计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.191000009

广义稠密对称特征问题标准化算法在GPU集群上的有效实现

刘世芳[1,2]　赵永华[1]　于天禹[1,2]　黄荣锋[1,2]
1 中国科学院计算机网络信息中心　北京 100080
2 中国科学院大学　北京 100080
(liushifang@cnic.cn)

摘　要　广义稠密对称特征问题的求解是许多应用科学和工程的主要任务,并且是计算电磁学、电子结构、有限元模型和量子化学等计算中的重要部分。将广义对称特征问题转化为标准对称特征问题是求解广义稠密对称特征问题的关键计算步骤。针对GPU集群,文中给出了广义稠密对称特征问题标准化块算法在GPU集群上基于MPI+CUDA的实现。为了适应GPU集群的架构,广义对称特征问题标准化算法将正定矩阵的Cholesky分解与传统的广义特征问题标准化块算法相结合,降低了标准化算法中不必要的通信开销,并且增强了算法的并行性。在基于MPI+CUDA的标准化算法中,GPU与CPU之间的数据传输操作被用来掩盖GPU内的数据拷贝操作,这消除了拷贝所花费的时间,进而提高了程序的性能。同时,文中还给出了矩阵在二维通信网格中行通信域和列通信域之间完全并行的点对点的转置算法和基于MPI+CUDA的具有多个右端项的三角矩阵方程 $\boldsymbol{BX}=\boldsymbol{A}$ 求解的并行块算法。在中科院计算机网络信息中心的超级计算机系统"元"上,每个计算节点配置2块Nvidia Tesla K20 GPGPU卡及2颗Intel E5-2680 V2处理器,使用多达32个GPU对不同规模矩阵的基于MPI+CUDA的广义对称特征问题标准化算法进行测试,取得了较好的加速效果与性能,并且具有良好的可扩展性。当使用32个GPU对50 000×50 000阶的矩阵进行测试时,峰值性能达到了约9.21 Tflops。

关键词:广义对称特征问题标准化算法;GPU集群;Cholesky分解;转置算法;三角矩阵方程
中图法分类号　TP301

Efficient Implementation of Generalized Dense Symmetric Eigenproblem Standardization Algorithm on GPU Cluster

LIU Shi-fang[1,2], ZHAO Yong-hua[1], YU Tian-yu[1,2] and HUANG Rong-feng[1,2]
1 Computer Network Information Center, Chinese Academy of Sciences, Beijing 100080, China
2 University of Chinese Academy of Sciences, Beijing 100080, China

Abstract　The solution of the generalized dense symmetric eigenproblem is the main task of many applied sciences and engineering, and is an important part in the calculation of electromagnetics, electronic structures, finite element models and quantum chemistry. Transforming generalized symmetric eigenproblem into a standard symmetric eigenproblem is an important computational step for solving the generalized dense symmetric eigenproblem. For the GPU cluster, the generalized blocked algorithm for generalized dense symmetric eigenproblem was presented based on MPI+CUDA on GPU cluster. In order to adapt to the architecture of the GPU cluster, the generalized symmetric eigenproblem standardization algorithm presented in this paper adopts the method of combining the Cholesky decomposition of the positive definite matrix with the traditional standardized blocked algorithm, which reduces the unnecessary communication overhead in the standardized algorithm and increases the parallelism of the algorithm. Moreover, In the MPI+CUDA based generalized symmetric eigenproblem standardization algorithm, the data transfer operation between the GPU and the CPU is utilized to mask the data copy operation in the GPU, which eliminates the time spent on copying, thereby improving the performance of the program. At the same time, a fully parallel point-to-point transposition algorithm between the row communication domain and the column communication domain in the two-dimensional communication grid was presented. In addition, a parallel blocked algorithm based on MPI+CUDA for the solution of the triangular matrix equation $\boldsymbol{BX}=\boldsymbol{A}$ with multiple right-end terms was also given. On the supercomputer system "Era" of the Computer Network Information Center of Chinese Academy of Sciences, each compute node is configured with 2 Nvidia Tesla K20 GPGPU cards and 2 Intel E5-

到稿日期:2019-10-08　返修日期:2019-12-06　本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
基金项目:国家重点研发计划项目(2017YFB02022);中国科学院战略性先导科技专项(C类)(XDC01040000)
This work was supported by the National Key Research and Development Program of China(2017YFB02022) and Strategic Priority Research Program of Chinese Academy of Sciences (XDC01040000).
通信作者:赵永华(yhzhao@sccas.cn)

2680 V2 processors. This paper tested different scale matrices using up to 32 GPUs. The implementation performance of the generalized symmetric eigenproblem standardization algorithm based on MPI+CUDA has achieved better acceleration and performance, and have good scalability. When tested with 50 000×50 000-order matrix using 32 GPUs, the peak performance reach approximately 9.21 Tflops.

Keywords Generalized symmetric eigenproblem standardization blocked algorithm, GPU cluster, Cholesky decomposition, Transpose algorithm, Triangular matrix equations

1 引言

本文描述的广义稠密对称特征问题可以被描述为以下形式：

$$\boldsymbol{Ax}=\lambda\boldsymbol{Bx} \tag{1}$$

其中，$\boldsymbol{A}\in R^{n\times n}$ 是稠密对称矩阵或 Hermitian 矩阵，$\boldsymbol{B}\in R^{n\times n}$ 是正定矩阵，$\lambda\in R$ 是对应于特征向量 $\boldsymbol{x}\in R^n$ 的特征值。当 $\boldsymbol{B}=\boldsymbol{I}$ 时，式(1)为标准特征问题。当 $\boldsymbol{B}\neq\boldsymbol{I}$ 时，可以通过将式(1)转化为标准稠密对称特征问题来进行求解。通常在材料科学和化学中电子结构计算等领域，需要解决以上广义稠密对称特征问题[1-3]。近几年，许多学者针对广义特征问题进行了研究[4-6]。求解广义特征问题一共需要 5 步，涉及到 5 个转换[7]，标准化是求解问题的第一步。求解大规模广义特征问题时需要大量的存储空间和时间。随着近几年超级计算机的飞速发展，目前已有很多学者针对大规模并行广义特征问题的求解进行了研究，例如 ScaLAPACK[7]，PLAPACK[8-9]，eigen_s 和 eigen_sx[10]，Elemental[11]，ELeMRRR[12] 等。同时，随着近年来异构计算的发展，基于众核的 PLASMA 和基于 GPU 的 MAGMA 也被开发出来[13-16]。标准化算法中主要涉及正定矩阵 $\boldsymbol{B}$ 的 Cholesky 分解即 $\boldsymbol{B}=\boldsymbol{LL}^{\mathrm{T}}$ 和矩阵 $\boldsymbol{C}=\boldsymbol{L}^{-1}\boldsymbol{AL}^{-\mathrm{T}}$ 的求解。广义特征问题标准化的过程如下：

