Received April 11, 2019, accepted May 15, 2019, date of publication May 20, 2019, date of current version June 4, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2917714

Dynamic Scheduling Strategy for Block Parallel Cholesky Factorization Based on Activity on Edge Network

RONGTENG WU
College of Computer and Control Engineering, Minjiang University, Fuzhou 350108, China
Fujian Provincial Key Laboratory of Information Processing and Intelligent Control, Minjiang University, Fuzhou 350108, China
e-mail: rongtengwu@163.com

This work was supported in part by the National Natural Science Foundation of China under Grant 61871204, in part by the National Natural Science Foundation of China under Grant 61703195, in part by the Natural Science Foundation of Fujian Province under Grant 2015J01659, 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 The efficient development of system software and design applications in parallel architecture is a notable challenge considering various aspects, such as load balancing, memory spaces, communication, and synchronization. This paper presents a block parallel Cholesky factorization algorithm for a multicore system, which is developed based on activity on edge network. First, the basic block computing tasks and their dependencies are taken as vertices and edges, respectively, and a directed acyclic graph corresponding to the specific block parallel Cholesky factorization is generated. Next, each edge of the directed acyclic graph is assigned to a weight equal to the processing time of the initial vertex of the edge, and the directed acyclic graph becomes an activity on edge network with only one starting and one ending vertex. Finally, a queuing algorithm is designed for the basic block computing tasks according to the edge activity on edge network, and a dynamic scheduling strategy is developed for block parallel Cholesky factorization. The results of the experiments concerning the parallel execution time of the algorithm in multicore systems with different configurations demonstrate that the proposed algorithm has notable advantages compared with the traditional static scheduling algorithm, and it exhibits satisfactory load balancing, parallelism, and scalability capacities.

INDEX TERMS Cholesky factorization, dense linear algebra, dynamic schedule strategy, load balancing, multicore computing.

I. INTRODUCTION

Parallelism appears to be the future of computing. In the past twenty years, parallel hardware architecture has evolved dramatically. Chip designers have turned to multicore processors and parallel programming to advance the quest for higher performance [1]–[3]. Multicore processor technologies, from supercomputers to embedded devices, have become ubiquitous in our everyday lives and have exerted a far-reaching impact on the high-performance computing world [4], [5]. The associated large market competence has prompted the industry and academia to develop various advanced technologies and architectures. The parallel programming technique must thus be extended to all areas of software design and development [6]–[8]. Programming in multicore or manycore systems is one of the most prominent challenges considering various aspects such as load balancing, memory spaces, communication, and synchronization [9], [10]. In the past, many attempts have been made to develop parallelizing compilers and parallel algorithms. Matrix factorization is a classical parallel programming algorithm and the key to a linear algebra system. Many problems, such as the solving of linear systems of equations, least squares fitting, and finding eigenvalues, can ultimately be reduced to matrix factorization. Cholesky, LU, and QR factorizations are commonly used to solve dense linear algebra systems and are widely

The associate editor coordinating the review of this manuscript and approving it for publication was Stavros Souravlas.

employed in scientific and engineering models [11]–[14]. Nearly all popular linear algebra libraries, such as the Linear Algebra Package (LAPACK) [15], Scalable Linear Algebra Package (ScaLAPACK) [16], and Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) [17], [18], include Cholesky, LU, and QR factorization routines. Tomov *et al.* [19] presented an approach to solve the dense linear algebra for multicore systems with GPU accelerators. The authors used a high-level parallel programming model and leveraged existing software infrastructure to accelerate the matrix factorization. A parallel Cholesky block factorization algorithm was presented in [10], in which a one-dimensional row block cyclic distribution strategy was employed to reduce the communication cost. Cojean *et al.* [20] proposed an approach that exploited the internal parallelism within tasks and improved the load balancing between CPUs and accelerators by combining a global and a local runtime system. In [21], a task-parallel algorithm was presented for sparse incomplete Cholesky factorization in which a task graph was used to deal with the data dependencies. Buttari *et al.* [22] presented a fine granularity parallel algorithm for the Cholesky, LU and QR factorizations. A heterogeneous factorization algorithm for multicores and multi-GPU systems was developed in [23], and the static data distribution was used to realize load balancing. Lastovetsky and Reddy [24] presented a static data distribution algorithm for dense linear algebra to fit the processors' heterogeneity and memory locality. A parallel LU factorization for heterogeneous systems was developed in [25] by using the static uniform block allocation policy to improve the performance of the algorithm. Endo *et al.* [26] developed a static load balancing algorithm in which all processors participated in updating the trail matrix. In [27], a parallel Cholesky block factorization that used a static block cycle data distribution policy for a multiple processor system was introduced. Generally, the static scheduling algorithm for Cholesky factorization is relatively easier to implement because the number of loops is definite. However, it is difficult to attain complete load balancing by using the static scheduling strategy for existing dependencies among basic block computing tasks from different iterations [22], [28], [29]. Abdelfattah *et al.* [30] discussed the static and dynamic scheduling strategies for Cholesky factorization to develop algorithm parallelism. Agullo et al. [31] performed an extensive analysis and comparison of static and dynamic Cholesky factorization algorithms for systems having multiple CPUs and GPUs. Deisher et al. [32] described a dynamic load balancing algorithm for matrix factorization in a multicore platform; however, they did not consider the parallelism on the first step of each iteration of the Cholesky factorization. In the current study, we develop a dynamic scheduling strategy for block parallel Cholesky factorization, which is based on an activity on edge (AOE) network.

