=64$，往往会获得较好的性能.

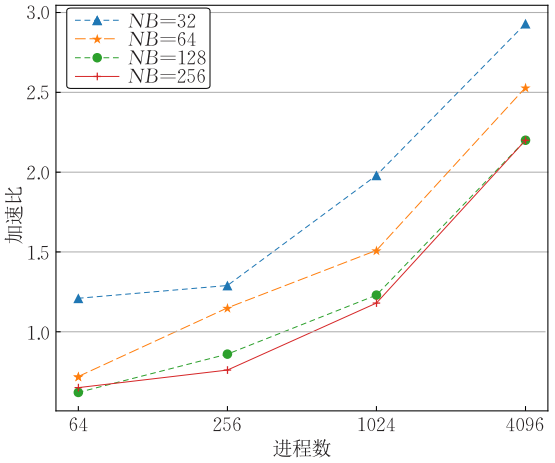


图 12 2.5D PUMMA 矩阵乘相对于 PDGEMM 的加速比

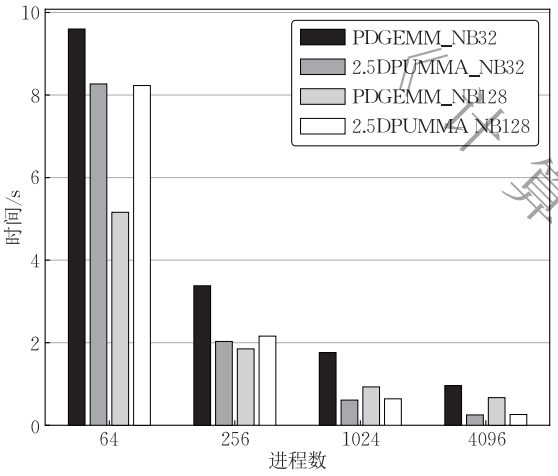


图 13 PDGEMM与2.5D PUMMA算法的墙上时间对比结果

4.2 矩阵规模等对性能的影响

下面通过选取不同矩阵规模 N、矩阵备份数 c 和进程数 NP，对 2.5D PUMMA 算法的性能进行更加详细的测试与分析. 在本算例中，统一选取 $NB=128$，同样采用随机矩阵进行测试，矩阵规模分别选用 8192 和 32768，并选用不同的 c 进行测试，其中 $c=1,2,4$.

当 $c=1$ 时，2.5D PUMMA 算法退化为标准的 PUMMA 算法. 相对 PDGEMM，PUMMA 算法需要额外的内存，并且无法实现计算与通信的重叠. 但是，PUMMA 算法每次计算和通信都尽力使得矩阵规模最大，提高了计算密度，数值结果表明标准的 PUMMA 算法的性能与 PDGEMM 的性能相当，甚至略好于 PDGEMM 算法.

当 $c=2$ 时，本文分别选用规模 $N=8192,16384$ 和 32768 的矩阵，选用进程数 $NP=64,256,1024$ 和 4096，并将进程拓扑按列划分为两组，如图 11 所示. 所有的测试结果都列在图 14，图中数据是本文算法相对 PDGEMM 的加速比. 与文献[20]采用的测试方法相同，本文在统计 2.5D 算法执行时间时，没有考虑创建进程拓扑的时间，因为 PDGEMM 函数的计算时间也不包含创建进程拓扑的时间. 随着 c 的增大，创建进程拓扑的时间占比也会增加.

需要说明的是当 $c=2$ 时，通过数据的重分配(即 DGSUM2D 函数)得到数据后，每个进程需在本地将数据按照 2D 块循环形式进行重新组织. 由于是本地操作，这部分的开销很小. 当 $c=2$ 时，由于选取 $NP=64,256,1024$ 和 4096 是某数的平方，因此每层的进程网格并非为正方形的，即每行的进程数不等于每列的进程数. 例如，当 $c=2,NP=64$ 时，本文的数值实验中 2.5D 进程拓扑的每层采用的是 8×4的进程网格，这说明 PUMMA 算法适用于任意的进程拓扑. 当 $c=2$ 时，进程数 NP 同样可选取 128,512,2048 等.