(1) 因为 $\boldsymbol{B}$ 是正定矩阵，所以可以得到 $\boldsymbol{B}$ 的 Cholesky 分解为 $\boldsymbol{B}=\boldsymbol{LL}^{\mathrm{T}}$，其中 $\boldsymbol{L}$ 为下三角矩阵；

(2)可以得到 $\boldsymbol{L}^{-1}\boldsymbol{Ax}=\boldsymbol{L}^{\mathrm{T}}\boldsymbol{x}\lambda\Rightarrow\boldsymbol{L}^{-1}\boldsymbol{A}(\boldsymbol{L}^{-1})^{\mathrm{T}}\boldsymbol{L}^{\mathrm{T}}\boldsymbol{x}=\boldsymbol{L}^{\mathrm{T}}\boldsymbol{x}\lambda$；

(3)令 $\boldsymbol{C}=\boldsymbol{L}^{-1}\boldsymbol{A}(\boldsymbol{L}^{-1})^{\mathrm{T}}$，$\boldsymbol{y}=\boldsymbol{L}^{\mathrm{T}}\boldsymbol{x}$，则得到标准稠密对称特征问题为 $\boldsymbol{Cy}=\lambda\boldsymbol{y}$。

在包括 ScaLAPACK[7]，PeIGS 和 HJS[13] 等许多并行软件包中，标准化采用的并行方法通常分两步进行：首先进行矩阵 $\boldsymbol{B}$ 的 Cholesky 分解，然后求解矩阵 $\boldsymbol{C}$。广义问题标准化算法的 GPU 实现中[14-15]，也采用了将矩阵 $\boldsymbol{B}$ 的 Cholesky 分解与求解矩阵 $\boldsymbol{C}$ 分开的方法。在并行计算过程中，因为矩阵分布在不同的处理器上，采用以上分两步进行的方法会增加不必要的消息传递，引起大量的通信开销。本文采用了将矩阵 $\boldsymbol{B}$ 的 Cholesky 分解和矩阵 $\boldsymbol{C}$ 的求解相结合的方法，避免了不必要的通信开销，增加了并行性。

本文描述的基于 MPI+CUDA 的并行标准化算法在 GPU 集群中的实现是在我们开发的 PSEPS[16](Parallel Symmetric Eigenproblem Solver)特征求解器软件包的基础上完成的。计算部分由 GPU 负责，通过调用 NVIDIA 的 cuBLAS 库完成，从而有效地保证了 GPU 实现的性能。通信部分在 CPU 上通过 MPI 进行，以充分发挥 CPU+GPU 异构平台的优势。在本文算法中，所有的操作几乎都是 BLAS3 操作，因此利用 GPU 进行加速能够充分发挥其性能。为了使得算法更适合 GPU 架构，使其能够充分利用 GPU 的计算能力，我们对算法做了一些优化改进：利用 CPU 与 GPU 之间的数据传输操作掩盖 GPU 内的数据拷贝操作，消除了拷贝的时间；同时，给出了矩阵在二维通信域中行通信域和列通信域之间的完全并行的点对点的转置算法，即将在处理器行通信域上的矩阵的转置拷贝到对应的列通信域上的矩阵；此外，给出了一种基于 MPI+CUDA 的具有多个右端项的三角矩阵方程 $\boldsymbol{BX}=\boldsymbol{A}$ 求解的并行块算法，通过最后一次并行求解完成，增加了算法的并行性，并改善了算法的负载平衡。在中科院超级计算机系统“元”上，使用多达 32 个 Nvidia Tesla K20 GPGPU 卡测试了不同规模的矩阵基于 MPI+CUDA 的并行标准化算法的实现性能，结果表明该算法取得了较好的加速效果与性能，并且具有良好的可扩展性。

2 利用 GPU 进行加速的广义稠密对称特征问题标准化串行块算法

广义稠密对称特征问题标准化算法主要涉及正定矩阵 $\boldsymbol{B}$ 的 Cholesky 分解即 $\boldsymbol{B}=\boldsymbol{LL}^{\mathrm{T}}$ 和矩阵 $\boldsymbol{C}=\boldsymbol{L}^{-1}\boldsymbol{AL}^{-\mathrm{T}}$ 的求解。显然，$\boldsymbol{C}$ 是一个对称矩阵。在计算矩阵 $\boldsymbol{C}$ 时，为了避免对下三角矩阵 $\boldsymbol{L}$ 的求逆运算以及直接的矩阵乘法运算，可以求解 $\boldsymbol{C}=\boldsymbol{L}^{-1}\boldsymbol{AL}^{-\mathrm{T}}$ 的等价形式：

$$\boldsymbol{LCL}^{\mathrm{T}}=\boldsymbol{A} \tag{2}$$

从而可以通过迭代的方法间接求解矩阵 $\boldsymbol{C}$。为了使用块算法，将矩阵 $\boldsymbol{A}$，$\boldsymbol{L}$ 和 $\boldsymbol{C}$ 划分成下面的块形式：

$$\boldsymbol{L}=\begin{pmatrix}\boldsymbol{L}_{11} & \boldsymbol{L}_{12}\\ \boldsymbol{L}_{21} & \boldsymbol{L}_{22}\end{pmatrix},\boldsymbol{C}=\begin{pmatrix}\boldsymbol{C}_{11} & \boldsymbol{C}_{12}\\ \boldsymbol{C}_{21} & \boldsymbol{C}_{22}\end{pmatrix},\boldsymbol{A}=\begin{pmatrix}\boldsymbol{A}_{11} & \boldsymbol{A}_{12}\\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22}\end{pmatrix}$$

其中，$\boldsymbol{L}_{11},\boldsymbol{C}_{11},\boldsymbol{A}_{11}\in R^{m\times m}$，则 $\boldsymbol{L}_{21},\boldsymbol{C}_{21},\boldsymbol{A}_{21}\in R^{(n-m)\times m}$，$\boldsymbol{L}_{12},\boldsymbol{C}_{12},\boldsymbol{A}_{12}\in R^{m\times(n-m)}$，$\boldsymbol{L}_{22},\boldsymbol{C}_{22},\boldsymbol{A}_{22}\in R^{(n-m)\times(n-m)}$。将上式代入式(2)，可以得到：

$$\boldsymbol{LCL}^{\mathrm{T}}=\begin{pmatrix}\boldsymbol{L}_{11}\boldsymbol{C}_{11}\boldsymbol{L}_{11}^{\mathrm{T}} & *\\ (\boldsymbol{L}_{21}\boldsymbol{C}_{11}+\boldsymbol{L}_{22}\boldsymbol{C}_{21})\boldsymbol{L}_{11}^{\mathrm{T}} & \boldsymbol{C}_{L}\end{pmatrix}=\begin{pmatrix}\boldsymbol{A}_{11} & \boldsymbol{A}_{12}\\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22}\end{pmatrix}$$

