Two-Stage Column Block Parallel LU Factorization Algorithm

RONGTENG WU1,2 AND XIAOHONG XIE3
1College of Computer and Control Engineering, Minjiang University, Fuzhou 350108, China
2Fujian Provincial Key Laboratory of Information Processing and Intelligent Control, Fuzhou 350108, China
3Department of Teaching Affairs, Minjiang University, Fuzhou 350108, China

Corresponding author: Rongteng Wu (rongtengwu@163.com)

This work was supported in part by the National Natural Science Foundation of China under Grant 61871204 and Grant 61703195, in part by the Natural Science Foundation of Fujian Province under Grant 2018J01544, in part by the Fujian Provincial Leading Project under Grant 2017H0030, and in part by the Key Project of College Youth Natural Science Foundation of Fujian Province under Grant JZ160467.

ABSTRACT Parallel computing is increasingly important in computer architectures, parallel architecture has become ubiquitous in our everyday lives. Novel architectures and programming models pose new challenges to algorithm design and system software development. This paper presents a two-stage column block parallel LU factorization algorithm for multiple-processor architectures. Any given matrix is first partitioned into large blocks, and then, every large block is partitioned into a number of small blocks according to the number of processors. Finally, the small column blocks are allocated to processors in an orderly “serpentine arrangement.” Each iteration of the column block parallel LU factorization is separated into two stages of operation. In the first stage, the first-step factorization operation is processed in advance and nonblocking communication is used to reduce the processor idle and waiting time and improve parallelism. In the second stage, the large blocks are used to satisfy more powerful processors, such as GPUs, which require more data to exploit their computing capabilities. Experiments are conducted on a multicore system and multi-GPU system with different configurations to test the algorithm’s performance. Compared with other related column block parallel LU factorizations, the two-stage algorithm exhibits better load balancing and parallel execution time performance.

INDEX TERMS LU factorization, load balancing, nonblocking communication, parallel execution time, scalability.

I. INTRODUCTION
Parallelism appears to be the future of computing. Innovations in hardware architecture, such as multi-GPU and hyperthreading processors, have made parallel computing resources available for universal desktop computers, laptop computers, and even embedded devices [1]–[3]. Parallel computing will be increasingly important in the modern economy and businesses, but conventional programming techniques may not work well in new architectures [4]–[7]. Novel architectures and programming models pose new challenges to algorithm design and system software development. Exploitation of these innovations requires the extension of parallel programming techniques to all areas of software design and development [8]–[10]. Many research studies have focused on parallel programming technology for multicore and multi-GPU architectures [11]–[14]. Gaussian elimination is a classic parallel programming algorithm used to solve dense linear algebra systems that are widely employed in scientific and engineering models [15], [16]. Cholesky, LU, and QR factorization routines are included in almost all popular linear algebra libraries, such as Linear Algebra Package (LAPACK) [17], Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) [18], and Matrix Algebra on GPU and Multicore Architectures (MAGMA) [19], [20]. MAGMA is a collection of next-generation linear algebra libraries for heterogeneous architectures. A high-level parallel programming model is used in [21] to implement an algorithm for dense linear algebra on multicore and GPU architectures. Kim et al. [22] present a task-parallel
algorithm to deal with the data dependencies for sparse incomplete Cholesky factorization. A right-looking algorithm for LU, Cholesky, and QR for GPUs is described by Volkov and Demmel [23] to optimize the performance of matrix factorization. A row block cyclic distribution strategy is used in [24] to improve load balancing and reduce the communication cost for Cholesky factorization. Endo et al. [25] describe a load balancing method for making the best use of the computational capability of all processors in a system to update a trail matrix. A static distribution policy is used in [26] for matrix factorization on heterogeneous architectures. Choi et al. [27] also use a static block cycle distribution strategy to develop a parallel block Cholesky factorization algorithm for multiple-processor systems. A block-cyclic redistribution scheme is presented in [28] to derive the optimal total cost for some particular instances of the redistribution problem. Park et al. [29] present an efficient algorithm for array redistribution to reduce the overall time of communication based on a generalized circulant matrix formalism. A dynamic scheduling policy is presented in [30] on the basis of activity on edge network for block parallel Cholesky factorization to improve parallelism and load balancing. Deisher et al. [31] also describe a dynamic load balancing strategy for matrix factorization in a multicore system. A column block uniform allocation policy is used in [32] for parallel LU factorization on heterogeneous platforms to improve load balancing and communication. Agullo et al. [33] perform an extensive analysis and comparison of static and dynamic Cholesky factorization algorithms for multiple-processor platforms. A pipeline technique is used in [34] to redistribute data on a multiprocessor grid during runtime. This pipeline technique can significantly reduce the amount of time required to complete a dynamic data transfer task. Dong et al. [35] design a heterogeneous algorithm for Cholesky factorization and QR factorization using a static data distribution to realize load balancing. However, they do not consider the parallelism in the first step of each iteration of the LU factorization. In this paper, we present a two-stage block parallel LU factorization algorithm to develop parallelism of LU factorization and further improve load balancing on multiple processors.

The remainder of this paper is organized as follows: Section II discusses the column block parallel LU factorization algorithm. Section III presents the two-stage column block parallel LU factorization algorithm for multiple-processor architectures. Section IV presents the experimental results, and a brief conclusion is presented in Section V.

II. COLUMN BLOCK PARALLEL LU FACTORIZATION ALGORITHM

The LU factorization of any given matrix $A$ uses a series of Gaussian eliminations to form $\pi A = LU$, where $\pi$ is a permutation matrix, $L$ is a unit lower triangular matrix, and $U$ is an upper triangular matrix. Because the LU factorization algorithm for an $N \times N$ square matrix can be easily modified to fit an $M \times N$ rectangular matrix, we only discuss the LU factorization algorithm for a square matrix in this paper. The computational complexity of the LU factorization is $O(2N^3/3)$ floating-point additions and multiplications, and partial pivoting increases a further $O(N^2 + N)/2$ floating-point additions and multiplications for an $N \times N$ matrix.