The remainder of this paper is organized as follows: Section II discusses the block parallel Cholesky factorization algorithm. Section III presents the dynamic load balancing strategy based on an AOE network for multicore architecture. Section IV presents the experimental results, and a brief conclusion is presented in Section V.

II. BLOCK PARALLEL CHOLESKY FACTORIZATION

Assuming that A is an $N \times N$ symmetric and positive-definite matrix, it can be factorized into a product of a lower triangular matrix L and its transpose L^T, that is, $A = LL^T$. Since the Cholesky factorization must be applied to a symmetric positive definite matrix, only the lower triangular portion of A is required to be stored. Hence, it can be assumed that the matrix A is stored in the lower triangle of a two-dimensional array, and the computed elements of L overwrite the given elements of A. Generally, if $N \times N$ matrices A and L are partitioned into $n \times n$ blocks of the same size and denoted as

$$A = \begin{pmatrix} A_{00} & & \\ \cdots & \cdots & \\ A_{(n-1)0} & \cdots & A_{(n-1)(n-1)} \end{pmatrix} \tag{1}$$

and

$$L = \begin{pmatrix} L_{00} & & \\ \cdots & \cdots & \\ L_{(n-1)0} & \cdots & L_{(n-1)(n-1)} \end{pmatrix} \tag{2}$$

the block parallel Cholesky factorization can be implemented in n loops.

Algorithm 1 Traditional Block Parallel Cholesky Factorization

```
for ($k$ =0; $k < n$; $k$++){
    DPOTF2($A_{kk}$,$L_{kk}$):
    for ($i = k$+1; $i < n$; $i$++)
        DTRSM($L_{kk}$,$A_{ik}$,$L_{ik}$):
    for ($i = k$+1; $i < n$; $i$++){
        for ($j = k$+1; $j < i$; $j$++)
            DGEMM($L_{ik}$, $L_{jk}$, $A_{ij}$);
        DSYRK($L_{ik}$, $L_{jk}$, $A_{ij}$);
        }
    }
```

Traditionally, block parallel Cholesky factorization can be easily implemented by calling four BLAS and LAPACK subroutines, as shown in algorithm 1, which contains three nested loops. The four subroutines for double precision data are DPOTF2, DTRSM, DGEMM, and DSYRK. The subroutine DPOTF2(A_{kk}, L_{kk}) is used to compute the Cholesky factorization of the matrix blocks on the basis of the equation $A_{kk} = L_{kk}L_{kk}^T$. Given an input matrix block A_{kk}, the output would be the lower triangular matrix block L_{kk}. The subroutine DTRSM(L_{kk}, A_{ik}, L_{ik}) is used to compute $L_{ik}(i > k)$ according to the equation $L_{ik} = A_{ik}L_{kk}^{-T}$. The subroutines DGEMM($L_{ik}$, L_{jk}, A_{ij}) and DSYRK(L_{ik}, L_{jk}, A_{ij}) are used to update the blocks A_{ij} in line with the equation $A_{ij} = A_{ij} - L_{ik}L_{jk}^T$ when $i > j$ and $i = j$, respectively.

All processors will be allocated with specific blocks statically. The k-th loop consists of three sequential steps and two synchronization operations. In step 1, the current processor runs DPOTF2 to factorize the block A_{kk} and get L_{kk}, and then broadcasts L_{kk} if need be. In step 2, all processors concurrently run DTRSM to compute the corresponding block below A_{kk} and get corresponding $L_{ik}(i > k)$, and then broadcast corresponding L_{ik} if need be. In step 3, all processors concurrently run DGEMM or DSYRK in parallel to update the corresponding lower right blocks of A_{kk}. Synchronization operation is required at the start of the both step 2 and step 3, which will lead to processor idle to some extent.

III. DYNAMIC LOAD BALANCING STRATEGY BASED ON AOE NETWORK

A. AOE NETWORK RELATED TO BLOCK PARALLEL CHOLESKY FACTORIZATION

For convenience, we denote the initial values of A and A_{ij} as $A^{(0)}$ and $A_{ij}^{(0)}$, respectively, i.e.,

$$A = A^{(0)} = \begin{pmatrix} A_{00}^{(0)} & & \\ \cdots & \cdots & \\ A_{(n-1)0}^{(0)} & \cdots & A_{(n-1)(n-1)}^{(0)} \end{pmatrix} \tag{3}$$

After the $(k-1)$-th loop, assuming that A is factorized into

$$A = \begin{pmatrix} L_{00} & & & \\ \cdots & \cdots & & \\ L_{k0} & \cdots & A_{kk}^{(k)} & \\ \cdots & \cdots & \cdots & \cdots \\ L_{(n-1)0} & \cdots & A_{(n-1)k}^{(k)} & \cdots & A_{(n-1)(n-1)}^{(k)} \end{pmatrix} \tag{4}$$

and $A^{(k)}$ is denoted as

$$A^{(k)} = \begin{pmatrix} A_{kk}^{(k)} & & \\ \cdots & \cdots & \\ A_{(n-1)k}^{(k)} & \cdots & A_{(n-1)(n-1)}^{(k)} \end{pmatrix} \tag{5}$$

the$k$-th $(0 \le k < n-1)$ loop includes three computing steps, which can be defined as follows:

Step 1: Perform Cholesky factorization $A_{kk}^{(k)}$, i.e., DPOTF2($A_{kk}^{(k)}, L_{kk}$).