其中，$\boldsymbol{C}_L=\boldsymbol{L}_{21}\boldsymbol{C}_{11}\boldsymbol{L}_{21}^{\mathrm{T}}+\boldsymbol{L}_{22}\boldsymbol{C}_{21}\boldsymbol{L}_{21}^{\mathrm{T}}+\boldsymbol{L}_{21}\boldsymbol{C}_{21}\boldsymbol{L}_{22}^{\mathrm{T}}+\boldsymbol{L}_{22}\boldsymbol{C}_{22}\boldsymbol{L}_{22}^{\mathrm{T}}$。

由上式可以得到 $\boldsymbol{L}_{11}\boldsymbol{C}_{11}\boldsymbol{L}_{11}^{\mathrm{T}}=\boldsymbol{A}_{11}$，如果 $m=1$，则可以直接得到 $\boldsymbol{C}_{11}$。下面给出利用归纳法求解 $\boldsymbol{C}_{21}$ 和 $\boldsymbol{C}_{22}$ 的过程：

$$\boldsymbol{Z}=\frac{1}{2}\boldsymbol{L}_{21}\boldsymbol{C}_{11}+\boldsymbol{L}_{22}\boldsymbol{C}_{21}=\boldsymbol{A}_{21}\boldsymbol{L}_{11}^{-\mathrm{T}}-\frac{1}{2}\boldsymbol{L}_{21}\boldsymbol{C}_{11} \tag{3}$$

$$\boldsymbol{L}_{22}\boldsymbol{C}_{22}\boldsymbol{L}_{22}^{\mathrm{T}}=\boldsymbol{A}_{22}-\boldsymbol{L}_{21}\boldsymbol{Z}^{\mathrm{T}}-\boldsymbol{Z}\boldsymbol{L}_{21}^{\mathrm{T}} \tag{4}$$

$$\boldsymbol{C}_{21}=\boldsymbol{L}_{22}^{-1}(\boldsymbol{Z}-\frac{1}{2}\boldsymbol{L}_{21}\boldsymbol{C}_{11}) \tag{5}$$

由式(3)的右端项可以求得矩阵 $\boldsymbol{Z}$，将 $\boldsymbol{Z}$ 代入式(5)可以求得 $\boldsymbol{C}_{21}$。式(4)与式(2)具有相同的形式，因而这个过程可以继续下去。若 $m=1$，则上述计算过程是许多的 BLAS1 向量-向量和 BLAS2 矩阵-向量操作，是一个非块算法，这样并不能充分利用 GPU 的计算能力。为了有尽可能多的 BLAS3 矩

阵-矩阵操作，以充分利用 GPU 的性能，则需要块算法。图 1 展示了将正定矩阵 $\boldsymbol{B}$ 的 Cholesky 分解与矩阵 $\boldsymbol{C}$ 的求解相结合的串行块算法，其中带“*”的步骤表示正定矩阵 $\boldsymbol{B}$ 的 Cholesky 分解。接下来对尾部矩阵 $\boldsymbol{A}_{22}$ 和 $\boldsymbol{B}_{22}$ 和执行相同的操作，直到最后一个条块。

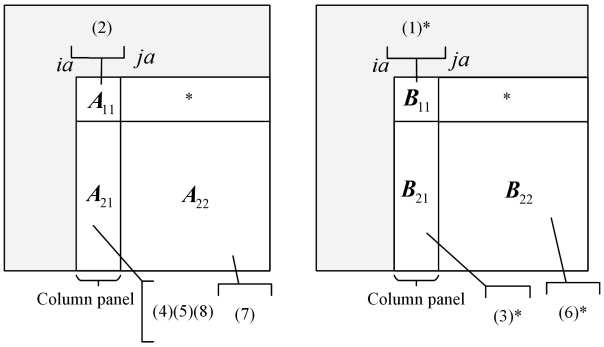


图 1 结合 Cholesky 分解的串行块算法

Fig. 1 Serial blocked algorithm with Cholesky decomposition

算法 1 描述了利用 GPU 加速的结合 Cholesky 分解的串行块算法，算法块的大小为 nb，则条块个数为 $bn=\lceil n/nb \rceil$。注意，在算法求解过程中得到的矩阵 $\boldsymbol{C}$ 和下三角矩阵 $\boldsymbol{L}$ 分别覆盖了矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的相应部分。

算法 1 结合 Cholesky 分解的串行块算法

begin

/* 将矩阵 **A** 和 **B** 传到 GPU 端 */

A→**dA**; **B**→**dB**

for each bt∈[0,bn−1]

/* 第 bt 个块的第一列/最后一列的列号 */

ia=bt×nb; ki=min(ia+nb−1, n−1)

$R_{11}=C_{11}=\{ia:ki\}$ /* 对角块的行/列号集合 */

$R_{21}=\{ki:n-1\}$, $\mathbf{C}_{21}=\{ia:ki\}$ /* $\mathbf{A}_{21}/\mathbf{B}_{21}$的行/列号集合 */

$R_{22}=C_{22}=\{ki:n-1\}$ /* $\mathbf{A}_{22}/\mathbf{B}_{22}$的行/列号集合 */

/* 将块矩阵 $\mathbf{A}_{11}$ 和 $\mathbf{B}_{11}$ 从 GPU 传到 CPU */

dA (R_{11},C_{11})→$\mathbf{A}_{11}$; **dB** (R_{11},C_{11})→$\mathbf{B}_{11}$

(1) * (in CPU) $\mathbf{L}_{11}=\mathrm{Chol}(\mathbf{B}_{11})$

(2) (in CPU) $\mathbf{C}_{11}=\mathrm{sygs2}(\mathbf{A}_{11},\mathbf{L}_{11})$

/* 将块矩阵 $\mathbf{L}_{11}$ 和 $\mathbf{C}_{11}$ 从 CPU 传回 GPU */

$\mathbf{C}_{11}$→**dA**(R_{11},C_{11}); $\mathbf{L}_{11}$→**dB**(R_{11},C_{11})

$\mathbf{C}_{11}$→$\mathbf{dC}_{11}$; $\mathbf{L}_{11}$→$\mathbf{dL}_{11}$

(3) * (in GPU)求解下三角 $\mathbf{L}_{21}$

$\mathbf{dB}(R_{21},C_{21})=\mathbf{dA}(R_{21},C_{21})\mathbf{dL}_{11}^{-\mathrm{T}}$

(4) (in GPU) **dA** $(R_{21},C_{21})=$**dB** $(R_{21},C_{21})\mathbf{dL}_{11}^{-\mathrm{T}}$

(5) (in GPU) **dA** $(R_{21},C_{21})-=\frac{1}{2}$**dB**$(R_{21},C_{21})\mathbf{dC}_{11}$

(6) * (in GPU) $\mathbf{dB}(R_{22},C_{22})-=\mathbf{dB}(R_{21},C_{21})$ **dB** $(R_{21},C_{21})^{\mathrm{T}}$