Given an $N \times N$ matrix $A$ to be partitioned into $n \times n$ blocks,

$$A = \begin{pmatrix}
A_{00} & \cdots & A_{0(n-1)} \\
\vdots & \ddots & \vdots \\
A_{(n-1)0} & \cdots & A_{(n-1)(n-1)}
\end{pmatrix}, \quad (1)$$

the column block parallel LU factorization can be implemented in $n$ iterations. In fact, no extra storage is required for the factorized results $L_{ij}$ and $U_{ij}$ because they will overwrite the corresponding blocks $A_{ij}$ of the original matrix $A$. All $\pi$ can be stored in an $N$-tuple vector. Assuming that, after the $(k-1)$-th iteration, $A$ is factorized into

$$A^{(k-1)} = \begin{pmatrix}
L_{00}U_{00} & \cdots & U_{0(k-1)} \\
\vdots & \ddots & \vdots \\
L_{(n-1)0}U_{(n-1)0} & \cdots & U_{(n-1)(k-1)}
\end{pmatrix}, \quad (2)$$

the $k$-th iteration will then involve the following four operational steps:

Step 1: Run the LU factorization algorithm on the $k$-th column block (panel) to compute

$$\pi_k \begin{pmatrix} A^{(k-1)}_{kk} \\
\vdots \\
A^{(k-1)}_{(n-1)k} \end{pmatrix} = \begin{pmatrix} L^{(k)}_{kk} \\
\vdots \\
L^{(k)}_{(n-1)k} \end{pmatrix} U_{kk}. \quad (3)$$

Obviously, $L^{(k)}_{kk} = L_{kk}$.

Step 2: Execute row swapping on the column blocks located in the left or right of the $k$-th column block based on $\pi_k$, i.e.,

$$\pi_k \begin{pmatrix}
L^{(k-1)}_{0k} & \cdots & L^{(k-1)}_{(k-1)k} \\
\vdots & \ddots & \vdots \\
L^{(k-1)}_{(n-1)k} & \cdots & L^{(k-1)}_{(n-1)(k-1)}
\end{pmatrix} \begin{pmatrix} A^{(k-1)}_{k(k+1)} & \cdots & A^{(k-1)}_{k(n-1)} \\
\vdots & \ddots & \vdots \\
A^{(k-1)}_{(n-1)(k+1)} & \cdots & A^{(k-1)}_{(n-1)(n-1)}
\end{pmatrix}$$

$$\Rightarrow \begin{pmatrix}
L^{(k)}_{0k} & \cdots & L^{(k)}_{(k-1)k} \\
\vdots & \ddots & \vdots \\
L^{(k)}_{(n-1)k} & \cdots & L^{(k)}_{(n-1)(k-1)}
\end{pmatrix} \begin{pmatrix} A^{(k)}_{k(k+1)} & \cdots & A^{(k)}_{k(n-1)} \\
\vdots & \ddots & \vdots \\
A^{(k)}_{(n-1)(k+1)} & \cdots & A^{(k)}_{(n-1)(n-1)}
\end{pmatrix}, \quad (4)$$

Step 3: Update the blocks $A^{(k)}_{k(k+1)} \cdots A^{(k)}_{(n-1)(n-1)}$ located on the right side of block $A_{kk}$ into $(U_{k(k+1)} \cdots U_{k(n-1)})$ according to the following equation:

$$L_{kk} (U_{k(k+1)} \cdots U_{k(n-1)}) = (A^{(k)}_{k(k+1)} \cdots A^{(k)}_{k(n-1)}). \quad (5)$$
Step 4: Update the blocks located in the lower right of block (trailing submatrix) \( A_{kk} \), i.e.,
\[
\begin{pmatrix}
A_{(k+1)(k+1)} & \cdots & A_{(k+1)(n-1)} \\
\vdots & \ddots & \vdots \\
A_{(n-1)(k+1)} & \cdots & A_{(n-1)(n-1)}
\end{pmatrix}
= \begin{pmatrix}
A_{k(k+1)} & \cdots & A_{k(n-1)} \\
\vdots & \ddots & \vdots \\
A_{(n-1)(k)} & \cdots & A_{(n-1)(n-1)}
\end{pmatrix}
\times \left( U_{k(k+1)} \cdots U_{k(n-1)} \right).
\tag{6}
\]

For convenience, we also use vector form to denote any partial column blocks in the later discussion, e.g.,
\[
\overrightarrow{A}_{ij} = \begin{pmatrix} A_{ij} \\ \vdots \\ A_{(n-1)j} \end{pmatrix}, \quad \overrightarrow{L}_{ij} = \begin{pmatrix} L_{ij} \\ \vdots \\ L_{(n-1)j} \end{pmatrix}, \text{ etc.}
\]

Typically, these four steps can be implemented by calling four BLAS and LAPACK subroutines, i.e., dgetrf(), dlaswp(), dtrsm(), and dgemm(), for double-precision data. We use the LAPACK subroutine dgetrf(\( \overrightarrow{A}_{kk}, \overrightarrow{L}_{kk}, U_{kk}, \pi_k \)) to represent LU factorization of the column block corresponding to Eq. (3), the LAPACK subroutine dlaswp(RestColumn(\( A_{kk} \), \( \pi_k \))) to perform the row exchange on the column blocks of A except for \( \overrightarrow{A}_{kk} \) corresponding to Eq. (4), the BLAS subroutine dtrsm(L_{kk}, RightBlock(\( A_{kk} \))) to update the column blocks located on the right side of block \( A_{kk} \) (or \( U_{kk} \)) corresponding to Eq. (5), and the BLAS subroutine dgemm(\( \overrightarrow{L}_{(k+1)k}, \text{LowerRight}(\( A_{kk} \))) to update the blocks located in the lower right of block \( A_{kk} \) corresponding to Eq. (6). Generally, \( n \) column blocks of the matrix A will be distributed to \( p \) processors \( P_0, \ldots, P_{p-1} \) cyclically for a system with \( p \) processors, and then any processor \( P_i \) \( (i = 0, \ldots, p-1) \) contains \( n/p \) column blocks \( i, i+p, \ldots, i+(n/p-1)p \), which form a submatrix S. Suppose that the \( k \)-th column block of A is located in the \( k \)-th column block of the submatrix S of P_y (\( y = k \mod p \)); then, \( \overrightarrow{A}_{0k} \) and \( \overrightarrow{A}_{kk} \) become \( S_{0k} \) and \( S_{kk} \) for P_y, respectively. The column block parallel LU factorization for multiprocessor systems can be described as follows in Algorithm 1.

The algorithm can easily be modified to support multi-GPU systems by using corresponding routines such as cublasDswap(), cublasDtrsm(), and cublasDgemm(). However, Algorithm 1 does not consider the parallelism on the first step of each iteration. In fact, in any iteration \( k \), every processor needs a broadcasting and synchronization operation, i.e., MPI_Bcast() in Algorithm 1, to wait for the current processor \( P_y \) (\( y = k \mod p \)) to run the routine dgetrf() and broadcast the results \( \overrightarrow{L}_{kk} \) and \( \pi_k \), which will lead to processor idle and degenerate the parallelism. Hence, we introduce a two-stage column block parallel LU factorization algorithm for multiple-processor architectures.