Step 2: Compute the lower left blocks of $A^{(k)}$, i.e., DTRSM($L_{kk}, A_{ik}^{(k)}, L_{ik}$), $(i > k)$.

Step 3: Update the lower right blocks of $A^{(k)}$ and obtain $A^{(k+1)}$, i.e., DGEMM(L_{ik}, L_{jk}, $A_{ij}^{(k)}$), $(k < j < i)$ and DSYRK($L_{ik}, L_{jk}, A_{ij}^{(k)}$), $(i = j > k)$.

The $(n-1)$-th loop only includes the first step, i.e., DPOTF2($A_{(n-1)(n-1)}^{(n-1)}, L_{(n-1)(n-1)}$).

In fact, all $A_{ij}^{(k)}$ $(k = 0, \cdots, j)$ are stored in the same storage location, i.e., the location of block A_{ij} of matrix A. We use superscript (k) to distinguish the updating results of different loops. The symbol L_{ij} in matrix A at any loop indicates that the corresponding storage location A_{ij} of matrix A is the final result L_{ij}, and it does not need to be updated in the subsequent loops.

Since dependencies of the three steps exist, it is difficult for the static task distribution strategy to process the parallelism of any two steps. For this reason, we present a dynamic scheduling strategy based on the AOE network. In any k-th $(0 \le k < n-1)$ loop, step 1 contains one basic computing task, i.e., DPOTF2($A_{kk}^{(k)}$, L_{kk}); step 2 contains $(n-1-k)$ basic computing tasks, i.e., DTRSM(L_{kk}, $A_{ik}^{(k)}$, L_{ik}) for $i = k+1, \cdots, n-1$; and step 3 contains $(n-1-k) \times (n-k)/2$ basic computing tasks, i.e., DGEMM($L_{ik}, L_{jk}, A_{ij}^{(k)}$) for $j = k+1, \cdots, n-1$ and $i = j+1, \cdots, n-1$ and DSYRK($L_{ik}, L_{jk}, A_{ij}^{(k)}$) for $i = j = k+1, \cdots, n-1$. These basic computing tasks can be expressed uniquely using the corresponding (i, j, k). For simplicity, we use $A_{ij}^{(k)}$ to represent the corresponding basic computing tasks, where $0 \le k \le j \le i \le n-1$. The number of basic computing tasks for all n loops is

$$\sum_{k=0}^{n-2}[1 + (n-1-k) + \frac{1}{2}(n-1-k)(n-k)] + 1 = \frac{1}{6}(n^3 + 3n^2 + 2n) \tag{6}$$

The computational dependencies among these basic tasks can be summarized as follows:

1) Task $A_{kk}^{(k)}$ $(k > 0)$ depends on the computational results of task $A_{kk}^{(k-1)}$; task $A_{00}^{(0)}$ is the first computational task and does not depend on any other task.

2) Task $A_{ik}^{(k)}$ $(k > 0$ and $i > k)$ depends on the computational results of tasks $A_{ik}^{(k1)}$ and $A_{kk}^{(k)}$, and task $A_{i0}^{(0)}(i > k)$ depends only on the computational results of task $A_{00}^{(0)}$.

3) Task $A_{ii}^{(k)}$ $(k > 0$ and $i > k)$ depends on the computational results of tasks $A_{ii}^{(k1)}$ and $A_{ik}^{(k)}$, and task $A_{ii}^{(0)}$ $(i > k)$ depends on the computational results of task $A_{ik}^{(k)}$.

4) Task $A_{ij}^{(k)}$ $(k > 0$ and $k < j < i)$ depends on the computational results of tasks $A_{ij}^{(k1)}$, $A_{ik}^{(k)}$, and $A_{jk}^{(k)}$; and task $A_{ij}^{(0)}$ $(k < j < i)$ depends on the computational results of tasks $A_{i0}^{(0)}$ and $A_{j0}^{(0)}$.

When the basic tasks are considered vertices and the dependencies of the tasks are considered as edges, a directed acyclic graph (DAG) G_A can be generated corresponding to the block parallel Cholesky factorization. Further, each edge $e_{ij} =< v_i, v_j >$ is assigned a weight equal to the processing time of the initial vertex v_i. As a result, G_A is a weighted DAG and has only one starting vertex and one ending vertex (In this paper, the terms "task" and "vertex" are used interchangeably). In this case, the weighted DAG is also referred to as an AOE network. Figure 1 shows the AOE network corresponding to the block parallel Cholesky factorization for an $N \times N$ matrix A divided into 4×4 blocks. The block Cholesky factorization consists of four loops, and the total number of tasks (or vertices) is $(n^3+3n^2+2n)/6 = 20$. The four weights w_1, w_2, w_3, and w_4 represent the corresponding processing times of four subroutines DPOTF2, DTRSM, DGEMM, and DSYRK for the basic matrix blocks.

The indegree and outdegree of each vertex $A_{ij}^{(k)}$ $(0 \le k \le j \le i \le n-1)$ of the AOE network can be computed easily according to the dependencies of the tasks.