(7) (in GPU) $\mathbf{dA}(R_{22},C_{22})-=\mathbf{dB}(R_{21},C_{21})\mathbf{dA}(R_{21},C_{21})^{\mathrm{T}}+\mathbf{dA}(R_{21},C_{21})\mathbf{dB}(R_{21},C_{21})^{\mathrm{T}}$

(8) (in GPU) $\mathbf{dA}(R_{21},C_{21})-=\frac{1}{2}\mathbf{dB}(R_{21},C_{21})\mathbf{dC}_{11}$

endfor

(in GPU) 将矩阵 **dA** 的对角块拷贝到 $\mathbf{dA}_{\mathrm{D}}$

(9) $\mathbf{dA}_{\mathrm{L}}:=\mathbf{dC}_{\mathrm{L}}=\mathbf{dB}_{\mathrm{L}}^{-1}$ $\mathbf{dA}_{\mathrm{L}}$/* 更新 **A** 的下三角部分 */

(in GPU) 将 $\mathbf{dA}_{\mathrm{D}}$ 拷贝到矩阵 **dA** 的对角块位置

end

第(1)步由 $\boldsymbol{B}_{11}$ 得到 $\boldsymbol{L}_{11}$，对 $\boldsymbol{B}_{11}$ 的 Cholesky 分解可以在 CPU 上通过调用 *dpotf2* 实现。第(2)步由 $\boldsymbol{A}_{11}$ 和 $\boldsymbol{L}_{11}$ 得到矩阵 $\boldsymbol{C}_{11}$，可以在 CPU 上通过调用 *dsygs2* 实现。在第(1)步和第(2)步开始之前，需要将 $\boldsymbol{A}_{11}$ 和 $\boldsymbol{B}_{11}$ 从 GPU 端传回 CPU，计算完成后，再将 $\boldsymbol{A}_{11}$ 和 $\boldsymbol{B}_{11}$ 传到 GPU 端进行其他 GPU 上的相关运算。第(3)步对应着矩阵方程 $\boldsymbol{X}\boldsymbol{dL}_{11}^{\mathrm{T}}=\boldsymbol{dA}_{21}$ 的求解，可以在 GPU 端通过调用 cuBLAS 的 *cublasDtrsm* 实现；第(4)步与第(3)步一样，可以通过调用 *cublasDtrsm* 实现。因为 $\boldsymbol{dC}_{11}$ 为对称矩阵，第(5)步可以通过调用一次 *cublasDsymm* 实现；同样，第(8)步可以通过调用一次 *cublasDsymm* 实现。第(6)步可以通过调用一次 *cublasDgemm* 实现。第(7)步可以通过调用两次 *cublasDgemm* 实现。

在串行算法中，对矩阵 $\boldsymbol{A}$ 的下三角块部分的更新放在整个循环之后，只进行了一次运算，即第(9)步。这样，将多个计算合并，使得计算密集，提高了 GPU 的利用率。在最后的更新计算中，因为对角块元素 $\boldsymbol{X}_{11},\boldsymbol{X}_{22},\cdots,\boldsymbol{X}_{kk}$ 已经在条块中更新，实际要求解的三角矩阵方程应为如图 2 所示的三角实线内部分。若通过调用 *cublasDtrsm* 求解，此时用到的矩阵方程是如图 2 所示的矩形虚线内部分，这会影响到 $\boldsymbol{dA}$ 的对角块元素 $\boldsymbol{X}_{11},\boldsymbol{X}_{22},\cdots,\boldsymbol{X}_{k-1,k-1}$，所以在 GPU 中先将其拷贝到 $\boldsymbol{dA}_{\mathrm{D}}$，等计算结束后，再将对角块元素拷贝回去。这里，矩阵对角块元素的拷贝在 GPU 中进行，从而减少了 CPU 与 GPU 之间的数据传输，并且利用了 GPU 的多线程编程，使拷贝操作可以并行进行，使得拷贝时间花费较少。

$$\begin{pmatrix} \boldsymbol{B}_{11} & 0 & \cdots & 0 \\ \boldsymbol{B}_{21} & \boldsymbol{B}_{22} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ \boldsymbol{B}_{k1} & \boldsymbol{B}_{k2} & \cdots & \boldsymbol{B}_{kk} \end{pmatrix} \begin{pmatrix} \boldsymbol{X}_{11} & 0 & \cdots & 0 \\ \boldsymbol{X}_{21} & \boldsymbol{X}_{22} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ \boldsymbol{X}_{k1} & \boldsymbol{X}_{k2} & \cdots & \boldsymbol{X}_{kk} \end{pmatrix} = \begin{pmatrix} \boldsymbol{A}_{11} & 0 & \cdots & 0 \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ \boldsymbol{A}_{k1} & \boldsymbol{A}_{k2} & \cdots & \boldsymbol{A}_{kk} \end{pmatrix}$$

图 2 三角矩阵方程 $\boldsymbol{BX}=\boldsymbol{A}$

Fig. 2 Triangular equations $\boldsymbol{BX}=\boldsymbol{A}$

3 基于 MPI+CUDA 的广义稠密对称特征问题标准化的并行块算法

基于串行标准化块算法 1，算法 2 给出了基于 MPI+CUDA 的广义稠密对称特征问题标准化的并行块算法在 GPU 集群上的实现。并行算法的数据分布采用二维块循环分布，二维块循环具有支持可拓展的高性能并行稠密线性代数编码的特性[17-19]。并行算法 2 中将矩阵 $\boldsymbol{A}$ 和矩阵 $\boldsymbol{B}$ 采用二维块循环分布到 $p=pr\times pc$ 个处理器网格中，存储块的大小为 nb。这里，采用算法块的大小与存储块的大小相同的策略，则条块个数为 $bn=\lceil n/nb \rceil$。假设每个处理器拥有矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的行号集合 RS 与列号集合 CS；并且在算法开始之前已经将部分矩阵 $\boldsymbol{A}(RS,CS)$ 和 $\boldsymbol{B}(RS,CS)$ 传到了对应的 GPU 中：$\boldsymbol{A}(RS,CS)$→$\boldsymbol{dA}(RS,CS)$，$\boldsymbol{B}(RS,CS)$→$\boldsymbol{dB}(RS,CS)$。算法 2 中，$\boldsymbol{C}_{\boldsymbol{A}},\boldsymbol{L}_{\boldsymbol{B}}\in R^{nb\times nb}$，$\boldsymbol{W}_{\boldsymbol{A}},\boldsymbol{W}_{\boldsymbol{B}}\in R^{(n/pr)\times nb}$，$\boldsymbol{U}_{\boldsymbol{A}},\boldsymbol{U}_{\boldsymbol{B}}\in R^{nb\times(n/pc)}$。前边加“$\boldsymbol{d}$”的变量表示在 GPU 端对应的变量。