Algorithm 1 Column Block Parallel LU Factorization Algorithm for Multiprocessor Architectures

\[
\text{for } (k = 1; k < n; k++) \\
\quad \text{if (process_id == y) } /\text{ y = k mod p} /\* Processor } P_y \text{ runs LU factorization on its } \overrightarrow{S}_{kk} \text{ /}\* \\
\quad \text{MPI_Bcast(} \overrightarrow{L}_{kk}, \pi_k, y); /\* Broadcasting and synchronization /\* \\
\quad \text{dlaswp(RestColumn}(S, \overrightarrow{A}_{kk}, \pi_k); /\* Each processor swaps its column blocks originally located on the left or right of } \overrightarrow{A}_{kk} \text{ /}\* \\
\quad \text{dtrsm}(L_{kk}, \text{RightBlock}(A_{kk})); /\* Each processor updates its blocks originally located on the right of } A_{kk} \text{ /}\* \\
\quad \text{dgemm}(L_{(k+1)k}, \text{LowerRight}(A_{kk})); /\* Each processor updates its blocks originally located in the lower right of } A_{kk} \text{ /}\* \\
\}
\]

III. TWO-STAGE COLUMN BLOCK PARALLEL LU FACTORIZATION ALGORITHM

The two-stage column block parallel LU factorization algorithm is still implemented in \( n \) iterations. However, each iteration will be separated into two stages of operation. In the first stage, the matrix’s small column blocks are allocated to processors in an orderly “serpentine arrangement” to obtain better load balancing, and then nonblocking communication is used to reduce processor idle and waiting time and improve the parallelism. The second stage will run on the large matrix blocks to satisfy more powerful processors, such as GPUs, which require more data to exploit their computing capabilities.

A. MATRIX PARTITION AND DATA DISTRIBUTION

Suppose that an \( N \times N \) matrix A is partitioned into \( n \times n \) large blocks, which are then each partitioned into \( 2p \times 2p \) small blocks again according to the number of processors \( p \). In other words, matrix A is partitioned into \( 2pn \times 2pn \) small blocks, which can be denoted as

\[
A = \begin{pmatrix}
a_{11} & \cdots & a_{1N} \\
\cdots & \ddots & \cdots \\
a_{N1} & \cdots & a_{NN}
\end{pmatrix} = \begin{pmatrix}
A_{11} & \cdots & A_{1n} \\
\cdots & \cdots & \cdots \\
A_{n1} & \cdots & A_{nn}
\end{pmatrix} = \begin{pmatrix}
A_{11} & \cdots & A_{1(2pn)} \\
\cdots & \cdots & \cdots \\
A_{(2pn)1} & \cdots & A_{(2pn)(2pn)}
\end{pmatrix}.
\tag{7}
\]

For any given large column block \( \overrightarrow{A}_{0i} \) (\( 0 \leq i \leq n - 1 \)), its \( 2pn \) small column blocks \( \overrightarrow{A}_{0(2pi+j)} \) (\( 0 \leq j \leq 2p - 1 \)) will be distributed to \( p \) processors in a
“serpentine arrangement.” Specifically, the $2p$ small column blocks $\mathbf{A}(0_{2p})$, $\cdots$, $\mathbf{A}(0_{2p(2p-1)})$ will be allocated to processors $P_0, P_1, \cdots, P_{p-1}, P_{p-1}, \cdots, P_1, P_0$ in an orderly manner; that is, any processor $P_i(0 \leq x \leq P - 1)$ will be allocated two small blocks $\mathbf{A}(0_{2p^i+x})$ and $\mathbf{A}(0_{2p^i+2p-1-x})$ for a given large column block $\mathbf{A}_{0i}$. Consequently, the processor $P_x$ contains $2n$ small column blocks $\mathbf{A}_x$, $\mathbf{A}(0_{2p-1-x}), \cdots$, $\mathbf{A}(0_{2p(n-1)+x}), \mathbf{A}(0_{2p(n-1)+2p-1-x})$, which form a submatrix and will be denoted as $S$ or $S_1$. Moreover, we assume that the $(2p^i+j)$-th column block of $A$ is allocated to one processor $P$ and is the $(2p^i+j)$-th column block of the submatrix $S$ of $P_i$: $\mathbf{A}(0_{2p^i+j})$, then becomes $S(0_{2p^i+j})$ for $P$. Hence, we use the different symbols $\mathbf{S}(2p^i+j)(2p^i+j)$ or $\mathbf{A}(2p^i+j)(2p^i+j)$ to denote the same column block in different situations. Figure 1 shows the matrix partition and data distribution in a four-processor system. For example, $P_1$ is allocated $2n$ small column blocks $\mathbf{A}(0_{2x}), \mathbf{A}(0_{2x+1}), \cdots, \mathbf{A}(0_{2x(n-1)+1}), \mathbf{A}(0_{2x(n-1)+2})$, which form a submatrix denoted as $S$ or $S_2$.

B. TWO-_STAGE COLUMN BLOCK PARALLEL LU FACTORIZATION ALGORITHM DESIGN

For a given matrix $A$ as shown in Formula (7), two-stage column block parallel LU factorization is implemented in $n$ iterations on a $p$-processor system. Initially, processor $P_0$ factorizes $\mathbf{A}_{00}$ (or $\mathbf{S}_{00}$) to obtain $\mathbf{L}_{00}$, $\mathbf{U}_{00}$, and $\pi_0$, as in Eq. (1). Then, each iteration will be separated into two stages of operation to obtain better parallelism and load balancing. The first stage is aimed at computing the $k$ large column blocks, i.e., $2p \times k$ small column blocks (the value of $k$ will be discussed later). For any given iteration $i$ ($i = 0, \ldots, n-1$), the first stage consists of $2p$ steps to process the $2p \times k$ small column blocks $2pi, \cdots, 2pi + (2pk - 1)$, i.e., $k$ large column blocks $i, \ldots, i + k - 1$. Any processor $P_i(0 \leq z \leq p)$ will process its $2k$ small column blocks $\mathbf{A}(0_{2p^i+z}), \mathbf{A}(0_{2p^i+2p-1-z}), \cdots, \mathbf{A}(0_{2p^i+k-1+z}), \mathbf{A}(0_{2p^i+k-1+z}(2p^i+j))$, which form a submatrix and will be denoted as $S(i)$ or $S(0)$. Figure 2 shows the relationship of $A$, $S$, and $S(0)$ at the start of the first stage of the $i$-th iteration for a four-processor system. In this case, the first $i$ row blocks $0, \ldots, i - 1$ of matrix $A$ have been processed completely, and their is no operation to be carried out in the later steps, as shown in Figure 2 (a). The four processors are allocated different submatrices $S$ partitioned from $A$, as shown in Figure 2 (b).