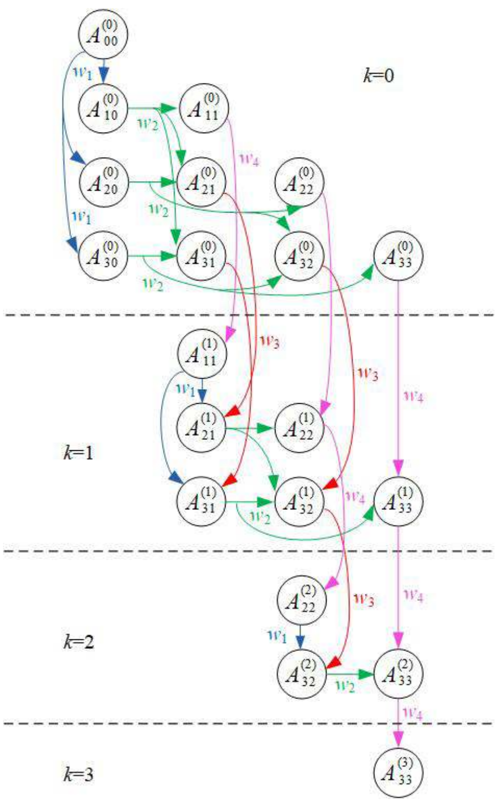

FIGURE 1. Example of an AOE network corresponding to the block parallel Cholesky factorization of a matrix with 4 × 4 blocks.

1) Indegree of vertex:

$$indegree(A_{kk}^{(k)}) = \begin{cases} 0 & k = 0 \\ 1 & k > 0 \end{cases} \tag{7}$$

$$indegree(A_{ik}^{(k)}) = \begin{cases} 1 & i > k = 0 \\ 2 & i > k > 0 \end{cases} \tag{8}$$

$$indegree(A_{ii}^{(k)}) = \begin{cases} 1 & i > k = 0 \\ 2 & i > k > 0 \end{cases} \tag{9}$$

$$indegree(A_{ij}^{(k)}) = \begin{cases} 2 & i > j > k = 0 \\ 3 & i > j > k > 0 \end{cases} \tag{10}$$

2) Outdegree of vertex:

$$outdegree(A_{kk}^{(k)}) = n - 1 - k \tag{11}$$

$$outdegree(A_{ik}^{(k)}) = n - 1 - k, \quad i > k \tag{12}$$

$$outdegree(A_{ij}^{(k)}) = 1, \quad j > k \tag{13}$$

It is thus evident that the successors of $A_{kk}^{(k)}$ are $A_{xk}^{(k)}$ $(x = k+1, \cdots, n-1)$, the successors of $A_{ik}^{(k)}$ $(i > k)$ are $A_{iy}^{(k)}$ $(y = k+1, \cdots, n-1)$ and $A_{zi}^{(k)}$ $(z = i+1, \cdots, n-1)$, and the successor of $A_{ij}^{(k)}$ $(j > k)$ is $A_{ij}^{(k+1)}$.

Similar to the earliest occurrence time and the latest occurrence time of the events (or vertices) of the AOE network, we define the earliest execution start time and the latest execution start time of tasks (vertices) $A_{ij}^{(k)}$ $(0 \leq k \leq j \leq i \leq n-1)$ as follows:

1) The earliest execution start time of task $A_{ij}^{(k)}$ is

$$e_t(A_{ij}^{(k)}) = \begin{cases} 0 & i = j = k = 0 \\ \max\limits_{A_p}\{e_t(A_p) + w_{<A_p, A_{ij}^{(k)}>}\} & \text{others} \end{cases} \tag{14}$$

where A_p is the predecessor of A_{ij}^k, and $w_{<A_p, A_{ij}^k>}$ is the weight of the edge $< A_p, A_{ij}^k >$.

2) The latest execution start time of task $A_{ij}^{(k)}$ is

$$l_t(A_{ij}^{(k)}) = \begin{cases} e_t(A_{(n-1)(n-1)}^{(n-1)}) & i = j = k = n-1 \\ \min\limits_{A_s}\{l_t(A_s) - w_{<A_{ij}^{(k)}, A_s>}\} & \text{others} \end{cases} \tag{15}$$

where A_s is the successor of A_{ij}^k, and $w_{<A_{ij}^k, A_s>}$ is the weight of edge $< A_{ij}^k, A_s >$.

B. QUEUING ALGORITHM

The tasks are divided into ready tasks (tasks of which all predecessors have been computed) and nonready tasks (tasks of which at least one predecessor has not been computed). According to the earliest execution start time e_t and latest execution start time l_t of tasks $A_{ij}^{(k)}$ $(0 \leq k \leq j \leq i \leq n-1)$, the ready queue including all current ready tasks can be created. The rules of queuing the ready tasks are as follows:

Rule 1: The task with a smaller l_t is queued ahead of the task with a larger l_t.

Rule 2: If two tasks have the same l_t, the task with smaller e_t is queued ahead of the task with larger e_t.

Rule 3: If two tasks have the same l_t and e_t, the task with smaller i is queued ahead of the task with larger i. Similarly, in the case of tasks with the same l_t, e_t and i, the task with a smaller j is queued ahead, and in the case of tasks with the same l_t, e_t, i, and j, the task with smaller k is queued ahead.

Rules 1 and 2 ensure that the tasks in the critical path of the AOE network start as early as possible. Rule 3 ensures faster queuing because the number of tasks with the same i is more than the number of tasks with the same j for same l_t and e_t and the number of tasks with the same j is more than the number of tasks with the same k for the same l_t, e_t, and i.