算法 2 基于 MPI+CUDA 的并行标准化算法

begin

for each bt∈[1,bn]

ia=(bt−1)×nb

ki=min(ia+nb−1,n)
bm=ki−ia+1
if ia∈RS and ia∈CS then
/* 将矩阵 **dA** 和 **dB** 的对角块从 GPU 取到 CPU */
(GC-1)**dA** (ia:ki,ia:ki)→$\mathbf{C}_A$
(GC-2)**dB** (ia:ki,ia:ki)→$\mathbf{L}_B$
/* 计算 $\mathbf{L}_B$ 的 Cholesky 分解 */
(1) * (in CPU) $\mathbf{L}_B:\mathbf{L}_B=\mathrm{Chol}(\mathbf{L}_B)$
(2) (in CPU) $\mathbf{C}_A=\mathrm{sygs2}(\mathbf{C}_A,\mathbf{L}_B)$
/* 更新 GPU 中矩阵 **dA** 和 **dB** 的对角块部分 */
(CG-3) $\mathbf{C}_A\rightarrow\mathbf{dA}$(ia:ki,ia:ki)
(CG-4) $\mathbf{L}_B\rightarrow\mathbf{dB}$(ia:ki,ia:ki)
(CC-1) Cbroadcast($\mathbf{C}_A$)/* 列播送 */
(CC-2) Cbroadcast($\mathbf{L}_B$)/* 列播送 */
(CG-5&6)$\mathbf{C}_A\rightarrow\mathbf{dC}_A;\mathbf{L}_B\rightarrow\mathbf{dL}_B$
endif
/* 从行号集合 RS 中移除 ia 到 ki 行 */
if ia∈RS then RS=RS\{ia:ki} endif
if ia∈CS then
$\Lambda=\{\mathrm{ia}:n\}\cap\mathrm{RS}$
(3) * (in GPU)计算 **L**(RS,ia:ki):(trsm)
$\mathbf{dB}(\mathrm{RS},\mathrm{ia}:\mathrm{ki})=\mathbf{dB}(\Lambda,\mathrm{ia}:\mathrm{ki})\mathbf{dL}_B^{-\mathrm{T}}$
(4) (in GPU)计算 **C**(RS,ia:ki):(trsm)
$\mathbf{dA}(\mathrm{RS},\mathrm{ia}:\mathrm{ki})=\mathbf{dA}(\Lambda,\mathrm{ia}:\mathrm{ki})\mathbf{dL}_B^{-\mathrm{T}}$
(5) (in GPU) (symm)
$\mathbf{dA}(\mathrm{RS},\mathrm{ia}:\mathrm{ki})=\mathbf{dA}(\Lambda,\mathrm{ia}:\mathrm{ki})-\mathbf{dB}(\Lambda,\mathrm{ia}:\mathrm{ki})\mathbf{dC}_A/2$
(GC-7) **dA** (RS,ia:ki)→$\mathbf{W}_A$
(GC-8) * **dB**(RS,ia:ki)→$\mathbf{W}_B$
(CC-3)Rbroadcast($\mathbf{W}_A$)/* 行播送 */
(CC-4)Rbroadcast($\mathbf{W}_B$) /* 行播送 */
(CG-9&10)$\mathbf{W}_A\rightarrow\mathbf{dW}_A$; $\mathbf{W}_B\rightarrow\mathbf{dW}_B$
endif
if ia∈CS then CS=CS\(ia:ki) endif
/* 二维通信域中矩阵转置 */
(T1)$\mathbf{dU}_A=\mathrm{TransposeM}(\mathbf{dW}_A)$
(T2)$\mathbf{dU}_B=\mathrm{TransposeM}(\mathbf{dW}_B)$
if ia∈RS then
(GC-11) $\mathbf{U}_A\leftarrow\mathbf{dU}_A$
(GC-12) $\mathbf{U}_B\leftarrow\mathbf{dU}_B$
(CC-5) Cbroadcast($\mathbf{U}_A$)/* 列播送 */
(CC-6)Cbroadcast($\mathbf{U}_B$)/* 列播送 */
(CG-13&14)$\mathbf{U}_A\rightarrow\mathbf{dU}_A$; $\mathrm{U}_B\rightarrow\mathbf{dU}_B$
endif
(6) * (in GPU)Performing rank-nb correction on the remaining elements of **B**: (gemm)
$\mathbf{dB}(\mathrm{RS},\mathrm{CS})-=\mathbf{dW}_B(\mathrm{RS},1:\mathrm{bm})\mathbf{dU}_B(1:\mathrm{bm},\mathrm{CS})$
(7)(in GPU)Performing rank-2nb correction on the remaining elements of **dA**: (gemm * 2)
$\mathbf{dA}(\mathrm{RS},\mathrm{CS})-=\mathbf{dW}_A(\mathrm{RS},1:\mathrm{bm})\mathbf{dU}_B(1:\mathrm{bm},\mathrm{CS})+\mathbf{dW}_B(\mathrm{RS},1:\mathrm{bm})\mathbf{dU}_A(1:\mathrm{bm},\mathrm{CS})$
if ia∈RS then
(8)(in GPU) (symm) $\mathbf{dA}(\Lambda,\mathrm{ia}:\mathrm{ki})-=\mathbf{dB}(\Lambda,\mathrm{ia}:\mathrm{ki})\mathbf{dC}_A/2$
endif
endfor
(9)调用算法 4 并行求解三角方程:**BX**=**A**
end

在算法 2 的描述中,(GC-数字)表示数据从 GPU 传到 CPU,(CG-数字)表示数据从 CPU 传到 GPU,(CC-数字)表示 CPU 之间的通信操作。算法 2 的通信操作 *Rbroadcast*(行播送)表示在处理器网格中的某处理器将数据广播到该处理器所在行通信域中的所有处理器;*Cbroadcast*(列播送)表示将数据广播到处理器所在列通信域中的所有处理器;*receive* 表示接收数据。此外,算法 2 中 *TransposeM* 表示矩阵转置操作:将在处理器行/列通信域上的矩阵转置拷贝到在处理器列/行通信域上对应的矩阵,算法 3 具体描述了此过程。

图 3 描述了算法 2 的流程,CPU 之间的通信包括如图 3 所示的 6 次操作,通信次数和通信量如表 1 所列。CPU+GPU 的异构编程需要 CPU 与 GPU 之间的数据传输,在 GPU 端进行加速计算,需要先将计算数据上传到 GPU 端的显存;因为使用的多个 GPU 分布在不同的节点上,所以 GPU 间的通信需要在 CPU 上通过 MPI 进行,首先将要通信的数据从 GPU 端传到 CPU 端,之后通过 MPI 通信,通信结束后再将数据传回 GPU 中。算法 2 中 CPU 与 GPU 的数据传输操作如图 3 所示,共有 14 次传输操作,数据传输量如表 2 所列。