在选取 $c=4$ 的情况下，本文对规模为 8192 的矩阵进行了测试，本文算法相对于 PDGEMM 的加速比也列在图 14 中. 从图中结果可以看出，当进程数较多时，2.5D PUMMA 算法要明显地快于 PDGEMM. 但是，当进程数较少时，2.5D PUMMA 算法相对 PDGEMM 并没有优势，这是由于进程数较少时，通信的开销影响不大，计算占主导因素(没必要采用 2.5D 算法)；另一方面是由于进程数较少时，从 2D 到 2.5D 的来回数据分配占据了大量的时间，参看图 15 中的结果.

图 14 的测试结果还表明，相对矩阵规模较大的矩阵，规模较小的矩阵的提速更明显，这是因为规模较小时，通信开销对性能的影响会更大. 图 15 列出了从 2D 到 2.5D 和 2.5D 到 2D 的数据分配时间占总体时间的百分比. 当进程数较少时，数据重分配时间占据了 50%的时间；但是，当进程数较多时，进程层间需要传输的数据量会明显降低，因此，时间开销也会降低.

总之，从图 14 和图 15 中的计算结果，可得到以下结论：

(1) 当进程数较少时，从 2D 到 2.5D 的数据重分配占据的时间较多，因此不建议采用 2.5D 算法；

(2) 当进程数较少时，计算占主导因素，通信的影响不大，采用通信优化的 2.5D 算法也没有加速效果；

(3) 当进程数较多时，从 2D 到 2.5D 的数据重

分配的时间占比很小，通信开销影响较大，建议采用 2.5D 算法，可获得很好的加速比.

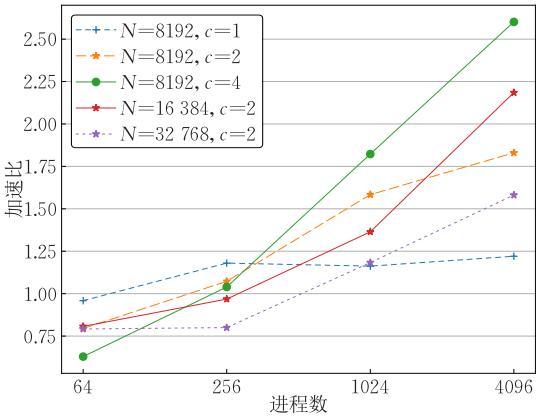


图 14　2.5D PUMMA 矩阵乘相对于 PDGEMM 的加速比

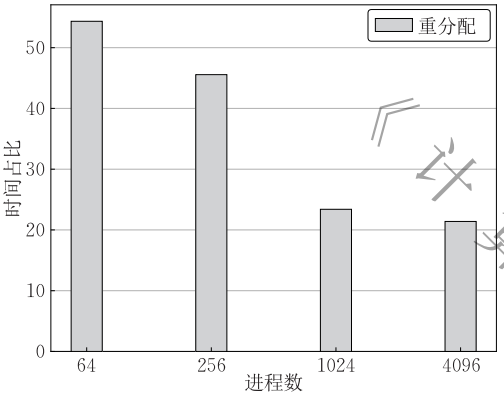


图 15　2D 到 2.5D 来回数据重分配的时间占总体时间的百分比

4.3　2.5D PUMMA 矩阵乘数值应用

并行矩阵乘在科学计算中具有重要作用，本小节将特征值问题作为应用算例，说明 2.5D PUMMA 算法可用于加速 ScaLAPACK 库中的计算对称特征值问题函数的性能. 首先简要介绍特征值问题的计算过程.

给定对称或者 Hermitian 矩阵[23] $\boldsymbol{A}\in\boldsymbol{F}^{n\times n}$，特征值问题是计算对角矩阵 $\boldsymbol{\Lambda}$ 和列正交矩阵 $\boldsymbol{Q}$，使其满足

$$\boldsymbol{AQ}=\boldsymbol{\Lambda Q} \tag{2}$$

其中 $\boldsymbol{\Lambda}$ 的对角元素称为特征值，$\boldsymbol{Q}$ 的列向量称为特征向量. 求解特征值分解的算法可分为“直接法”和“迭代法”. 直接法的基本步骤是通过一系列正交变换将(稠密)矩阵化为三对角形式，然后求解三对角矩阵的特征值问题，最后通过对特征向量矩阵进行后向变换(back-transform)来得到原来特征值问题的解. 算法的具体流程如下所示(可参考文献[1,24]).