The primary operations pertaining to the ready queue are initialization, enqueuing, and dequeuing. Every node in the ready queue contains the parameters l_t, e_t, i, j, and k, etc.

1) Ready queue initialization is intended to create a ready queue with only one node corresponding to the computing task $A_{00}^{(0)}$ for the nonempty matrix A.

2) Enqueuing involves inserting the node corresponding to a ready task into the ready queue according to the values of l_t, e_t, i, j, and k. The task becomes a ready task when all its predecessors have been computed, that is, the indegree of the task's corresponding vertex is 0.

3) Dequeuing involves removing the head node of the ready queue. When a task has been removed from the ready queue

and has been computed, the indegree of all its successors is subtracted by 1.

The dynamic scheduling strategy for parallel block Cholesky factorization can be described as follows:

Step 1: Initiate the ready queue to create a ready queue with only one node corresponding to $A_{00}^{(0)}$.

Step 2: Run the dequeue operation to remove the head node $A_{ij}^{(k)}$ from the ready queue.

Step 3: Compute task $A_{ij}^{(k)}$.

Step 4: Subtract the indegree of all successors of node $A_{ij}^{(k)}$ by 1.

Step 5: Run the enqueue operation. If the indegree of any successor of the node $A_{ij}^{(k)}$ is equal to 0, the successor is inserted into the ready queue according to the values of l_t, e_t, i, j, and k.

Repeat steps 2 to 5 until the ready queue is empty or all $(n^3 + 3n^2 + 2n)/6$ tasks are computed.

Obviously, the parallelism of the algorithm is a task level parallelism, and the algorithm is well suited to multiple instruction stream, multiple data stream (MIMD) model architecture with shared memory, typically, like multicore system, but not the single instruction stream, multiple data stream (SIMD) model architecture such as GPU processor or very long instruction word (VLIW) processor.

Algorithm 2 Block Parallel Cholesky Factorization Algorithm Implemented by Using the Dynamic Load Balancing Strategy Based on AOE Network

```
InitializeQueue( ); //Initialize the ready queue
pragma omp parallel
While (Queue != empty) {
    #pragma omp critical
    DeQueue ($i$, $j$, $k$); // Dequeuing operation (critical region)
    if ($i == k$)
        DPOTF2 ($A_{kk}^{(k)}, L_{kk}$);
    else if ($j == k$)
        DTRSM ($L_{kk}, A_{ik}^{(k)}, L_{ik}$);
    else if ($i == j$ && $j > k$)
        DSYRK($L_{jk}, L_{jk}, A_{jj}^{(k)}$);
    else
        DGEMM ($L_{ik}, L_{jk}, A_{ij}^{(k)}$);
    Decreasing the indegree of all successors of $A_{ij}^{(k)}$.
    #pragma omp critical
    EnQueue ($i$, $j$, $k$); // Enqueuing operation (critical region)
}
```

IV. EXPERIMENTAL RESULTS

The experiment is performed on the AMAX XG-48201GK system that consists of two Intel Xeon E5-2620 v4 CPUs and eight 16 GB memories. Every CPU contains eight cores, and the frequency is 2.1 GHz.

In our experiment, the routines DPOTF2 in LAPACK and DTRSM, DGEMM and DSYRK in BLAS are used to process the block computing basic tasks for double precision data. Algorithm 2 is the block parallel Cholesky factorization algorithm for a multiple core system with shared memory architecture, implemented by using the dynamic load balancing strategy based on an AOE network.

In algorithm 2, the main thread run is InitializeQueue(), which creates a ready queue with only one node corresponding to the computing task $A_{00}^{(0)}$ for any nonempty matrix A. Both DeQueue(i, j, k) and EnQueue(i, j, k) act as critical regions. The former removes the head node of the ready queue and returns the corresponding (i, j, k), and the latter inserts the ready successors of $A_{ij}^{(k)}$ into the ready queue based on rules 1, 2, and 3.

A. PARALLEL EXECUTION TIME

The parallel execution time of the parallel block Cholesky factorization algorithm based on an AOE network for a given $N \times N$ double precision matrix is related to the number of CPU cores available and the number of matrix blocks, n. Figure 2 shows the performance of the algorithm for a double precision matrix with $N = 10000$ and $N = 20000$. The x-axis represents the number of CPU cores from 2 to 16; the y-axis shows the number of matrix blocks, n, from 20 to 200; and the z-axis indicates the parallel execution time in seconds. A larger number of cores available in the system pertains to lower parallel execution time for a given matrix and fixed n. However, when the block number n is increased, the parallel execution time first decreases and later increases gradually. The optimum number of blocks n for a two CPU core system for a double precision matrix with sizes of both 10000×10000 and 20000×20000 is 25. However, the optimum number of blocks, n, is approximately 50 or 80 for $N = 10000$ and approximately 50 for $N = 20000$ in the system having more than four cores.

To compare the performance of the presented algorithm with the conventional parallel algorithm described as algorithm 1, the experiment is performed on a multicore system for a double precision matrix with sizes $N = 10000$ and $N = 20000$. Figure 3 shows the performance results of the two algorithms for systems with different configurations and different numbers of blocks, n. The parallel execution times of both the presented algorithm and traditional algorithm decrease gradually when the number of cores is increased from 2 to 16, and the best performance is obtained when n is approximately 25 for the system with two cores and approximately 50 and 80 for systems with more than two cores. The best performance of our algorithm is thus better than that of the traditional algorithm.