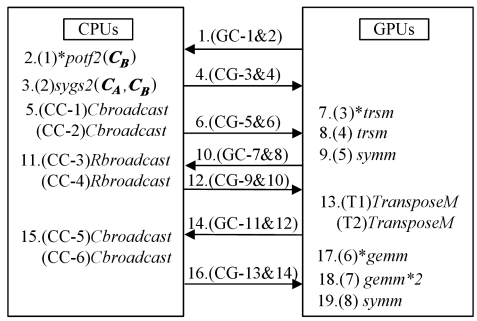


图 3 算法 2 的流程

Fig. 3 Flow of algorithm 2

表 1 算法 2 中的通信

Table 1 Communications in algorithm 2

Step	Message start times	Message volume
(CC-1)|(CC-2)	$\log_2 pc$	$nb\times nb$
(CC-3)|(CC-4)	$\log_2 pr$	$((n-bt)/pc)\times nb$
(CC-5)|(CC-6)	$\log_2 pc$	$((n-bt)/pr)\times nb$

表 2 算法 2 中 CPU 与 GPU 之间的数据传输

Table 2 Data transmission between CPU and GPU in algorithm 2

Step	Data transfer volume
(GC-1&2)|(CG-3&4)|(CG-5&6)	$nb\times nb$
(GC-7&8)|(CG-9&10)|(GC-11&12)	$((n-bt)/pc)\times nb$
(CG-13&14)	$((n-bt)/pr)\times nb$

计算 $\boldsymbol{B}_{11}$ 的 Cholesky 分解时需要将 $\boldsymbol{C}_A$ 和 $\boldsymbol{L}_B$ 从矩阵 ***dA*** 和 ***dB*** 中拷贝出来。因为 ***dA*** 和 ***dB*** 已将在 GPU 端了,而 $\boldsymbol{C}_A$ 和 $\boldsymbol{L}_B$ 需要从 GPU 端取到 CPU 进行计算,这里利用了 GPU 与 CPU 之间的数据传输操作(GC-1)和(GC-2),直接将 $\boldsymbol{C}_A$ 和 $\boldsymbol{L}_B$ 在 ***dA*** 与 ***dB*** 中对应的部分直接从 GPU 传到 CPU 上的 $\boldsymbol{C}_A$ 和 $\boldsymbol{L}_B$:$\boldsymbol{dA}(ia:ki,ia:ki)\rightarrow\boldsymbol{C}_A$,$\boldsymbol{dB}(ia:ki,ia:ki)\rightarrow\boldsymbol{L}_B$。计算完

成后，再将 $\boldsymbol{C}_A$ 和 $\boldsymbol{L}_B$ 上传到 GPU 端 $\boldsymbol{dA}$ 与 $\boldsymbol{dB}$ 对应的部分(CG-3)和(CG-4)。这样便利用 CPU 与 GPU 之间的数据传输掩盖了数据的拷贝操作，消除了拷贝的时间。(GC-7)和(GC-8)也同样利用了数据传输操作掩盖了 GPU 内的数据拷贝操作。

算法 2 中的(T1)和(T2)对应矩阵在处理器行和列间的转置操作：将在行通信域上的矩阵转置拷贝到对应的在列通信域上的矩阵。该矩阵转置操作对应一个完全并行的点对点的通信方式。这里以(T1)为例，通过(CC-3)和(CG-9)，每个行通信域中的处理器对应的 GPU 中已经包含了完整的 $\boldsymbol{dW}_A$。这样，每个处理器行上的处理器可以并行操作：每个行处理器发送一个块矩阵，对应的列处理器接收相应的块矩阵。若 $sp==rp$，则不需要 GPU 与 CPU 之间的数据传输，在 GPU 内直接利用多线程编程并行地将部分块矩阵转置拷贝到目标块矩阵；否则，在发送数据的处理器中首先将要发送的部分块矩阵从 GPU 取到 CPU，然后在 CPU 上通过 MPI 发送数据；若当前处理器为接收数据的处理器，则接收数据，然后将接收到的块矩阵从 CPU 传到 GPU，最后在 GPU 内并行地将部分块矩阵转置拷贝到目标块矩阵。因此，此算法为一个完全并行的点对点的转置算法。算法 3 具体描述了此过程。其中，sp 表示发送数据的处理器，rp 表示接收数据的处理器，$R1/C1$ 表示发送的块矩阵的行/列号集合，$R2/C2$ 表示接收的块矩阵的行/列号集合。

算法 3 $\boldsymbol{dC}=TransposeM(\boldsymbol{dR})$

```
begin
  for each i∈[bt+1,bn-1] then
    (1)发送的块矩阵在 sp 上的起始位置：
      ia1=bt×nb; ja1=i×nb;
    接收的块矩阵在 rp 上的起始位置：
      ia2=ja1; ja2=ia1;
    (2)传递的块矩阵的大小(行数、列数)：
      bm=min(ia1+nb-1,n-1)-ia1+1
      bn=min(ja1+nb-1,n-1)-ja1+1
    (3)获取(ia1,ja1)在 sp 上的本地位置(i1,j1)
      获取(ia2,ja2)在 rp 上的本地位置(i2,j2)
      R1={ i1:i1+bm}; C1={ j1:j1+bn }
      R2={ i2:i2+bn }; C2={ j2:j2+bm}
    (4-1)if sp==rp then /*转置拷贝*/
      (in GPU)dC(R2,C2)←dR(R1,C1)^T
    (4-2)else if 当前处理器为 sp then
      (GC-1)将块矩阵从 GPU 传到 CPU：
        R(R1,C1)←dR(R1,C1)
      (CC-1)发送块矩阵 R(R1,C1)
    (4-3)else if 当前处理器为 rp then
      (CC-2)接收块矩阵 R(R1,C1)
      /*将块矩阵从 CPU 传到 GPU*/
      (GC-2)R(R1,C1)→dR(R1,C1)
        (in GPU)dC(R2,C2)←dR(R1,C1)^T
      endif
  endfor
end
```

算法 4 描述了并行求解如图 2 所示的三角实线内矩阵方程的过程。因为需要求解的是 $\boldsymbol{X}$ 主对角线下的三角部分，所以仅需要 $\boldsymbol{B}$ 的第二个列块以后的下三角部分以及 $\boldsymbol{A}$ 的主对角线下的三角部分。在算法 4 中，这通过在开始时从 RS/CS 中移除第一个列块对应的行/列序号实现。最后将三角方程右端的矩阵 $\boldsymbol{B}$ 的各个列条块 *Rbroadcast* 给本行的处理器进行求解，两个主要的计算步骤是(1)和(2)，这两部分计算可以通过调用 cuBLAS 的 *cublasDtrsm* 和 *cublasDgemm*，利用 GPU 进行加速。