(1) 将 $\boldsymbol{A}$ 化为三对角形式；

$$T=\boldsymbol{UAU}^{\mathrm{T}} \tag{3}$$

其中 $\boldsymbol{U}=\boldsymbol{U}_n\cdots\boldsymbol{U}_2\boldsymbol{U}_1$ 是一系列 Householder 矩阵乘积.

(2) 求解三对角矩阵的特征值分解；

$$\boldsymbol{T\hat{Q}}=\boldsymbol{\Lambda\hat{Q}} \tag{4}$$

(3) 对感兴趣的 k 个特征向量进行反变换，得到标准特征值问题的特征分解；

$$\tilde{\boldsymbol{Q}}=\boldsymbol{U\hat{Q}} \tag{5}$$

在通过正交变换将矩阵化为三对角矩阵后，需要进一步地用 DC、QR 或 MRRR(Multiple Relatively Robust Representations)等算法进行求解. 本文将新提出的 2.5D PUMMA 乘法用于加速三对角 DC 算法，即上述步骤 II. 在三对角 DC 算法中，最耗时的部分是计算特征向量，此步骤需要计算两个稠密矩阵的乘积，因此可用本文提出的 2.5D 矩阵乘法来代替原来的 PDGEMM 算法.

在文献[21]中，作者分别用 HSS 矩阵来加速三对角矩阵的 DC 算法，并采用 Clement 型、Hermite 型和 Toeplitz 型矩阵进行数值实验，来说明算法有效性. 与前面的工作类似，本文仍采用 Clement 和 Toeplitz 型的三对角矩阵来说明 2.5D PUMMA 矩阵乘对于加速三对角 DC 算法的计算效果.

三对角 Clement 型矩阵的定义如下：

$$\boldsymbol{T}=\mathrm{tridiag}\begin{pmatrix}1 & 1 & & 1 & 1\\ 2 & 2 & \cdots & 2 & 2\\ 1 & 1 & & 1 & 1\end{pmatrix} \tag{6}$$

三对角 Toeplitz 型矩阵的定义如下：

$$\boldsymbol{T}=\mathrm{tridiag}\begin{pmatrix}\sqrt{n} & \sqrt{2(n-1)} & \sqrt{(n-1)2} & \sqrt{n}\\ 0 & 0 & \cdots & 0 & 0\\ \sqrt{n} & \sqrt{2(n-1)} & \sqrt{(n-1)2} & \sqrt{n}\end{pmatrix} \tag{7}$$

本算例选取矩阵的维数为 $N=15\,000$，当矩阵规模大于 $SMB=10\,000$ 时，我们将原来 ScaLAPACK 函数中的 PDGEMM 替换为 2.5D PUMMA 矩阵乘法. 这是因为当矩阵维数较小时，并行矩阵乘法的耗时很小，2.5D PUMMA 的提速效果不明显，性能与 PDGEMM 算法没很大区别. 通过选取不同的进程数 $NP=256$，1024 和 4096，来进行数值实验，最终的测试结果如图 16 所示. 数值结果表明用 2.5D PUMMA 算法加速后的 ScaLAPACK 函数 PDSTEDC，可获得 1.20x 倍的加速比.

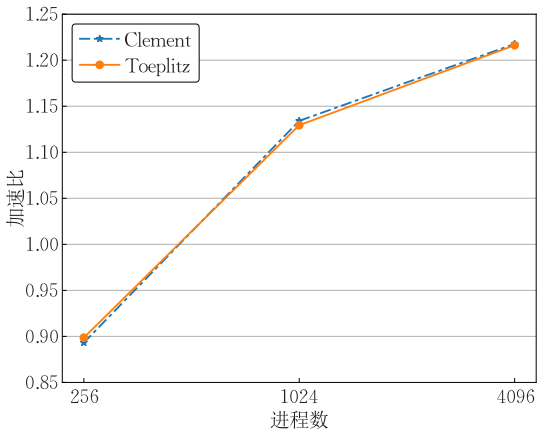


图 16 用 2.5D PUMMA 矩阵乘法改进后的 DC 算法相对 PDSTEDC 的加速比($N=15000$)