The operations of the $j$-th step of the first stage at the $i$-th iteration of the two-stage algorithm are described as follows:

1. Every processor $P_i$ receives $\mathbf{L}(2p_{i+j})$ and $\pi_{2pi+j}$, which are the factorization results of the $(2p^i+j)$-th small column blocks $\mathbf{A}(2p^i+j)(2p^i+j)$ computed by processor $P_x$ in the previous step, where

$$
\begin{aligned}
x & = \begin{cases}
  j, & (j \leq p - 1) \\
  2p - 1 - j, & (p - 1 < j \leq 2p - 1).
\end{cases}
\end{aligned}
$$

(8)

2. If the processor number is $y$, where

$$
\begin{aligned}
y & = \begin{cases}
  j + 1, & (j < p - 1) \\
  2p - 2 - j, & (p - 1 \leq j < 2p - 1) \\
  0, & (j = 2p - 1).
\end{cases}
\end{aligned}
$$

(9)

then $P_y$ processes the $(2pi+j+1)$-th small column block in advance, i.e.,

1. Run column swapping on $\mathbf{A}(2pi+j)(2pi+j+1)$ according to $\pi_{2pi+j}$. 

FIGURE 1. Matrix partition and data distribution for a system with four processors.

FIGURE 2. Relationship of $A$, $S$, and $S(0)$ at the start of the first stage of the $i$-th iteration for a four-processor system.
2) Compute $U(2p_i+j)(2p_i+j+1)$ using $A(2p_i+j)(2p_i+j+1)$ and $L(2p_i+j)(2p_i+j)$. 
3) Update $A(2p_i+j+1)(2p_i+j+1)$ using $L(2p_i+j+1)(2p_i+j)$ and $U(2p_i+j+1)(2p_i+j+1)$. 
4) Factorize $A(2p_i+j+1)(2p_i+j+1)$ and obtain $L(2p_i+j+1)(2p_i+j+1)$, $U(2p_i+j+1)(2p_i+j+1)$, and $\pi_j+1$.
5) $P_y$ broadcasts $L(2p_i+j+1)(2p_i+j+1)$ and $\pi_j+1$ to all processors in nonblocking broadcast mode.

According to $\pi_{2p_i+j}$, every processor $P_z$ runs row swapping on its small column blocks in the $i$th large column, as in Eq. (4), but not including the small column blocks $2p_i+j$ and $2p_i+j+1$.

(5) Every processor $P_z$ uses $L(2p_i+j)(2p_i+j)$ to update its corresponding blocks in $A(2p_i+j)(2p_i+j+2)\ldots A(2p_i+j)(2p_i+2p_i-1)$ and located on the right of $A(2p_i+j)(2p_i+j+1)$, as in Eq. (5).

(6) Every processor $P_z$ updates its corresponding blocks in $A(2p_i+j+1)(2p_i+j+2)\ldots A(2p_i+j+1)(2p_i+2p_i-1)$ and located on the lower right of $A(2p_i+j)(2p_i+j+1)$, as in Eq. (6).

The second stage of the $i$th iteration will process other column blocks and can be described as follows:

1) All processors gather the $i$th large column block and obtain:

$$
\tilde{L}_{ii} = \begin{pmatrix}
L_{ii} & \cdots \\
\cdots & L_{ii+2p_i-1}(2p_i+2p-1) & \cdots
\end{pmatrix}
$$

(10)

2) Every processor $P_z$ runs row swapping on its column blocks on the left or right of $(\tilde{A}_{ii}, \ldots, \tilde{A}_{(i+k-1)})$ according to $\pi_i$ as in Eq. (4), where $\pi_i$ is composed of $\pi_{2p_i+j}(j = 0, \ldots, 2p_i-1)$.

3) Every processor $P_z$ updates its corresponding blocks in $A_{(i+k)}(\ldots, A_{(i+k-1)}$ and located on the right of $A_{(i+k-1)}$ by using $L_{ii}$, as in Eq. (5).

4) Every processor $P_z$ updates its corresponding blocks in $A_{(i+k)}(\ldots, A_{(i+k-1)}$ and located on the lower right of $A_{(i+k-1)}$, as in Eq. (6).

Obviously, the communication in the first stage is non-blocking, which will improve load balancing and parallelism because the task is approximately equal, and the processors hardly have idle and waiting time.

When the BLAS and LAPACK routines dgetrf(), dlaswp(), dtrsm(), and dgemm() are used correspondingly, the two-stage column block parallel LU factorization algorithm for multiple-processor architectures can be described as shown in Algorithm 2.

In Algorithm 2, the values of $x$ and $y$ are determined by Eqs. (8) and (9), respectively. LeftColumn($S^{(i)}$, $\tilde{A}_{(2p_i+j)(2p_i+j)}$) represents the column blocks belonging to the corresponding $S^{(i)}$ and located originally on the left of $\tilde{A}_{(2p_i+j)(2p_i+j)}$, and LeftColumn($S$, $\tilde{A}_{ii}$) represents the column blocks belonging to the corresponding $S$ and located originally on the left of $\tilde{A}_{ii}$. RightColumn($S^{(i)}$, $\tilde{A}_{(2p_i+j)(2p_i+j)}$) represents the small column blocks belonging to the corresponding $S^{(i)}$ and located originally on the right of $\tilde{A}_{(2p_i+j)(2p_i+j)}$, and RightColumn($S$, $\tilde{A}_{ii}$) represents the column blocks belonging to the corresponding $S$ and located originally on the right of $\tilde{A}_{ii}$. RightBlock($S^{(i)}$, $A_{(2p_i+j)(2p_i+j)}$) represents the small blocks belonging to the corresponding $S^{(i)}$ and located originally on the right of $\tilde{A}_{(2p_i+j)(2p_i+j)}$, and RightBlock($S$, $A_{ii}$) represents the small blocks located originally on the left of $\tilde{A}_{ii}$.
belonging to the corresponding $S^{(i)}$ and located originally on the lower right of $\mathcal{A}_{(2p_i+j)(2p_i+j)}$ and LowerRight($S$, $\mathcal{A}_{(i+k-1)})$ represents the blocks belonging to the corresponding $S$ and located originally on the lower right of $\mathcal{A}_{(i+k-1)}$.

For a shared-memory multicore system, at the $j$-th step of the first stage of any iteration $i$, the broadcast operation MPI_Bcast() is not necessary because all CPU cores can directly access matrix $A$ in shared memory, but the operation corresponding to the wait operation MPI_Wait(req[j]) is needed to verify the previous step’s factorization results $\mathcal{L} (2pi+j)(2pi+j)$ and $\pi_{2pi+j}$. Alternatively, Algorithm 2 can be easily modified to run on multi-GPU systems by using related routines, such as cublasDswap(), cublasDtrsm(), and cublasDgemm().

C. PERFORMANCE ANALYSIS

According to the two-stage column block parallel algorithm, because the small column blocks are allocated to processors in an orderly “serpentine arrangement,” all processors are allocated identical matrix data and have almost identical computing tasks. Therefore, all processors hardly have idle and waiting time only if all processors can receive the factorization results $\mathcal{L} (2pi+j)(2pi+j)$ and $\pi_{2pi+j}$ in the $j$-th step at the first stage of every iteration $i$ in a timely manner because a nonblocking communication operation is used. In fact, we draw the following conclusion:

If $k \geq \frac{1}{2}(p+3)$, then all processors can receive the factorization results in a timely manner and hardly have idle and waiting time in the first stage.