算法 4 并行求解三角方程 $\boldsymbol{BX}=\boldsymbol{A}$

```
begin
  if 0∈RS then /*从 RS 中移除第一个列块对应的行号*/
    RS=RS\(1,min(n,nb)) endif
  if 0∈CS then /*从 CS 中移除第一个列块对应的列号*/
    CS=CS\(1:min(n,nb)) endif
  for each bt∈[1,bn-1] then
    ia=bt×nb;ki=min(ia+nb-1,n-1);bm=ki-ia+1
    if ia∈CS then
      (GC-1)B_R←dB(RS,ia:ki)
      (CC-1)Rbroadcast(B_R)/*行播送*/
      (CG-2)B_R→dB_R
    endif
    Λ={1:ia}∩CS
    if ia∈RS then
      (1) (in GPU)求解 X 的第 bt 个行块
        dA(ia:ki,Λ)=dB_R^{-1}(ia:ki,ia:ki) dA(ia:ki,Λ)
      RS=RS\(ia:ki)
      (GC-3)A_C←dA(ia:ki,Λ)
      (CC-2) Cbroadcast(A_C)/*列播送*/
      (CG-4)A_C→dA_C
    endif
      (2)(in GPU)修正 dA(ki+1:n,1:ia)中的元素
        dA(RS,Λ)-=dB_R(RS,1:bm) dA_C(1:bm,Λ)
  endfor
end
```

4 实验结果与分析

为了测试算法的性能和准确性，本文计算了 Frank 矩阵 $\boldsymbol{A}=(a_{ij})$ 的特征值和特征向量，其中 $a_{ij}=n-\max(i,j)-1(i,j=1,2,\cdots,n)$。Frank 矩阵的特征值为：

$$\lambda_k=\frac{1}{2(1+\cos((\frac{2k-1}{2n+1})\pi)},\ k=1,2,\cdots,n$$

在中科院计算机网络中心超级计算机系统“元”上实现了文中描述的广义特征问题标准化算法。每台 GPGPU 计算节点配置 2 块 Nvidia Tesla K20 GPGPU 卡(峰值双精度浮点性能(板卡)1.17 Tflops)，2 颗 Intel E5-2680 V2(Ivy Bridge 10C|2.8 GHz)处理器，64 GB DDR3 ECC 1866 MHz 内存。实验使用的 cuBLAS 版本是 6.0.37，MPI 是 openmpi-1.7.5。在测试时，为了使得多个 GPU 并行加速计算，每个节点申请两个进程，一个进程对应一个 GPU。测试的数据是双精度浮点数。

4.1 串行算法的性能分析

当 nb 取不同的值时，通信次数、负载平衡以及利用 GPU 计算的性能都会受到影响，这将会影响整个标准化算法的性能，因此需要找到一个合适的值，使得算法达到最好的性能。图 4 展示了不同规模的矩阵串行块算法中当块的大小 nb 取不同的值(32，64，128，256，512)时的性能。可以发现，当 nb=256 时，算法取得的性能最高，因此在下面的分析中，均取 nb=256。从图 4 中可以看出，矩阵规模越大，算法的性能越好，当矩阵规模为 16000×16000，nb 取 256 时，算法达到的性能约为 721.09 Gflop/s，此时的 GPU 利用率约达到了 61.63%。

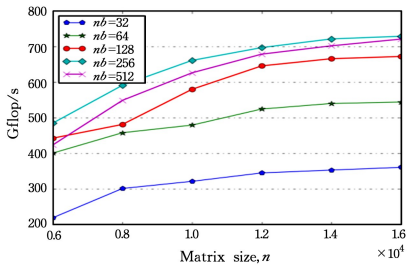


图 4 在单个 GPU 上 nb 取不同值时算法的性能

Fig. 4 Algorithm performance of nb with different values on a GPU

4.2 基于 MPI+CUDA 的并行标准化算法的性能分析

表 3 对比了 16000×16000 阶矩阵标准化算法在 CPU 上调用 MKL 2013 的 BLAS 库与调用 GPU 的 cuBLAS 库时，在不同进程数(1，2，4，8，16，32)条件下的耗时。从表中可以看出，随着处理器数目的增多，加速倍数在不断降低。这是因为随着处理器数目的增多，在单个 GPU 上的负载不断减小，而 GPU 适用于数据密集型的任务，负载越小，加速效果越不明显；同时，随着进程数目的增多，通信时间以及 CPU 与 GPU 之间数据传输时间等所占比例增大，从而导致加速效果不如进程数目少时好。总的来说，相较于 CPU 版本，GPU 版本取得了较好的加速效果。

表 3 使用 CPU 或 GPU 的时间比较

Table 3 Comparison of time using CPUs or GPUs

Number of processes	CPU/s	GPU/s	Speedup
1	410.94	15.51	26.50
2	235.89	11.82	19.96
4	113.16	8.98	12.60
8	69.82	6.87	10.16
16	36.24	4.6	7.88
32	24.75	4.11	6.02

图 5 展示了在不同矩阵规模(10000×10000，15000×15000，20000×20000，25000×25000 和 30000×30000)的条件下使用不同 GPU 个数时的加速比。从图中可以看出，随着 GPU 使用个数的增加，加速比不断增加，但是增加的幅度在不断减小。这是因为随着 GPU 个数的增加，GPU 的负载不断降低，计算时间的占比不断降低，而 GPU 与 CPU 之间的数据传输、通信等开销所占的比例则会增加。另外还可以看出，在使用相同的 GPU 个数时，问题规模越大，加速比越大。这是由于问题规模越大，在 GPU 上的负载越大，越能够充分地利用 GPU 的计算能力，取得的加速效果越好，得到的加速比也就越大。规模越大，加速比越好，因此本文描述的基于 MPI+CUDA 的并行标准化算法具有良好的可扩展性。

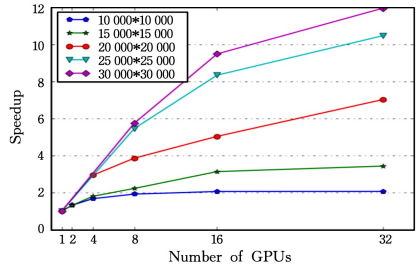


图 5 不同矩阵规模的加速比

Fig. 5 Speedup curves for different matrix sizes

图 6 给出 20000×20000 阶矩阵标准化算法中各个主要部分所花费的时间。其中，$compu_gpu$ 表示在 GPU 上的计算时间，$compu_cpu$ 表示在 CPU 上的计算时间，$memcpy$ 表示 GPU 与 CPU 之间的数据传输时间，$ctime$ 表示 CPU 之间的通信时间，$trsm$ 表示算法 4 并行求解三角矩阵方程的时间，$others$ 表示其他时间。从图中可以发现，随着 GPU 使用数目的增加，在 GPU 上的计算时间 $compu_gpu$ 逐渐递减；$compu_cpu$，$memcpy$，$trsm$ 等时间也在随着 GPU 使用数目的增加不断减少。总的来说，利用多个 GPU 进行加速计算的标准化算法的总耗时随着 GPU 使用数目的增加而不断减少，表现出了良好的可扩展性。