B. OVERHEAD

The overhead must be considered in the parallel block Cholesky algorithm based on an AOE network because the queue is introduced to implement the dynamic

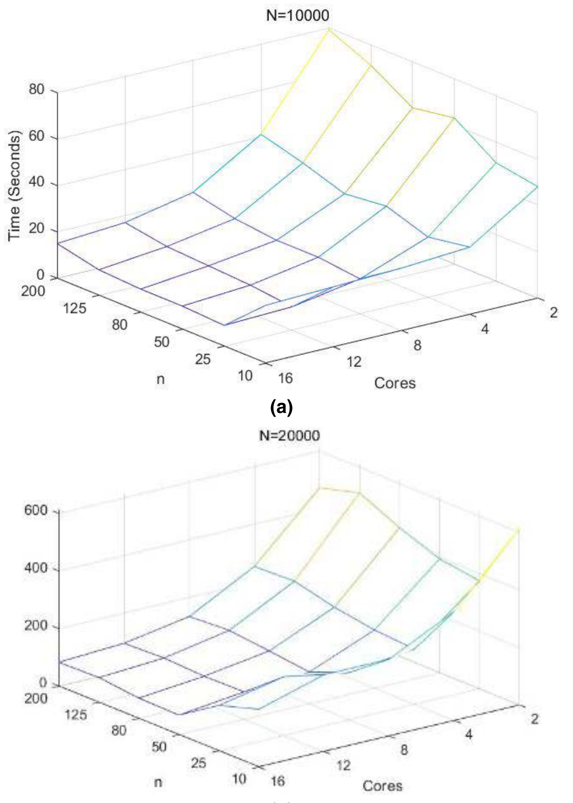

FIGURE 2. Performance of the dynamic scheduling Cholesky factorization algorithm based on AOE network for double precision $N \times N$ matrix. (a) $N = 10000$. (b) $N = 20000$.

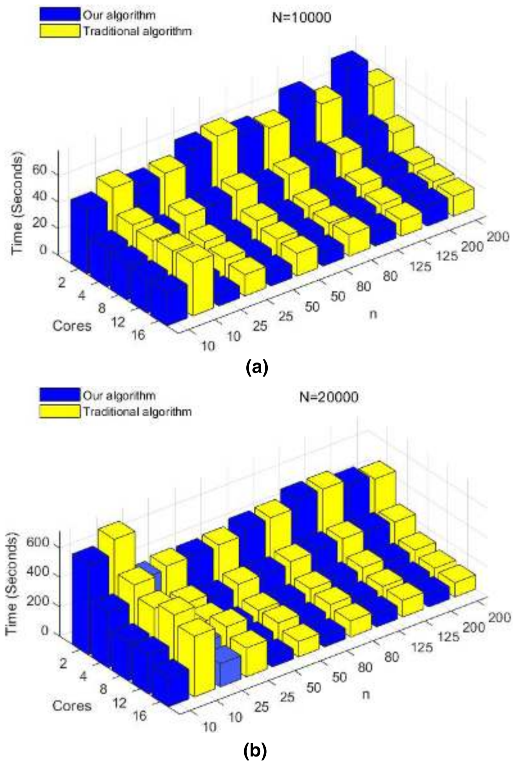

FIGURE 3. Performance comparison of the dynamic scheduling Cholesky factorization algorithm based on AOE network and the traditional block parallel Cholesky factorization algorithm for different multicore configurations and different numbers of blocks, n, (a) $N = 10000$, (b) $N = 20000$.

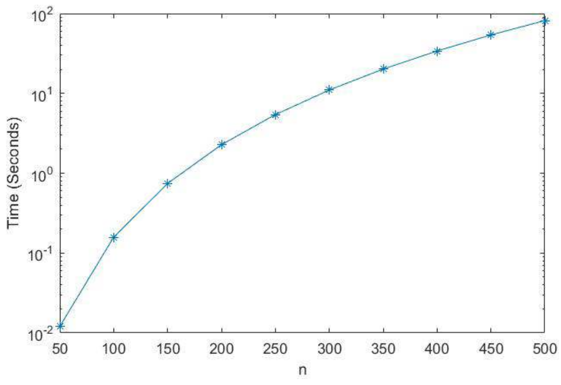

FIGURE 4. Queuing time of the dynamic scheduling Cholesky factorization algorithm based on AOE network for different numbers of blocks, n.

scheduling strategy. This section discusses the queuing time of the algorithm for a multicore system. The total queuing time consists primarily of the enqueue and the dequeue operations; therefore, the queuing time depends mostly on the number of blocks, n and the performance of the cores, and not on the matrix size N. The number of matrix blocks, n must be increased with increase in the available cores to obtain a better performance of the general parallel block Cholesky factorization algorithm in a multicore system. However, a larger n means that a larger number of smaller tasks are scheduled, which results in performance degradation. Therefore, n cannot be excessively large or small.

Figure 4 shows the queuing time of the algorithm for different numbers of blocks, n. The queuing time increases rapidly with increase in n. In the experiment, the queuing time is approximately 0.15 s when $n = 100$ and 81 s when $n = 500$; however, its value is less than 3 s when n is less than 200. In the performance evaluation experiment, the optimum number of blocks, n, is approximately 50 or 80, and the queuing time is less than 1 s, which is representative of less than 1 percent for each core. Generally, the queuing time is small and can be ignored for each core when n is not large.