In the $j$-th step of the first stage of the $i$-th iteration, the main computing operations include dlaswp($\pi_{2pi+j}$, $\mathcal{A}_{(2pi+j)(2pi+j)}$), $dt校선$($\mathcal{L}_{(2pi+j)(2pi+j)}$, $A_{(2pi+j)(2pi+j)}$), $dt校선$($\mathcal{L}_{(2pi+j+1)(2pi+j)}$, $A_{(2pi+j+1)(2pi+j)}$) ($j < j_2 \leq 2p - 1$), dgemm($\mathcal{L}_{(2pi+j+1)(2pi+j)}$, $\mathcal{A}_{(2pi+j+1)(2pi+j)}$), and dgetrf($\mathcal{A}_{(2pi+j+1)(2pi+j+1)}$), which will be simply denoted as dgetrf($\mathcal{A}_{(2pi+j+1)(2pi+j+1)}$), $dt校선$($\mathcal{L}_{(2pi+j)(2pi+j)}$, $A_{(2pi+j)(2pi+j)}$) and dgemm($\mathcal{L}_{(2pi+j)(2pi+j)}$, $\mathcal{A}_{(2pi+j)(2pi+j)}$) computed by dtrsm() and the data $\mathcal{A}_{(2pi+j)(2pi+j)}$ computed by dgemm() is exactly $\mathcal{A}_{(2pi+j)(2pi+j)}$. Thus, we denote $dt校선$($\mathcal{L}_{(2pi+j)(2pi+j)}$, $A_{(2pi+j)(2pi+j)}$) and dgemm($\mathcal{L}_{(2pi+j)(2pi+j)}$, $\mathcal{A}_{(2pi+j)(2pi+j)}$) together as $dt校线$($\mathcal{A}_{(2pi+j)(2pi+j)}$). Compared with $dt校线$($\mathcal{A}_{(2pi+j)(2pi+j)}$) or dgetrf($\mathcal{A}_{(2pi+j)(2pi+j)}$), the execution time of dlaswp($\pi_{2pi+j}$, $\mathcal{A}_{(2pi+j)(2pi+j)}$) can be ignored. Hence, we consider only the operations $dt校线$() and dgetrf() to reasonably discuss the parallelism and load balancing. According to the time complexity, the execution time of dgetrf($\mathcal{A}_{(2pi+j+1)}$) is approximately half that of $dt校线$($\mathcal{A}_{(2pi+j+1)(2pi+j+1)}$) ($j < j_2 \leq 2p - 1$). Let the execution time of $dt校线$($\mathcal{A}_{(2pi+j+1)(2pi+j+1)}$) and dgetrf($\mathcal{A}_{(2pi+j+1)(2pi+j+1)}$) be $t_1$ and $t_2$, respectively. Then, for the $j$-th step of the first stage of the $i$-th iteration, the task load of $P_0$ is

$$P_0_{\text{Task}_j} = \begin{cases} (2k - 1)t_1, & 0 \leq j < 2p - 2 \\ (2k - 1)t_1 + t_2, & j = 2p - 2 \\ (2k - 2)t_1 + t_2, & j = 2p - 1 \end{cases} \quad (11)$$

and the task load of $P_z (z = 1, \ldots, p - 1)$ is

$$P_z_{\text{Task}_j} = \begin{cases} 2kt_1, & 0 \leq j < z - 1 \\ 2kt_1 + t_2, & j = z - 1 \\ (2k - 1)t_1, & z - 1 < j < 2p - 2 - z \\ (2k - 1)t_1 + t_2, & j = 2p - 2 - z \\ (2k - 2)t_1, & 2p - 2 - z < j \leq 2p - 1. \end{cases} \quad (12)$$

The total task load of $P_0$ at the first stage of the $i$-th iteration is

$$P_0_{\text{Task}} = \sum_{j=0}^{2p-1} P_0_{\text{Task}_j} = (4pk - 2p - 1) t_1 + 2t_2 \quad (13)$$

The total task load of $P_z$ at the first stage of the $i$-th iteration is

$$P_z_{\text{Task}} = \sum_{j=0}^{2p-1} P_z_{\text{Task}_j} = (4pk - 2p - 1) t_1 + 2t_2. \quad (14)$$

In other words, all processors have equal total task loads at the first stage of any iteration. According to Eq. (9) and Eq. (10), for any given $z (1 \leq z \leq p - 1)$ and $j_0 (0 \leq j_0 \leq 2p - 1)$, we have

$$\sum_{j=0}^{j_0} P_0_{\text{Task}_j} \leq \sum_{j=0}^{j_0} P_z_{\text{Task}_j}. \quad (15)$$

Hence, for any given step $j_0 > 0$, any processor $P_z$ can receive the results of dgetrf($\mathcal{A}_{(2pi+j)(2pi+j)}$), which is run by $P_z$ at step $j_0 - 1$, in a timely manner only if the operation dgetrf($\mathcal{A}_{(2pi+j)(2pi+j)}$) has been completed before processor $P_0$ starts step $j_0$, where $x$ is determined by Eq. (8).

1) When $j_0 = 0$, the corresponding operation dgetrf($\mathcal{A}_{(2pi-1)(2pi-1)}$) has been completed in the first stage of the previous iteration $i - 1$.

2) When $j_0 = 2p - 1$, then $P_z$ becomes $P_0$. Obviously, according to Eq. (15), any processor $P_z$ can receive the results of dgetrf($\mathcal{A}_{(2pi-2p-1+2pi-2p-1)}$) run by $P_0$ in a timely manner.