5 总结与展望

本文提出一种基于 BLACS 的 2.5D 稠密矩阵乘算法,与 PDGEMM 具有相同的函数接口,可直接相互替换,但是相比于 PDGEMM 具有更好的可扩展性.在进程较多时,例如 4096 进程时,可比 PDGEMM 快 2.20～2.93 倍.在进程数较少时,例如 64 或 256 进程时,新提出的 2.5D 矩阵乘法的性能可能会比 PDGEMM 差,这是因为数据重分配等带来的额外开销,并且当进程数较少时,通信对性能的影响不大.在一台具有 300 个计算节点的巨型机系统上,用大量的数值算例测试了本文新提出算法的性能,分别选取了不同矩阵规模 N、分块大小 NB、矩阵备份数 c 和进程数 NP.测试结果表明当进程数较多时,2.5D PUMMA 算法有明显的提速效果.为了降低巨型机的系统扰动的影响,本文的测试结果均是对函数进行多次调用,然后取平均值.进一步,本文将新提出的 2.5D PUMMA 矩阵乘用于加速三对角矩阵的特征值分解,在采用 4096 进程时,可获得 1.2 倍以上的加速比.

相较于其它 2.5D 矩阵乘算法,本文 2.5D PUMMA 的突出优势是可利用矩阵块的低秩性,可参考文献[24].接下来的工作是研究针对特殊矩阵的 2.5D PUMMA 算法,例如 Cauchy、Toeplitz 和 Vandermonde 等矩阵,该几类矩阵在科学计算和工业中都有广泛应用.

致 谢 本文作者非常感谢审稿人提出的宝贵建议,极大地改进了本文的内容的呈现形式,同时感谢丁胜杰帮忙绘制了部分的示意图!

参 考 文 献

[1] Anderson E, Bai Z, Dongarra J, et al. Lapack users' guide. 3rd Edition. Philadelphia, USA: Society for Industrial and Applied Mathematics, 1999

[2] Blackford L S, Choi J, Cleary A J, et al. ScaLAPACK: A linear algebra library for message-passing computers//Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing. Minneapolis, USA, 1997

[3] Schatz M, Geijn R, Poulson J, et al. Parallel matrix multiplication: A systematic journey. SIAM Journal on Scientific Computing, 2016, 38(6): 748-781

[4] Irony D, Toledo S, Tiskin A. Communication lower bounds for distributed-memory matrix multiplication. Journal of Parallel and Distributed Computing, 2004, 64(9): 1017-1026

[5] Cannon L E. A Cellular Computer to Implement the Kalman Filter Algorithm [Ph. D. dissertation]. Montana State University, Bozeman, USA, 1969

[6] Fox G, Johnson M, Lyzenga G, et al. Matrix algorithms on a hypercube I: Matrix multiplication. Parallel Computing, 1987, 4(1): 17-31

[7] Van de Geijn Robert A, Watts J. SUMMA: Scalable universal matrix multiplication algorithm. Concurrency and Computation: Practice and Experience, 1997, 9(4): 255-274

[8] Agarwal R C, Balle S M, Gustavson F G, et al. A three-dimensional approach to parallel matrix multiplication. IBM Journal of Research and Development, 1995, 39: 575-582

[9] Bell C, Bonachea D, Nishtala R, et al. Optimizing bandwidth limited problems using one-sided communication and overlap//Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium. Rhodes Island, Greece, 2006: 84-102

[10] Nishtala R, Hargrove P H, Bonachea D O, et al. Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap//Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing. Rome, Italy, 2009

[11] Berntsen J. Communication efficient matrix multiplication on hypercubes. Parallel Compute, 1989, 12: 335-342

[12] Gupta A, Kumar V. Scalability of matrix multiplication algorithms on parallel computers//Proceedings of the 1993 International Conference on Parallel Processing. Syracuse, New York, USA, 1993, (III): 115-123

[13] Johnsson S L. Minimizing the communication time for matrix multiplication on multiprocessors. Parallel Computing, 1997, 19(11): 1235-1257

[14] Choi J. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. Concurrency and Computation: Practice and Experience, 1998, 10(8): 655-670

[15] Solomonik E, Demmel J. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms//Proceedings of the 17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011). Bordeaux, France, 2011: 90-109

[16] Georganas E, Gonzalez-Dominguez J, Solomonik E, et al. Communication avoiding and overlapping for numerical linear algebra//Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12). Salt Lake City, USA, 2012, 100: 1-11

[17] Solomonik E, Bhatele A, Demmel J. Improving communication performance in dense linear algebra via topology aware collectives//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11). Seattle, USA, 2011, (77): 1-11

[18] Lazzaro A, VandeVondele J, Hutter J, et al. Increasing the efficiency of sparse matrix-matrix multiplication with a 2.5D algorithm and one-sided MPI//Proceedings of the Platform for Advanced Scientific Computing Conference. Lugano, Switzerland, 2017: 1-9