C. SCALABILITY

The scalability is used to evaluate the acceleration of an algorithm to solve a specific problem when more computing resources are available; it is also used to evaluate the capability of an algorithm to solve potentially larger problems. First, we tested how the parallel execution time changes when the number of CPU cores is increased gradually for a fixed-size matrix. Next, we measured the parallel execution time while increasing the matrix size to test the scalability.

Figure 5 shows the parallel execution times of the presented algorithm and the traditional algorithm for a 20000×20000 double precision matrix, when the algorithms are run on different systems configured with different numbers of cores. When the number of cores is increased

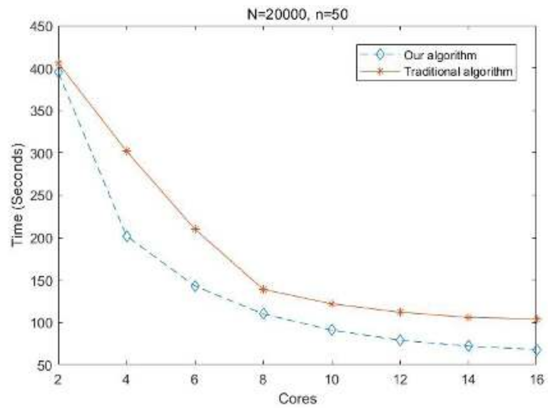

FIGURE 5. Parallel execution times of the dynamic scheduling Cholesky factorization algorithm based on AOE network and the traditional block parallel Cholesky factorization algorithm for double precision 20000 × 20000 matrix on systems with different configurations.

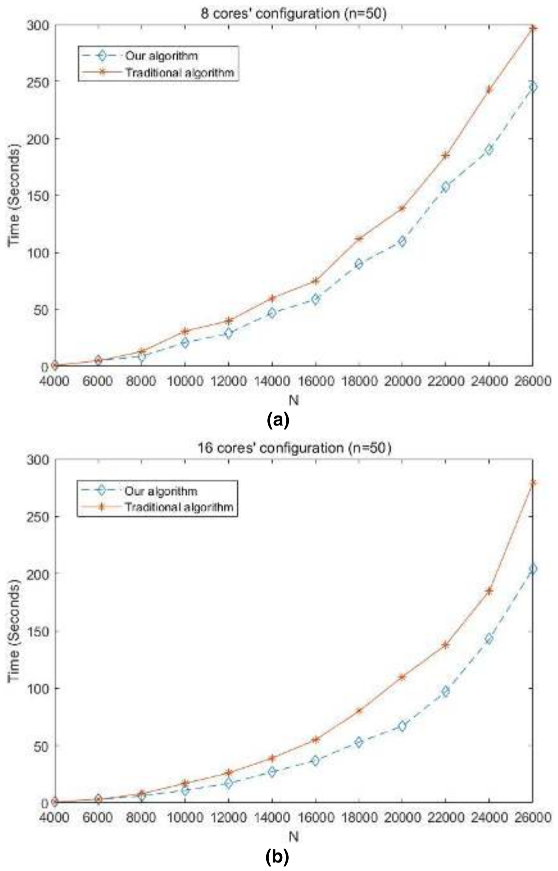

FIGURE 6. Parallel execution times of the dynamic scheduling Cholesky factorization algorithm based on AOE network and the traditional block parallel Cholesky factorization algorithm for different sized matrices on systems having 8 and 16 cores. (a) 8 cores, (b) 16 cores.

from 2 to 16, the parallel execution times of both the algorithms decrease gradually. The two algorithms exhibit nearly the same scalability. However, the presented algorithm has a smaller parallel execution time in different system configurations.

Figure 6 shows the parallel execution times for the presented algorithm and traditional algorithm for different sized matrices. The x-axis presents the matrix size from 4000 to 26000. The y-axis indicates the parallel execution of the algorithm. Each subfigure consists of two curves that correspond to the presented algorithm and the traditional algorithm. Every curve approximately represents the graph of a function similar to $y = a \cdot x^3$, which is consistent with the theoretical results. Furthermore, the presented algorithm demonstrates better time performance for different sized matrices.

V. CONCLUSION

Multicore technologies, from supercomputers to laptops and embedded devices, have become ubiquitous in our everyday lives. The parallel programming technique must thus be extended to all areas of software development. In this paper, we present a block parallel Cholesky factorization algorithm for a multicore system. Traditionally, the block parallel Cholesky factorization algorithm consists of $n-1$ loops and $(n^3 + 3n^2 + 2n)/6$ basic block computing tasks. When the basic block computing tasks are taken as vertices and the dependencies of the tasks are considered edges, a directed acyclic graph can be generated in accordance with the block parallel Cholesky factorization. We assign each edge a weight that is equal to the processing time of the graph's initial vertex; the DAG thus becomes an AOE network with only one starting vertex and one ending vertex. On the basis of this AOE network, we present a queuing algorithm for basic block computing tasks and develop a dynamic load balancing algorithm for block parallel Cholesky factorization. Experiments to determine the parallel execution time in multicore systems with different configurations demonstrate that the proposed algorithm has notable advantages compared with the traditional static scheduling algorithm. The experimental results also indicate that the queuing time is small and can be ignored for each core when n is not large, and that the proposed algorithm exhibits satisfactory scalability.