3) When $0 < j_0 \leq p - 1$, the task load of $P_x$ before dgetrf($\mathcal{A}_{(2pi+j)(2pi+j)}$) has been completed is

$$T_1 = \sum_{j=0}^{j_0-1} P_x_{\text{Task}_j} - (2k - 1) t_1 = 2kj_0 t_1 + t_2 - 2kt_1 + t_1 \quad (16)$$

and the task load of $P_0$ before starting step $j_0$ is

$$T_2 = \sum_{j=0}^{j_0-1} P_0_{\text{Task}_j} = (2k - 1)j_0 t_1. \quad (17)$$
According to $T_1 \leq T_2$, we obtain
\[ 2kt_1 \geq (j_0 + 1)t_1 + t_2. \]  
(18)

Because $t_2 < t_1$, so long as $2kt_1 \geq (j_0 + 1)t_1 + t_1$, that is,
\[ 2k \geq j_0 + 2, \]  
(19)

and $j_0 \leq p - 1$, only if $2k \geq p + 1$, that is
\[ k \geq \frac{1}{2}(p + 1). \]  
(20)

4) When $p \leq j_0 \leq 2p - 2$, the task load of $P_i$ before dgetrf($A_{(2pi+j_0)/(2pi+j_0)}$) has been completed is
\[ T_i' = \sum_{j=0}^{j_0-1} P_{x(Task)}j - (2k - 2)t_1 \]
\[ = 2kj_0t_1 + 2pt_1 - 2j_0t_1 - 2kt_1 + t_1 + 2t_2, \]  
(21)

and the task load of $P_0$ before starting step $j_0$ is
\[ T_0' = \sum_{j=0}^{j_0-1} P_{0(Task)}j = 2kj_0t_1 - j_0t_1, \]  
(22)

According to $T_i' \leq T_0'$, we obtain
\[ 2pt_1 - j_0t_1 - 2kt_1 + t_1 + 2t_2 \leq 0. \]  
(23)

Because $t_2 < t_1$ and $j_0 \geq p$, we have
\[ 2pt_1 - j_0t_1 - 2kt_1 + t_1 + 2t_2 \]
\[ \leq 2pt_1 - pt_1 - 2kt_1 + t_1 + 2t_1 = (p + 3 - 2k)t_1, \]  
(24)

Hence, only if $(p + 3 - 2k)t_1 \leq 0$, that is
\[ k \geq \frac{1}{2}(p + 3). \]  
(25)

To summarize 1), 2), 3), and 4), when $k \geq \frac{1}{2}(p + 3)$, all processors can receive the factorization results in a timely manner and hardly have idle and waiting time in the first stage. All processors have almost identical task loads, and the algorithm exhibits better parallelism.

Figure 3 shows the execution sequence and dependency of all basic computing tasks in the first stage of the first iteration in a three-processor system. In this case, the value of $k$ is 3, which satisfies Eq. (25). Then, the total task load of every processor includes 29 dTrGe() tasks and 2 dgetrf() tasks, i.e., 29$t_1+2t_2$, which is consistent with Eqs. (13) and (14). Every red arrow indicates the dependency of the two tasks belonging to different processors. It is observed from these arrows that no processors wait idly for other processor to process their predecessor tasks. For example, considering the computing task dTrGe($A_{9,11}$, $A_{9,12}$, $A_{9,17}$, $A_{9,18}$, $A_{9,23}$), the total task load that needs to be processed by $P_0$ before it is 15$t_1$, and the total task load that needs to be processed by $P_2$ before it is 13$t_1+2t_2$. Hence, $P_0$ can receive the results of dgetrf($A_{9,9}$) in a timely manner and has no idle and waiting time.

IV. EXPERIMENTAL RESULTS
The experiment is performed on an AMAX XG-48201GK system consisting of two Intel Xeon E5-2620 v4 CPUs and eight NVIDIA Titan XP GPUs. Every CPU contains 8 cores, and the frequency is 2.1 GHz. Table 1 shows the main hardware resources in our experiments.

| TABLE 1. Experimental environment. |
|-----------------------------------|
| **CPU** | **GPU** |
| Processor type | Intel Xeon E5-2620 v4 | NVIDIA Titan XP |
| Processor number | 2 | 8 |
| Core number | 24 | 32 |
| Frequency | 2.1 GHz | 3840 Cores/GPU |
| Memory | 16 GB | 1582 MHz |

A. LOAD BALANCING
For the traditional algorithm described as Algorithm 1, the factorization operation on $A_{ik}$ in every iteration is only run by a single processor; then, the operation is the broadcast of related data and synchronization, so the waiting time would be considerable for a system with more processors. The two-stage algorithm presented in the paper improves the load balancing and reduces processors’ idle and waiting time by distributing data in a “serpentine arrangement” style and using a nonblocking communication mode in the first stage in every iteration. To evaluate the processor idle time and load balancing of the algorithm, the metric Average idle ratio is
defined as follows:

\[
\text{Average idle ratio} = \frac{\text{Average idle time}}{\text{Parallel execution time}} \quad (26)
\]

The smaller \( \text{Average idle ratio} \) is, the lower the waiting time and the better the load balancing performance are for a given algorithm. The best is when the \( \text{Average idle ratio} \) is equal to 0, which means no synchronization waiting time and the best load balancing; on the contrary, the worst is when the \( \text{Average idle ratio} \) approaches 1.

In our experiment, we test the \( \text{Average idle ratio} \) for the traditional algorithm described in Algorithm 1 and the two-stage algorithm presented in this paper. The experiment is performed on the multicore system configured with 8 cores and 16 cores, respectively. Figure 4 shows the \( \text{Average idle ratio} \) comparison of the traditional algorithm and the two-stage algorithm for different multicore configurations and different matrix sizes \( N \). The two algorithms run with larger \( \text{Average idle ratios} \) on a 16-core system than on an 8-core system. The reason could be that, when the CPU cores increase from 8 to 16, the parallel execution time decreases, but the communication time does not decrease. The \( \text{Average idle ratios} \) of the two algorithms in the different systems decrease when the matrix sizes increase from 8,192 to 36,384. Generally, when the matrix is larger, the parallel execution time will be larger and the synchronization waiting time will be relatively smaller. Most of the \( \text{Average idle ratios} \) of the traditional algorithm are between 2% and 10%. However, the \( \text{Average idle ratios} \) of the two-stage algorithm are less than 1% for the different systems. Compared with the traditional algorithm, the two-stage algorithm exhibits better parallelism and load balancing.

**B. PARALLEL EXECUTION TIME**

The parallel execution time of an algorithm is related to the load balancing strategy, storage mode, communication and synchronization method, etc. for a specific system. Parallel execution time is the most important metric for a parallel algorithm. Parallel execution time is tested in our experiments to measure the performance of the algorithm. We compare the two-stage algorithm with the traditional algorithm on a multicore system and then compare it with the two representative LU factorization algorithms “magma_dgetrf” and “magma_dgetrf_mgpu” in MAGMA on a multi-GPU architecture. MAGMA is the state of the art in dense linear algebra library development for multicore and multi-GPU hybrid architectures. The magma_dgetrf and magma_dgetrf_mgpu algorithms are representative LU factorization algorithms for multicore and multi-GPU hybrid systems in MAGMA. These two magma_dgetrf and magma_dgetrf_mgpu algorithms are right-looking Level-3 BLAS versions of the algorithm for computing the LU factorization of a general \( M \times N \) matrix \( A \) using partial pivoting with row interchanges. The former is a hybrid CPU/GPU routine where the matrix is initially in the CPU host memory, and the latter is a hybrid CPU/multiple-GPU routine where the matrix is distributed across multiple GPUs’ device memories.