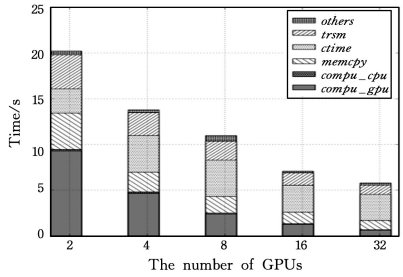


图 6 并行标准化算法各部分的耗时

Fig. 6 Time of each part of parallel standardization algorithm

图 7 给出了不同矩阵规模使用不同数目的 GPU 个数(1，2，4，8，16，32)的并行标准化算法时达到的性能。从图中可以看出，当使用的 GPU 个数一样时，问题规模越大，得到的性能越高。这是因为问题规模越大，在 GPU 上的负载越大，更能够充分地利用 GPU 的计算能力；并且规模越大，计算时间所占的比例会越大，加速效果越明显，因此取得的性能就越好。文中描述的基于 MPI+CUDA 的并行标准化算法取得了较好的性能。

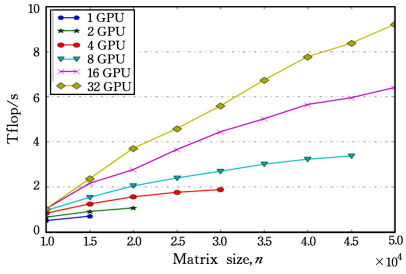


图 7 并行标准化算法的性能

Fig. 7 Performance of parallel standardization algorithm

结束语 本文给出了基于 MPI+CUDA 的广义稠密对称特征问题标准化算法在 GPU 集群上的实现，利用多个 GPU 对计算部分进行并行加速计算，通信通过在 CPU 上的 MPI 完成。该标准化算法采用将正定矩阵 $\boldsymbol{B}$ 的 Cholesky 分解与传统标准化算法相结合的策略，降低了通信开销，增加了算法的并行性。在基于 MPI+CUDA 的并行算法中，通过利用 CPU 与 GPU 之间的数据传输操作掩盖了数据在 GPU 内的拷贝操作；同时给出了矩阵在二维通信网格中行和列通信域之间的转置算法；另外，还给出了一个基于 MPI+CUDA 的具有多个右端项的三角矩阵方程的并行块求解算法。在基于 MPI+CUDA 的并行算法中，块的大小将会影响通信次数、负载平衡以及利用 GPU 计算的性能，在使用时需要权衡。本文描述的算法取得了较好的加速效果与性能，并且具有良好的可扩展性。

参 考 文 献

[1] KENT P C. Computational challenges of large-scale, long-time, first-principles molecular dynamics[J]. Journal of Physics: Conference Series, 2008, 125: 012058.

[2] AUCKENTHALER T, BLUM V, BUNGARTZ H J, et al. Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations[J]. Parallel Computing, 2011, 37(12): 783-794.

[3] SINGH D J. Introduction to the LAPW method[M]// Planewaves, Pseudopotentials and the LAPW Method. Boston, MA: Springer US, 1994: 35-43.

[4] CHEN X S, VONG S W, LI W, et al. Noda iterations for generalized eigenproblems following Perron-Frobenius theory[J]. Numerical Algorithms, 2019, 80(3): 937-955.

[5] CUCURINGU M, DAVIES P, GLIELMO A, et al. SPONGE: A generalized eigenproblem for clustering signed networks[J]. arXiv: 1904.08575, 2019.

[6] BUTKOVIČ P, JONES D. On special cases of the generalized max-plus eigenproblem[J]. SIAM Journal on Matrix Analysis and Applications, 2016, 37(3): 1002-1021.

[7] BLACKFORD L S, CHOI J, CLEARY A, et al. ScaLAPACK users' guide[M]. Philadelphia: SIAM, 1997.

[8] ALPATOV P, BAKER G, EDWARDS C, et al. PLAPACK Parallel Linear Algebra Package Design Overview[C]// Proceedings of the 1997 ACM/IEEE Conference on Supercomputing. IEEE, 1997: 29-29.

[9] BAKER G, GUNNELS J, MORROW G, et al. PLAPACK: high performance through high-level abstraction[C]// 1998 International Conference on Parallel Processing (Cat. No. 98EX205). IEEE, 1998.

[10] IMAMURA T, YAMADA S, MACHIDA M. Development of a high-performance eigensolver on a peta-scale next-generation supercomputer system[J]. Progress in Nuclear Science and Technology, 2011, 2: 643-650.

[11] POULSON J, MARKER B, GEIJN R A V D, et al. Elemental: A new framework for distributed memory dense matrix computations [J]. ACM Transactions on Mathematical Software (TOMS), 2013, 39(2): 13.

[12] PETSCHOW M, PEISE E, BIENTINESI P. High-performance solvers for dense Hermitian eigenproblems[J]. SIAM Journal on Scientific Computing, 2013, 35(1): C1-C22.

[13] HENDRICKSON B, JESSUP E, SMITH C. Toward an efficient parallel eigensolver for dense symmetric matrices[J]. SIAM Journal on Scientific Computing, 1998, 20(3): 1132-1154.

[14] TOMOV S, NATH R, DU P, et al. MAGMA Users' Guide[EB/OL]. ICL, UTK (November 2009). https://icl.cs.utk.edu/projectsfiles/magma/doxygen/.

[15] HAIDAR A, SOLCÁ R, GATES M, et al. Leading edge hybrid multi-GPU algorithms for generalized eigenproblems in electronic structure calculations[M]// Lecture Notes in Computer Science. Berlin: Springer, 2013: 67-80.

[16] ZHAO Y H, CHI X B, CHENG Q. Efficient parallel blocked algorithms for generalized Hermitian eigenproblem[J]. Journal of Computer Research and Development, 2007, 44(10): 1724-1732.

[17] LICHTENSTEIN W, JOHNSSON S L. Block-cyclic dense linear algebra[J]. SIAM Journal on Scientific Computing, 1993, 14(6): 1259-1288.

[18] BOLEN J, DAVIS A, DAZEY B, et al. Massively parallel distributed computing: World's first 281 gigaflop supercomputer [EB/OL]. Proceedings of the Intel Supercomputer Users Group. https://pdfs.semanticscholar.org/52ed/b836555ead0769708968adbb58a64120d34b.pdf.

[19] BISSELING R H, VAN DE VORST J G G. Parallel LU decomposition on a transputer network[C]// Conference Organized by Koninklijke/Shell-Laboratory, Amsterdam. Berlin: Springer, 1988: 61-77.



LIU Shi-fang, born in 1994, Ph.D. Her main research interests include numerical parallel algorithm library implementation and optimization of parallel algorithm library under heterogeneous parallel machine.



ZHAO Yong-hua, born in 1966, Ph.D, professor, Ph.D supervisor. His main research interests include parallel algorithm and software, parallel programming model, spectral clustering data analysis and so on.