[19] Mukunoki D, Imamura T. Performance analysis of 2D-compatible 2.5D-PDGEMM on knights landingcluster//Proceedings of the 18th International Conference on Computational Science. Wuxi, China, 2018: 853-858

[20] Mukunoki D, Imamura T. Implementation and performance analysis of 2.5D-PDGEMM on the K computer//Proceedings of the International Conference on Parallel Processing and Applied Mathematics. Lublin, Poland, 2017: 348-358

[21] Li S, Liu J, Du Y. A high performance implementation of Zolo-SVD algorithm on distributed memory systems. Parallel Computing, 2019, 86: 57-65

[22] Sukkari D E, Ltaief H, Keyes D. High performance polar decomposition on distributed memory systems//Proceedings of the 22nd International European Conference on Parallel and Distributed Computing (Euro-Par 2016). Grenoble, France, 2016: 605-616

[23] Akemann G. Higher genus correlators for the Hermitian matrix model with multiple cuts. Nuclear Physics, 1996, 482(1): 403-430

[24] Liao X, Li S, Lu Y, Roman J E. A parallel structured divide-and-conquer algorithm for symmetric tridiagonal eigenvalue problems. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(2): 367-378

[25] Choi J, Dongarra J J, Walker D W. PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. Concurrency: Practice and Experience, 1994, 6(7): 543-570

[26] Deshpande V, Sawyer W. An MPI implementation of the BLACS//Proceedings of the 3rd International Conference on High-Performance Computing Data and Analytics. Trivandrum, India, 1996: 463-468

[27] Dekel E, Nassimi D, Sahni S. Parallel matrix and graph algorithms. SIAM Journal on Computing, 1981, 10(4): 657-675

[28] Buluc A, Gilbert J. R. Challenges and advances in parallel sparse matrix-matrix multiplication//Proceedings of the 37th International Conference on Parallel Processing. Portland, USA, 2008: 503-510

[29] Wu Jian-Ping, Chi Xue-Bin. Matrix multiplication on distributed system. Mathematica Numerica Sinica, 1999, 21(1): 99-108 (in Chinese)
(吴建平，迟学斌. 分布式系统上并行矩阵乘法. 计算数学，1999，21(1)：99-108)

[30] Solomonik E, Matthews D A, Hammond J, Demmel J. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions//Proceedings of the 27th International Symposium on Parallel and Distributed Processing. Boston, USA, 2013: 813-824

[31] Demmel J, Eliahu D, Fox A, et al. Communication-optimal parallel recursive rectangular matrix multiplication//Proceedings of the 27th International Symposium on Parallel and Distributed Processing. Boston, USA, 2013: 261-272

LIAO Xia, Ph. D. candidate. Her research interests include high performance computing, big data analysis and processing and parallel computing.

LI Sheng-Guo, Ph. D., assistant researcher. His research interests include high performance computing, CFD, numerical linear algebra and eigenvalue problems.

LU Yu-Tong, Ph. D., professor. Her research interests include high performance computing, fault tolerant computing and system software.

YANG Can-Qun, Ph. D., professor. His research interests include high performance computing, heterogeneous computing and large-scale parallel software optimization.

Background

Parallel matrix multiplication is an important operation in numerical linear algebra, big data analysis, and scientific computing. How to improve the scalability of the PDGEMM algorithm has an important role in improving the utilization of future E-class computers.

Based on the BLACS and PUMMA algorithms, a new 2.5D matrix multiplication algorithm is proposed in this paper, and its scalability is much better than PDGEMM. When the number of processes is large, a speedup of 2.93 can be obtained. It has the same function interface as PDGEMM and can directly replace PDGEMM in the ScaLAPACK function, thereby improving the performance of the corresponding function.

This research is partly supported by Key R&D projects of the Ministry of Science and Technology (2018YFB0204301), the National Key R&D Program of China (2018YFB0204303), the National Natural Science Foundation of China (No. 61872392, No. U1811461, No. NNW2019ZT6-B20, No. NNW2019ZT6-B21 and No. NNW2019ZT5-A10), Guangdong Natural Science Foundation (2018B030312002), the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant (No. 2016ZT06D211), the NSF of Hunan (No. 2019JJ40339) and the NSF of NUDT (No. ZK18-03-01).