Figure 5 shows a performance comparison of the two-stage algorithm and traditional algorithm for double-precision matrices on different multicore systems configured with 8 cores and 16 cores. Figure 6 gives a performance comparison of the two-stage algorithm, magma_dgetrf, and magma_dgetrf_mgpu for double-precision matrices on different multi-GPU systems configured with four GPUs and eight GPUs. In Figures 5 and 6, the \( x \)-axis represents the different algorithms and different system configurations, the \( y \)-axis indicates the size \( N \) of the double-precision matrices.
matrices, and the z-axis shows the parallel execution time of the different algorithms in seconds. When the size of matrices $N$ is increased from 8,192 to 20,480 for the multicore system or from 16,384 to 40,960 for the multi-GPU system, the parallel execution time increases gradually. In most cases, having more processors available in the system leads to lower parallel execution time for a given algorithm and matrix. However, when the matrix size is smaller, such as $N = 16,384$ and 20,480, the parallel execution time of magma_dgetrf on four GPUs is less than that on eight GPUs. The reason is that the smaller matrix cannot provide sufficient computation for more GPUs, but the greater number of GPUs means more data transfer between the CPU and GPU, and the communication and synchronization cost accounts for a considerable proportion of the parallel execution time for magma_dgetrf. The performance of the two-stage algorithm is better than that of the traditional algorithm on the multicore system and that of magma_dgetrf and magma_dgetrf_mgpu on the multi-GPU system.

In multicore and multi-GPU heterogeneous system, the panel is factorized on the CPU core and the trailing submatrix is updated on the GPU for the algorithms magma_dgetrf and magma_dgetrf_mgpu, while the panel factorization and the trailing submatrix updating are run on the GPU for the two-stage algorithm. The former overlaps the left panel factorization and trailing submatrix computed by using the right-looking ahead technique to decrease the waiting time of GPUs, while the latter uses a data distribution strategy and partial asynchronous operation among GPUs to decrease the waiting time of GPUs. Since the panel is factorized on the CPU core and the trailing submatrix is updated on the GPU, the algorithm magma_dgetrf or magma_dgetrf_mgpu has to copy the column block data from the GPU to the corresponding CPU core and then broadcast the factorization results from this CPU core to other CPU cores and to corresponding GPUs. By contrast, the two-stage algorithm only needs to broadcast the data from the GPU to other GPUs directly because the panel factorization and trailing submatrix updating are run on the GPU. Obviously, the communication cost of the algorithm magma_dgetrf or magma_dgetrf_mgpu is larger than that of the two-stage algorithm. Increasing the CPU cores and GPUs in the system will lead to a rapid increase in the communication costs. In fact, the time of panel factorization run by the GPU is much less than that run by the CPU core or the communication cost. The communication cost of the algorithm magma_dgetrf or magma_dgetrf_mgpu will exceed the total costs of the communication time and the panel factorization time of the two-stage algorithm when we increase CPU cores and GPUs in the system. As shown in Figure 6, the performance of the two-stage algorithm is better than that of magma_dgetrf and magma_dgetrf_mgpu on the multi-GPU system when the number of GPUs is increased to four in our experiment. Furthermore, the communication cost is related to matrix size, column block size, hardware architecture, bandwidth, etc. Figure 7 shows the comparison of the execution time of the panel factorization time and the communication cost in different systems. The x-axis represents the matrix size from 16,384 to 40,960, and the y-axis indicates the panel factorization time and the communication cost. The left blue bar is the total time of panel factorization run on the GPU, the middle red bar is the total time of panel factorization run on the CPU core, and the right yellow bar is the communication cost when the panel is factorized on the CPU core. Obviously, the panel factorization time of the GPU is much less than the panel factorization time of the CPU core and the communication cost, especially in the system with 8 CPU cores and 8 GPUs.

**C. SCALABILITY**

Scalability is often used to evaluate the acceleration of an algorithm when solving a definite problem and the capability of an algorithm to solve potentially larger problems when more computing resources are available. An experiment is conducted on the multicore system to measure how the parallel execution time accelerates when the number of CPU cores is increased gradually for a fixed-size matrix. Then, the performance variation trend is tested when the size of the matrix and the number of CPU cores increase simultaneously. Figure 8 shows the parallel execution times of the two-stage algorithm and the traditional algorithm for a
A two-stage column block parallel LU factorization algorithm for multiple-processor architectures. Any given matrix is first partitioned into large blocks to satisfy more powerful processors, such as GPUs, which require more data to exploit their computing capabilities. Then, these large blocks are partitioned into a number of small blocks according to the number of processors, and the small column blocks are allocated to processors in an orderly “serpentine arrangement” to improve load balancing and parallelism. Each iteration of the column block parallel LU factorization algorithm is separated into two stages of operations. In the first stage, the first-step factorization operation is processed in advance, and non-blocking communication is used to reduce processor idle and waiting time and improve parallelism. In the second stage, the large column blocks are used to provide sufficient data to satisfy more powerful processors, such as GPUs. Experiments compare the load balancing and parallel execution time of the two-stage algorithm and the traditional algorithm on a multicore system and then compare the parallel execution time of the two-stage algorithm and two representative algorithms on a multi-GPU system. The two-stage algorithm exhibits better load balancing and parallel execution time performance. Furthermore, experiments on scalability also show good acceleration performance of the two-stage algorithm when increasing the number of processors for larger problems.

**REFERENCES**

[1] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemeranier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos, “A taxonomy of task-based parallel programming technologies for high-performance computing,” *J. Supercomput.*, vol. 74, no. 4, pp. 1422–1434, Apr. 2018.

[2] B. Zhou and R. Buyya, “Augmentation techniques for mobile cloud computing: A taxonomy, survey, and future directions,” *CSURACM Comput. Surv.*, vol. 51, no. 1, pp. 1–38, Jan. 2018.

[3] A. Geist and D. A. Reed, “A survey of high-performance computing scaling challenges,” *Int. J. High Perform. Comput. Appl.*, vol. 31, no. 1, pp. 104–113, Jan. 2017.

[4] G. Kan, T. Lei, K. Liang, J. Li, L. Ding, X. He, H. Yu, D. Zhang, D. Zuo, Z. Bao, M. Amo-Boateng, Y. Hu, and M. Zhang, “A multi-core CPU and many-core GPU based fast parallel shuffled complex evolution global optimization approach,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, pp. 332–344, Feb. 2017.

[5] K. Li, “Scheduling parallel tasks with energy and time constraints on multiple manycore processors in a cloud computing environment,” *Future Generation Comput. Syst.*, vol. 82, pp. 591–605, May 2018.

[6] Z. Cheng, E. Shaffer, R. Yeh, G. Zagaris, and L. Olson, “Efficient parallel optimization of volume meshes on heterogeneous computing systems,” *Eng. Comput.*, vol. 33, no. 4, pp. 717–726, Oct. 2017.