REFERENCES

[1] G. Blake, R. G. Dreslinski, and T. Mudge, "A survey of multicore processors," *IEEE Signal Process. Mag.*, vol. 26, no. 6, pp. 26–37, Nov. 2009.

[2] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin, "An evaluation of emerging many-core parallel programming models," in *Proc. ACM 7th Int. Workshop Program. Models Appl. Multicores Manycores*, New York, NY, USA, 2016, pp. 1–10.

[3] P. Thoman *et al.*, "A taxonomy of task-based parallel programming technologies for high-performance computing," *J. Supercomput.*, vol. 74, no. 4, pp. 1422–1434, Apr. 2018.

[4] J. Sahuquillo, S. Petit, V. Selfa, and M. E. Gómez, "A research-oriented course on advanced multicore architecture," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshop*, Hyderabad, India, May 2015, pp. 760–765.

[5] 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, Aug. 2017.

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

[7] R. Mahmud, R. Kotagiri, and R. Buyya, "Fog computing: A taxonomy, survey and future directions," in *Internet of Everything*. Singapore: Springer, 2018, pp. 103–130.

[8] 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.

[9] P. Valero-Lara *et al.*, "Multicore and manycore: Hybrid computing architectures and applications," in *Innovative Research and Applications in Next-Generation High Performance Computing*. Hershey, PA, USA: IGI Global, 2016, pp. 107–158.
[10] R. Wu, "A heterogeneous parallel Cholesky block factorization algorithm," *IEEE Access*, vol. 6, pp. 14071–14077, 2018.
[11] 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.
[12] 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, May 2018.
[13] T. Dong, A. Haidar, P. Luszczek, J. A. Harris, S. Tomov, and J. Dongarra, "LU factorization of small matrices: Accelerating batched DGETRF on the GPU," in *Proc. HPCC/CSS/ICESS*, 2014, pp. 157–160.
[14] J. Dongarra *et al.*, "The design and performance of batched BLAS on modern high-performance computing systems," *Procedia Comput. Sci.*, vol. 108, pp. 495–504, Jun. 2017.
[15] E. Anderson *et al.*, *LAPACK Users' Guide*. Philadelphia, PA, USA: SIAM, 1999.
[16] J. Choi *et al.*, "ScaLAPACK: A portable linear algebra library for distributed memory computers—Design issues and performance," *Comput. Phys. Commun.*, vol. 97, nos. 1–2, pp. 1–15, 1996.
[17] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, "Scheduling dense linear algebra operations on multicore processors," *Concurrency Comput., Pract. Exper.*, vol. 22, no. 1, pp. 15–44, Jan. 2009.
[18] A. YarKhan, J. Kurzak, P. Luszczek, and J. Dongarra, "Porting the PLASMA numerical library to the OpenMP standard," *Int. J. Parallel Program.*, vol. 45, no. 3, pp. 612–633, Jun. 2017.
[19] 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.
[20] T. Cojean, A. Guermouche, A. Hugo, R. Namyst, and P. A. Wacrenier, "Resource aggregation for task-based Cholesky factorization on top of modern architectures," *Parallel Comput.*, vol. 83, pp. 73–92, Apr. 2019.
[21] K. Kim, S. Rajamanickam, G. Stelle, H. C. Edwards, and S. L. Olivier, "Task parallel incomplete Cholesky factorization using 2D partitioned-block layout," Sandia Corp., Albuquerque, NM, USA, Tech. Rep. DE-AC04-94-AL85000, 2016.
[22] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Comput.*, vol. 35, no. 1, pp. 38–53, Jan. 2009.
[23] 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.
[24] A. Lastovetsky and R. Reddy, "Data distribution for dense factorization on computers with memory heterogeneity," *Parallel Comput.*, vol. 33, pp. 757–779, Dec. 2007.
[25] R. Wu and X. Xie, "A heterogeneous parallel LU factorization algorithm based on a basic column block uniform allocation strategy," *Math. Problems Eng.*, vol. 2019, Feb. 2019, Art. no. 3720450.
[26] T. Endo, S. Matsuoka, A. Nukada, and N. Maruyama, "Linpack evaluation on a supercomputer with heterogeneous accelerators," in *Proc. IPDPS*, Atlanta, GA, USA, 2010, pp. 1–8.
[27] J. Choi *et al.*, "Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines," *Sci. Program.*, vol. 5, no. 3, pp. 173–184, 1996.
[28] J. Kurzak, P. Luszczek, M. Faverge, and J. Dongarra, "LU factorization with partial pivoting for a multicore system with accelerators," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 8, pp. 1613–1621, Aug. 2013.
[29] S. S. Catalán *et al.*, "Static versus dynamic task scheduling of the Lu factorization on ARM big. LITTLE architectures," in *Proc. IPDPSW*, Lake Buena Vista, FL, USA, 2017, pp. 733–742.
[30] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Fast Cholesky factorization on GPUs for batch and native modes in MAGMA," *J. Comput. Sci.*, vol. 20, pp. 85–93, May 2017.
[31] E. Agullo, O. Beaumont, L. Eyraud-Dubois, and S. Kumar, "Are static schedules so bad? A case study on Cholesky factorization," in *Proc. IEEE IPDPS*, Chicago, IL, USA, May 2016, pp. 1021–1030.
[32] 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. Develop.*, vol. 26, nos. 3–4, pp. 211–220, Jun. 2011.

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.