[7] T. Rauber and G. Rünger, “Parallel programming models,” in *Parallel Programming: For Multicore and Cluster Systems*. Berlin, Germany: Springer, 2013, pp. 93–98.

[8] A. Arunaran, D. Manjula, and V. Sugamaran, “Task scheduling techniques in cloud computing: A literature survey,” *Future Genes Comput. Syst.*, vol. 91, pp. 407–415, Feb. 2019.

[9] R. Mahmud, R. Kotagiri, and R. Buyya, “Fog computing: A taxonomy, survey and future directions,” *Future Gener. Comput. Syst.* vol. 51, no. 1, pp. 1–38, Jan. 2018.

[10] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, “Mobile edge computing: A survey,” *IEEE Internet Things J.*, vol. 5, no. 1, pp. 450–465, Feb. 2018.

[11] E. Rios, L. S. Ochi, C. Boeres, V. N. Coelho, I. M. Coelho, and R. Farias, “Exploring parallel multi-GPU local search strategies in a metaheuristic framework,” *J. Parallel Distrib. Comput.*, vol. 111, pp. 59–95, Jan. 2018.
[12] N. Melab, J. Gmrs, M. Mezmaz, and D. Tuytten, “Multi-core versus many-core computing for many-task Branch-and-Bound applied to big optimization problems,” Future Gener. Comput. Syst., vol. 82, pp. 472–481, May 2018.

[13] A. Dabah, A. Bendjoudi, A. AitZai, D. El-Baz, and N. N. Taboudjemar, “Hybrid multi-core CPU and GPU-based B&B approaches for the blocking job shop scheduling problem,” J. Parallel Distrib. Comput., vol. 117, pp. 73–86, Jul. 2018.

[14] T.-T. Yu and B. Derbel, “Parallel Branch-and-Bound in multi-core multi-CPU multi-GPU heterogeneous environments,” Future Generation Comput. Syst., vol. 56, pp. 95–109, Mar. 2016.

[15] L. Grigori, S. Cayrols, and J. W. Demmel, “Low rank approximation of a sparse matrix based on LU factorization with column and row tournament pivoting,” SIAM J. Sci. Comput., vol. 40, no. 2, pp. C181–C209, Jan. 2018.

[16] E. Anderson, Z. Bai, and C. Bischof, LAPACK Users’ Guide. Philadelphia, PA, USA: SIAM, 1999.

[17] C. Chen, J. Fang, T. Tang, and C. Yang, “LU factorization on heterogeneous systems: An energy-efficient approach towards high performance,” Computing, vol. 99, no. 8, pp. 791–811, Aug. 2017.

[18] E. Anderson, Z. Bai, and C. Bischof, LAPACK Users’ Guide. Philadelphia, PA, USA: SIAM, 1999.

[19] A. Yarkhan, J. Kurzak, P. Luszczek, and J. Dongarra, “Porting the PLASMA numerical library to the OpenMP standard,” Int. J. Parallel Prog., vol. 45, no. 3, pp. 612–633, Jun. 2017.

[20] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, “Design and implementation of the ScalAPACK LU, QR, and Cholesky factorization routines,” Sci. Program., vol. 5, no. 3, pp. 173–184, Mar. 1996.

[21] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert, “Scheduling block-cyclic array redistribution,” IEEE Trans. Parallel Distrib. Syst., vol. 9, no. 2, pp. 192–205, Feb. 1998.

[22] R. Wu, “A heterogeneous parallel cholesky block factorization algorithm,” in Proc. IPDPS, Atlanta, GA, USA, Apr. 2010, pp. 232–240, Jun. 2010.

[23] J. Dongarra, M. Gates, and A. Haidar, “Accelerating numerical dense linear algebra calculations with GPUs,” in Numerical Computations With GPUs. Cham, Switzerland: Springer, 2014, pp. 3–28.

[24] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, “Dense linear algebra solvers for multicore with GPU accelerators,” in Proc. IEEE Int. Symp. Parallel Distrib. Process., Workshops Phd Forum (IPDPSW), Atlanta, GA, USA, Apr. 2010, pp. 1–8.

[25] K. Kim, S. Rajamanickam, and G. Stelle, “Task parallel incomplete cholesky factorization using 2D partitioned-block layout,” Sandia Corp., Albuquerque, NM, USA, Tech. Rep. DE-AC04-94-AL85000, 2006.

[26] V. Volkov and J. W. Demmel, “LU, QR and Cholesky factorizations using vector capabilities of GPUs,” Dept. EECS, Univ. California, Berkeley, CA, USA;Tech. Rep. UCb/Eecs-2008-49, 2008.

[27] R. Wu, “A heterogeneous parallel cholesky block factorization algorithm,” IEEE Access, vol. 6, pp. 14071-14077, 2018.

[28] R. Wu, “Dynamic scheduling strategy for block parallel cholesky factorization based on activity on edge network,” IEEE Access, vol. 7, pp. 66317–66324, 2019.

[29] M. Deisher, M. Smelyanskiy, B. Nickerson, V. W. Lee, M. Chuvelev, and P. Dubey, “Designing and dynamically load balancing hybrid LU for multi/many-core,” Comput. Sci. Res. Dev., vol. 26, nos. 3–4, pp. 211–220, Jun. 2011.

[30] R. Wu and X. Xie, “A heterogeneous parallel LU factorization algorithm based on a basic column block uniform allocation strategy,” Math. Probl. Eng., vol. 2019, Feb. 2019, Art. no. 3720450.

[31] E. Agullo, O. Beaumont, and L. Eyraud-Dubois, “Are static schedules so bad? A case study on Cholesky factorization,” in Proc. IPDPS, Chicago, IL, USA, 2016, pp. 1021–1030.

[32] S. Souravlas and M. Roumeliotis, “A pipeline technique for dynamic data transfer on a multiprocessor grid,” Int. J. Parallel Program., vol. 32, no. 5, pp. 361–388, Oct. 2004.

[33] F. Dong, S. Tomov, and J. Dongarra, “Efficient support for matrix computations on heterogeneous multi-core and multi-GPU architectures,” Univ. Tennessee Knoxville, Knoxville, TN, USA, Tech. Rep. UT-CS-11-668, Jun. 2011.

[34] RONGTENG WU received the M.S. degree from Fuzhou University, China, in 2004, and the Ph.D. degree from Tianjin University, China, in 2008. He is currently an Associate Professor with the College of Computer and Control Engineering, Minjiang University, China. His research interests include parallel and distributed computing, cloud computing and GPU computing, and image processing.

[35] XIAOHONG XIE received the B.S. degree from Fujian Agriculture and Forestry University, China, in 2008, and the M.S. degree from Fuzhou University, China, in 2013. She is currently an Assistant Research Fellow with the Department of Teaching Affairs, Minjiang University, China. Her current research interests include parallel and distributed computing, cloud computing, and image processing.